ОБЩЕСТВО С ОГРАНИЧЕННОЙ ОТВЕТСТВЕННОСТЬЮ «ИТС ЛАБ»
(ООО «ИТС ЛАБ»)

УТВЕРЖДАЮ
Генеральный директор
ООО «ИТС ЛАБ»

_____ Е.И.Ткаченко
«28» марта 2025 г.

СПЕЦИАЛЬНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ
КОНТРОЛЬНЫЕ ЛИСТЫ СВЕТОФОРНЫХ ОБЪЕКТОВ
(реестровая запись в РРПО №)

Текст программы

ЛИСТ УТВЕРЖДЕНИЯ

RU.94076251.00004-01 16 1-ЛУ

СОГЛАСОВАНО

Ведущий инженер
_____ М.О. Галиновский
«28» марта 2025 г.

2025

УТВЕРЖДЕН
RU.94076251.00004-01 16 1-ЛУ

СПЕЦИАЛЬНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ КОНТРОЛЬНЫЕ ЛИСТЫ СВЕТОФОРНЫХ ОБЪЕКТОВ

(реестровая запись в РРПО №)

Текст программы

RU.94076251.00004-01 16 1-ЛУ

Листов 92

2025

АННОТАЦИЯ

Настоящий документ представляет собой текст программы специального программного обеспечения «Контрольные листы светофорных объектов» (реестровая запись в РРПО №) (далее – СПО «КЛ СО»).

Документ разработан согласно ГОСТ 19.401-78, структура и оформление документа соответствуют ГОСТ 19.105-78, основные надписи титульной части – по ГОСТ 19.104-78, выполнен печатным способом согласно ГОСТ 19.106-78.

Основная часть документа содержит символическую запись на исходном языке.

СОДЕРЖАНИЕ

СПО КОНТРОЛЬНЫЕ ЛИСТЫ СВЕТОФОРНЫХ ОБЪЕКТОВ
(РЕЕСТРОВАЯ ЗАПИСЬ В РРПО №)

Программа СПО КЛ СО состоит из следующих скриптов:

– скрипт основной функции СПО КЛ СО;

– скрипт настройки подключения к базе данных PostgreSQL;

– скрипт для настройки подключения к хранилищу объектов S3;

– скрипт предназначен для определения моделей базы данных с использованием SQLAlchemy ORM;

– скрипт предназначен для формирования и экспорта данных в формате XLSX;

– скрипт для обеспечения взаимодействия с базой данных;

– скрипт для реализации вспомогательных функций;

– скрипт для реализации вспомогательных функций;

– скрипт для обработки API-запросов;

– скрипт для обработки API-запросов;

– скрипт для обработки API-запросов, связанных с видами перекрестков;

1. Скрипт основной функции СПО КЛ СО

Скрипт main.py, являющийся неотъемлемой частью текста программы, предназначен для инициализации и запуска FastAPI приложения. Регистрирует все маршрутизаторы API и запускает веб-сервер для обработки запросов.

Входные данные: параметры запуска из настроек приложения.

Выходные данные: запущенное FastAPI приложение, обрабатывающее HTTP-запросы Входные данные отсутствуют. Выходные данные отсутствуют.

Символическая запись на исходном языке:

```
import uvicorn
from fastapi import FastAPI
```

```
from settings import settings
from api.routers.main_router import main_router
from api.routers.user import user_router
from api.routers.junction_kind import junctions_kind_router
from api.routers.junction import junction_router
from api.routers.supports import supports_router
from api.routers.controllers import controllers_router
from api.routers.power_backup import power_backups_router
from api.routers.cables import cables_router
from api.routers.wells import wells_router
from api.routers.traffic_lights import traffic_lights_router
from api.routers.tvp import tvp_router
from api.routers.accessible_devices import accessible_devices_router
from api.routers.road_markings import road_markings_router
from api.routers.autonoms_power_sources import autonoms_power_sources_router
from api.routers.toov import toov_router
from api.routers.detector_camera import detectors_cameras_router
from api.routers.tsodd import tsodd_router
from api.routers.light_sections import light_sections_router
from api.routers.additional_info import additional_info_router
from api.routers.files import files_router
from api.routers.export import export_router

app = FastAPI(title='Check list backend',
description='Бэкенд сервиса "Контрольные листы"')

app.include_router(main_router)
app.include_router(user_router)
app.include_router(junctions_kind_router)
```

```
app.include_router(junction_router)
app.include_router(files_router)
app.include_router(supports_router)
app.include_router(controllers_router)
app.include_router(power_backups_router)
app.include_router(cables_router)
app.include_router(wells_router)
app.include_router(traffic_lights_router)
app.include_router(tvp_router)
app.include_router(accessible_devices_router)
app.include_router(road_markings_router)
app.include_router(autonoms_power_sources_router)
app.include_router(toov_router)
app.include_router(detectors_cameras_router)
app.include_router(tsodd_router)
app.include_router(light_sections_router)
app.include_router(additional_info_router)
app.include_router(export_router)

if __name__ == '__main__':
    uvicorn.run("main:app", host=settings.app_host, port=settings.app_port)
```

2. Скрипт настройки подключения к базе данных PostgreSQL

Скрипт orm.py, являющийся неотъемлемой частью текста программы, предназначен для настройки подключения к базе данных PostgreSQL с использованием SQLAlchemy ORM. Реализует инициализацию асинхронного движка базы данных и создание сессий.

Входные данные: параметры подключения к базе данных из настроек приложения.

Выходные данные: экземпляр класса Database для взаимодействия с базой данных.

Символическая запись на исходном языке:

```
from sqlalchemy.ext.asyncio import create_async_engine, async_sessionmaker
from sqlalchemy.ext.asyncio import AsyncSession
from sqlalchemy.ext.declarative import declarative_base
from src.settings import settings
class Database:
    def __init__(self):
        self.postgres_host: str = settings.postgres_host
        self.postgres_port: int = settings.postgres_port
        self.postgres_user: str = settings.postgres_user
        self.postgres_pass: str = settings.postgres_pass
        self.postgres_db: str = settings.postgres_db
        self.engine = create_async_engine(
            f"postgresql+asyncpg://{self.postgres_user}:{self.postgres_pass}@"
            f"{self.postgres_host}:{self.postgres_port}/{self.postgres_db}",
        )
        self.AsyncSession = async_sessionmaker(bind=self.engine, class_=AsyncSession,
expire_on_commit=False)
        self.Base = declarative_base()
    async def get_db(self):
        """Предоставляет сессию базы данных для взаимодействия."""
        # database = self.AsyncSession()
        # try:
        #     yield database
        # finally:
        #     database.close()
```

```
    async with self.AsyncSession() as session:
        yield session
db = Database()
```

3. Скрипт для настройки подключения к хранилищу объектов S3

Скрипт s3.py, являющийся неотъемлемой частью текста программы, предназначен для настройки подключения к хранилищу объектов S3. Реализует инициализацию клиента S3, проверку наличия и создание корзины для хранения файлов.

Входные данные: параметры подключения к S3 из настроек приложения.

Выходные данные: экземпляр клиента S3 для взаимодействия с хранилищем объектов.

```
import boto3
from settings import settings
from botocore.exceptions import EndpointConnectionError
class S3Connector:
  def __init__(self):
    self.s3_host: str = str(settings.s3_host)
    self.s3_port: str = str(settings.s3_port)
    self.s3_access_key: str = str(settings.s3_username)
    self.s3_secret_key: str = str(settings.s3_password)
    self.s3_bucket: str = str(settings.s3_bucket)
    self.session = boto3.session.Session()
    self.client = self.session.client(service_name='s3',
                                      endpoint_url=f'http://{self.s3_host}:{self.s3_port}',
                                      use_ssl=False,
                                      aws_access_key_id=self.s3_access_key,
```

```
                                aws_secret_access_key=self.s3_secret_key,
                                )
        # создание корзины, если не создана ранее
        try:
            if not self.check_bucket_exists(self.s3_bucket):
                self.create_bucket(self.s3_bucket)
        except EndpointConnectionError as ECE:
            print(f'Не удается подключиться к S3 - {ECE}')
    def get_client(self):
        print('Создание клиента S3')
        return self.client
    def check_bucket_exists(self, bucket_name: str) -> bool:
        is_exist = False
        list_buckets = self.client.list_buckets()
        if list_buckets and bucket_name in [bucket['Name'] for bucket in
list_buckets['Buckets']]:
            is_exist = True
        print(f'Корзина S3 {bucket_name} существует') if is_exist else print(
            f'Корзина S3 {bucket_name} не существует')
        return is_exist

    def create_bucket(self, bucket_name: str):
        print(f'Создание корзины в S3: {self.s3_bucket}')
        self.client.create_bucket(Bucket=bucket_name)
s3_connector = S3Connector()
s3_client = s3_connector.get_client()
```

4. Скрипт предназначен для определения моделей базы данных с использованием SQLAlchemy ORM

Скрипт db_models.py, являющийся неотъемлемой частью текста программы, предназначен для определения моделей базы данных с использованием SQLAlchemy ORM. Содержит описания таблиц базы данных, их полей и связей между ними.

Входные данные: нет прямых входных данных.

Выходные данные: определения классов моделей для взаимодействия с базой данных.

Символическая запись на исходном языке:

```
from sqlalchemy import (Column, String, Text, ForeignKey, TIMESTAMP,
DECIMAL, Enum, Boolean, func)
from sqlalchemy.orm import relationship
from .orm import db
import datetime
import enum
class User(db.Base):
    __tablename__ = "user"
    __table_args__ = {'comment': 'Таблица пользователей '}
    # Поля таблицы
    id = Column(String(128), primary_key=True, index=True,
comment='Идентификатор (CUID2)')
    name = Column(String(128), nullable=False, comment='ФИО пользователя')
    login = Column(String(128), unique=True, nullable=False, comment='Логин')
    password = Column(String(128), nullable=False, comment='Пароль')
    created_by = Column(String(128), ForeignKey("user.id", ondelete='SET NULL'),
                        comment='ID пользователя, добавившего запись')
```

```
    created_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
comment='Время создания')
    updated_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
onupdate=func.now(),
                        comment='Время последнего обновления')
    creator = relationship("User", remote_side=[id], backref="created_users")  # Ссылка
на добавившего пользователя
# Таблица видов перекрестков
class JunctionsKind(db.Base):
    __tablename__ = "junctions_kind"
    __table_args__ = {'comment': 'Таблица видов перекрестков'}

    id = Column(String(128), primary_key=True, index=True,
comment='Идентификатор (CUID2)')
    kind = Column(String(128), nullable=False, comment='Название вида
перекрестка')
    created_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
comment='Время создания')
    updated_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
onupdate=func.now(),
                        comment='Время последнего обновления')

    # Связь с таблицей junctions
    junctions = relationship("Junctions", back_populates="kind")

# Enum для статуса осмотра
class InspectionStatus(enum.Enum):
    INSPECTED = "Inspected"
    NOT_INSPECTED = "Not inspected"
```

```
# Таблица объектов осмотра (перекрестков)
class Junctions(db.Base):
    __tablename__ = "junctions"
    __table_args__ = {'comment': 'Таблица объектов осмотра (перекрестков)'}

    id = Column(String(128), primary_key=True, index=True,
comment='Идентификатор (CUID2)')
    kind_id = Column(String(128), ForeignKey("junctions_kind.id"), nullable=False,
                     comment='Вид перекрестка (ссылка на junctions_kind)')
    region = Column(Text, nullable=True, comment='Наименование региона (субъекта
РФ)')
    locality = Column(Text, nullable=True, comment='Наименование населенного
пункта')
    address = Column(Text, nullable=True, comment='Адрес расположения')
    longitude = Column(DECIMAL(9, 6), nullable=False, comment='Долгота')
    latitude = Column(DECIMAL(9, 6), nullable=False, comment='Широта')
    status = Column(Boolean, nullable=True, default=False, server_default='false',
                    comment='Узел оборудованный/не оборудованный')
    number = Column(String(10), nullable=False, comment='Номер документа')
    project = Column(Text, nullable=True, comment='Наименование проекта')
    inspection_status = Column(Enum(InspectionStatus), nullable=True,
default=InspectionStatus.NOT_INSPECTED,
                               server_default=InspectionStatus.NOT_INSPECTED.name,
                               comment='Статус осмотра')
    inspection_at = Column(TIMESTAMP(timezone=True), comment='Дата осмотра')

    executor_name = Column(Text, nullable=True, comment='ФИО пользователя')
```

```
    executor_position = Column(Text, nullable=True, comment='Должность
пользователя')
    comment = Column(Text, nullable=True, comment='Комментарий')
    created_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
comment='Время создания')
    updated_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
onupdate=func.now(),
                        comment='Время последнего обновления')
    kind = relationship("JunctionsKind", back_populates="junctions")  # Связь с
junctions_kind
    files = relationship("Files", back_populates="junction", cascade="all, delete")
    supports = relationship("Supports", back_populates="junction", cascade="all,
delete")
    controllers = relationship("Controllers", back_populates="junction", cascade="all,
delete")
    power_backup = relationship("PowerBackup",
back_populates="junction",cascade="all, delete")
    cables = relationship("Cables", back_populates="junction", cascade="all, delete")
    wells = relationship("Wells", back_populates="junction",  cascade="all, delete")
    traffic_lights = relationship("TrafficLights", back_populates="junction",
cascade="all, delete")
    tvp = relationship("TVP", back_populates="junction", cascade="all, delete")
    accessible_signaling_devices = relationship("AccessibleSignalingDevices",
                                                back_populates="junction",
                                                cascade="all, delete")
    road_markings = relationship("RoadMarkings", back_populates="junction",
cascade="all, delete")
    autonoms_power_sources = relationship("AutonomsPowerSources",
back_populates="junction",cascade="all, delete")
```

```
    toov = relationship("TOOV", back_populates="junction", cascade="all, delete")
    detector_camera = relationship("DetectorCamera", back_populates="junction",
cascade="all, delete")
    tsodd = relationship("TSODD", back_populates="junction",cascade="all, delete")
    light_sections = relationship("LightSections",
back_populates="junction",cascade="all, delete")
    additional_info = relationship("AdditionalInfo", back_populates="junction",
cascade="all, delete")
class Files(db.Base):
    __tablename__ = "files"
    __table_args__ = {'comment': 'Таблица прикрепленных к осмотру файлов'}

    id = Column(String(128), primary_key=True, index=True,
comment='Идентификатор (CUID2)')
    junction_id = Column(String(128), ForeignKey("junctions.id",
ondelete="CASCADE"), index=True, nullable=True,
                         comment='id объекта/осмотра')
    link = Column(Text, nullable=False, comment='Ссылка на файл')
    created_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
comment='Время создания')
    updated_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
onupdate=func.now(),
                        comment='Время последнего обновления')

    junction = relationship("Junctions", back_populates='files')  # Связь с таблицей
junction

# Enum для состояния объекта
class CheckObjectState(enum.Enum):
```

```
    # INFO:
    # Добавление уже существующего enum в новые таблицы требует ручной модификации миграций alembic ->
    # В миграции добавить "from sqlalchemy.dialects.postgresql import ENUM"
    # В миграции перед def upgrade() добавить "check_object_state_enum = Enum(CheckObjectState, name="checkobjectstate", create_type=False)"
    # Поменять в миграции sa.Enum('GOOD', 'BAD', name='checkobjectstate') на check_object_state_enum.

    GOOD = "Good"
    BAD = "Bad"
check_object_state_enum = Enum(CheckObjectState, name="checkobjectstate", create_type=False)
class Supports(db.Base):
    __tablename__ = "supports"
    __table_args__ = {'comment': 'Конструкции для размещения элементов светофорного объекта'}

    id = Column(String(128), primary_key=True, index=True, comment='Идентификатор (CUID2)')
    junction_id = Column(String(128), ForeignKey("junctions.id", ondelete="CASCADE"), index=True, nullable=False,
                         comment='id объекта/осмотра')
    name = Column(String(128), nullable=False, comment='Название осматриваемого элемента')

    verticality_sagging_state = Column(check_object_state_enum, nullable=True, default=None,
```

```
                                comment='Вертикальность стойки/опоры, провисание
троса')
    verticality_sagging_description = Column(String(128), nullable=True,
comment='Описание')

    base_state = Column(check_object_state_enum, nullable=True, default=None,
                        comment='Состояние фундамента')
    base_description = Column(String(128), nullable=True, comment='Описание')

    inscriptions_advertisement_state = Column(check_object_state_enum,
nullable=True, default=None,
                                              comment='Наличие несанкционированных надписей и
рекламы')
    inscriptions_advertisement_description = Column(String(128), nullable=True,
                                                    comment='Описание')

    dirt_sagging_state = Column(check_object_state_enum, nullable=True,
default=None,
                                comment='Наличие внешних загрязнений')
    dirt_sagging_description = Column(String(128), nullable=True,
comment='Описание')

    rust_color_state = Column(check_object_state_enum, nullable=True, default=None,
                              comment='Наличие ржавчины и повреждений лакокрасочного
покрытия')
    rust_color_description = Column(String(128), nullable=True, comment='Описание')

    hatch_state = Column(check_object_state_enum, nullable=True, default=None,
```

```
                comment='Наличие повреждений (дефектов) люка (люков) на
опоре')
    hatch_description = Column(String(128), nullable=True, comment='Описание')

    broken_constructions_state = Column(check_object_state_enum, nullable=True,
default=None,
                                        comment='Наличие повреждений конструкций')
    broken_constructions_description = Column(String(128), nullable=True,
                                              comment='Описание')

    created_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
comment='Время создания')
    updated_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
onupdate=func.now(),
                        comment='Время последнего обновления')

    junction = relationship("Junctions", back_populates='supports')  # Связь с таблицей
junction
class Controllers(db.Base):
    __tablename__ = "controllers"
    __table_args__ = {'comment': 'Дорожные контроллеры и ВПУ'}

    id = Column(String(128), primary_key=True, index=True,
comment='Идентификатор (CUID2)')
    junction_id = Column(String(128), ForeignKey("junctions.id",
ondelete="CASCADE"), index=True, nullable=False,
                         comment='id объекта/осмотра')
    name = Column(String(128), nullable=False, comment='Название осматриваемого
элемента')
```

```
    dk_name = Column(Text, nullable=True, comment='Наименование дорожного
контроллера')
    dk_serial = Column(Text, nullable=True, comment='Серийный номер дорожного
контроллера')

    position_state = Column(check_object_state_enum, nullable=True, default=None,
                            comment='Контроль положения')
    position_description = Column(String(128), nullable=True, comment='Описание')

    inscriptions_advertisement_state = Column(check_object_state_enum,
nullable=True, default=None,
                                              comment='Наличие несанкционированных надписей и
рекламы')
    inscriptions_advertisement_description = Column(String(128), nullable=True,
comment='Описание')

    dirt_state = Column(check_object_state_enum, nullable=True, default=None,
                        comment='Наличие внешних загрязнений')
    dirt_description = Column(String(128), nullable=True, comment='Описание')

    rust_color_state = Column(check_object_state_enum, nullable=True, default=None,
                              comment='Наличие ржавчины и повреждений лакокрасочного
покрытия')
    rust_color_description = Column(String(128), nullable=True, comment='Описание')

    cabinet_state = Column(check_object_state_enum, nullable=True, default=None,
                           comment='Наличие повреждений конструкции шкафа')
    cabinet_description = Column(String(128), nullable=True, comment='Описание')
```

```
    cabinet_hardware_state = Column(check_object_state_enum, nullable=True,
default=None,
                                    comment='Состояние фурнитуры (петли дверей, замок)')
    cabinet_hardware_description = Column(String(128), nullable=True,
comment='Описание')

    dirt_inside_state = Column(check_object_state_enum, nullable=True, default=None,
                               comment='Наличие загрязнений внутри')
    dirt_inside_description = Column(String(128), nullable=True, comment='Описание')

    base_state = Column(check_object_state_enum, nullable=True, default=None,
                        comment='Состояние фундамента шкафа')
    base_description = Column(String(128), nullable=True, comment='Описание')

    time_accuracy_state = Column(check_object_state_enum, nullable=True,
default=None,
                                 comment='Точность времени в элементах управления')
    time_accuracy_description = Column(String(128), nullable=True,
comment='Описание')

    passport_state = Column(check_object_state_enum, nullable=True, default=None,
                            comment='Наличие и правильность ведения паспорта
светофорного объекта')
    passport_description = Column(String(128), nullable=True, comment='Описание')

    cable_marking_state = Column(check_object_state_enum, nullable=True,
default=None,
                                 comment='Маркировка кабелей')
```

```
    cable_marking_description = Column(String(128), nullable=True,
comment='Описание')

    contact_density_state = Column(check_object_state_enum, nullable=True,
default=None,
                                   comment='Плотность контактов соединительных клемм')
    contact_density_description = Column(String(128), nullable=True,
comment='Описание')

    cable_isolation_state = Column(check_object_state_enum, nullable=True,
default=None,
                                   comment='Наличие повреждений изоляции кабелей')
    cable_isolation_description = Column(String(128), nullable=True,
comment='Описание')

    grounding_cable_state = Column(check_object_state_enum, nullable=True,
default=None,
                                   comment='Состояние заземляющего кабеля')
    grounding_cable_description = Column(String(128), nullable=True,
comment='Описание')

    fastening_reliability_state = Column(check_object_state_enum, nullable=True,
default=None,
                                         comment='Надежность крепления')
    fastening_reliability_description = Column(String(128), nullable=True,
comment='Описание')

    created_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
comment='Время создания')
```

```
    updated_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
onupdate=func.now(),
                        comment='Время последнего обновления')

    junction = relationship("Junctions", back_populates='controllers')  # Связь с
таблицей junction

class PowerBackup(db.Base):
    __tablename__ = "power_backup"
    __table_args__ = {'comment': 'Резервные источники питания (ИБП,
аккумуляторные батареи)'}

    id = Column(String(128), primary_key=True, index=True,
comment='Идентификатор (CUID2)')
    junction_id = Column(String(128), ForeignKey("junctions.id",
ondelete="CASCADE"), index=True, nullable=False,
                         comment='id объекта/осмотра')
    name = Column(String(128), nullable=False, comment='Название осматриваемого
элемента')

    body_damage_state = Column(check_object_state_enum, nullable=True,
default=None,
                               comment='Наличие повреждений (дефектов) корпуса источника
питания')
    body_damage_description = Column(String(128), nullable=True,
comment='Описание')

    battery_outside_state = Column(check_object_state_enum, nullable=True,
default=None,
```

```
                        comment='Внешнее состояние аккумуляторных батарей')
    battery_outside_description = Column(String(128), nullable=True,
comment='Описание')

    electrical_parameters_state = Column(check_object_state_enum, nullable=True,
default=None,
                                         comment='Электрические параметры аккумуляторных
батарей')
    electrical_parameters_description = Column(String(128), nullable=True,
comment='Описание')

    indication_state = Column(check_object_state_enum, nullable=True, default=None,
                              comment='Состояние индикации ИБП')
    indication_description = Column(String(128), nullable=True, comment='Описание')

    functionality_state = Column(check_object_state_enum, nullable=True,
default=None,
                                 comment='Работоспособность ИБП')
    functionality_description = Column(String(128), nullable=True,
comment='Описание')

    created_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
comment='Время создания')
    updated_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
onupdate=func.now(),
                        comment='Время последнего обновления')

    junction = relationship("Junctions", back_populates='power_backup')  # Связь с
таблицей junction
```

```
class Cables(db.Base):
    __tablename__ = "cables"
    __table_args__ = {'comment': 'Кабели различного назначения'}

    id = Column(String(128), primary_key=True, index=True,
comment='Идентификатор (CUID2)')
    junction_id = Column(String(128), ForeignKey("junctions.id",
ondelete="CASCADE"), index=True, nullable=False,
                         comment='id объекта/осмотра')
    name = Column(String(128), nullable=False, comment='Название осматриваемого
элемента')

    tension_bends_state = Column(check_object_state_enum, nullable=True,
default=None,
                                 comment='Наличие натяжений, критичных изгибов')
    tension_bends_description = Column(String(128), nullable=True,
comment='Описание')

    grounding_rail_state = Column(check_object_state_enum, nullable=True,
default=None,
                                  comment='Состояние соединений заземляющей рейки,
контроль заземления')
    grounding_rail_description = Column(String(128), nullable=True,
comment='Описание')

    equipment_connection_state = Column(check_object_state_enum, nullable=True,
default=None,
```

```
                                comment='Состояние подключений к оборудованию')
    equipment_connection_description = Column(String(128), nullable=True,
comment='Описание')

    created_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
comment='Время создания')
    updated_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
onupdate=func.now(),
                        comment='Время последнего обновления')

    junction = relationship("Junctions", back_populates='cables')  # Связь с таблицей
junction



class Wells(db.Base):
    __tablename__ = "wells"
    __table_args__ = {'comment': 'Колодцы кабельной канализации'}

    id = Column(String(128), primary_key=True, index=True,
comment='Идентификатор (CUID2)')
    junction_id = Column(String(128), ForeignKey("junctions.id",
ondelete="CASCADE"), index=True, nullable=False,
                         comment='id объекта/осмотра')
    name = Column(String(128), nullable=False, comment='Название осматриваемого
элемента')

    hatch_state = Column(check_object_state_enum, nullable=True, default=None,
                         comment='Наличие крышки люка')
    hatch_description = Column(String(128), nullable=True, comment='Описание')
```

```
    trash_state = Column(check_object_state_enum, nullable=True, default=None,
                         comment='Наличие мусора и иловых отложений в колодце')
    trash_description = Column(String(128), nullable=True, comment='Описание')

    water_state = Column(check_object_state_enum, nullable=True, default=None,
                         comment='Наличие воды в колодце')
    water_description = Column(String(128), nullable=True, comment='Описание')

    created_at = Column(TIMESTAMP(timezone=True), server_default=func.now(), comment='Время создания')
    updated_at = Column(TIMESTAMP(timezone=True), server_default=func.now(), onupdate=func.now(),
                        comment='Время последнего обновления')

    junction = relationship("Junctions", back_populates='wells')  # Связь с таблицей junction
class TrafficLightMode(enum.Enum):
    """Режимы из документации к ДК циркон 'Протокол SXTP версия 1.6 от 11.12.2023.docx', приложение 3."""

    OFF = "Off"  # Выключен
    WORKING_NORMAL_MODE = "Working_normal_mode"  # Рабочий штатный режим
    CRITICAL_FAIL = "Critical_fail"  # Критическая неисправность (нельзя установить удаленно из АСУДД)
    MANUAL_CONTROL_VPU = "Manual_control_VPU"  # Ручное управление ВПУ (нельзя установить удаленно из АСУДД)
```

```
    TECHNOLOGICAL_MODE = "Technological_mode"  # Технологический режим (ручная настройка режимов через инженерный пульт)
    YELLOW_FLASHING = "Yellow_flashing"  # Желтое мигание (ЖМ)
    RED_FOR_EVERYONE = "Red_for_everyone"  # Всем красный
    REMOTE_CONTROL_ASUDD = "Remote_control_ASUDD"  # Удаленное управление из АСУДД (режим переключения фаз)
    GIVEN_PROGRAM = "Given_program"  # Работа по заданной программе пользователем из АРМ «Мониторинг и управление дорожным хозяйством» (не по расписанию)
    COORDINATED_CONTROL_MODE = "Coordinated_control_mode"  # Режим координированного управления
    LOCAL_ADAPTIVE_MODE = "Local_adaptive_mode"  # Режим локального адаптивного управления

class TrafficLights(db.Base):
    __tablename__ = "traffic_lights"
    __table_args__ = {'comment': 'Светофоры'}

    id = Column(String(128), primary_key=True, index=True, comment='Идентификатор (CUID2)')
    junction_id = Column(String(128), ForeignKey("junctions.id", ondelete="CASCADE"), index=True, nullable=False,
                         comment='id объекта/осмотра')
    name = Column(String(128), nullable=False, comment='Название осматриваемого элемента')

    functionality_state = Column(check_object_state_enum, nullable=True, default=None,
                                 comment='Работоспособность светофора')
```

```
    functionality_description = Column(String(128), nullable=True,
comment='Описание')

    broken_lamps_state = Column(check_object_state_enum, nullable=True,
default=None,
                                comment='Наличие вышедших из строя ламп (у светофоров
ламповых типов)')
    broken_lamps_description = Column(String(128), nullable=True,
comment='Описание')

    broken_leds_state = Column(check_object_state_enum, nullable=True,
default=None,
                                comment='Наличие вышедших из строя светодиодных модулей
(у светофоров светодиодных типов)')
    broken_leds_description = Column(String(128), nullable=True,
comment='Описание')

    dirt_outside_state = Column(check_object_state_enum, nullable=True,
default=None,
                                comment='Наличие загрязнений корпуса')
    dirt_outside_description = Column(String(128), nullable=True,
comment='Описание')

    dirt_optical_state = Column(check_object_state_enum, nullable=True, default=None,
                                comment='Наличие загрязнений оптической части светофора
(линзы, отражателя и прочего)')
    dirt_optical_description = Column(String(128), nullable=True,
comment='Описание')
```

```
    peak_state = Column(check_object_state_enum, nullable=True, default=None,
                        comment='Целостность и центровка козырьков (бленд) светофора')
    peak_description = Column(String(128), nullable=True, comment='Описание')

    design_position_state = Column(check_object_state_enum, nullable=True,
default=None,
                                   comment='Проектное положение светофора')
    design_position_description = Column(String(128), nullable=True,
comment='Описание')

    visibility_100m_state = Column(check_object_state_enum, nullable=True,
default=None,
                                   comment='Видимость сигналов с расстояния не менее 100 м')
    visibility_100m_description = Column(String(128), nullable=True,
comment='Описание')

    # mode_state = Column(Enum(TrafficLightMode), nullable=True, default=None,  #
тут свой отдельный enum
    #                   comment='Установленный режим работы')

    mode_state = Column(String(128), nullable=True, comment='Установленный
режим работы')  # строка вместо enum

    mode_description = Column(String(128), nullable=True, comment='Описание')

    manual_mode_state = Column(check_object_state_enum, nullable=True,
default=None,
                               comment='Режим ручного управления')
```

```
    manual_mode_description = Column(String(128), nullable=True,
comment='Описание')

    fastening_reliability_state = Column(check_object_state_enum, nullable=True,
default=None,
                                         comment='Надежность крепления')
    fastening_reliability_description = Column(String(128), nullable=True,
comment='Описание')

    created_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
comment='Время создания')
    updated_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
onupdate=func.now(),
                        comment='Время последнего обновления')

    junction = relationship("Junctions", back_populates='traffic_lights')  # Связь с
таблицей junction
class TVP(db.Base):
    __tablename__ = "tvp"
    __table_args__ = {'comment': 'ТВП (табло вызывное пешеходное)'}

    id = Column(String(128), primary_key=True, index=True,
comment='Идентификатор (CUID2)')
    junction_id = Column(String(128), ForeignKey("junctions.id",
ondelete="CASCADE"), index=True, nullable=False,
                         comment='id объекта/осмотра')
    name = Column(String(128), nullable=False, comment='Название осматриваемого
элемента')
```

```
    dirt_state = Column(check_object_state_enum, nullable=True, default=None,
                        comment='Наличие загрязнений корпуса')
    dirt_description = Column(String(128), nullable=True, comment='Описание')

    inscriptions_advertisement_state = Column(check_object_state_enum,
nullable=True, default=None,
                                              comment='Наличие несанкционированных надписей и
рекламы')
    inscriptions_advertisement_description = Column(String(128), nullable=True,
comment='Описание')

    broken_state = Column(check_object_state_enum, nullable=True, default=None,
                          comment='Наличие повреждений (дефектов) корпуса, кнопки
ТВП')
    broken_description = Column(String(128), nullable=True, comment='Описание')

    functionality_state = Column(check_object_state_enum, nullable=True,
default=None,
                                 comment='Работоспособность ТВП (совершение контрольного
вызова)')
    functionality_description = Column(String(128), nullable=True,
comment='Описание')

    waiting_state = Column(check_object_state_enum, nullable=True, default=None,
                           comment='Работоспособность индикатора ожидания')
    waiting_description = Column(String(128), nullable=True, comment='Описание')

    fastening_reliability_state = Column(check_object_state_enum, nullable=True,
default=None,
```

```
                                comment='Надежность крепления')
    fastening_reliability_description = Column(String(128), nullable=True,
comment='Описание')


    created_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
comment='Время создания')
    updated_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
onupdate=func.now(),
                        comment='Время последнего обновления')


    junction = relationship("Junctions", back_populates='tvp')  # Связь с таблицей
junction
class AccessibleSignalingDevices(db.Base):
    __tablename__ = "accessible_signaling_devices"
    __table_args__ = {'comment': 'Дополнительное оборудование светофорного
объекта для лиц с нарушением функций зрения '
                                 'и лиц с нарушением функций зрения и слуха'}


    id = Column(String(128), primary_key=True, index=True,
comment='Идентификатор (CUID2)')
    junction_id = Column(String(128), ForeignKey("junctions.id",
ondelete="CASCADE"), index=True, nullable=False,
                         comment='id объекта/осмотра')
    name = Column(String(128), nullable=False, comment='Название осматриваемого
элемента')
    uzsp_state = Column(check_object_state_enum, nullable=True, default=None,
                        comment='Работоспособность УЗСП (для сигналов ориентации и
перехода)')
    uzsp_description = Column(String(128), nullable=True, comment='Описание')
```

```
    orientation_signal_zone_state = Column(check_object_state_enum, nullable=True,
default=None,
                                           comment='Зона слышимости сигнала ориентации')
    orientation_signal_zone_description = Column(String(128), nullable=True,
comment='Описание')

    orientation_signal_duration_state = Column(check_object_state_enum,
nullable=True, default=None,
                                               comment='Продолжительность сигнала ориентации')
    orientation_signal_duration_description = Column(String(128), nullable=True,
comment='Описание')

    crossing_signal_zone_state = Column(check_object_state_enum, nullable=True,
default=None,
                                        comment='Зона слышимости сигнала перехода')
    crossing_signal_zone_description = Column(String(128), nullable=True,
comment='Описание')

    crossing_signal_duration_state = Column(check_object_state_enum, nullable=True,
default=None,
                                            comment='Продолжительность сигнала перехода')
    crossing_signal_duration_description = Column(String(128), nullable=True,
comment='Описание')

    uzsp_mounting_height_state = Column(check_object_state_enum, nullable=True,
default=None,
                                        comment='Высота установки УЗСП')
```

```
    uzsp_mounting_height_description = Column(String(128), nullable=True,
comment='Описание')

    tactile_alarm_state = Column(check_object_state_enum, nullable=True,
default=None,
                                 comment='Работоспособность устройства тактильной
сигнализации')
    tactile_alarm_description = Column(String(128), nullable=True,
comment='Описание')

    tactile_alarm_duration_state = Column(check_object_state_enum, nullable=True,
default=None,
                                      comment='Продолжительность действия тактильного
сигнала')
    tactile_alarm_duration_description = Column(String(128), nullable=True,
comment='Описание')

    tactile_alarm_mounting_height_state = Column(check_object_state_enum,
nullable=True, default=None,
                                         comment='Высота установки устройства тактильной
сигнализации')
    tactile_alarm_mounting_height_description = Column(String(128), nullable=True,
comment='Описание')

    signal_matching_state = Column(check_object_state_enum, nullable=True,
default=None,
                                   comment='Соответствие между звуковым и тактильным
сигналами и сигналами светофора')
```

```
    signal_matching_description = Column(String(128), nullable=True,
comment='Описание')

    fastening_reliability_state = Column(check_object_state_enum, nullable=True,
default=None,
                                         comment='Надежность крепления')
    fastening_reliability_description = Column(String(128), nullable=True,
comment='Описание')

    created_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
comment='Время создания')
    updated_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
onupdate=func.now(),
                        comment='Время последнего обновления')

    junction = (
        relationship("Junctions", back_populates='accessible_signaling_devices'))  # Связь
с таблицей junction



class RoadMarkings(db.Base):
    __tablename__ = "road_markings"
    __table_args__ = {'comment': 'Дорожная разметка'}

    id = Column(String(128), primary_key=True, index=True,
comment='Идентификатор (CUID2)')
    junction_id = Column(String(128), ForeignKey("junctions.id",
ondelete="CASCADE"), index=True, nullable=False,
                         comment='id объекта/осмотра')
```

```
    name = Column(String(128), nullable=False, comment='Название осматриваемого элемента')

    markings_state = Column(check_object_state_enum, nullable=True, default=None,
                            comment='Состояние дорожной разметки в зоне действия датчиков светофорного объекта')
    markings_description = Column(String(128), nullable=True, comment='Описание')

    created_at = Column(TIMESTAMP(timezone=True), server_default=func.now(), comment='Время создания')
    updated_at = Column(TIMESTAMP(timezone=True), server_default=func.now(), onupdate=func.now(),
                        comment='Время последнего обновления')

    junction = relationship("Junctions", back_populates='road_markings')  # Связь с таблицей junction

class AutonomsPowerSources(db.Base):
    __tablename__ = "autonoms_power_sources"
    __table_args__ = {'comment': 'Источники автономного питания'}

    id = Column(String(128), primary_key=True, index=True, comment='Идентификатор (CUID2)')
    junction_id = Column(String(128), ForeignKey("junctions.id", ondelete="CASCADE"), index=True, nullable=False,
                         comment='id объекта/осмотра')
    name = Column(String(128), nullable=False, comment='Название осматриваемого элемента')
```

```
    functionality_state = Column(check_object_state_enum, nullable=True,
default=None,
                                 comment='Работоспособность')
    functionality_description = Column(String(128), nullable=True,
comment='Описание')

    broken_body_state = Column(check_object_state_enum, nullable=True,
default=None,
                               comment='Наличие повреждений корпуса')
    broken_body_description = Column(String(128), nullable=True,
comment='Описание')

    broken_panel_state = Column(check_object_state_enum, nullable=True,
default=None,
                                comment='Наличие повреждений панелей')
    broken_panel_description = Column(String(128), nullable=True,
comment='Описание')

    dirt_state = Column(check_object_state_enum, nullable=True, default=None,
                        comment='Наличие загрязнений')
    dirt_description = Column(String(128), nullable=True, comment='Описание')

    fastening_reliability_state = Column(check_object_state_enum, nullable=True,
default=None,
                                         comment='Надежность крепления')
    fastening_reliability_description = Column(String(128), nullable=True,
comment='Описание')
```

```
    created_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
comment='Время создания')
    updated_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
onupdate=func.now(),
                        comment='Время последнего обновления')

    junction = relationship("Junctions", back_populates='autonoms_power_sources')  #
Связь с таблицей junction

class TOOV(db.Base):
    __tablename__ = "toov"
    __table_args__ = {'comment': 'TOOB табло обратного отсчета времени'}

    id = Column(String(128), primary_key=True, index=True,
comment='Идентификатор (CUID2)')
    junction_id = Column(String(128), ForeignKey("junctions.id",
ondelete="CASCADE"), index=True, nullable=False,
                         comment='id объекта/осмотра')
    name = Column(String(128), nullable=False, comment='Название осматриваемого
элемента')

    functionality_state = Column(check_object_state_enum, nullable=True,
default=None,
                                 comment='Работоспособность')
    functionality_description = Column(String(128), nullable=True,
comment='Описание')

    broken_body_state = Column(check_object_state_enum, nullable=True,
default=None,
```

```
                        comment='Наличие повреждений корпуса')
    broken_body_description = Column(String(128), nullable=True,
comment='Описание')

    dirt_state = Column(check_object_state_enum, nullable=True, default=None,
                        comment='Наличие загрязнений')
    dirt_description = Column(String(128), nullable=True, comment='Описание')

    indicator_state = Column(check_object_state_enum, nullable=True, default=None,
                             comment='Состояние индикации')
    indicator_description = Column(String(128), nullable=True, comment='Описание')

    fastening_reliability_state = Column(check_object_state_enum, nullable=True,
default=None,
                                         comment='Надежность крепления')
    fastening_reliability_description = Column(String(128), nullable=True,
comment='Описание')

    created_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
comment='Время создания')
    updated_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
onupdate=func.now(),
                        comment='Время последнего обновления')

    junction = relationship("Junctions", back_populates='toov')  # Связь с таблицей
junction

class DetectorCamera(db.Base):
    __tablename__ = "detector_camera"
```

```
    __table_args__ = {'comment': 'Датчики, детекторы, видеокамеры'}

    id = Column(String(128), primary_key=True, index=True,
comment='Идентификатор (CUID2)')
    junction_id = Column(String(128), ForeignKey("junctions.id",
ondelete="CASCADE"), index=True, nullable=False,
                         comment='id объекта/осмотра')
    name = Column(String(128), nullable=False, comment='Название осматриваемого
элемента')

    functionality_state = Column(check_object_state_enum, nullable=True,
default=None,
                                 comment='Работоспособность')
    functionality_description = Column(String(128), nullable=True,
comment='Описание')

    broken_body_state = Column(check_object_state_enum, nullable=True,
default=None,
                               comment='Наличие повреждений корпуса')
    broken_body_description = Column(String(128), nullable=True,
comment='Описание')

    dirt_state = Column(check_object_state_enum, nullable=True, default=None,
                        comment='Наличие загрязнений')
    dirt_description = Column(String(128), nullable=True, comment='Описание')

    fastening_reliability_state = Column(check_object_state_enum, nullable=True,
default=None,
                                         comment='Надежность крепления')
```

```
    fastening_reliability_description = Column(String(128), nullable=True,
comment='Описание')

    created_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
comment='Время создания')
    updated_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
onupdate=func.now(),
                        comment='Время последнего обновления')

    junction = relationship("Junctions", back_populates='detector_camera')  # Связь с
таблицей junction

class TSODD(db.Base):
    __tablename__ = "tsodd"
    __table_args__ = {'comment': 'ТСОДД в составе СО (технические средства
организации дорожного движения)'}

    id = Column(String(128), primary_key=True, index=True,
comment='Идентификатор (CUID2)')
    junction_id = Column(String(128), ForeignKey("junctions.id",
ondelete="CASCADE"), index=True, nullable=False,
                         comment='id объекта/осмотра')
    name = Column(String(128), nullable=False, comment='Название осматриваемого
элемента')

    inscriptions_advertisement_state = Column(check_object_state_enum,
nullable=True, default=None,
                                              comment='Наличие несанкционированных надписей и
рекламы')
```

```
    inscriptions_advertisement_description = Column(String(128), nullable=True,
comment='Описание')

    dirt_state = Column(check_object_state_enum, nullable=True, default=None,
                        comment='Наличие внешних загрязнений')
    dirt_description = Column(String(128), nullable=True, comment='Описание')

    reflective_film_state = Column(check_object_state_enum, nullable=True,
default=None,
                                   comment='Наличие повреждений световозвращающей
пленки')
    reflective_film_description = Column(String(128), nullable=True,
comment='Описание')

    position_state = Column(check_object_state_enum, nullable=True, default=None,
                            comment='Контроль положения')
    position_description = Column(String(128), nullable=True, comment='Описание')

    fastening_reliability_state = Column(check_object_state_enum, nullable=True,
default=None,
                                         comment='Надежность крепления')
    fastening_reliability_description = Column(String(128), nullable=True,
comment='Описание')

    created_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
comment='Время создания')
    updated_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
onupdate=func.now(),
                        comment='Время последнего обновления')
```

```
    junction = relationship("Junctions", back_populates='tsodd')  # Связь с таблицей
junction


class LightSections(db.Base):
    __tablename__ = "light_sections"
    __table_args__ = {'comment': 'Информационные световые секции'}

    id = Column(String(128), primary_key=True, index=True,
comment='Идентификатор (CUID2)')
    junction_id = Column(String(128), ForeignKey("junctions.id",
ondelete="CASCADE"), index=True, nullable=False,
                         comment='id объекта/осмотра')
    name = Column(String(128), nullable=False, comment='Название осматриваемого
элемента')

    functionality_state = Column(check_object_state_enum, nullable=True,
default=None,
                                 comment='Работоспособность')
    functionality_description = Column(String(128), nullable=True,
comment='Описание')

    mode_state = Column(check_object_state_enum, nullable=True, default=None,
                        comment='Установленный режим работы')
    mode_description = Column(String(128), nullable=True, comment='Описание')

    indicator_state = Column(check_object_state_enum, nullable=True, default=None,
                             comment='Состояние индикации')
```

```
    indicator_description = Column(String(128), nullable=True, comment='Описание')

    inscriptions_advertisement_state = Column(check_object_state_enum,
nullable=True, default=None,
                                              comment='Наличие несанкционированных надписей и
рекламы')
    inscriptions_advertisement_description = Column(String(128), nullable=True,
comment='Описание')

    dirt_state = Column(check_object_state_enum, nullable=True, default=None,
                        comment='Наличие внешних загрязнений')
    dirt_description = Column(String(128), nullable=True, comment='Описание')

    design_position_state = Column(check_object_state_enum, nullable=True,
default=None,
                                   comment='Проектное положение')
    design_position_description = Column(String(128), nullable=True,
comment='Описание')

    fastening_reliability_state = Column(check_object_state_enum, nullable=True,
default=None,
                                         comment='Надежность крепления')
    fastening_reliability_description = Column(String(128), nullable=True,
comment='Описание')

    created_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
comment='Время создания')
    updated_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
onupdate=func.now(),
```

```
                comment='Время последнего обновления')

    junction = relationship("Junctions", back_populates='light_sections')  # Связь с таблицей junction


# Enum для категорий дополнительной информации
class AdditionalInfoCategory(enum.Enum):
    """Категории являются названием таблиц с сущностями осмотра"""

    SUPPORTS = "supports"
    CONTROLLERS = "controllers"
    POWER_BACKUP = "power_backup"
    CABLES = "cables"
    WELLS = "wells"
    TRAFFIC_LIGHT = "traffic_lights"
    TVP = "tvp"
    ACCESSIBLE_SIGNALING_DEVICES = "accessible_signaling_devices"
    ROAD_MARKINGS = "road_markings"
    AUTONOMS_POWER_SOURCES = "autonoms_power_sources"
    TOOV = "toov"
    DETECTOR_CAMERA = "detector_camera"
    TSODD = "tsodd"
    LIGHT_SECTIONS = "light_sections"

class AdditionalInfo(db.Base):
    __tablename__ = "additional_info"
    __table_args__ = {'comment': 'Таблица для сохранения дополнительных пунктов осмотра, добавленных пользователем'}
```

```
    id = Column(String(128), primary_key=True, index=True,
comment='Идентификатор (CUID2)')
    junction_id = Column(String(128), ForeignKey("junctions.id",
ondelete="CASCADE"), index=True, nullable=False,
                         comment='id объекта/осмотра')
    name = Column(String(128), nullable=False, comment='Название осматриваемого
элемента')

    category = Column(Enum(AdditionalInfoCategory), nullable=True, default=None,
                      comment='Категория для которой добавляется запись из
additional_info')
    element_id = Column(String(128), nullable=False, comment='id элемента осмотра
для записи из таблиц с элементами')

    field_name = Column(String(128), nullable=False,
                        comment='Название поля введенного пользователем (для
отображения в интерфейсе)')
    field_state = Column(check_object_state_enum, nullable=True, default=None,
                         comment='Состояние элемента')
    field_description = Column(String(128), nullable=True, comment='Описание')

    created_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
comment='Время создания')
    updated_at = Column(TIMESTAMP(timezone=True), server_default=func.now(),
onupdate=func.now(),
                        comment='Время последнего обновления')
    junction = relationship("Junctions", back_populates='additional_info')  # Связь с
таблицей junction
```

5. Скрипт предназначен для формирования и экспорта данных в формате XLSX

Скрипт export.py, являющийся неотъемлемой частью текста программы, предназначен для формирования и экспорта данных в формате XLSX. Реализует функциональность создания контрольных листов осмотра с форматированием согласно требованиям ГОСТ Р 59103-2020.

Входные данные: данные осмотров из базы данных.

Выходные данные: сформированные файлы XLSX с данными осмотров.

Символическая запись на исходном языке:

```
import re
from copy import deepcopy
import openpyxl
from io import BytesIO
from openpyxl.cell.text import InlineFont
from openpyxl.styles import Alignment, Border, Side, Font, PatternFill
from openpyxl.cell.rich_text import CellRichText, TextBlock
from openpyxl.utils import get_column_letter

import api.db_requests as db_requests
import connectors.db_models as db_models

# сопоставление названий и моделей базы данных
db_model_mapper: dict = {
    'Конструкции для размещения элементов светофорного объекта':
db_models.Supports,
    'Дорожные контроллеры и ВПУ': db_models.Controllers,
```

```
    'Резервные источники питания (ИБП, аккумуляторные батареи), при наличии': db_models.PowerBackup,
    'Кабели различного назначения': db_models.Cables,
    'Колодцы кабельной канализации': db_models.Wells,
    'Светофоры': db_models.TrafficLights,
    'ТВП': db_models.TVP,
    'Дополнительное оборудование светофорного объекта для лиц с нарушением функций зрения '
    'и лиц с нарушением функций зрения и слуха': db_models.AccessibleSignalingDevices,
    'Дорожная разметка': db_models.RoadMarkings,
    'Источники автономного питания': db_models.AutonomsPowerSources,
    'ТООВ': db_models.TOOV,
    'Датчики, детекторы, видеокамеры': db_models.DetectorCamera,
    'ТСОДД в составе СО': db_models.TSODD,
    'Информационные световые секции': db_models.LightSections,
}

# сопоставление названий категорий и функций получения данных для категорий
db_requests_mapper: dict = {
    'Конструкции для размещения элементов светофорного объекта': db_requests.get_supports_for_junction,
    'Дорожные контроллеры и ВПУ': db_requests.get_controllers_for_junction,
    'Резервные источники питания (ИБП, аккумуляторные батареи), при наличии': db_requests.get_power_backups_for_junction,
    'Кабели различного назначения': db_requests.get_cables_for_junction,
    'Колодцы кабельной канализации': db_requests.get_wells_for_junction,
    'Светофоры': db_requests.get_traffic_lights_for_junction,
    'ТВП': db_requests.get_tvp_for_junction,
```

```
    'Дополнительное оборудование светофорного объекта для лиц с нарушением функций зрения '
    'и лиц с нарушением функций зрения и слуха':
db_requests.get_accessible_devices_for_junction,
    'Дорожная разметка': db_requests.get_road_markings_for_junction,
    'Источники автономного питания': db_requests.get_power_sources_for_junction,
    'ТООВ': db_requests.get_toov_for_junction,
    'Датчики, детекторы, видеокамеры':
db_requests.get_detectors_cameras_for_junction,
    'ТСОДД в составе СО': db_requests.get_tsodd_for_junction,
    'Информационные световые секции': db_requests.get_light_sections_for_junction,
}

# unused_fields: list = ['id', 'junction_id', 'created_at', 'updated_at', 'dk_name',
'dk_serial']
unused_fields: list = ['junction_id', 'created_at', 'updated_at', 'dk_name', 'dk_serial']



async def get_wrapped_text_lines(ws, cell):
    """Определение количества строк текста в ячейке"""

    column_letter = cell.column_letter
    column_width = ws.column_dimensions[column_letter].width or 25  # если ширина не задана, используем 25

    text = cell.value or ''

    if not isinstance(text, str):
        return 1  # если не строка, считаем за одну строку
```

```
    avg_char_width = 1.0  # Примерная ширина символа

    max_chars_per_line = int(column_width / avg_char_width)

    text_row = len(text) / max_chars_per_line
    text_row = int(text_row) + 1 if text_row > 1 else int(text_row)
    return text_row


async def get_field_description(category_name: str, field_name: str) -> str:
    """Получение названия полей из описаний моделей sqlalchemy"""

    global db_model_mapper

    model = db_model_mapper.get(category_name)
    rus_name = getattr(model.__table__.c[field_name], "comment", None)
    return rus_name


async def get_category_empty_data(category_name: str) -> list:
    """
    Получение данных для пустой категории.
    """

    global db_model_mapper

    model = db_model_mapper.get(category_name)
    if not model:
```

```
        return list()

    return [{column.name: None for column in model.__table__.columns}]



async def remove_null_keys(category_data):
    """Удаление пустых элементов из данных о категориях"""

    data_copy = deepcopy(category_data)

    # собираем все ключи, которые есть в категориях
    all_keys = set()
    for item in data_copy:
        all_keys.update(item.keys())

    # сбор ключей, у которых во всех словарях значение None
    keys_to_remove = {key for key in all_keys if all(item.get(key) is None for item in
data_copy)}

    # удаление ключей, у которых во всех словарях значение None
    for item in data_copy:
        for key in keys_to_remove:
            del item[key]

    return data_copy



async def category_work(ws, junction_id, short):
```

```
    """Добавление данных о категориях осмотра в создаваемый контрольный
лист"""

    global db_requests_mapper

    cell_fill = PatternFill(start_color="FFFF00", end_color="FFFF00", fill_type="solid")

    cnt_category = 1  # счетчик добавленных категорий

    for category_name, func in db_requests_mapper.items():

        items_id_dict = dict()

        cnt_elem = 1  # счетчик добавленных элементов осмотра (прим.
Вертикальность опоры, Состояние фундамента...)

        category_data = await func(junction_id)
        category_data = sorted(category_data, key=lambda x: x["name"])

        if len(category_data) == 0 and short is False:
            # если для осмотра нет данных в категории, подставляем пустые значения
            category_data = await get_category_empty_data(category_name)
        elif len(category_data) == 0 and short is True:
            # не добавляем пустые категории если выбран short файл
            continue

        all_keys = set()
        for item in category_data:
            # убираем лишние данные из категории
```

```
            for item_key in list(item.keys()):
                if item_key in unused_fields:
                    del item[item_key]

            all_keys.update(item.keys())

        # # заголовки
        filled_rows = ws.max_row
        # название категории, имя элемента
        ws[f'A{filled_rows + 1}'] = f'{cnt_category} {category_name}'  # прим. 2 Дорожные контроллеры и ВПУ
        ws[f'B{filled_rows + 1}'] = item.get('name')
        ws.merge_cells(f'A{filled_rows + 1}:A{filled_rows + 2}')
        ws[f'A{filled_rows + 1}'].alignment = Alignment(horizontal='left', vertical='center', wrap_text=True)
        ws[f'A{filled_rows + 1}'].fill = cell_fill

        # заголовки
        row = filled_rows + 1
        col = 2
        for cnt, item in enumerate(category_data):
            item_name = item['name'] if item['name'] else ''
            item_name = f'{cnt + 1}) Эл-т № {item_name}'  # Прим. '1) Эл-т № ОП1'

            item_id = item['id']

            name_cell = ws.cell(row=row, column=col, value=item_name)
            item_id = item['id']
```

```
            items_id_dict[item_id] = [name_cell.row, name_cell.column]  # id элемента и
его координаты

            ws.merge_cells(f'{get_column_letter(col)}{row}:{get_column_letter(col +
1)}{row}')
            ws.cell(row=row + 1, column=col, value='Оценка состояния')
            ws[f'{get_column_letter(col)}{row + 1}'].alignment =
Alignment(horizontal='center', vertical='center',
                                                    wrap_text=True)
            # ws[f'{get_column_letter(col)}{row +
1}'].row_dimensions[filled_rows+2].height = 40  # высота строки
            ws.row_dimensions[row + 1].height = 40  # высота строки
            ws.cell(row=row + 1, column=col + 1, value='Примечание')
            ws[f'{get_column_letter(col + 1)}{row + 1}'].alignment =
Alignment(horizontal='center', vertical='center',
                                                      wrap_text=True)

            col += 2

        start_row = ws.max_row + 1
        col = 1

        if short:
            # удаление не заполненных пунктов осмотра если выбран short файл
            category_data = await remove_null_keys(category_data)

        for item_ctn, item in enumerate(category_data):
            row = start_row
```

```
        for field_name, field_data in item.items():

            if state_field := str(field_name).endswith('_state'):
                base_field_name = re.sub(r'_state$', '', field_name)
                description = base_field_name + '_description'

                if col == 1:
                    field_rus_name = await get_field_description(category_name,
field_name)
                    # заполнение названий пунктов проверки
                    field_rus_name = f'{cnt_category}.{cnt_elem} {field_rus_name}'  #
прим. 1.7 Состояние фундамента
                    ws.cell(row=row, column=col, value=field_rus_name)
                    ws[f'A{row}'].alignment = Alignment(horizontal='left', vertical='center',
wrap_text=True)

                    count_row = await get_wrapped_text_lines(ws, ws[f'A{row}'])  #
количество строк в ячейке

                    ws.row_dimensions[
                        row].height = 28 * count_row  # установка высоты ячейки в
зависимости от количества строк в ней
                    cnt_elem += 1  # увеличиваем счетчик элементов осмотра после

                # замена состояний объекта
                if field_data is not None:
                    if category_name == 'Светофоры' and field_name == 'mode_state':
                        # для режима работы светофоров значения не подставляем, берем
напрямую из строки
```

```
                pass
            elif field_data.value == 'Good':
                field_data = 'У'
            elif field_data.value == 'Bad':
                field_data = 'Н'
        else:
            field_data = '-'

        # состояние элемента
        ws.cell(row=row, column=col + 1, value=str(field_data))
        ws[f'{get_column_letter(col + 1)}{row}'].alignment =
Alignment(horizontal='center',
                                                vertical='center',
                                                wrap_text=True)

        # описание элемента
        item_description = item.get(description)
        if not item_description:
            item_description = ''

        ws.cell(row=row, column=col + 2, value=item_description)
        ws[f'{get_column_letter(col + 2)}{row}'].alignment = (
            Alignment(
                horizontal='center', vertical='center',
                wrap_text=True))
        # col += 1
        row += 1

    # дополнительные поля
```

```
            if field_name == 'additional_info':
                row_additional_data = ws.max_row + 1

                for additional_item in field_data:

                    field_for_element = additional_item.get('element_id')
                    element_coordinates = items_id_dict.get(field_for_element)

                    additional_field_name = additional_item.get('field_name')

                    additional_field_name = f'{cnt_category}.{cnt_elem}
{additional_field_name}'  # прим. 1.7 Состояние фундамента
                    cnt_elem += 1  # увеличиваем счетчик элементов осмотра после
                    ws.cell(row=row_additional_data, column=1,
value=additional_field_name)
                    ws[f'A{row_additional_data}'].alignment = Alignment(horizontal='left',
vertical='center',
                                                        wrap_text=True)
                    count_row = await get_wrapped_text_lines(ws, ws[
                        f'A{row_additional_data}'])  # количество строк в ячейке
                    ws.row_dimensions[
                        row_additional_data].height = 28 * count_row  # установка высоты
ячейки в зависимости от количества строк в ней

                    additional_state = additional_item.get('field_state')

                    if additional_state is not None:
                        if additional_state.value == 'Good':
                            field_state = 'У'
```

```
                elif additional_state.value == 'Bad':
                    field_state = 'H'
                else:
                    field_state = '-'

                # Проверяем, есть ли уже записи для данного элемента
                while ws.cell(row=row_additional_data,
column=element_coordinates[1]).value:
                    row_additional_data += 1  # Ищем свободную строку

                # состояние дополнительного элемента
                ws.cell(row=row_additional_data, column=element_coordinates[1],
value=field_state)

ws[f'{get_column_letter(element_coordinates[1])}{row_additional_data}'].alignment =
Alignment(
                    horizontal='center', vertical='center', wrap_text=True)
                count_row = await get_wrapped_text_lines(ws, ws[
                    f'A{row_additional_data}'])  # количество строк в ячейке
                ws.row_dimensions[
                    row_additional_data].height = 28 * count_row  # установка высоты
ячейки в зависимости от количества строк в ней

                # описание дополнительного элемента
                additional_item_description = additional_item.get('field_description')
                if not additional_item_description:
                    additional_item_description = ''
                ws.cell(row=row_additional_data, column=element_coordinates[1] + 1,
                        value=additional_item_description)
```

```
                    ws[f'{get_column_letter(element_coordinates[1] +
1)}{row_additional_data}'].alignment = (
                        Alignment(
                            horizontal='center', vertical='center',
                            wrap_text=True))

                    row_additional_data += 1
                    row += 1
                row += 1
            col += 2

        cnt_category += 1  # увеличиваем счетчик категорий после данных для
категории



async def export_data(junction_id: str, short: bool):
    """Формирования файла для экспорта"""

    # получение осмотра

    junction = await db_requests.get_junction(junction_id)

    junction_kind_id = junction['kind_id']
    kind = await db_requests.get_junction_kind(junction_kind_id)
    kind_name = kind.get('kind')
    junction['kind'] = kind_name
    del junction['kind_id']

    wb = openpyxl.Workbook()
```

```
ws = wb.active
ws.title = "КЛ ГОСТ"

ws.column_dimensions['A'].width = 30  # ширина столбца
ws.column_dimensions['B'].width = 10  # ширина столбца
ws.column_dimensions['C'].width = 12  # ширина столбца
ws.column_dimensions['D'].width = 10  # ширина столбца
ws.column_dimensions['E'].width = 12  # ширина столбца
ws.column_dimensions['F'].width = 10  # ширина столбца
ws.column_dimensions['G'].width = 12  # ширина столбца

ws.merge_cells('A1:G1')

# нижнее обрамление
border_bottom_style = Border(bottom=Side(style='thin', color='000000'))

# head
# Размер шрифта
head_font_style = Font(name='Times New Roman', size=12)

# Установка размера шрифта к диапазону ячеек
for row in ws['A1:G7']:
    for cell in row:
        cell.font = head_font_style

# Устанавливаем текст в объединённую ячейку
ws['A1'] = 'Контрольный лист осмотра светофорного объекта'
# Выставляем выравнивание по центру (по горизонтали и вертикали)
ws['A1'].alignment = Alignment(horizontal='center', vertical='center')
```

```
    ws['A3'] = 'Номер документа:'
    ws.merge_cells('B3:G3')
    ws['B3'] = junction.get('number')
    ws['B3'].border = border_bottom_style

    ws['A5'] = 'Адрес объекта:'
    ws.merge_cells('B5:G5')
    ws['B5'] = junction.get('address')
    ws['B5'].border = border_bottom_style

    ws['A7'] = 'Дата/время осмотра:'
    ws.merge_cells('B7:G7')
    inspection_at = junction.get('inspection_at')
    formatted_date = inspection_at.strftime("%d.%m.%Y %H.%M")
    ws['B7'] = formatted_date
    ws['B7'].border = border_bottom_style

    ws['A9'] = 'Наименование группы \n элементов и пункт проверки'
    ws['A9'].alignment = Alignment(horizontal='center', vertical='center',
wrap_text=True)
    ws.merge_cells('B9:G9')
    ws['B9'].alignment = Alignment(horizontal='center', vertical='center',
wrap_text=True)
    ws['B9'] = 'Номер (идентификатор) элемента в составе светофорного объекта и
результат проверки'
    ws.row_dimensions[9].height = 30  # высота строки
```

```
    await category_work(ws, junction_id, short)  # добавление данных о категориях осмотра

    # установка стиля шрифта и обрамления
    document_font_style = Font(name='Times New Roman', size=11)
    border_cell_style = Border(
        top=Side(style='thin', color='000000'),  # Верхняя граница
        bottom=Side(style='thin', color='000000'),  # Нижняя граница
        left=Side(style='thin', color='000000'),  # Левая граница
        right=Side(style='thin', color='000000')  # Правая граница
    )
    for row in ws.iter_rows(min_row=9):
        for cell in row:
            cell.font = document_font_style
            cell.border = border_cell_style

    filled_categories = True
    if filled_categories:
        legend_row = ws.max_row + 1

        rich_text = CellRichText(
            TextBlock(text="Примечание - ", font=InlineFont(b=True)),  # Жирный фрагмент
            TextBlock(
                text=' В графе "Оценка состояния" проставляют оценку "удовлетворительно" ("У") при отсутствии замечаний, "неудовлетворительно" ("Н") - при наличии замечаний.',
                font=InlineFont(b=False))  # Обычный фрагмент
        )
```

```
        ws[f'A{legend_row}'].value = rich_text
        ws[f'A{legend_row}'].alignment = Alignment(horizontal='left', vertical='center',
wrap_text=True)
        ws[f'A{legend_row}'].font = document_font_style

        ws.merge_cells(f'A{legend_row}:G{legend_row}')
        ws.row_dimensions[legend_row].height = 30  # высота строки

    # bottom
    # Получаем количество заполненных строк
    filled_rows = ws.max_row + 2  # последняя заполненная строка + отступ

    # ws.merge_cells(f'A{filled_rows}:G{filled_rows}')
    ws[f'A{filled_rows}'] = 'Контрольный лист осмотра выполнен в соответствии с
формой Приложения А ГОСТ Р 59103-2020.'

    ws[f'A{filled_rows + 2}'] = 'От Заказчика:'

    column_for_signed = ['A', 'C', 'D', 'F', 'G']
    for column in column_for_signed:
        ws[f'{column}{filled_rows + 4}'].border = border_bottom_style

    description_font_style = Font(name='Times New Roman', size=8)

    ws[f'A{filled_rows + 5}'] = 'Должность'
    ws[f'A{filled_rows + 5}'].alignment = Alignment(horizontal='center',
vertical='center')
```

```
    ws[f'C{filled_rows + 5}'] = 'Подпись'
    ws.merge_cells(f'C{filled_rows + 5}:D{filled_rows + 5}')
    ws[f'C{filled_rows + 5}'].alignment = Alignment(horizontal='center',
vertical='center')

    ws[f'F{filled_rows + 5}'] = 'ФИО'
    ws.merge_cells(f'F{filled_rows + 5}:G{filled_rows + 5}')
    ws[f'F{filled_rows + 5}'].alignment = Alignment(horizontal='center',
vertical='center')

    ws[f'A{filled_rows + 7}'] = 'От Исполнителя:'

    for column in column_for_signed:
        ws[f'{column}{filled_rows + 8}'].border = border_bottom_style

    ws[f'A{filled_rows + 8}'] = junction.get('executor_position')
    ws[f'A{filled_rows + 8}'].alignment = Alignment(horizontal='center',
vertical='center')

    ws[f'F{filled_rows + 8}'] = junction.get('executor_name')
    ws.merge_cells(f'F{filled_rows + 8}:G{filled_rows + 8}')
    ws[f'F{filled_rows + 8}'].alignment = Alignment(horizontal='center',
vertical='center')

    ws[f'A{filled_rows + 9}'] = 'Должность'
    ws[f'A{filled_rows + 9}'].alignment = Alignment(horizontal='center',
vertical='center')

    ws.merge_cells(f'C{filled_rows + 9}:D{filled_rows + 9}')
```

```
    ws[f'C{filled_rows + 9}'] = 'Подпись'
    ws[f'C{filled_rows + 9}'].alignment = Alignment(horizontal='center',
vertical='center')

    ws.merge_cells(f'F{filled_rows + 9}:G{filled_rows + 9}')
    ws[f'F{filled_rows + 9}'] = 'ФИО'
    ws[f'F{filled_rows + 9}'].alignment = Alignment(horizontal='center',
vertical='center')

    # установка шрифта
    for row in ws.iter_rows(min_row=filled_rows):
        for cell in row:
            cell.font = document_font_style
    # уменьшение шрифта для пунктов "подпись", "должность", "ФИО"
    ws[f'A{filled_rows + 5}'].font = description_font_style
    ws[f'C{filled_rows + 5}'].font = description_font_style
    ws[f'F{filled_rows + 5}'].font = description_font_style
    ws[f'A{filled_rows + 9}'].font = description_font_style
    ws[f'C{filled_rows + 9}'].font = description_font_style
    ws[f'F{filled_rows + 9}'].font = description_font_style
    output = BytesIO()
    wb.save(output)
    output.seek(0)
    return output
```

6. Скрипт для обеспечения взаимодействия с базой данных

Скрипт db_requests.py, являющийся неотъемлемой частью текста программы, предназначен для обеспечения взаимодействия с базой данных.

Содержит функции для выполнения CRUD-операций (создание, чтение, обновление, удаление) для различных сущностей системы.

Входные данные: параметры запросов к базе данных, идентификаторы записей, данные для создания и обновления записей.

Выходные данные: результаты запросов к базе данных, объекты данных, статусы выполнения операций.

Символическая запись на исходном языке:

```
from copy import deepcopy
import openpyxl
from io import BytesIO
from openpyxl.cell.text import InlineFont
from openpyxl.styles import Alignment, Border, Side, Font, PatternFill
from openpyxl.cell.rich_text import CellRichText, TextBlock
from openpyxl.utils import get_column_letter
import api.db_requests as db_requests
import connectors.db_models as db_models
# сопоставление названий и моделей базы данных
db_model_mapper: dict = {
```

```
    'Конструкции для размещения элементов светофорного объекта':
db_models.Supports,
    'Дорожные контроллеры и ВПУ': db_models.Controllers,
    'Резервные источники питания (ИБП, аккумуляторные батареи), при
наличии': db_models.PowerBackup,
    'Кабели различного назначения': db_models.Cables,
    'Колодцы кабельной канализации': db_models.Wells,
    'Светофоры': db_models.TrafficLights,
    'ТВП': db_models.TVP,
    'Дополнительное оборудование светофорного объекта для лиц с
нарушением функций зрения '
    'и лиц с нарушением функций зрения и слуха':
db_models.AccessibleSignalingDevices,
    'Дорожная разметка': db_models.RoadMarkings,
    'Источники автономного питания': db_models.AutonomsPowerSources,
    'ТООВ': db_models.TOOV,
    'Датчики, детекторы, видеокамеры': db_models.DetectorCamera,
    'ТСОДД в составе СО': db_models.TSODD,
    'Информационные световые секции': db_models.LightSections,
}

# сопоставление названий категорий и функций получения данных для
категорий
db_requests_mapper: dict = {
    'Конструкции для размещения элементов светофорного объекта':
db_requests.get_supports_for_junction,
    'Дорожные контроллеры и ВПУ': db_requests.get_controllers_for_junction,
    'Резервные источники питания (ИБП, аккумуляторные батареи), при
наличии': db_requests.get_power_backups_for_junction,
```

```
    'Кабели различного назначения': db_requests.get_cables_for_junction,
    'Колодцы кабельной канализации': db_requests.get_wells_for_junction,
    'Светофоры': db_requests.get_traffic_lights_for_junction,
    'ТВП': db_requests.get_tvp_for_junction,
    'Дополнительное оборудование светофорного объекта для лиц с
нарушением функций зрения '
    'и лиц с нарушением функций зрения и слуха':
db_requests.get_accessible_devices_for_junction,
    'Дорожная разметка': db_requests.get_road_markings_for_junction,
    'Источники автономного питания':
db_requests.get_power_sources_for_junction,
    'ТООВ': db_requests.get_toov_for_junction,
    'Датчики, детекторы, видеокамеры':
db_requests.get_detectors_cameras_for_junction,
    'ТСОДД в составе СО': db_requests.get_tsodd_for_junction,
    'Информационные световые секции':
db_requests.get_light_sections_for_junction,
}

# unused_fields: list = ['id', 'junction_id', 'created_at', 'updated_at', 'dk_name',
'dk_serial']
unused_fields: list = ['junction_id', 'created_at', 'updated_at', 'dk_name',
'dk_serial']



async def get_wrapped_text_lines(ws, cell):
    """Определение количества строк текста в ячейке"""

    column_letter = cell.column_letter
```

```
    column_width = ws.column_dimensions[column_letter].width or 25  # если ширина не задана, используем 25

    text = cell.value or ''

    if not isinstance(text, str):
        return 1  # если не строка, считаем за одну строку

    avg_char_width = 1.0  # Примерная ширина символа

    max_chars_per_line = int(column_width / avg_char_width)

    text_row = len(text) / max_chars_per_line
    text_row = int(text_row) + 1 if text_row > 1 else int(text_row)
    return text_row


async def get_field_description(category_name: str, field_name: str) -> str:
    """Получение названия полей из описаний моделей sqlalchemy"""

    global db_model_mapper

    model = db_model_mapper.get(category_name)
    rus_name = getattr(model.__table__.c[field_name], "comment", None)
    return rus_name


async def get_category_empty_data(category_name: str) -> list:
    """
```

```
    Получение данных для пустой категории.
    """

    global db_model_mapper

    model = db_model_mapper.get(category_name)
    if not model:
        return list()

    return [{column.name: None for column in model.__table__.columns}]



async def remove_null_keys(category_data):
    """Удаление пустых элементов из данных о категориях"""

    data_copy = deepcopy(category_data)

    # собираем все ключи, которые есть в категориях
    all_keys = set()
    for item in data_copy:
        all_keys.update(item.keys())

    # сбор ключей, у которых во всех словарях значение None
    keys_to_remove = {key for key in all_keys if all(item.get(key) is None for item
in data_copy)}

    # удаление ключей, у которых во всех словарях значение None
    for item in data_copy:
        for key in keys_to_remove:
```

```
            del item[key]

    return data_copy


async def category_work(ws, junction_id, short):
    """Добавление данных о категориях осмотра в создаваемый контрольный лист"""

    global db_requests_mapper

    cell_fill = PatternFill(start_color="FFFF00", end_color="FFFF00", fill_type="solid")

    cnt_category = 1  # счетчик добавленных категорий

    for category_name, func in db_requests_mapper.items():

        items_id_dict = dict()

        cnt_elem = 1  # счетчик добавленных элементов осмотра (прим. Вертикальность опоры, Состояние фундамента...)

        category_data = await func(junction_id)
        category_data = sorted(category_data, key=lambda x: x["name"])

        if len(category_data) == 0 and short is False:
            # если для осмотра нет данных в категории, подставляем пустые значения
```

```
            category_data = await get_category_empty_data(category_name)
        elif len(category_data) == 0 and short is True:
            # не добавляем пустые категории если выбран short файл
            continue

        all_keys = set()
        for item in category_data:
            # убираем лишние данные из категории
            for item_key in list(item.keys()):
                if item_key in unused_fields:
                    del item[item_key]

            all_keys.update(item.keys())

        # # заголовки
        filled_rows = ws.max_row
        # название категории, имя элемента
        ws[f'A{filled_rows + 1}'] = f'{cnt_category} {category_name}'  # прим. 2 Дорожные контроллеры и ВПУ
        ws[f'B{filled_rows + 1}'] = item.get('name')
        ws.merge_cells(f'A{filled_rows + 1}:A{filled_rows + 2}')
        ws[f'A{filled_rows + 1}'].alignment = Alignment(horizontal='left', vertical='center', wrap_text=True)
        ws[f'A{filled_rows + 1}'].fill = cell_fill

        # заголовки
        row = filled_rows + 1
        col = 2
        for cnt, item in enumerate(category_data):
```

```
            item_name = item['name'] if item['name'] else ''
            item_name = f'{cnt + 1}) Эл-т № {item_name}'  # Прим. '1) Эл-т №
ОП1'

            item_id = item['id']

            name_cell = ws.cell(row=row, column=col, value=item_name)
            item_id = item['id']
            items_id_dict[item_id] = [name_cell.row, name_cell.column]  # id
элемента и его координаты

            ws.merge_cells(f'{get_column_letter(col)}{row}:{get_column_letter(col
+ 1)}{row}')
            ws.cell(row=row + 1, column=col, value='Оценка состояния')
            ws[f'{get_column_letter(col)}{row + 1}'].alignment =
Alignment(horizontal='center', vertical='center',
                                                    wrap_text=True)
            # ws[f'{get_column_letter(col)}{row +
1}'].row_dimensions[filled_rows+2].height = 40  # высота строки
            ws.row_dimensions[row + 1].height = 40  # высота строки
            ws.cell(row=row + 1, column=col + 1, value='Примечание')
            ws[f'{get_column_letter(col + 1)}{row + 1}'].alignment =
Alignment(horizontal='center', vertical='center',
                                                  wrap_text=True)

            col += 2

        start_row = ws.max_row + 1
        col = 1
```

```
        if short:
            # удаление не заполненных пунктов осмотра если выбран short файл
            category_data = await remove_null_keys(category_data)

        for item_ctn, item in enumerate(category_data):
            row = start_row

            for field_name, field_data in item.items():

                if state_field := str(field_name).endswith('_state'):
                    base_field_name = re.sub(r'_state$', '', field_name)
                    description = base_field_name + '_description'

                    if col == 1:
                        field_rus_name = await get_field_description(category_name,
field_name)
                        # заполнение названий пунктов проверки
                        field_rus_name = f'{cnt_category}.{cnt_elem} {field_rus_name}'
# прим. 1.7 Состояние фундамента
                        ws.cell(row=row, column=col, value=field_rus_name)
                        ws[f'A{row}'].alignment = Alignment(horizontal='left',
vertical='center', wrap_text=True)

                        count_row = await get_wrapped_text_lines(ws, ws[f'A{row}'])  #
количество строк в ячейке

                        ws.row_dimensions[
```

```
                row].height = 28 * count_row  # установка высоты ячейки в
зависимости от количества строк в ней
                cnt_elem += 1  # увеличиваем счетчик элементов осмотра
после

            # замена состояний объекта
            if field_data is not None:
                if category_name == 'Светофоры' and field_name ==
'mode_state':
                    # для режима работы светофоров значения не подставляем,
берем напрямую из строки
                    pass
                elif field_data.value == 'Good':
                    field_data = 'У'
                elif field_data.value == 'Bad':
                    field_data = 'Н'
            else:
                field_data = '-'

            # состояние элемента
            ws.cell(row=row, column=col + 1, value=str(field_data))
            ws[f'{get_column_letter(col + 1)}{row}'].alignment =
Alignment(horizontal='center',
                                                    vertical='center',
                                                    wrap_text=True)

            # описание элемента
            item_description = item.get(description)
            if not item_description:
```

```
                item_description = ''

                ws.cell(row=row, column=col + 2, value=item_description)
                ws[f'{get_column_letter(col + 2)}{row}'].alignment = (
                    Alignment(
                        horizontal='center', vertical='center',
                        wrap_text=True))
                # col += 1
                row += 1

            # дополнительные поля
            if field_name == 'additional_info':
                row_additional_data = ws.max_row + 1

                for additional_item in field_data:

                    field_for_element = additional_item.get('element_id')
                    element_coordinates = items_id_dict.get(field_for_element)

                    additional_field_name = additional_item.get('field_name')

                    additional_field_name = f'{cnt_category}.{cnt_elem}
{additional_field_name}'  # прим. 1.7 Состояние фундамента
                    cnt_elem += 1  # увеличиваем счетчик элементов осмотра
после
                    ws.cell(row=row_additional_data, column=1,
value=additional_field_name)
                    ws[f'A{row_additional_data}'].alignment =
Alignment(horizontal='left', vertical='center',
```

```
                                                wrap_text=True)
                count_row = await get_wrapped_text_lines(ws, ws[
                    f'A{row_additional_data}'])  # количество строк в ячейке
                ws.row_dimensions[
                    row_additional_data].height = 28 * count_row  # установка
высоты ячейки в зависимости от количества строк в ней

                additional_state = additional_item.get('field_state')

                if additional_state is not None:
                    if additional_state.value == 'Good':
                        field_state = 'У'
                    elif additional_state.value == 'Bad':
                        field_state = 'Н'
                else:
                    field_state = '-'

                # Проверяем, есть ли уже записи для данного элемента
                while ws.cell(row=row_additional_data,
column=element_coordinates[1]).value:
                    row_additional_data += 1  # Ищем свободную строку

                # состояние дополнительного элемента
                ws.cell(row=row_additional_data,
column=element_coordinates[1], value=field_state)

ws[f'{get_column_letter(element_coordinates[1])}{row_additional_data}'].align
ment = Alignment(
                    horizontal='center', vertical='center', wrap_text=True)
```

```
                count_row = await get_wrapped_text_lines(ws, ws[
                    f'A{row_additional_data}'])  # количество строк в ячейке
                ws.row_dimensions[
                    row_additional_data].height = 28 * count_row  # установка высоты ячейки в зависимости от количества строк в ней

                # описание дополнительного элемента
                additional_item_description = additional_item.get('field_description')
                if not additional_item_description:
                    additional_item_description = ''
                ws.cell(row=row_additional_data, column=element_coordinates[1] + 1,
                        value=additional_item_description)
                ws[f'{get_column_letter(element_coordinates[1] + 1)}{row_additional_data}'].alignment = (
                    Alignment(
                        horizontal='center', vertical='center',
                        wrap_text=True))

                row_additional_data += 1
                row += 1
            row += 1
        col += 2

    cnt_category += 1  # увеличиваем счетчик категорий после данных для категории
```

```
async def export_data(junction_id: str, short: bool):
    """Формирования файла для экспорта"""

    # получение осмотра

    junction = await db_requests.get_junction(junction_id)

    junction_kind_id = junction['kind_id']
    kind = await db_requests.get_junction_kind(junction_kind_id)
    kind_name = kind.get('kind')
    junction['kind'] = kind_name
    del junction['kind_id']

    wb = openpyxl.Workbook()
    ws = wb.active
    ws.title = "КЛ ГОСТ"

    ws.column_dimensions['A'].width = 30  # ширина столбца
    ws.column_dimensions['B'].width = 10  # ширина столбца
    ws.column_dimensions['C'].width = 12  # ширина столбца
    ws.column_dimensions['D'].width = 10  # ширина столбца
    ws.column_dimensions['E'].width = 12  # ширина столбца
    ws.column_dimensions['F'].width = 10  # ширина столбца
    ws.column_dimensions['G'].width = 12  # ширина столбца

    ws.merge_cells('A1:G1')

    # нижнее обрамление
    border_bottom_style = Border(bottom=Side(style='thin', color='000000'))
```

```
# head
# Размер шрифта
head_font_style = Font(name='Times New Roman', size=12)

# Установка размера шрифта к диапазону ячеек
for row in ws['A1:G7']:
    for cell in row:
        cell.font = head_font_style

# Устанавливаем текст в объединённую ячейку
ws['A1'] = 'Контрольный лист осмотра светофорного объекта'
# Выставляем выравнивание по центру (по горизонтали и вертикали)
ws['A1'].alignment = Alignment(horizontal='center', vertical='center')

ws['A3'] = 'Номер документа:'
ws.merge_cells('B3:G3')
ws['B3'] = junction.get('number')
ws['B3'].border = border_bottom_style

ws['A5'] = 'Адрес объекта:'
ws.merge_cells('B5:G5')
ws['B5'] = junction.get('address')
ws['B5'].border = border_bottom_style

ws['A7'] = 'Дата/время осмотра:'
ws.merge_cells('B7:G7')
inspection_at = junction.get('inspection_at')
formatted_date = inspection_at.strftime("%d.%m.%Y %H.%M")
```

```
    ws['B7'] = formatted_date
    ws['B7'].border = border_bottom_style

    ws['A9'] = 'Наименование группы \n элементов и пункт проверки'
    ws['A9'].alignment = Alignment(horizontal='center', vertical='center',
wrap_text=True)
    ws.merge_cells('B9:G9')
    ws['B9'].alignment = Alignment(horizontal='center', vertical='center',
wrap_text=True)
    ws['B9'] = 'Номер (идентификатор) элемента в составе светофорного
объекта и результат проверки'
    ws.row_dimensions[9].height = 30  # высота строки

    await category_work(ws, junction_id, short)  # добавление данных о
категориях осмотра

    # установка стиля шрифта и обрамления
    document_font_style = Font(name='Times New Roman', size=11)
    border_cell_style = Border(
        top=Side(style='thin', color='000000'),  # Верхняя граница
        bottom=Side(style='thin', color='000000'),  # Нижняя граница
        left=Side(style='thin', color='000000'),  # Левая граница
        right=Side(style='thin', color='000000')  # Правая граница
    )
    for row in ws.iter_rows(min_row=9):
        for cell in row:
            cell.font = document_font_style
            cell.border = border_cell_style
```

```
    filled_categories = True
    if filled_categories:
        legend_row = ws.max_row + 1

        rich_text = CellRichText(
            TextBlock(text="Примечание - ", font=InlineFont(b=True)),  # Жирный
фрагмент
            TextBlock(
                text=' В графе "Оценка состояния" проставляют оценку
"удовлетворительно" ("У") при отсутствии замечаний,
"неудовлетворительно" ("Н") - при наличии замечаний.',
                font=InlineFont(b=False))  # Обычный фрагмент
        )

        ws[f'A{legend_row}'].value = rich_text
        ws[f'A{legend_row}'].alignment = Alignment(horizontal='left',
vertical='center', wrap_text=True)
        ws[f'A{legend_row}'].font = document_font_style

        ws.merge_cells(f'A{legend_row}:G{legend_row}')
        ws.row_dimensions[legend_row].height = 30  # высота строки

    # bottom
    # Получаем количество заполненных строк
    filled_rows = ws.max_row + 2  # последняя заполненная строка + отступ

    # ws.merge_cells(f'A{filled_rows}:G{filled_rows}')
    ws[f'A{filled_rows}'] = 'Контрольный лист осмотра выполнен в
соответствии с формой Приложения А ГОСТ Р 59103-2020.'
```

```
    ws[f'A{filled_rows + 2}'] = 'От Заказчика:'

    column_for_signed = ['A', 'C', 'D', 'F', 'G']
    for column in column_for_signed:
        ws[f'{column}{filled_rows + 4}'].border = border_bottom_style

    description_font_style = Font(name='Times New Roman', size=8)

    ws[f'A{filled_rows + 5}'] = 'Должность'
    ws[f'A{filled_rows + 5}'].alignment = Alignment(horizontal='center',
vertical='center')

    ws[f'C{filled_rows + 5}'] = 'Подпись'
    ws.merge_cells(f'C{filled_rows + 5}:D{filled_rows + 5}')
    ws[f'C{filled_rows + 5}'].alignment = Alignment(horizontal='center',
vertical='center')

    ws[f'F{filled_rows + 5}'] = 'ФИО'
    ws.merge_cells(f'F{filled_rows + 5}:G{filled_rows + 5}')
    ws[f'F{filled_rows + 5}'].alignment = Alignment(horizontal='center',
vertical='center')

    ws[f'A{filled_rows + 7}'] = 'От Исполнителя:'

    for column in column_for_signed:
        ws[f'{column}{filled_rows + 8}'].border = border_bottom_style

    ws[f'A{filled_rows + 8}'] = junction.get('executor_position')
```

```
    ws[f'A{filled_rows + 8}'].alignment = Alignment(horizontal='center',
vertical='center')

    ws[f'F{filled_rows + 8}'] = junction.get('executor_name')
    ws.merge_cells(f'F{filled_rows + 8}:G{filled_rows + 8}')
    ws[f'F{filled_rows + 8}'].alignment = Alignment(horizontal='center',
vertical='center')

    ws[f'A{filled_rows + 9}'] = 'Должность'
    ws[f'A{filled_rows + 9}'].alignment = Alignment(horizontal='center',
vertical='center')

    ws.merge_cells(f'C{filled_rows + 9}:D{filled_rows + 9}')
    ws[f'C{filled_rows + 9}'] = 'Подпись'
    ws[f'C{filled_rows + 9}'].alignment = Alignment(horizontal='center',
vertical='center')

    ws.merge_cells(f'F{filled_rows + 9}:G{filled_rows + 9}')
    ws[f'F{filled_rows + 9}'] = 'ФИО'
    ws[f'F{filled_rows + 9}'].alignment = Alignment(horizontal='center',
vertical='center')

    # установка шрифта
    for row in ws.iter_rows(min_row=filled_rows):
        for cell in row:
            cell.font = document_font_style

    # уменьшение шрифта для пунктов "подпись", "должность", "ФИО"
    ws[f'A{filled_rows + 5}'].font = description_font_style
```

```
    ws[f'C{filled_rows + 5}'].font = description_font_style
    ws[f'F{filled_rows + 5}'].font = description_font_style
    ws[f'A{filled_rows + 9}'].font = description_font_style
    ws[f'C{filled_rows + 9}'].font = description_font_style
    ws[f'F{filled_rows + 9}'].font = description_font_style

    output = BytesIO()
    wb.save(output)
    output.seek(0)

    return output
"""
Скрипт db_requests.py, являющийся неотъемлемой частью текста
программы, предназначен для
обеспечения взаимодействия с базой данных. Содержит функции для
выполнения CRUD-операций
(создание, чтение, обновление, удаление) для различных сущностей
системы.
Входные данные: параметры запросов к базе данных, идентификаторы
записей, данные для создания и обновления записей.
Выходные данные: результаты запросов к базе данных, объекты данных,
статусы выполнения операций.
"""

import asyncio
from datetime import datetime, timedelta
from dateutil import parser
import pytz
```

```
from sqlalchemy.future import select
from sqlalchemy import update, desc, asc, cast, DateTime

from connectors.db_models import Supports
from connectors.orm import db
from connectors import db_models
from api.util import cuid2_generator, get_object_state



async def add_user(user_id, name, login, password, created_by):
    if not user_id:
        user_id: str = await cuid2_generator()

    new_user = db_models.User(
        id=user_id,
        name=name,
        login=login,
        password=password,
        created_by=created_by
    )

    async with db.AsyncSession() as session:
        session.add(new_user)
        await session.commit()
        await session.refresh(new_user)
        return new_user



async def check_user(login):
```

```
    """Проверка существования пользователя с переданным login"""

    async with db.AsyncSession() as session:
        query = select(db_models.User).where(db_models.User.login == login)
        result = await session.execute(query)
        user = result.first()

        return user


async def get_all_users():
    """Получение всех пользователей"""

    async with db.AsyncSession() as session:
        query = select(db_models.User)
        result = await session.execute(query)
        users = result.scalars().all()

        return [{'id': user.id,
                 'login': user.login,
                 'name': user.name,
                 'created_by': user.created_by,
                 'created_at': user.created_at,
                 'updated_at': user.updated_at} for user in users]


async def get_user(user_id: str = None, login: str = None):
    """Получение пользователя по его id или login"""
```

```
    async with db.AsyncSession() as session:
        if user_id and not login:
            query = select(db_models.User).where(db_models.User.id == user_id)
        elif not user_id and login:
            query = select(db_models.User).where(db_models.User.login == login)
        result = await session.execute(query)
        user_data = result.scalar_one_or_none()

        if not user_data:
            raise ValueError(f"Запись с id {user_id} не найдена.")

        user_data_result = {'id': user_data.id,
                            'login': user_data.login,
                            'name': user_data.name,
                            'created_by': user_data.created_by,
                            'created_at': user_data.created_at,
                            'updated_at': user_data.updated_at,
                            }

        return user_data_result


async def get_user_pass(user_id: str = None):
    """Получение пароля пользователя по его id"""

    async with db.AsyncSession() as session:
        query = select(db_models.User.password).where(db_models.User.id ==
user_id)
        result = await session.execute(query)
```

```
        user_pass = result.scalar_one_or_none()

        if not user_pass:
            raise ValueError(f"Запись с id {user_id} не найдена.")

        return user_pass


async def delete_user(user_id: str):
    async with db.AsyncSession() as session:
        async with session.begin():
            query = select(db_models.User).where(db_models.User.id == user_id)
            result = await session.execute(query)
            user_to_delete = result.scalar_one_or_none()

            if not user_to_delete:
                raise ValueError(f"Запись с id {user_id} не найдена.")

            await session.delete(user_to_delete)
            await session.commit()
            return True


async def update_user(user_id: str, new_user_name: str):
    async with db.AsyncSession() as session:
        async with session.begin():
            query = (
                update(db_models.User)
                .where(db_models.User.id == user_id)
```

```
            .values(name=new_user_name)
        )

        result = await session.execute(query)
        await session.commit()
        return result.rowcount


async def get_all_junction_kinds():
    """Получить все виды перекрестков."""

    async with db.AsyncSession() as session:
        query = select(db_models.JunctionsKind)
        result = await session.execute(query)
        kinds = result.scalars().all()

        return [{c.name: getattr(kind, c.name) for c in kind.__table__.columns} for
kind in kinds]


async def get_junction_kind(junction_kind_id: str):
    """Получить объект осмотра по его id"""

    async with db.AsyncSession() as session:
        query =
select(db_models.JunctionsKind).where(db_models.JunctionsKind.id ==
junction_kind_id)
        result = await session.execute(query)
        junction_kind = result.scalar_one_or_none()
```

```
        if not junction_kind:
            raise ValueError(f"Запись с id {junction_kind_id} не найдена.")

        return {c.name: getattr(junction_kind, c.name) for c in
junction_kind.__table__.columns}



async def add_junctions_kind(kind_id, kind_name):
    if not kind_id:
        kind_id: str = await cuid2_generator()

    new_kind = db_models.JunctionsKind(
        id=kind_id,
        kind=kind_name,
    )

    async with db.AsyncSession() as session:
        session.add(new_kind)
        await session.commit()
        await session.refresh(new_kind)
        return new_kind



async def delete_junctions_kind(kind_id: str):
    async with db.AsyncSession() as session:
        async with session.begin():
            query =
select(db_models.JunctionsKind).where(db_models.JunctionsKind.id == kind_id)
```

```
        result = await session.execute(query)
        kind_to_delete = result.scalar_one_or_none()

        if not kind_to_delete:
            raise ValueError(f"Запись с id {kind_id} не найдена.")

        await session.delete(kind_to_delete)
        await session.commit()
        return True


async def update_junctions_kind(kind_id: str, new_kind_name):
    async with db.AsyncSession() as session:
        async with session.begin():  # Открываем транзакцию
            # Создаем запрос на обновление
            query = (
                update(db_models.JunctionsKind)
                .where(db_models.JunctionsKind.id == kind_id)
                .values(kind=new_kind_name)
            )

            # Выполняем запрос
            result = await session.execute(query)

            # Сохраняем изменения
            await session.commit()

            # Возвращаем количество обновленных строк
            return result.rowcount
```

```
async def add_junctions(junction_id, kind_id, address, longitude, latitude,
                        status, number, inspection_status,
                        inspection_at, executor_name, executor_position, comment,
                        region, locality, project):
    if not junction_id:
        junction_id: str = await cuid2_generator()

    try:
        inspection_status = db_models.InspectionStatus(inspection_status)
    except ValueError:
        inspection_status = db_models.InspectionStatus.NOT_INSPECTED

    new_junction = db_models.Junctions(
        id=junction_id,
        kind_id=kind_id,
        address=address,
        longitude=longitude,
        latitude=latitude,
        status=status,
        number=number,
        inspection_status=inspection_status,
        inspection_at=inspection_at,
        executor_name=executor_name,
        executor_position=executor_position,
        comment=comment,
        region=region,
        locality=locality,
```

```
        project=project,
    )

    async with db.AsyncSession() as session:
        session.add(new_junction)
        await session.commit()
        await session.refresh(new_junction)
        return new_junction


async def get_all_junctions(sort_by: str, sort_direction: str,
                            search_by_number: str,
                            search_by_address: str,
                            search_by_executor_name: str,
                            search_by_created_at: str,
                            search_by_locality: str,
                            ):
    """Получить все объекты осмотров"""

    async with db.AsyncSession() as session:

        if not hasattr(db_models.Junctions, sort_by):
            raise ValueError(f"Поле сортировки отсутствует: {sort_by}")

        order_by_clause = asc(getattr(db_models.Junctions, sort_by)) if
sort_direction == "asc" else desc(
            getattr(db_models.Junctions, sort_by))

        query = select(db_models.Junctions).order_by(order_by_clause)
```

```
    if search_by_number:
        query =
query.filter(db_models.Junctions.number.ilike(f"%{search_by_number}%"))

    if search_by_address:
        query =
query.filter(db_models.Junctions.address.ilike(f"%{search_by_address}%"))

    if search_by_locality:
        query =
query.filter(db_models.Junctions.locality.ilike(f"%{search_by_locality}%"))

    if search_by_executor_name:
        query =
query.filter(db_models.Junctions.executor_name.ilike(f"%{search_by_executor_
name}%"))

    if search_by_created_at:
        try:
            parsed_date = parser.parse(search_by_created_at)

            # Приводим к UTC, если дата без временной зоны
            if parsed_date.tzinfo is None:
                parsed_date = parsed_date.replace(tzinfo=pytz.UTC)

            if parsed_date.time() == datetime.min.time():
                # Если указана только дата, ищем записи в течение дня (00:00 -
23:59)
```

```
                start_date = parsed_date.replace(hour=0, minute=0, second=0,
microsecond=0)
                end_date = start_date + timedelta(days=1)

                query = query.filter(
                    db_models.Junctions.created_at >= start_date,
                    db_models.Junctions.created_at < end_date
                )
            else:
                # Если указано точное время, ищем с разбросом в 1 секунду,
чтобы нивелировать милисекунды

                start_time = parsed_date
                end_time = parsed_date + timedelta(seconds=1)

                query = query.filter(
                    db_models.Junctions.created_at >= start_time,
                    db_models.Junctions.created_at < end_time
                )

        except Exception as e:
            raise ValueError(f"Некорректный формат даты:
{search_by_created_at}. Ошибка: {e}")

    result = await session.execute(query)
    junctions = result.scalars().all()

    return [{c.name: getattr(junction, c.name) for c in
junction.__table__.columns} for junction in junctions]
```

```
async def get_junction(junction_id: str):
    """Получить объект осмотра по его id"""

    async with db.AsyncSession() as session:
        query = select(db_models.Junctions).where(db_models.Junctions.id ==
junction_id)
        result = await session.execute(query)
        junction = result.scalar_one_or_none()

        if not junction:
            raise ValueError(f"Запись с id {junction_id} не найдена.")

        return {c.name: getattr(junction, c.name) for c in
junction.__table__.columns}



async def delete_junction(junction_id: str):
    async with db.AsyncSession() as session:
        async with session.begin():
            query = select(db_models.Junctions).where(db_models.Junctions.id ==
junction_id)
            result = await session.execute(query)
            junction_to_delete = result.scalar_one_or_none()

            if not junction_to_delete:
                raise ValueError(f"Запись с id {junction_id} не найдена.")
```

```
        await session.delete(junction_to_delete)
        await session.commit()
        return True


async def update_junction(junction_id, kind_id, address, longitude, latitude,
                          status, number, inspection_status,
                          inspection_at, executor_name, executor_position, comment,
                          region, locality, project):
    async with db.AsyncSession() as session:
        async with session.begin():
            query = select(db_models.Junctions).where(db_models.Junctions.id ==
junction_id)
            result = await session.execute(query)
            junction_to_update = result.scalar_one_or_none()

            if not junction_to_update:
                raise ValueError(f"Запись с id {junction_id} не найдена.")

            if kind_id:
                junction_to_update.kind_id = kind_id
            if region:
                junction_to_update.region = region
            if locality:
                junction_to_update.locality = locality
            if address:
                junction_to_update.address = address
            if longitude:
                junction_to_update.longitude = longitude
```

```
        if latitude:
            junction_to_update.latitude = latitude
        if status:
            junction_to_update.status = status
        if number:
            junction_to_update.number = number
        if project:
            junction_to_update.project = project
        if inspection_status:
            try:
                inspection_status = db_models.InspectionStatus(inspection_status)
            except ValueError:
                inspection_status = db_models.InspectionStatus.NOT_INSPECTED
            junction_to_update.inspection_status = inspection_status
        if inspection_at:
            junction_to_update.inspection_at = inspection_at
        if executor_name:
            junction_to_update.executor_name = executor_name
        if executor_position:
            junction_to_update.executor_position = executor_position
        if comment:
            junction_to_update.comment = comment

        await session.commit()
        return True


async def add_support(support_id, junction_id, name, verticality_sagging_state,
                      verticality_sagging_description, base_state, base_description,
```

```
                inscriptions_advertisement_state,
inscriptions_advertisement_description,
                dirt_sagging_state, dirt_sagging_description, rust_color_state,
rust_color_description,
                hatch_state, hatch_description, broken_constructions_state,
broken_constructions_description,
                additional_info):
    if not support_id:
        support_id: str = await cuid2_generator()

    try:
        verticality_sagging_state =
db_models.CheckObjectState(verticality_sagging_state)
    except ValueError:
        verticality_sagging_state = None

    try:
        base_state = db_models.CheckObjectState(base_state)
    except ValueError:
        base_state = None

    try:
        inscriptions_advertisement_state =
db_models.CheckObjectState(inscriptions_advertisement_state)
    except ValueError:
        inscriptions_advertisement_state = None

    try:
        dirt_sagging_state = db_models.CheckObjectState(dirt_sagging_state)
```

```
    except ValueError:
        dirt_sagging_state = None

    try:
        rust_color_state = db_models.CheckObjectState(rust_color_state)
    except ValueError:
        rust_color_state = None

    try:
        hatch_state = db_models.CheckObjectState(hatch_state)
    except ValueError:
        hatch_state = None

    try:
        broken_constructions_state =
db_models.CheckObjectState(broken_constructions_state)
    except ValueError:
        broken_constructions_state = None

    new_support = db_models.Supports(
        id=support_id,
        junction_id=junction_id,
        name=name,
        verticality_sagging_state=verticality_sagging_state,
        verticality_sagging_description=verticality_sagging_description,
        base_state=base_state,
        base_description=base_description,
        inscriptions_advertisement_state=inscriptions_advertisement_state,
```

```
inscriptions_advertisement_description=inscriptions_advertisement_description,
        dirt_sagging_state=dirt_sagging_state,
        dirt_sagging_description=dirt_sagging_description,
        rust_color_state=rust_color_state,
        rust_color_description=rust_color_description,
        hatch_state=hatch_state,
        hatch_description=hatch_description,
        broken_constructions_state=broken_constructions_state,
        broken_constructions_description=broken_constructions_description,
    )

    async with db.AsyncSession() as session:
        session.add(new_support)
        await session.commit()
        await session.refresh(new_support)

        if additional_info:
            for item in additional_info:
                additional_id: str = await cuid2_generator()
                junction_id: str = junction_id
                element_id = new_support.id
                name: str = item.name
                category = db_models.AdditionalInfoCategory.SUPPORTS
                field_name: str = item.field_name
                try:
                    field_state = db_models.CheckObjectState(item.field_state)
                except ValueError:
                    field_state = None
```

```
            field_description = item.field_description

            new_additional_info = db_models.AdditionalInfo(
                id=additional_id,
                junction_id=junction_id,
                element_id=element_id,
                name=name,
                category=category,
                field_name=field_name,
                field_state=field_state,
                field_description=field_description,
            )

            session.add(new_additional_info)
        await session.commit()

    return new_support


async def get_supports_for_junction(junction_id: str):
    """Получить все опоры для осмотра"""

    async with db.AsyncSession() as session:
        query = select(db_models.Supports).where(db_models.Supports.junction_id
== junction_id)
        result = await session.execute(query)
        supports = result.scalars().all()

        supports_result = (
```

```
        [{c.name: getattr(support, c.name) for c in support.__table__.columns} for
support in supports])

    for support in supports_result:
        support_id = support.get('id')
        query =
select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.element_id
== support_id,
                                               db_models.AdditionalInfo.category ==
db_models.AdditionalInfoCategory.SUPPORTS
                                               )

        additional_data = await session.execute(query)

        if additional_data:
            additional_data = additional_data.scalars().all()
            support['additional_info'] = [{c.name: getattr(add_data, c.name) for c in
add_data.__table__.columns}
                                          for add_data in additional_data]

    return supports_result



async def get_support_for_junction(support_id: str):
    """Получить запись 'Опора' осмотра по ее id"""

    async with db.AsyncSession() as session:
        query = select(db_models.Supports).where(db_models.Supports.id ==
support_id)
```

```
    result = await session.execute(query)
    support = result.scalar_one_or_none()

    if not support:
        raise ValueError(f"Запись с id {support_id} не найдена.")

    support_result = {c.name: getattr(support, c.name) for c in
support.__table__.columns}

    query =
select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.element_id
== support_id,
                                           db_models.AdditionalInfo.category ==
db_models.AdditionalInfoCategory.SUPPORTS
                                           )

    additional_data = await session.execute(query)

    if additional_data:
        additional_data = additional_data.scalars().all()
        support_result['additional_info'] = [
            {c.name: getattr(add_data, c.name) for c in
add_data.__table__.columns}
            for add_data in additional_data]

    return support_result


async def delete_support(support_id: str):
```

```
    """Удаление опоры по ее id"""

    async with db.AsyncSession() as session:
        async with session.begin():
            query = select(db_models.Supports).where(db_models.Supports.id ==
support_id)
            result = await session.execute(query)
            support_to_delete = result.scalar_one_or_none()

            additional_data_query = select(db_models.AdditionalInfo).where(
                db_models.AdditionalInfo.element_id == support_id)
            additional_data_result = await session.execute(additional_data_query)
            additional_data_to_delete = additional_data_result.scalars().all()

            if not support_to_delete:
                raise ValueError(f"Запись с id {support_id} не найдена.")

            await session.delete(support_to_delete)
            if additional_data_to_delete:
                for item_to_delete in additional_data_to_delete:
                    await session.delete(item_to_delete)
            await session.commit()
            return True


async def delete_supports_for_junction(junction_id: str):
    """Удаление всех опор привязанных к осмотру"""

    async with db.AsyncSession() as session:
```

```
    async with session.begin():
        query =
select(db_models.Supports).where(db_models.Supports.junction_id ==
junction_id)
        result = await session.execute(query)
        support_to_delete = result.scalars().all()

        for support in support_to_delete:
            support_id = support.id

            await session.delete(support)

            additional_info_query = select(db_models.AdditionalInfo).where(
                db_models.AdditionalInfo.element_id == support_id)
            additional_info_result = await session.execute(additional_info_query)
            additional_info_to_delete = additional_info_result.scalars().all()

            await asyncio.gather(*(session.delete(item) for item in
additional_info_to_delete))

        await session.commit()
        return True



async def update_support(support_id, junction_id, name,
verticality_sagging_state,
                         verticality_sagging_description, base_state, base_description,
                         inscriptions_advertisement_state,
inscriptions_advertisement_description,
```

```
                dirt_sagging_state, dirt_sagging_description, rust_color_state,
                rust_color_description, hatch_state, hatch_description,
                broken_constructions_state, broken_constructions_description,
                additional_info):
    async with db.AsyncSession() as session:
        async with session.begin():
            query = select(db_models.Supports).where(db_models.Supports.id ==
support_id)
            result = await session.execute(query)
            support_to_update = result.scalar_one_or_none()

            if not support_to_update:
                raise ValueError(f"Запись с id {support_id} не найдена.")

            if junction_id:
                support_to_update.junction_id = junction_id
            if name:
                support_to_update.name = name
            if verticality_sagging_state:
                support_to_update.verticality_sagging_state = await
get_object_state(verticality_sagging_state)
            if verticality_sagging_description:
                support_to_update.verticality_sagging_description =
verticality_sagging_description
            if base_state:
                support_to_update.base_state = await get_object_state(base_state)
            if base_description:
                support_to_update.base_description = base_description
            if inscriptions_advertisement_state:
```

```
            support_to_update.inscriptions_advertisement_state = await
get_object_state(
                inscriptions_advertisement_state)
        if inscriptions_advertisement_description:
            support_to_update.inscriptions_advertisement_description =
inscriptions_advertisement_description
        if dirt_sagging_state:
            support_to_update.dirt_sagging_state = await
get_object_state(dirt_sagging_state)
        if dirt_sagging_description:
            support_to_update.dirt_sagging_description = dirt_sagging_description
        if rust_color_state:
            support_to_update.rust_color_state = await
get_object_state(rust_color_state)
        if rust_color_description:
            support_to_update.rust_color_description = rust_color_description
        if hatch_state:
            support_to_update.hatch_state = await get_object_state(hatch_state)
        if hatch_description:
            support_to_update.hatch_description = hatch_description
        if broken_constructions_state:
            support_to_update.broken_constructions_state = await
get_object_state(broken_constructions_state)
        if broken_constructions_description:
            support_to_update.broken_constructions_description =
broken_constructions_description

        if additional_info:
            for item in additional_info:
```

```
            additional_id = item.id
            additional_name = item.name
            additional_field_name = item.field_name
            additional_field_state = item.field_state
            additional_field_description = item.field_description

            additional_info_query = select(db_models.AdditionalInfo).where(
                db_models.AdditionalInfo.id == additional_id)
            additional_info_result = await
session.execute(additional_info_query)
            additional_info_update =
additional_info_result.scalar_one_or_none()

            if additional_name:
                additional_info_update.name = additional_name
            if additional_field_name:
                additional_info_update.field_name = additional_field_name
            if additional_field_state:
                additional_info_update.field_state = await
get_object_state(additional_field_state)
            if additional_field_description:
                additional_info_update.additional_field_description =
additional_field_description

        await session.commit()
        return True
```

```
async def add_controller(controller_id, junction_id, name, dk_name, dk_serial,
position_state, position_description,
                         inscriptions_advertisement_state,
inscriptions_advertisement_description,
                         dirt_state, dirt_description, rust_color_state,
rust_color_description,
                         cabinet_state, cabinet_description, cabinet_hardware_state,
cabinet_hardware_description,
                         dirt_inside_state, dirt_inside_description, base_state,
base_description,
                         time_accuracy_state, time_accuracy_description, passport_state,
passport_description,
                         cable_marking_state, cable_marking_description,
contact_density_state,
                         contact_density_description, cable_isolation_state,
cable_isolation_description,
                         grounding_cable_state, grounding_cable_description,
fastening_reliability_state,
                         fastening_reliability_description, additional_info,
                         ):
    if not controller_id:
        controller_id: str = await cuid2_generator()

    position_state = await get_object_state(position_state)
    inscriptions_advertisement_state = await
get_object_state(inscriptions_advertisement_state)
    dirt_state = await get_object_state(dirt_state)
    rust_color_state = await get_object_state(rust_color_state)
    cabinet_state = await get_object_state(cabinet_state)
```

```
    cabinet_hardware_state = await get_object_state(cabinet_hardware_state)
    dirt_inside_state = await get_object_state(dirt_inside_state)
    base_state = await get_object_state(base_state)
    time_accuracy_state = await get_object_state(time_accuracy_state)
    passport_state = await get_object_state(passport_state)
    cable_marking_state = await get_object_state(cable_marking_state)
    contact_density_state = await get_object_state(contact_density_state)
    cable_isolation_state = await get_object_state(cable_isolation_state)
    grounding_cable_state = await get_object_state(grounding_cable_state)
    fastening_reliability_state = await get_object_state(fastening_reliability_state)

    new_support = db_models.Controllers(
        id=controller_id,
        junction_id=junction_id,
        name=name,
        dk_name=dk_name,
        dk_serial=dk_serial,
        position_state=position_state,
        position_description=position_description,
        inscriptions_advertisement_state=inscriptions_advertisement_state,

inscriptions_advertisement_description=inscriptions_advertisement_description,
        dirt_state=dirt_state,
        dirt_description=dirt_description,
        rust_color_state=rust_color_state,
        rust_color_description=rust_color_description,
        cabinet_state=cabinet_state,
        cabinet_description=cabinet_description,
        cabinet_hardware_state=cabinet_hardware_state,
```

```
    cabinet_hardware_description=cabinet_hardware_description,
    dirt_inside_state=dirt_inside_state,
    dirt_inside_description=dirt_inside_description,
    base_state=base_state,
    base_description=base_description,
    time_accuracy_state=time_accuracy_state,
    time_accuracy_description=time_accuracy_description,
    passport_state=passport_state,
    passport_description=passport_description,
    cable_marking_state=cable_marking_state,
    cable_marking_description=cable_marking_description,
    contact_density_state=contact_density_state,
    contact_density_description=contact_density_description,
    cable_isolation_state=cable_isolation_state,
    cable_isolation_description=cable_isolation_description,
    grounding_cable_state=grounding_cable_state,
    grounding_cable_description=grounding_cable_description,
    fastening_reliability_state=fastening_reliability_state,
    fastening_reliability_description=fastening_reliability_description,
)

async with db.AsyncSession() as session:
    session.add(new_support)
    await session.commit()
    await session.refresh(new_support)

    if additional_info:
        for item in additional_info:
            additional_id: str = await cuid2_generator()
```

```
            junction_id: str = junction_id
            element_id = new_support.id
            name: str = item.name
            category = db_models.AdditionalInfoCategory.CONTROLLERS
            field_name: str = item.field_name

            field_state = await get_object_state(item.field_state)

            field_description = item.field_description

            new_additional_info = db_models.AdditionalInfo(
                id=additional_id,
                junction_id=junction_id,
                element_id=element_id,
                name=name,
                category=category,
                field_name=field_name,
                field_state=field_state,
                field_description=field_description,
            )

            session.add(new_additional_info)
        await session.commit()

    return new_support


async def get_controllers_for_junction(junction_id: str):
    """Получить все контроллеры для осмотра"""
```

```
    async with db.AsyncSession() as session:
        query =
select(db_models.Controllers).where(db_models.Controllers.junction_id ==
junction_id)
        result = await session.execute(query)
        controllers = result.scalars().all()

        controllers_result = (
            [{c.name: getattr(controller, c.name) for c in
controller.__table__.columns} for controller in controllers])

        for controller in controllers_result:
            controller_id = controller.get('id')
            query =
select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.element_id
== controller_id,
                                                db_models.AdditionalInfo.category ==
db_models.AdditionalInfoCategory.CONTROLLERS
                                                )

            additional_data = await session.execute(query)

            if additional_data:
                additional_data = additional_data.scalars().all()
                controller['additional_info'] = [{c.name: getattr(add_data, c.name) for c
in add_data.__table__.columns}
                                                 for add_data in additional_data]
```

```
        return controllers_result


async def get_controller(controller_id: str):
    """Получить запись 'контроллер' осмотра по ее id"""

    async with db.AsyncSession() as session:
        query = select(db_models.Controllers).where(db_models.Controllers.id ==
controller_id)
        result = await session.execute(query)
        controller = result.scalar_one_or_none()

        if not controller:
            raise ValueError(f"Запись с id {controller_id} не найдена.")

        controller_result = {c.name: getattr(controller, c.name) for c in
controller.__table__.columns}

        query =
select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.element_id
== controller_id,
                                               db_models.AdditionalInfo.category ==
db_models.AdditionalInfoCategory.CONTROLLERS
                                               )

        additional_data = await session.execute(query)

        if additional_data:
            additional_data = additional_data.scalars().all()
```

```
            controller_result['additional_info'] = [
                {c.name: getattr(add_data, c.name) for c in
add_data.__table__.columns}
                for add_data in additional_data]

        return controller_result



async def delete_controller(controller_id: str):
    """Удаление контроллера по ее id"""

    async with db.AsyncSession() as session:
        async with session.begin():
            query = select(db_models.Controllers).where(db_models.Controllers.id
== controller_id)
            result = await session.execute(query)
            controller_to_delete = result.scalar_one_or_none()

            additional_data_query = select(db_models.AdditionalInfo).where(
                db_models.AdditionalInfo.element_id == controller_id)
            additional_data_result = await session.execute(additional_data_query)
            additional_data_to_delete = additional_data_result.scalars().all()

            if not controller_to_delete:
                raise ValueError(f"Запись с id {controller_id} не найдена.")

            await session.delete(controller_to_delete)
            if additional_data_to_delete:
                for item_to_delete in additional_data_to_delete:
```

```
                await session.delete(item_to_delete)
            await session.commit()
            return True


async def delete_controllers_for_junction(junction_id: str):
    """Удаление всех контроллеров привязанных к осмотру"""

    async with db.AsyncSession() as session:
        async with session.begin():
            query =
select(db_models.Controllers).where(db_models.Controllers.junction_id ==
junction_id)
            result = await session.execute(query)
            controllers_to_delete = result.scalars().all()

            for controller in controllers_to_delete:
                controller_id = controller.id

                await session.delete(controller)

                additional_info_query = select(db_models.AdditionalInfo).where(
                    db_models.AdditionalInfo.element_id == controller_id)
                additional_info_result = await session.execute(additional_info_query)
                additional_info_to_delete = additional_info_result.scalars().all()

                await asyncio.gather(*(session.delete(item) for item in
additional_info_to_delete))
```

```
            await session.commit()
            return True


async def update_controller(
        controller_id, **kwargs
):
    async with db.AsyncSession() as session:
        async with session.begin():
            query = select(db_models.Controllers).where(db_models.Controllers.id
== controller_id)
            result = await session.execute(query)
            controller_to_update = result.scalar_one_or_none()

            if not controller_to_update:
                raise ValueError(f"Запись с id {controller_id} не найдена.")

            state_fields = {
                "position_state",
                "inscriptions_advertisement_state",
                "dirt_state",
                "rust_color_state",
                "cabinet_state",
                "cabinet_hardware_state",
                "dirt_inside_state",
                "base_state",
                "time_accuracy_state",
                "passport_state",
                "cable_marking_state",
```

```
            "contact_density_state",
            "cable_isolation_state",
            "grounding_cable_state",
            "fastening_reliability_state",
        }

        updates = {}
        for field, value in kwargs.items():
            if value:
                updates[field] = await get_object_state(value) if field in state_fields
else value

        for attr, val in updates.items():
            setattr(controller_to_update, attr, val)

        if "additional_info" in kwargs and kwargs["additional_info"]:
            for item in kwargs["additional_info"]:
                additional_id = item.get('id')
                additional_name = item.get('name')
                additional_field_name = item.get('field_name')
                additional_field_state = item.get('field_state')
                additional_field_description = item.get('field_description')

                additional_info_query = select(db_models.AdditionalInfo).where(
                    db_models.AdditionalInfo.id == additional_id)
                additional_info_result = await
session.execute(additional_info_query)
                additional_info_update =
additional_info_result.scalar_one_or_none()
```

```
            if additional_name:
                additional_info_update.name = additional_name
            if additional_field_name:
                additional_info_update.field_name = additional_field_name
            if additional_field_state:
                additional_info_update.field_state = await
get_object_state(additional_field_state)
            if additional_field_description:
                additional_info_update.additional_field_description =
additional_field_description

        await session.commit()
        return True


async def add_power_backup(power_backup_id, junction_id, name,
body_damage_state, body_damage_description,
                           battery_outside_state, battery_outside_description,
electrical_parameters_state,
                           electrical_parameters_description, indication_state,
indication_description,
                           functionality_state, functionality_description, additional_info
                           ):
    if not power_backup_id:
        power_backup_id: str = await cuid2_generator()

    body_damage_state = await get_object_state(body_damage_state)
    battery_outside_state = await get_object_state(battery_outside_state)
```

```
    electrical_parameters_state = await
get_object_state(electrical_parameters_state)
    indication_state = await get_object_state(indication_state)
    functionality_state = await get_object_state(functionality_state)

    new_power_backup = db_models.PowerBackup(
        id=power_backup_id,
        junction_id=junction_id,
        name=name,
        body_damage_state=body_damage_state,
        body_damage_description=body_damage_description,
        battery_outside_state=battery_outside_state,
        battery_outside_description=battery_outside_description,
        electrical_parameters_state=electrical_parameters_state,
        electrical_parameters_description=electrical_parameters_description,
        indication_state=indication_state,
        indication_description=indication_description,
        functionality_state=functionality_state,
        functionality_description=functionality_description
    )

    async with db.AsyncSession() as session:
        session.add(new_power_backup)
        await session.commit()
        await session.refresh(new_power_backup)

        if additional_info:
            for item in additional_info:
                additional_id: str = await cuid2_generator()
```

```
            junction_id: str = junction_id
            element_id = new_power_backup.id
            name: str = item.name
            category = db_models.AdditionalInfoCategory.POWER_BACKUP
            field_name: str = item.field_name

            field_state = await get_object_state(item.field_state)

            field_description = item.field_description

            new_additional_info = db_models.AdditionalInfo(
                id=additional_id,
                junction_id=junction_id,
                element_id=element_id,
                name=name,
                category=category,
                field_name=field_name,
                field_state=field_state,
                field_description=field_description,
            )

            session.add(new_additional_info)
        await session.commit()

    return new_power_backup


async def get_power_backups_for_junction(junction_id: str):
    """Получить все источинки питания для осмотра"""
```

```
    async with db.AsyncSession() as session:
        query =
select(db_models.PowerBackup).where(db_models.PowerBackup.junction_id ==
junction_id)
        result = await session.execute(query)
        power_backups = result.scalars().all()

        power_backups_result = (
            [{c.name: getattr(power_backup, c.name) for c in
power_backup.__table__.columns} for power_backup in
             power_backups])

        for power_backup in power_backups_result:
            power_backup_id = power_backup.get('id')
            query =
select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.element_id
== power_backup_id,
                                                db_models.AdditionalInfo.category ==
db_models.AdditionalInfoCategory.POWER_BACKUP
                                                )

            additional_data = await session.execute(query)

            if additional_data:
                additional_data = additional_data.scalars().all()
                power_backup['additional_info'] = [
                    {c.name: getattr(add_data, c.name) for c in
add_data.__table__.columns}
```

```
                for add_data in additional_data]

    return power_backups_result


async def get_power_backup(power_backup_id: str):
    """Получить 'Резервный источник' питания для осмотра"""

    async with db.AsyncSession() as session:
        query = select(db_models.PowerBackup).where(db_models.PowerBackup.id
== power_backup_id)
        result = await session.execute(query)
        power_backup = result.scalar_one_or_none()

        if not power_backup:
            raise ValueError(f"Запись с id {power_backup_id} не найдена.")

        power_backup_result = {c.name: getattr(power_backup, c.name) for c in
power_backup.__table__.columns}

        query =
select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.element_id
== power_backup_id,
                                       db_models.AdditionalInfo.category ==
db_models.AdditionalInfoCategory.POWER_BACKUP
                                       )

        additional_data = await session.execute(query)
```

```
        if additional_data:
            additional_data = additional_data.scalars().all()
            power_backup_result['additional_info'] = [
                {c.name: getattr(add_data, c.name) for c in
add_data.__table__.columns}
                for add_data in additional_data]

        return power_backup_result



async def delete_power_backup(power_backup_id: str):
    """Удаление элемента питания по ее id"""

    async with db.AsyncSession() as session:
        async with session.begin():
            query =
select(db_models.PowerBackup).where(db_models.PowerBackup.id ==
power_backup_id)
            result = await session.execute(query)
            power_backup_to_delete = result.scalar_one_or_none()

            additional_data_query = select(db_models.AdditionalInfo).where(
                db_models.AdditionalInfo.element_id == power_backup_id)
            additional_data_result = await session.execute(additional_data_query)
            additional_data_to_delete = additional_data_result.scalars().all()

            if not power_backup_to_delete:
                raise ValueError(f"Запись с id {power_backup_id} не найдена.")
```

```
            await session.delete(power_backup_to_delete)
            if additional_data_to_delete:
                for item_to_delete in additional_data_to_delete:
                    await session.delete(item_to_delete)
            await session.commit()
            return True


async def delete_power_backups_for_junction(junction_id: str):
    """Удаление всех контроллеров привязанных к осмотру"""

    async with db.AsyncSession() as session:
        async with session.begin():
            query =
select(db_models.PowerBackup).where(db_models.PowerBackup.junction_id ==
junction_id)
            result = await session.execute(query)
            power_backups_to_delete = result.scalars().all()

            for power_backup in power_backups_to_delete:
                power_backup_id = power_backup.id

                await session.delete(power_backup)

                additional_info_query = select(db_models.AdditionalInfo).where(
                    db_models.AdditionalInfo.element_id == power_backup_id)
                additional_info_result = await session.execute(additional_info_query)
                additional_info_to_delete = additional_info_result.scalars().all()
```

```
            await asyncio.gather(*(session.delete(item) for item in
additional_info_to_delete))

        await session.commit()
        return True


async def update_power_backup(power_backup_id, **kwargs):
    async with db.AsyncSession() as session:
        async with session.begin():
            query =
select(db_models.PowerBackup).where(db_models.PowerBackup.id ==
power_backup_id)
            result = await session.execute(query)
            power_backup_to_update = result.scalar_one_or_none()

            if not power_backup_to_update:
                raise ValueError(f"Запись с id {power_backup_id} не найдена.")

            state_fields = {
                'body_damage_state',
                'battery_outside_state',
                'electrical_parameters_state',
                'indication_state',
                'functionality_state',
            }

            updates = {}
            for field, value in kwargs.items():
```

```
            if value:
                updates[field] = await get_object_state(value) if field in state_fields
else value

        for attr, val in updates.items():
            setattr(power_backup_to_update, attr, val)

        if "additional_info" in kwargs and kwargs["additional_info"]:
            for item in kwargs["additional_info"]:
                additional_id = item.get('id')
                additional_name = item.get('name')
                additional_field_name = item.get('field_name')
                additional_field_state = item.get('field_state')
                additional_field_description = item.get('field_description')

                additional_info_query = select(db_models.AdditionalInfo).where(
                    db_models.AdditionalInfo.id == additional_id)
                additional_info_result = await
session.execute(additional_info_query)
                additional_info_update =
additional_info_result.scalar_one_or_none()

                if additional_name:
                    additional_info_update.name = additional_name
                if additional_field_name:
                    additional_info_update.field_name = additional_field_name
                if additional_field_state:
                    additional_info_update.field_state = await
get_object_state(additional_field_state)
```

```
                if additional_field_description:
                    additional_info_update.additional_field_description =
additional_field_description

        await session.commit()
        return True


async def add_cables(cable_id, junction_id, name, tension_bends_state,
tension_bends_description,
                grounding_rail_state, grounding_rail_description,
                equipment_connection_state, equipment_connection_description,
                additional_info
                ):
    if not cable_id:
        cable_id: str = await cuid2_generator()

    tension_bends_state = await get_object_state(tension_bends_state)
    grounding_rail_state = await get_object_state(grounding_rail_state)
    equipment_connection_state = await
get_object_state(equipment_connection_state)

    new_cable = db_models.Cables(
        id=cable_id,
        junction_id=junction_id,
        name=name,
        tension_bends_state=tension_bends_state,
        tension_bends_description=tension_bends_description,
        grounding_rail_state=grounding_rail_state,
```

```
    grounding_rail_description=grounding_rail_description,
    equipment_connection_state=equipment_connection_state,
    equipment_connection_description=equipment_connection_description
)

async with db.AsyncSession() as session:
    session.add(new_cable)
    await session.commit()
    await session.refresh(new_cable)

    if additional_info:
        for item in additional_info:
            additional_id: str = await cuid2_generator()
            junction_id: str = junction_id
            element_id = new_cable.id
            name: str = item.name
            category = db_models.AdditionalInfoCategory.CABLES
            field_name: str = item.field_name

            field_state = await get_object_state(item.field_state)

            field_description = item.field_description

            new_additional_info = db_models.AdditionalInfo(
                id=additional_id,
                junction_id=junction_id,
                element_id=element_id,
                name=name,
                category=category,
```

```
                field_name=field_name,
                field_state=field_state,
                field_description=field_description,
            )

            session.add(new_additional_info)
        await session.commit()

    return new_cable


async def get_cables_for_junction(junction_id: str):
    """Получить все кабели для осмотра"""

    async with db.AsyncSession() as session:
        query = select(db_models.Cables).where(db_models.Cables.junction_id ==
junction_id)
        result = await session.execute(query)
        cables = result.scalars().all()

        cables_result = (
            [{c.name: getattr(cable, c.name) for c in cable.__table__.columns} for
cable in
             cables])

        for cable in cables_result:
            cable_id = cable.get('id')
```

```
            query =
select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.element_id
== cable_id,
                                                db_models.AdditionalInfo.category ==
db_models.AdditionalInfoCategory.CABLES
                                                )

            additional_data = await session.execute(query)

            if additional_data:
                additional_data = additional_data.scalars().all()
                cable['additional_info'] = [
                    {c.name: getattr(add_data, c.name) for c in
add_data.__table__.columns}
                    for add_data in additional_data]

        return cables_result



async def get_cable(cable_id: str):
    """Получить запись 'Кабель' по ее id"""

    async with db.AsyncSession() as session:
        query = select(db_models.Cables).where(db_models.Cables.id == cable_id)
        result = await session.execute(query)
        cable = result.scalar_one_or_none()

        if not cable:
            raise ValueError(f"Запись с id {cable_id} не найдена.")
```

```
        cable_result = {c.name: getattr(cable, c.name) for c in
cable.__table__.columns}

        query =
select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.element_id
== cable_id,
                                               db_models.AdditionalInfo.category ==
db_models.AdditionalInfoCategory.CABLES
                                               )

        additional_data = await session.execute(query)

        if additional_data:
            additional_data = additional_data.scalars().all()
            cable_result['additional_info'] = [
                {c.name: getattr(add_data, c.name) for c in
add_data.__table__.columns}
                for add_data in additional_data]

        return cable_result



async def delete_cable(cables_id: str):
    """Удаление кабеля по ее id"""

    async with db.AsyncSession() as session:
        async with session.begin():
```

```
        query = select(db_models.Cables).where(db_models.Cables.id ==
cables_id)
        result = await session.execute(query)
        power_backup_to_delete = result.scalar_one_or_none()

        additional_data_query = select(db_models.AdditionalInfo).where(
            db_models.AdditionalInfo.element_id == cables_id)
        additional_data_result = await session.execute(additional_data_query)
        additional_data_to_delete = additional_data_result.scalars().all()

        if not power_backup_to_delete:
            raise ValueError(f"Запись с id {cables_id} не найдена.")

        await session.delete(power_backup_to_delete)
        if additional_data_to_delete:
            for item_to_delete in additional_data_to_delete:
                await session.delete(item_to_delete)
        await session.commit()
        return True



async def delete_cables_for_junction(junction_id: str):
    """Удаление всех кабелей привязанных к осмотру"""

    async with db.AsyncSession() as session:
        async with session.begin():
            query = select(db_models.Cables).where(db_models.Cables.junction_id
== junction_id)
            result = await session.execute(query)
```

```
        cables_to_delete = result.scalars().all()

        for cable in cables_to_delete:
            cable_id = cable.id

            await session.delete(cable)

            additional_info_query = select(db_models.AdditionalInfo).where(
                db_models.AdditionalInfo.element_id == cable_id)
            additional_info_result = await session.execute(additional_info_query)
            additional_info_to_delete = additional_info_result.scalars().all()

            await asyncio.gather(*(session.delete(item) for item in
additional_info_to_delete))

        await session.commit()
        return True



async def update_cable(cable_id, **kwargs):
    async with db.AsyncSession() as session:
        async with session.begin():
            query = select(db_models.Cables).where(db_models.Cables.id ==
cable_id)
            result = await session.execute(query)
            cable_to_update = result.scalar_one_or_none()

            if not cable_to_update:
                raise ValueError(f"Запись с id {cable_id} не найдена.")
```

```
        state_fields = {
            'tension_bends_state',
            'grounding_rail_state',
            'equipment_connection_state',
        }

        updates = {}
        for field, value in kwargs.items():
            if value:
                updates[field] = await get_object_state(value) if field in state_fields
else value

        for attr, val in updates.items():
            setattr(cable_to_update, attr, val)

        if "additional_info" in kwargs and kwargs["additional_info"]:
            for item in kwargs["additional_info"]:
                additional_id = item.get('id')
                additional_name = item.get('name')
                additional_field_name = item.get('field_name')
                additional_field_state = item.get('field_state')
                additional_field_description = item.get('field_description')

                additional_info_query = select(db_models.AdditionalInfo).where(
                    db_models.AdditionalInfo.id == additional_id)
                additional_info_result = await
session.execute(additional_info_query)
```

```
            additional_info_update =
additional_info_result.scalar_one_or_none()

            if additional_name:
                additional_info_update.name = additional_name
            if additional_field_name:
                additional_info_update.field_name = additional_field_name
            if additional_field_state:
                additional_info_update.field_state = await
get_object_state(additional_field_state)
            if additional_field_description:
                additional_info_update.additional_field_description =
additional_field_description

        await session.commit()
        return True


async def add_well(well_id, junction_id, name, hatch_state, hatch_description,
trash_state,
                   trash_description, water_state, water_description, additional_info,
                   ):
    if not well_id:
        well_id: str = await cuid2_generator()

    hatch_state = await get_object_state(hatch_state)
    trash_state = await get_object_state(trash_state)
    water_state = await get_object_state(water_state)
```

```
new_well = db_models.Wells(
    id=well_id,
    junction_id=junction_id,
    name=name,
    hatch_state=hatch_state,
    hatch_description=hatch_description,
    trash_state=trash_state,
    trash_description=trash_description,
    water_state=water_state,
    water_description=water_description
)

async with db.AsyncSession() as session:
    session.add(new_well)
    await session.commit()
    await session.refresh(new_well)

    if additional_info:
        for item in additional_info:
            additional_id: str = await cuid2_generator()
            junction_id: str = junction_id
            element_id = new_well.id
            name: str = item.name
            category = db_models.AdditionalInfoCategory.WELLS
            field_name: str = item.field_name

            field_state = await get_object_state(item.field_state)

            field_description = item.field_description
```

```
            new_additional_info = db_models.AdditionalInfo(
                id=additional_id,
                junction_id=junction_id,
                element_id=element_id,
                name=name,
                category=category,
                field_name=field_name,
                field_state=field_state,
                field_description=field_description,
            )

            session.add(new_additional_info)
        await session.commit()

    return new_well


async def get_wells_for_junction(junction_id: str):
    """Получить все колодцы для осмотра"""

    async with db.AsyncSession() as session:
        query = select(db_models.Wells).where(db_models.Wells.junction_id ==
junction_id)
        result = await session.execute(query)
        wells = result.scalars().all()

        wells_result = (
```

```
        [{c.name: getattr(well, c.name) for c in well.__table__.columns} for well
in
         wells])

    for well in wells_result:
        well_id = well.get('id')
        query =
select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.element_id
== well_id,
                                               db_models.AdditionalInfo.category ==
db_models.AdditionalInfoCategory.WELLS
                                               )

        additional_data = await session.execute(query)

        if additional_data:
            additional_data = additional_data.scalars().all()
            well['additional_info'] = [
                {c.name: getattr(add_data, c.name) for c in
add_data.__table__.columns}
                for add_data in additional_data]

    return wells_result


async def get_well(well_id: str):
    """Получить запись 'Колодец кабельной канализации' по ее id"""

    async with db.AsyncSession() as session:
```

```
    query = select(db_models.Wells).where(db_models.Wells.id == well_id)
    result = await session.execute(query)
    well = result.scalar_one_or_none()

    if not well:
        raise ValueError(f"Запись с id {well_id} не найдена.")

    well_result = {c.name: getattr(well, c.name) for c in
well.__table__.columns}

    query =
select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.element_id
== well_id,
                                        db_models.AdditionalInfo.category ==
db_models.AdditionalInfoCategory.WELLS
                                        )

    additional_data = await session.execute(query)

    if additional_data:
        additional_data = additional_data.scalars().all()
        well_result['additional_info'] = [
            {c.name: getattr(add_data, c.name) for c in
add_data.__table__.columns}
            for add_data in additional_data]

    return well_result
```

```
async def delete_well(well_id: str):
    """Удаление колодца по его id"""

    async with db.AsyncSession() as session:
        async with session.begin():
            query = select(db_models.Wells).where(db_models.Wells.id == well_id)
            result = await session.execute(query)
            well_to_delete = result.scalar_one_or_none()

            additional_data_query = select(db_models.AdditionalInfo).where(
                db_models.AdditionalInfo.element_id == well_id)
            additional_data_result = await session.execute(additional_data_query)
            additional_data_to_delete = additional_data_result.scalars().all()

            if not well_to_delete:
                raise ValueError(f"Запись с id {well_id} не найдена.")

            await session.delete(well_to_delete)
            if additional_data_to_delete:
                for item_to_delete in additional_data_to_delete:
                    await session.delete(item_to_delete)
            await session.commit()
            return True


async def delete_wells_for_junction(junction_id: str):
    """Удаление всех колодцев привязанных к осмотру"""

    async with db.AsyncSession() as session:
```

```
        async with session.begin():
            query = select(db_models.Wells).where(db_models.Wells.junction_id ==
junction_id)
            result = await session.execute(query)
            wells_to_delete = result.scalars().all()

            for well in wells_to_delete:
                well_id = well.id

                await session.delete(well)

                additional_info_query = select(db_models.AdditionalInfo).where(
                    db_models.AdditionalInfo.element_id == well_id)
                additional_info_result = await session.execute(additional_info_query)
                additional_info_to_delete = additional_info_result.scalars().all()

                await asyncio.gather(*(session.delete(item) for item in
additional_info_to_delete))

            await session.commit()
            return True


async def update_well(well_id, **kwargs):
    async with db.AsyncSession() as session:
        async with session.begin():
            query = select(db_models.Wells).where(db_models.Wells.id == well_id)
            result = await session.execute(query)
            cable_to_update = result.scalar_one_or_none()
```

```
        if not cable_to_update:
            raise ValueError(f"Запись с id {well_id} не найдена.")

        state_fields = {
            'hatch_state',
            'trash_state',
            'water_state',
        }

        updates = {}
        for field, value in kwargs.items():
            if value:
                updates[field] = await get_object_state(value) if field in state_fields
else value

        for attr, val in updates.items():
            setattr(cable_to_update, attr, val)

        if "additional_info" in kwargs and kwargs["additional_info"]:
            for item in kwargs["additional_info"]:
                additional_id = item.get('id')
                additional_name = item.get('name')
                additional_field_name = item.get('field_name')
                additional_field_state = item.get('field_state')
                additional_field_description = item.get('field_description')

                additional_info_query = select(db_models.AdditionalInfo).where(
                    db_models.AdditionalInfo.id == additional_id)
```

```
            additional_info_result = await
session.execute(additional_info_query)
            additional_info_update =
additional_info_result.scalar_one_or_none()

            if additional_name:
                additional_info_update.name = additional_name
            if additional_field_name:
                additional_info_update.field_name = additional_field_name
            if additional_field_state:
                additional_info_update.field_state = await
get_object_state(additional_field_state)
            if additional_field_description:
                additional_info_update.additional_field_description =
additional_field_description

        await session.commit()
        return True


async def add_traffic_lights(traffic_lights_id, junction_id, name,
functionality_state, functionality_description,
                             broken_lamps_state, broken_lamps_description,
                             broken_leds_state, broken_leds_description,
                             dirt_outside_state, dirt_outside_description,
                             dirt_optical_state, dirt_optical_description,
                             peak_state, peak_description,
                             design_position_state, design_position_description,
                             visibility_100m_state, visibility_100m_description,
```

```
                    mode_state, mode_description,
                    manual_mode_state, manual_mode_description,
                    fastening_reliability_state, fastening_reliability_description,
                    additional_info
                    ):
    if not traffic_lights_id:
        traffic_lights_id: str = await cuid2_generator()

    functionality_state = await get_object_state(functionality_state)
    broken_lamps_state = await get_object_state(broken_lamps_state)
    broken_leds_state = await get_object_state(broken_leds_state)
    dirt_outside_state = await get_object_state(dirt_outside_state)
    dirt_optical_state = await get_object_state(dirt_optical_state)
    peak_state = await get_object_state(peak_state)
    design_position_state = await get_object_state(design_position_state)
    visibility_100m_state = await get_object_state(visibility_100m_state)
    # mode_state = db_models.TrafficLightMode(mode_state) if mode_state else
None  # режим работы, отдельный enum
    manual_mode_state = await get_object_state(manual_mode_state)
    fastening_reliability_state = await get_object_state(fastening_reliability_state)

    new_traffic_light = db_models.TrafficLights(
        id=traffic_lights_id,
        junction_id=junction_id,
        name=name,
        functionality_state=functionality_state,
        functionality_description=functionality_description,
        broken_lamps_state=broken_lamps_state,
        broken_lamps_description=broken_lamps_description,
```

```
    broken_leds_state=broken_leds_state,
    broken_leds_description=broken_leds_description,
    dirt_outside_state=dirt_outside_state,
    dirt_outside_description=dirt_outside_description,
    dirt_optical_state=dirt_optical_state,
    dirt_optical_description=dirt_optical_description,
    peak_state=peak_state,
    peak_description=peak_description,
    design_position_state=design_position_state,
    design_position_description=design_position_description,
    visibility_100m_state=visibility_100m_state,
    visibility_100m_description=visibility_100m_description,
    mode_state=mode_state,
    mode_description=mode_description,
    manual_mode_state=manual_mode_state,
    manual_mode_description=manual_mode_description,
    fastening_reliability_state=fastening_reliability_state,
    fastening_reliability_description=fastening_reliability_description,
)

async with db.AsyncSession() as session:
    session.add(new_traffic_light)
    await session.commit()
    await session.refresh(new_traffic_light)

    if additional_info:
        for item in additional_info:
            additional_id: str = await cuid2_generator()
            junction_id: str = junction_id
```

```
            element_id = new_traffic_light.id
            name: str = item.name
            category = db_models.AdditionalInfoCategory.TRAFFIC_LIGHT
            field_name: str = item.field_name

            field_state = await get_object_state(item.field_state)

            field_description = item.field_description

            new_additional_info = db_models.AdditionalInfo(
                id=additional_id,
                junction_id=junction_id,
                element_id=element_id,
                name=name,
                category=category,
                field_name=field_name,
                field_state=field_state,
                field_description=field_description,
            )

            session.add(new_additional_info)
        await session.commit()

    return new_traffic_light


async def get_traffic_lights_for_junction(junction_id: str):
    """Получить все светофоры для осмотра"""
```

```
    async with db.AsyncSession() as session:
        query =
select(db_models.TrafficLights).where(db_models.TrafficLights.junction_id ==
junction_id)
        result = await session.execute(query)
        traffic_lights = result.scalars().all()

        traffic_lights_result = (
            [{c.name: getattr(traffic_light, c.name) for c in
traffic_light.__table__.columns} for traffic_light in
             traffic_lights])

        for traffic_lights in traffic_lights_result:
            traffic_lights_id = traffic_lights.get('id')
            query =
select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.element_id
== traffic_lights_id,
                                                   db_models.AdditionalInfo.category ==
db_models.AdditionalInfoCategory.TRAFFIC_LIGHT
                                                   )

            additional_data = await session.execute(query)

            if additional_data:
                additional_data = additional_data.scalars().all()
                traffic_lights['additional_info'] = [
                    {c.name: getattr(add_data, c.name) for c in
add_data.__table__.columns}
                    for add_data in additional_data]
```

```
    return traffic_lights_result


async def get_traffic_light(traffic_light_id: str):
    """Получить запись 'Светофор' по ее id"""

    async with db.AsyncSession() as session:
        query = select(db_models.TrafficLights).where(db_models.TrafficLights.id
== traffic_light_id)
        result = await session.execute(query)
        traffic_light = result.scalar_one_or_none()

        if not traffic_light:
            raise ValueError(f"Запись с id {traffic_light_id} не найдена.")

        traffic_light_result = {c.name: getattr(traffic_light, c.name) for c in
traffic_light.__table__.columns}

        query =
select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.element_id
== traffic_light_id,
                                               db_models.AdditionalInfo.category ==
db_models.AdditionalInfoCategory.TRAFFIC_LIGHT
                                               )

        additional_data = await session.execute(query)

        if additional_data:
```

```
            additional_data = additional_data.scalars().all()
            traffic_light_result['additional_info'] = [
                {c.name: getattr(add_data, c.name) for c in
add_data.__table__.columns}
                for add_data in additional_data]

        return traffic_light_result



async def delete_traffic_light(traffic_light_id: str):
    """Удаление светофора по его id"""

    async with db.AsyncSession() as session:
        async with session.begin():
            query =
select(db_models.TrafficLights).where(db_models.TrafficLights.id ==
traffic_light_id)
            result = await session.execute(query)
            traffic_light_to_delete = result.scalar_one_or_none()

            additional_data_query = select(db_models.AdditionalInfo).where(
                db_models.AdditionalInfo.element_id == traffic_light_id)
            additional_data_result = await session.execute(additional_data_query)
            additional_data_to_delete = additional_data_result.scalars().all()

            if not traffic_light_to_delete:
                raise ValueError(f"Запись с id {traffic_light_id} не найдена.")

            await session.delete(traffic_light_to_delete)
```

```
            if additional_data_to_delete:
                for item_to_delete in additional_data_to_delete:
                    await session.delete(item_to_delete)
            await session.commit()
            return True


async def delete_traffic_lights_for_junction(junction_id: str):
    """Удаление всех колодцев привязанных к осмотру"""

    async with db.AsyncSession() as session:
        async with session.begin():
            query =
select(db_models.TrafficLights).where(db_models.TrafficLights.junction_id ==
junction_id)
            result = await session.execute(query)
            traffic_lights_to_delete = result.scalars().all()

            for traffic_light in traffic_lights_to_delete:
                traffic_light_id = traffic_light.id

                await session.delete(traffic_light)

                additional_info_query = select(db_models.AdditionalInfo).where(
                    db_models.AdditionalInfo.element_id == traffic_light_id)
                additional_info_result = await session.execute(additional_info_query)
                additional_info_to_delete = additional_info_result.scalars().all()
```

```
            await asyncio.gather(*(session.delete(item) for item in
additional_info_to_delete))

        await session.commit()
        return True


async def update_traffic_light(traffic_light_id, **kwargs):
    async with db.AsyncSession() as session:
        async with session.begin():
            query =
select(db_models.TrafficLights).where(db_models.TrafficLights.id ==
traffic_light_id)
            result = await session.execute(query)
            traffic_light_to_update = result.scalar_one_or_none()

            if not traffic_light_to_update:
                raise ValueError(f"Запись с id {traffic_light_id} не найдена.")

            state_fields = {
                'functionality_state',
                'broken_lamps_state',
                'broken_leds_state',
                'dirt_outside_state',
                'dirt_optical_state',
                'peak_state',
                'design_position_state',
                'visibility_100m_state',
                'manual_mode_state',
```

```
            'fastening_reliability_state',
        }

        # if 'mode_state' in kwargs:
        #     mode_state_value = kwargs['mode_state']
        #     traffic_light_to_update.mode_state =
db_models.TrafficLightMode(mode_state_value).name
        #     kwargs.pop('mode_state')

        updates = {}
        for field, value in kwargs.items():

            if value:
                updates[field] = await get_object_state(value) if field in state_fields
else value

        for attr, val in updates.items():
            setattr(traffic_light_to_update, attr, val)

        if "additional_info" in kwargs and kwargs["additional_info"]:
            for item in kwargs["additional_info"]:
                additional_id = item.get('id')
                additional_name = item.get('name')
                additional_field_name = item.get('field_name')
                additional_field_state = item.get('field_state')
                additional_field_description = item.get('field_description')

                additional_info_query = select(db_models.AdditionalInfo).where(
                    db_models.AdditionalInfo.id == additional_id)
```

```
            additional_info_result = await
session.execute(additional_info_query)
            additional_info_update =
additional_info_result.scalar_one_or_none()

            if additional_name:
                additional_info_update.name = additional_name
            if additional_field_name:
                additional_info_update.field_name = additional_field_name
            if additional_field_state:
                additional_info_update.field_state = await
get_object_state(additional_field_state)
            if additional_field_description:
                additional_info_update.additional_field_description =
additional_field_description

        await session.commit()
        return True


async def add_tvp(tvp_id, junction_id, name, dirt_state, dirt_description,
            inscriptions_advertisement_state,
inscriptions_advertisement_description,
            broken_state, broken_description, functionality_state,
            functionality_description, waiting_state, waiting_description,
            fastening_reliability_state, fastening_reliability_description,
            additional_info
            ):
    if not tvp_id:
```

```
        tvp_id: str = await cuid2_generator()

    dirt_state = await get_object_state(dirt_state)
    inscriptions_advertisement_state = await
get_object_state(inscriptions_advertisement_state)
    broken_state = await get_object_state(broken_state)
    functionality_state = await get_object_state(functionality_state)
    waiting_state = await get_object_state(waiting_state)
    fastening_reliability_state = await get_object_state(fastening_reliability_state)

    new_tvp = db_models.TVP(
        id=tvp_id,
        junction_id=junction_id,
        name=name,
        dirt_state=dirt_state,
        dirt_description=dirt_description,
        inscriptions_advertisement_state=inscriptions_advertisement_state,

inscriptions_advertisement_description=inscriptions_advertisement_description,
        broken_state=broken_state,
        broken_description=broken_description,
        functionality_state=functionality_state,
        functionality_description=functionality_description,
        waiting_state=waiting_state,
        waiting_description=waiting_description,
        fastening_reliability_state=fastening_reliability_state,
        fastening_reliability_description=fastening_reliability_description,
    )
```

```
async with db.AsyncSession() as session:
    session.add(new_tvp)
    await session.commit()
    await session.refresh(new_tvp)

    if additional_info:
        for item in additional_info:
            additional_id: str = await cuid2_generator()
            junction_id: str = junction_id
            element_id = new_tvp.id
            name: str = item.name
            category = db_models.AdditionalInfoCategory.TVP
            field_name: str = item.field_name

            field_state = await get_object_state(item.field_state)

            field_description = item.field_description

            new_additional_info = db_models.AdditionalInfo(
                id=additional_id,
                junction_id=junction_id,
                element_id=element_id,
                name=name,
                category=category,
                field_name=field_name,
                field_state=field_state,
                field_description=field_description,
            )
```

```
            session.add(new_additional_info)
        await session.commit()

    return new_tvp


async def get_tvp_for_junction(junction_id: str):
    """Получить все 'ТВП (табло вызывное пешеходное)' для осмотра"""

    async with db.AsyncSession() as session:
        query = select(db_models.TVP).where(db_models.TVP.junction_id ==
junction_id)
        result = await session.execute(query)
        all_tvp = result.scalars().all()

        tvp_result = (
            [{c.name: getattr(tvp, c.name) for c in tvp.__table__.columns} for tvp in
             all_tvp])

        for tvp in tvp_result:
            tvp_id = tvp.get('id')
            query =
select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.element_id
== tvp_id,
                                                db_models.AdditionalInfo.category ==
db_models.AdditionalInfoCategory.TVP
                                                )

            additional_data = await session.execute(query)
```

```
        if additional_data:
            additional_data = additional_data.scalars().all()
            tvp['additional_info'] = [
                {c.name: getattr(add_data, c.name) for c in
add_data.__table__.columns}
                for add_data in additional_data]

    return tvp_result


async def get_one_tvp(tvp_id: str):
    """Получить запись 'Светофор' по ее id"""

    async with db.AsyncSession() as session:
        query = select(db_models.TVP).where(db_models.TVP.id == tvp_id)
        result = await session.execute(query)
        tvp = result.scalar_one_or_none()

        if not tvp:
            raise ValueError(f"Запись с id {tvp_id} не найдена.")

        tvp_result = {c.name: getattr(tvp, c.name) for c in tvp.__table__.columns}

        query =
select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.element_id
== tvp_id,
                                               db_models.AdditionalInfo.category ==
db_models.AdditionalInfoCategory.TVP
```

```
)

        additional_data = await session.execute(query)

        if additional_data:
            additional_data = additional_data.scalars().all()
            tvp_result['additional_info'] = [
                {c.name: getattr(add_data, c.name) for c in
add_data.__table__.columns}
                for add_data in additional_data]

        return tvp_result


async def delete_tvp(tvp_id: str):
    """Удаление светофора по его id"""

    async with db.AsyncSession() as session:
        async with session.begin():
            query = select(db_models.TVP).where(db_models.TVP.id == tvp_id)
            result = await session.execute(query)
            traffic_light_to_delete = result.scalar_one_or_none()

            additional_data_query = select(db_models.AdditionalInfo).where(
                db_models.AdditionalInfo.element_id == tvp_id)
            additional_data_result = await session.execute(additional_data_query)
            additional_data_to_delete = additional_data_result.scalars().all()

            if not traffic_light_to_delete:
```

```
            raise ValueError(f"Запись с id {tvp_id} не найдена.")

        await session.delete(traffic_light_to_delete)
        if additional_data_to_delete:
            for item_to_delete in additional_data_to_delete:
                await session.delete(item_to_delete)
        await session.commit()
        return True


async def delete_all_tvp_for_junction(junction_id: str):
    """Удаление всех 'ТВП (табло вызывное пешеходное)' привязанных к осмотру"""

    async with db.AsyncSession() as session:
        async with session.begin():
            query = select(db_models.TVP).where(db_models.TVP.junction_id == junction_id)
            result = await session.execute(query)
            all_tvp_to_delete = result.scalars().all()

            for tvp in all_tvp_to_delete:
                tvp_id = tvp.id

                await session.delete(tvp)

                additional_info_query = select(db_models.AdditionalInfo).where(
                    db_models.AdditionalInfo.element_id == tvp_id)
                additional_info_result = await session.execute(additional_info_query)
```

```
            additional_info_to_delete = additional_info_result.scalars().all()

            await asyncio.gather(*(session.delete(item) for item in
additional_info_to_delete))

        await session.commit()
        return True



async def update_tvp(tvp_id, **kwargs):
    async with db.AsyncSession() as session:
        async with session.begin():
            query = select(db_models.TVP).where(db_models.TVP.id == tvp_id)
            result = await session.execute(query)
            traffic_light_to_update = result.scalar_one_or_none()

            if not traffic_light_to_update:
                raise ValueError(f"Запись с id {tvp_id} не найдена.")

            state_fields = {
                'dirt_state',
                'inscriptions_advertisement_state',
                'broken_state',
                'functionality_state',
                'waiting_state',
                'fastening_reliability_state',
            }

            updates = {}
```

```
        for field, value in kwargs.items():
            if value:
                updates[field] = await get_object_state(value) if field in state_fields
else value

        for attr, val in updates.items():
            setattr(traffic_light_to_update, attr, val)

        if "additional_info" in kwargs and kwargs["additional_info"]:
            for item in kwargs["additional_info"]:
                additional_id = item.get('id')
                additional_name = item.get('name')
                additional_field_name = item.get('field_name')
                additional_field_state = item.get('field_state')
                additional_field_description = item.get('field_description')

                additional_info_query = select(db_models.AdditionalInfo).where(
                    db_models.AdditionalInfo.id == additional_id)
                additional_info_result = await
session.execute(additional_info_query)
                additional_info_update =
additional_info_result.scalar_one_or_none()

                if additional_name:
                    additional_info_update.name = additional_name
                if additional_field_name:
                    additional_info_update.field_name = additional_field_name
                if additional_field_state:
```

```
                additional_info_update.field_state = await
get_object_state(additional_field_state)
            if additional_field_description:
                additional_info_update.additional_field_description =
additional_field_description

        await session.commit()
        return True


async def add_accessible_devices(accessible_devices_id, junction_id, name,
uzsp_state, uzsp_description,
                                 orientation_signal_zone_state,
orientation_signal_zone_description,
                                 orientation_signal_duration_state,
orientation_signal_duration_description,
                                 crossing_signal_zone_state,
crossing_signal_zone_description,
                                 crossing_signal_duration_state,
crossing_signal_duration_description,
                                 uzsp_mounting_height_state,
uzsp_mounting_height_description,
                                 tactile_alarm_state, tactile_alarm_description,
                                 tactile_alarm_duration_state,
tactile_alarm_duration_description,
                                 tactile_alarm_mounting_height_state,
tactile_alarm_mounting_height_description,
                                 signal_matching_state, signal_matching_description,
                                 fastening_reliability_state, fastening_reliability_description,
```

```
                    additional_info
                    ):
    if not accessible_devices_id:
        accessible_devices_id: str = await cuid2_generator()

    uzsp_state = await get_object_state(uzsp_state)
    orientation_signal_zone_state = await
get_object_state(orientation_signal_zone_state)
    orientation_signal_duration_state = await
get_object_state(orientation_signal_duration_state)
    crossing_signal_zone_state = await
get_object_state(crossing_signal_zone_state)
    crossing_signal_duration_state = await
get_object_state(crossing_signal_duration_state)
    uzsp_mounting_height_state = await
get_object_state(uzsp_mounting_height_state)
    tactile_alarm_state = await get_object_state(tactile_alarm_state)
    tactile_alarm_duration_state = await
get_object_state(tactile_alarm_duration_state)
    tactile_alarm_mounting_height_state = await
get_object_state(tactile_alarm_mounting_height_state)
    signal_matching_state = await get_object_state(signal_matching_state)
    fastening_reliability_state = await get_object_state(fastening_reliability_state)

    new_accessible_devices = db_models.AccessibleSignalingDevices(
        id=accessible_devices_id,
        junction_id=junction_id,
        name=name,
        uzsp_state=uzsp_state,
```

```
        uzsp_description=uzsp_description,
        orientation_signal_zone_state=orientation_signal_zone_state,
        orientation_signal_zone_description=orientation_signal_zone_description,
        orientation_signal_duration_state=orientation_signal_duration_state,

orientation_signal_duration_description=orientation_signal_duration_description,
        crossing_signal_zone_state=crossing_signal_zone_state,
        crossing_signal_zone_description=crossing_signal_zone_description,
        crossing_signal_duration_state=crossing_signal_duration_state,
        crossing_signal_duration_description=crossing_signal_duration_description,
        uzsp_mounting_height_state=uzsp_mounting_height_state,
        uzsp_mounting_height_description=uzsp_mounting_height_description,
        tactile_alarm_state=tactile_alarm_state,
        tactile_alarm_description=tactile_alarm_description,
        tactile_alarm_duration_state=tactile_alarm_duration_state,
        tactile_alarm_duration_description=tactile_alarm_duration_description,
        tactile_alarm_mounting_height_state=tactile_alarm_mounting_height_state,

tactile_alarm_mounting_height_description=tactile_alarm_mounting_height_desc
ription,
        signal_matching_state=signal_matching_state,
        signal_matching_description=signal_matching_description,
        fastening_reliability_state=fastening_reliability_state,
        fastening_reliability_description=fastening_reliability_description,
    )

    async with db.AsyncSession() as session:
        session.add(new_accessible_devices)
        await session.commit()
```

```
        await session.refresh(new_accessible_devices)

        if additional_info:
            for item in additional_info:
                additional_id: str = await cuid2_generator()
                junction_id: str = junction_id
                element_id = new_accessible_devices.id
                name: str = item.name
                category =
db_models.AdditionalInfoCategory.ACCESSIBLE_SIGNALING_DEVICES
                field_name: str = item.field_name

                field_state = await get_object_state(item.field_state)

                field_description = item.field_description

                new_additional_info = db_models.AdditionalInfo(
                    id=additional_id,
                    junction_id=junction_id,
                    element_id=element_id,
                    name=name,
                    category=category,
                    field_name=field_name,
                    field_state=field_state,
                    field_description=field_description,
                )

                session.add(new_additional_info)
            await session.commit()
```

```
    return new_accessible_devices


async def get_accessible_devices_for_junction(junction_id: str):
    """Получить все записи 'Дополнительное оборудование для лиц с нарушениями органов чувств' для осмотра"""

    async with db.AsyncSession() as session:
        query = select(db_models.AccessibleSignalingDevices).where(
            db_models.AccessibleSignalingDevices.junction_id == junction_id)
        result = await session.execute(query)
        accessible_devices = result.scalars().all()

        traffic_lights_result = (
            [{c.name: getattr(device, c.name) for c in device.__table__.columns} for device in
             accessible_devices])

        for device in traffic_lights_result:
            device_id = device.get('id')
            query = select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.element_id == device_id,
                                                           db_models.AdditionalInfo.category == db_models.AdditionalInfoCategory.ACCESSIBLE_SIGNALING_DEVICES
                                                           )

            additional_data = await session.execute(query)
```

```
            if additional_data:
                additional_data = additional_data.scalars().all()
                device['additional_info'] = [
                    {c.name: getattr(add_data, c.name) for c in
add_data.__table__.columns}
                    for add_data in additional_data]

        return traffic_lights_result


async def get_accessible_device(accessible_device_id: str):
    """Получить запись 'Дополнительное оборудование для лиц с
нарушениями органов чувств' по ее id"""

    async with db.AsyncSession() as session:
        query = select(db_models.AccessibleSignalingDevices).where(
            db_models.AccessibleSignalingDevices.id == accessible_device_id)
        result = await session.execute(query)
        accessible_device = result.scalar_one_or_none()

        if not accessible_device:
            raise ValueError(f"Запись с id {accessible_device_id} не найдена.")

        accessible_device_result = {c.name: getattr(accessible_device, c.name) for c
in
                                    accessible_device.__table__.columns}
```

```
        query =
select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.element_id
== accessible_device_id,
                                               db_models.AdditionalInfo.category ==
db_models.AdditionalInfoCategory.ACCESSIBLE_SIGNALING_DEVICES
                                               )

        additional_data = await session.execute(query)

        if additional_data:
            additional_data = additional_data.scalars().all()
            accessible_device_result['additional_info'] = [
                {c.name: getattr(add_data, c.name) for c in
add_data.__table__.columns}
                for add_data in additional_data]

        return accessible_device_result



async def delete_accessible_device(accessible_device_id: str):
    """Удаление записи 'Дополнительное оборудование для лиц с
нарушениями органов чувств' по ее id"""

    async with db.AsyncSession() as session:
        async with session.begin():
            query = select(db_models.AccessibleSignalingDevices).where(
                db_models.AccessibleSignalingDevices.id == accessible_device_id)
            result = await session.execute(query)
            accessible_device_to_delete = result.scalar_one_or_none()
```

```
            additional_data_query = select(db_models.AdditionalInfo).where(
                db_models.AdditionalInfo.element_id == accessible_device_id)
            additional_data_result = await session.execute(additional_data_query)
            additional_data_to_delete = additional_data_result.scalars().all()

            if not accessible_device_to_delete:
                raise ValueError(f"Запись с id {accessible_device_id} не найдена.")

            await session.delete(accessible_device_to_delete)
            if additional_data_to_delete:
                for item_to_delete in additional_data_to_delete:
                    await session.delete(item_to_delete)
            await session.commit()
            return True


async def delete_accessible_devices_for_junction(junction_id: str):
    """Удаление всех записей 'Дополнительное оборудование для лиц с нарушениями органов чувств' привязанных к осмотру"""

    async with db.AsyncSession() as session:
        async with session.begin():
            query = select(db_models.AccessibleSignalingDevices).where(
                db_models.AccessibleSignalingDevices.junction_id == junction_id)
            result = await session.execute(query)
            accessible_devices_to_delete = result.scalars().all()

            for accessible_device in accessible_devices_to_delete:
```

```
            accessible_device_id = accessible_device.id

            await session.delete(accessible_device)

            additional_info_query = select(db_models.AdditionalInfo).where(
                db_models.AdditionalInfo.element_id == accessible_device_id)
            additional_info_result = await session.execute(additional_info_query)
            additional_info_to_delete = additional_info_result.scalars().all()

            await asyncio.gather(*(session.delete(item) for item in
additional_info_to_delete))

        await session.commit()
        return True


async def update_accessible_device(accessible_device_id, **kwargs):
    async with db.AsyncSession() as session:
        async with session.begin():
            query = select(db_models.AccessibleSignalingDevices).where(
                db_models.AccessibleSignalingDevices.id == accessible_device_id)
            result = await session.execute(query)
            accessible_device_to_update = result.scalar_one_or_none()

            if not accessible_device_to_update:
                raise ValueError(f"Запись с id {accessible_device_id} не найдена.")

            state_fields = {
                'uzsp_state',
```

```
            'orientation_signal_zone_state',
            'orientation_signal_duration_state',
            'crossing_signal_zone_state',
            'crossing_signal_duration_state',
            'uzsp_mounting_height_state',
            'tactile_alarm_state',
            'tactile_alarm_duration_state',
            'tactile_alarm_mounting_height_state',
            'signal_matching_state',
            'fastening_reliability_state',
        }

        updates = {}
        for field, value in kwargs.items():
            if value:
                updates[field] = await get_object_state(value) if field in state_fields
else value

        for attr, val in updates.items():
            setattr(accessible_device_to_update, attr, val)

        if "additional_info" in kwargs and kwargs["additional_info"]:
            for item in kwargs["additional_info"]:
                additional_id = item.get('id')
                additional_name = item.get('name')
                additional_field_name = item.get('field_name')
                additional_field_state = item.get('field_state')
                additional_field_description = item.get('field_description')
```

```
            additional_info_query = select(db_models.AdditionalInfo).where(
                db_models.AdditionalInfo.id == additional_id)
            additional_info_result = await
session.execute(additional_info_query)
            additional_info_update =
additional_info_result.scalar_one_or_none()

            if additional_name:
                additional_info_update.name = additional_name
            if additional_field_name:
                additional_info_update.field_name = additional_field_name
            if additional_field_state:
                additional_info_update.field_state = await
get_object_state(additional_field_state)
            if additional_field_description:
                additional_info_update.additional_field_description =
additional_field_description

        await session.commit()
        return True


async def add_road_marking(road_marking_id, junction_id, name,
markings_state, markings_description, additional_info):
    if not road_marking_id:
        road_marking_id: str = await cuid2_generator()

    markings_state = await get_object_state(markings_state)
```

```
new_road_marking = db_models.RoadMarkings(
    id=road_marking_id,
    junction_id=junction_id,
    name=name,
    markings_state=markings_state,
    markings_description=markings_description,
)

async with db.AsyncSession() as session:
    session.add(new_road_marking)
    await session.commit()
    await session.refresh(new_road_marking)

    if additional_info:
        for item in additional_info:
            additional_id: str = await cuid2_generator()
            junction_id: str = junction_id
            element_id = new_road_marking.id
            name: str = item.name
            category = db_models.AdditionalInfoCategory.ROAD_MARKINGS
            field_name: str = item.field_name

            field_state = await get_object_state(item.field_state)

            field_description = item.field_description

            new_additional_info = db_models.AdditionalInfo(
                id=additional_id,
                junction_id=junction_id,
```

```
                element_id=element_id,
                name=name,
                category=category,
                field_name=field_name,
                field_state=field_state,
                field_description=field_description,
            )

            session.add(new_additional_info)
        await session.commit()

    return new_road_marking


async def get_road_markings_for_junction(junction_id: str):
    """Получить все записи 'Дорожная разметка' для осмотра"""

    async with db.AsyncSession() as session:
        query =
select(db_models.RoadMarkings).where(db_models.RoadMarkings.junction_id
== junction_id)
        result = await session.execute(query)
        road_markings = result.scalars().all()

        road_markings_result = (
            [{c.name: getattr(road_marking, c.name) for c in
road_marking.__table__.columns} for road_marking in
             road_markings])
```

```
        for road_mark in road_markings_result:
            road_mark_id = road_mark.get('id')
            query =
select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.element_id
== road_mark_id,
                                                   db_models.AdditionalInfo.category ==
db_models.AdditionalInfoCategory.ROAD_MARKINGS
                                                   )

            additional_data = await session.execute(query)

            if additional_data:
                additional_data = additional_data.scalars().all()
                road_mark['additional_info'] = [
                    {c.name: getattr(add_data, c.name) for c in
add_data.__table__.columns}
                    for add_data in additional_data]

        return road_markings_result


async def get_road_marking(road_marking_id: str):
    """Получение записи "Дорожная разметка" по ее id"""

    async with db.AsyncSession() as session:
        query =
select(db_models.RoadMarkings).where(db_models.RoadMarkings.id ==
road_marking_id)
        result = await session.execute(query)
```

```
    road_marking = result.scalar_one_or_none()

    if not road_marking:
        raise ValueError(f"Запись с id {road_marking_id} не найдена.")

    road_marking_result = {c.name: getattr(road_marking, c.name) for c in
road_marking.__table__.columns}

    query =
select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.element_id
== road_marking_id,
                                           db_models.AdditionalInfo.category ==
db_models.AdditionalInfoCategory.ROAD_MARKINGS
                                           )

    additional_data = await session.execute(query)

    if additional_data:
        additional_data = additional_data.scalars().all()
        road_marking_result['additional_info'] = [
            {c.name: getattr(add_data, c.name) for c in
add_data.__table__.columns}
            for add_data in additional_data]

    return road_marking_result


async def delete_road_marking(road_marking_id: str):
    """Удаление записи 'Дорожная разметка' по ее id"""
```

```
    async with db.AsyncSession() as session:
        async with session.begin():
            query =
select(db_models.RoadMarkings).where(db_models.RoadMarkings.id ==
road_marking_id)
            result = await session.execute(query)
            road_marking_to_delete = result.scalar_one_or_none()

            additional_data_query = select(db_models.AdditionalInfo).where(
                db_models.AdditionalInfo.element_id == road_marking_id)
            additional_data_result = await session.execute(additional_data_query)
            additional_data_to_delete = additional_data_result.scalars().all()

            if not road_marking_to_delete:
                raise ValueError(f"Запись с id {road_marking_id} не найдена.")

            await session.delete(road_marking_to_delete)
            if additional_data_to_delete:
                for item_to_delete in additional_data_to_delete:
                    await session.delete(item_to_delete)
            await session.commit()
            return True


async def delete_road_markings_for_junction(junction_id: str):
    """Удаление всех записей 'Дорожная разметка' привязанных к осмотру"""

    async with db.AsyncSession() as session:
```

```
        async with session.begin():
            query =
select(db_models.RoadMarkings).where(db_models.RoadMarkings.junction_id
== junction_id)
            result = await session.execute(query)
            road_markings_to_delete = result.scalars().all()

            for road_marking in road_markings_to_delete:
                road_marking_id = road_marking.id

                await session.delete(road_marking)

                additional_info_query = select(db_models.AdditionalInfo).where(
                    db_models.AdditionalInfo.element_id == road_marking_id)
                additional_info_result = await session.execute(additional_info_query)
                additional_info_to_delete = additional_info_result.scalars().all()

                await asyncio.gather(*(session.delete(item) for item in
additional_info_to_delete))

            await session.commit()
            return True



async def update_road_marking(road_marking_id, **kwargs):
    async with db.AsyncSession() as session:
        async with session.begin():
```

```
        query =
select(db_models.RoadMarkings).where(db_models.RoadMarkings.id ==
road_marking_id)
        result = await session.execute(query)
        road_marking_to_update = result.scalar_one_or_none()

        if not road_marking_to_update:
            raise ValueError(f"Запись с id {road_marking_id} не найдена.")

        state_fields = {
            'markings_state',
        }

        updates = {}
        for field, value in kwargs.items():
            if value:
                updates[field] = await get_object_state(value) if field in state_fields
else value

        for attr, val in updates.items():
            setattr(road_marking_to_update, attr, val)

        if "additional_info" in kwargs and kwargs["additional_info"]:
            for item in kwargs["additional_info"]:
                additional_id = item.get('id')
                additional_name = item.get('name')
                additional_field_name = item.get('field_name')
                additional_field_state = item.get('field_state')
                additional_field_description = item.get('field_description')
```

```
            additional_info_query = select(db_models.AdditionalInfo).where(
                db_models.AdditionalInfo.id == additional_id)
            additional_info_result = await
session.execute(additional_info_query)
            additional_info_update =
additional_info_result.scalar_one_or_none()

            if additional_name:
                additional_info_update.name = additional_name
            if additional_field_name:
                additional_info_update.field_name = additional_field_name
            if additional_field_state:
                additional_info_update.field_state = await
get_object_state(additional_field_state)
            if additional_field_description:
                additional_info_update.additional_field_description =
additional_field_description

        await session.commit()
        return True


async def add_power_supply(power_supply_id, junction_id, name,
                           functionality_state, functionality_description,
                           broken_body_state, broken_body_description,
                           broken_panel_state, broken_panel_description,
                           dirt_state, dirt_description,
                           fastening_reliability_state, fastening_reliability_description,
```

```
                additional_info
                ):
if not power_supply_id:
    power_supply_id: str = await cuid2_generator()

functionality_state = await get_object_state(functionality_state)
broken_body_state = await get_object_state(broken_body_state)
broken_panel_state = await get_object_state(broken_panel_state)
dirt_state = await get_object_state(dirt_state)
fastening_reliability_state = await get_object_state(fastening_reliability_state)

new_power_supply = db_models.AutonomsPowerSources(
    id=power_supply_id,
    junction_id=junction_id,
    name=name,
    functionality_state=functionality_state,
    functionality_description=functionality_description,
    broken_body_state=broken_body_state,
    broken_body_description=broken_body_description,
    broken_panel_state=broken_panel_state,
    broken_panel_description=broken_panel_description,
    dirt_state=dirt_state,
    fastening_reliability_state=fastening_reliability_state,
    fastening_reliability_description=fastening_reliability_description,
    dirt_description=dirt_description,
)

async with db.AsyncSession() as session:
    session.add(new_power_supply)
```

```
        await session.commit()
        await session.refresh(new_power_supply)

        if additional_info:
            for item in additional_info:
                additional_id: str = await cuid2_generator()
                junction_id: str = junction_id
                element_id = new_power_supply.id
                name: str = item.name
                category =
db_models.AdditionalInfoCategory.AUTONOMS_POWER_SOURCES
                field_name: str = item.field_name

                field_state = await get_object_state(item.field_state)

                field_description = item.field_description

                new_additional_info = db_models.AdditionalInfo(
                    id=additional_id,
                    junction_id=junction_id,
                    element_id=element_id,
                    name=name,
                    category=category,
                    field_name=field_name,
                    field_state=field_state,
                    field_description=field_description,
                )

                session.add(new_additional_info)
```

```
            await session.commit()

    return new_power_supply


async def get_power_sources_for_junction(junction_id: str):
    """Получить все записи 'Источники автономного питания' для осмотра"""

    async with db.AsyncSession() as session:
        query = select(db_models.AutonomsPowerSources).where(
            db_models.AutonomsPowerSources.junction_id == junction_id)
        result = await session.execute(query)
        power_sources = result.scalars().all()

        power_sources_result = (
            [{c.name: getattr(source, c.name) for c in source.__table__.columns} for
source in
             power_sources])

        for source in power_sources_result:
            source_id = source.get('id')
            query =
select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.element_id
== source_id,
                                                db_models.AdditionalInfo.category ==
db_models.AdditionalInfoCategory.AUTONOMS_POWER_SOURCES
                                                )

            additional_data = await session.execute(query)
```

```
            if additional_data:
                additional_data = additional_data.scalars().all()
                source['additional_info'] = [
                    {c.name: getattr(add_data, c.name) for c in
add_data.__table__.columns}
                    for add_data in additional_data]

        return power_sources_result


async def get_power_source(power_source_id: str):
    """Получить запись 'Получение записи "Источник автономного питания"
по ее id' по ее id"""

    async with db.AsyncSession() as session:
        query =
select(db_models.AutonomsPowerSources).where(db_models.AutonomsPowerS
ources.id == power_source_id)
        result = await session.execute(query)
        power_source = result.scalar_one_or_none()

        if not power_source:
            raise ValueError(f"Запись с id {power_source_id} не найдена.")

        power_source_result = {c.name: getattr(power_source, c.name) for c in
power_source.__table__.columns}
```

```
        query =
select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.element_id
== power_source_id,
                                               db_models.AdditionalInfo.category ==
db_models.AdditionalInfoCategory.AUTONOMS_POWER_SOURCES
                                               )

        additional_data = await session.execute(query)

        if additional_data:
            additional_data = additional_data.scalars().all()
            power_source_result['additional_info'] = [
                {c.name: getattr(add_data, c.name) for c in
add_data.__table__.columns}
                for add_data in additional_data]

        return power_source_result



async def delete_power_source(power_source_id: str):
    """Удаление записи 'Источники автономного питания' по ее id"""

    async with db.AsyncSession() as session:
        async with session.begin():
            query = select(db_models.AutonomsPowerSources).where(
                db_models.AutonomsPowerSources.id == power_source_id)
            result = await session.execute(query)
            power_source_to_delete = result.scalar_one_or_none()
```

```
        additional_data_query = select(db_models.AdditionalInfo).where(
            db_models.AdditionalInfo.element_id == power_source_id)
        additional_data_result = await session.execute(additional_data_query)
        additional_data_to_delete = additional_data_result.scalars().all()

        if not power_source_to_delete:
            raise ValueError(f"Запись с id {power_source_id} не найдена.")

        await session.delete(power_source_to_delete)
        if additional_data_to_delete:
            for item_to_delete in additional_data_to_delete:
                await session.delete(item_to_delete)
        await session.commit()
        return True


async def delete_power_sources_for_junction(junction_id: str):
    """Удаление всех записей 'Источники автономного питания' привязанных к осмотру"""

    async with db.AsyncSession() as session:
        async with session.begin():
            query = select(db_models.AutonomsPowerSources).where(
                db_models.AutonomsPowerSources.junction_id == junction_id)
            result = await session.execute(query)
            power_sources_to_delete = result.scalars().all()

            for source in power_sources_to_delete:
                source_id = source.id
```

```
            await session.delete(source)

            additional_info_query = select(db_models.AdditionalInfo).where(
                db_models.AdditionalInfo.element_id == source_id)
            additional_info_result = await session.execute(additional_info_query)
            additional_info_to_delete = additional_info_result.scalars().all()

            await asyncio.gather(*(session.delete(item) for item in
additional_info_to_delete))

        await session.commit()
        return True


async def update_power_source(power_source_id, **kwargs):
    async with db.AsyncSession() as session:
        async with session.begin():
            query = select(db_models.AutonomsPowerSources).where(
                db_models.AutonomsPowerSources.id == power_source_id)
            result = await session.execute(query)
            power_source_to_update = result.scalar_one_or_none()

            if not power_source_to_update:
                raise ValueError(f"Запись с id {power_source_id} не найдена.")

            state_fields = {
                'functionality_state',
                'broken_body_state',
```

```
            'broken_panel_state',
            'dirt_state',
            'fastening_reliability_state',
        }

        updates = {}
        for field, value in kwargs.items():
            if value:
                updates[field] = await get_object_state(value) if field in state_fields
else value

        for attr, val in updates.items():
            setattr(power_source_to_update, attr, val)

        if "additional_info" in kwargs and kwargs["additional_info"]:
            for item in kwargs["additional_info"]:
                additional_id = item.get('id')
                additional_name = item.get('name')
                additional_field_name = item.get('field_name')
                additional_field_state = item.get('field_state')
                additional_field_description = item.get('field_description')

                additional_info_query = select(db_models.AdditionalInfo).where(
                    db_models.AdditionalInfo.id == additional_id)
                additional_info_result = await
session.execute(additional_info_query)
                additional_info_update =
additional_info_result.scalar_one_or_none()
```

```
            if additional_name:
                additional_info_update.name = additional_name
            if additional_field_name:
                additional_info_update.field_name = additional_field_name
            if additional_field_state:
                additional_info_update.field_state = await
get_object_state(additional_field_state)
            if additional_field_description:
                additional_info_update.additional_field_description =
additional_field_description

        await session.commit()
        return True


async def add_toov(toov_id, junction_id, name,
                   functionality_state, functionality_description,
                   broken_body_state, broken_body_description,
                   dirt_state, dirt_description,
                   indicator_state, indicator_description,
                   fastening_reliability_state, fastening_reliability_description,
                   additional_info
                   ):
    if not toov_id:
        toov_id: str = await cuid2_generator()

    functionality_state = await get_object_state(functionality_state)
    broken_body_state = await get_object_state(broken_body_state)
    dirt_state = await get_object_state(dirt_state)
```

```
indicator_state = await get_object_state(indicator_state)
fastening_reliability_state = await get_object_state(fastening_reliability_state)

new_toov = db_models.TOOV(
    id=toov_id,
    junction_id=junction_id,
    name=name,
    functionality_state=functionality_state,
    functionality_description=functionality_description,
    broken_body_state=broken_body_state,
    broken_body_description=broken_body_description,
    dirt_state=dirt_state,
    dirt_description=dirt_description,
    indicator_state=indicator_state,
    indicator_description=indicator_description,
    fastening_reliability_state=fastening_reliability_state,
    fastening_reliability_description=fastening_reliability_description,
)

async with db.AsyncSession() as session:
    session.add(new_toov)
    await session.commit()
    await session.refresh(new_toov)

    if additional_info:
        for item in additional_info:
            additional_id: str = await cuid2_generator()
            junction_id: str = junction_id
            element_id = new_toov.id
```

```
            name: str = item.name
            category = db_models.AdditionalInfoCategory.TOOV
            field_name: str = item.field_name

            field_state = await get_object_state(item.field_state)

            field_description = item.field_description

            new_additional_info = db_models.AdditionalInfo(
                id=additional_id,
                junction_id=junction_id,
                element_id=element_id,
                name=name,
                category=category,
                field_name=field_name,
                field_state=field_state,
                field_description=field_description,
            )

            session.add(new_additional_info)
        await session.commit()

    return new_toov


async def get_toov_for_junction(junction_id: str):
    """Получить все записи 'TOOB - табло обратного отсчета времени' для
осмотра"""
```

```
    async with db.AsyncSession() as session:
        query = select(db_models.TOOV).where(db_models.TOOV.junction_id ==
junction_id)
        result = await session.execute(query)
        toov = result.scalars().all()

        toov_result = (
            [{c.name: getattr(device, c.name) for c in device.__table__.columns} for
device in
             toov])

        for device in toov_result:
            device_id = device.get('id')
            query =
select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.element_id
== device_id,
                                                db_models.AdditionalInfo.category ==
db_models.AdditionalInfoCategory.TOOV
                                                )

            additional_data = await session.execute(query)

            if additional_data:
                additional_data = additional_data.scalars().all()
                device['additional_info'] = [
                    {c.name: getattr(add_data, c.name) for c in
add_data.__table__.columns}
                    for add_data in additional_data]
```

```
        return toov_result


async def get_toov(toov_id: str):
    """Получить запись 'ТООВ - табло обратного отсчета времени' по ее id"""

    async with db.AsyncSession() as session:
        query = select(db_models.TOOV).where(db_models.TOOV.id == toov_id)
        result = await session.execute(query)
        toov = result.scalar_one_or_none()

        if not toov:
            raise ValueError(f"Запись с id {toov_id} не найдена.")

        toov_result = {c.name: getattr(toov, c.name) for c in
toov.__table__.columns}

        query =
select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.element_id
== toov_id,
                                               db_models.AdditionalInfo.category ==
db_models.AdditionalInfoCategory.TOOV
                                               )

        additional_data = await session.execute(query)

        if additional_data:
            additional_data = additional_data.scalars().all()
            toov_result['additional_info'] = [
```

```
            {c.name: getattr(add_data, c.name) for c in
add_data.__table__.columns}
            for add_data in additional_data]

    return toov_result


async def delete_toov(toov_id: str):
    """Удаление записи 'TOOB - табло обратного отсчета времени' по ее id"""

    async with db.AsyncSession() as session:
        async with session.begin():
            query = select(db_models.TOOV).where(db_models.TOOV.id ==
toov_id)
            result = await session.execute(query)
            toov_to_delete = result.scalar_one_or_none()

            additional_data_query = select(db_models.AdditionalInfo).where(
                db_models.AdditionalInfo.element_id == toov_id)
            additional_data_result = await session.execute(additional_data_query)
            additional_data_to_delete = additional_data_result.scalars().all()

            if not toov_to_delete:
                raise ValueError(f"Запись с id {toov_id} не найдена.")

            await session.delete(toov_to_delete)
            if additional_data_to_delete:
                for item_to_delete in additional_data_to_delete:
                    await session.delete(item_to_delete)
```

```
            await session.commit()
            return True


async def delete_toov_for_junction(junction_id: str):
    """Удаление всех записей 'ТООВ - табло обратного отсчета времени' привязанных к осмотру"""

    async with db.AsyncSession() as session:
        async with session.begin():
            query = select(db_models.TOOV).where(db_models.TOOV.junction_id == junction_id)
            result = await session.execute(query)
            toov_to_delete = result.scalars().all()

            for toov in toov_to_delete:
                toov_id = toov.id

                await session.delete(toov)

                additional_info_query = select(db_models.AdditionalInfo).where(
                    db_models.AdditionalInfo.element_id == toov_id)
                additional_info_result = await session.execute(additional_info_query)
                additional_info_to_delete = additional_info_result.scalars().all()

                await asyncio.gather(*(session.delete(item) for item in additional_info_to_delete))

            await session.commit()
```

```
        return True


async def update_toov(toov_id, **kwargs):
    async with db.AsyncSession() as session:
        async with session.begin():
            query = select(db_models.TOOV).where(db_models.TOOV.id ==
toov_id)
            result = await session.execute(query)
            toov_to_update = result.scalar_one_or_none()

            if not toov_to_update:
                raise ValueError(f"Запись с id {toov_id} не найдена.")

            state_fields = {
                'functionality_state',
                'broken_body_state',
                'dirt_state',
                'indicator_state',
                'fastening_reliability_state',
            }

            updates = {}
            for field, value in kwargs.items():
                if value:
                    updates[field] = await get_object_state(value) if field in state_fields
else value

            for attr, val in updates.items():
```

```
            setattr(toov_to_update, attr, val)

        if "additional_info" in kwargs and kwargs["additional_info"]:
            for item in kwargs["additional_info"]:
                additional_id = item.get('id')
                additional_name = item.get('name')
                additional_field_name = item.get('field_name')
                additional_field_state = item.get('field_state')
                additional_field_description = item.get('field_description')

                additional_info_query = select(db_models.AdditionalInfo).where(
                    db_models.AdditionalInfo.id == additional_id)
                additional_info_result = await
session.execute(additional_info_query)
                additional_info_update =
additional_info_result.scalar_one_or_none()

                if additional_name:
                    additional_info_update.name = additional_name
                if additional_field_name:
                    additional_info_update.field_name = additional_field_name
                if additional_field_state:
                    additional_info_update.field_state = await
get_object_state(additional_field_state)
                if additional_field_description:
                    additional_info_update.additional_field_description =
additional_field_description

        await session.commit()
```

```
        return True


async def add_detector_camera(detector_camera_id, junction_id, name,
                              functionality_state, functionality_description,
                              broken_body_state, broken_body_description,
                              dirt_state, dirt_description,
                              fastening_reliability_state, fastening_reliability_description,
                              additional_info
                              ):
    if not detector_camera_id:
        detector_camera_id: str = await cuid2_generator()

    functionality_state = await get_object_state(functionality_state)
    broken_body_state = await get_object_state(broken_body_state)
    dirt_state = await get_object_state(dirt_state)
    fastening_reliability_state = await get_object_state(fastening_reliability_state)

    new_detector_camera = db_models.DetectorCamera(
        id=detector_camera_id,
        junction_id=junction_id,
        name=name,
        functionality_state=functionality_state,
        functionality_description=functionality_description,
        broken_body_state=broken_body_state,
        broken_body_description=broken_body_description,
        dirt_state=dirt_state,
        dirt_description=dirt_description,
        fastening_reliability_state=fastening_reliability_state,
```

```
        fastening_reliability_description=fastening_reliability_description,
    )

    async with db.AsyncSession() as session:
        session.add(new_detector_camera)
        await session.commit()
        await session.refresh(new_detector_camera)

        if additional_info:
            for item in additional_info:
                additional_id: str = await cuid2_generator()
                junction_id: str = junction_id
                element_id = new_detector_camera.id
                name: str = item.name
                category =
db_models.AdditionalInfoCategory.DETECTOR_CAMERA
                field_name: str = item.field_name

                field_state = await get_object_state(item.field_state)

                field_description = item.field_description

                new_additional_info = db_models.AdditionalInfo(
                    id=additional_id,
                    junction_id=junction_id,
                    element_id=element_id,
                    name=name,
                    category=category,
                    field_name=field_name,
```

```
                field_state=field_state,
                field_description=field_description,
            )

            session.add(new_additional_info)
        await session.commit()

    return new_detector_camera


async def get_detectors_cameras_for_junction(junction_id: str):
    """Получить все записи 'Датчики, детекторы, видеокамеры' для осмотра"""

    async with db.AsyncSession() as session:
        query = select(db_models.DetectorCamera).where(db_models.DetectorCamera.junction_id == junction_id)
        result = await session.execute(query)
        detectors_cameras = result.scalars().all()

        detectors_cameras_result = (
            [{c.name: getattr(device, c.name) for c in device.__table__.columns} for device in
             detectors_cameras])

        for device in detectors_cameras_result:
            device_id = device.get('id')
```

```
            query =
select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.element_id
== device_id,
                                                db_models.AdditionalInfo.category ==
db_models.AdditionalInfoCategory.DETECTOR_CAMERA
                                                )

            additional_data = await session.execute(query)

            if additional_data:
                additional_data = additional_data.scalars().all()
                device['additional_info'] = [
                    {c.name: getattr(add_data, c.name) for c in
add_data.__table__.columns}
                    for add_data in additional_data]

        return detectors_cameras_result



async def get_detector_camera(detector_camera_id: str):
    """Получить запись 'Датчики, детекторы, видеокамеры' по ее id"""

    async with db.AsyncSession() as session:
        query =
select(db_models.DetectorCamera).where(db_models.DetectorCamera.id ==
detector_camera_id)
        result = await session.execute(query)
        detector_camera = result.scalar_one_or_none()
```

```
    if not detector_camera:
        raise ValueError(f"Запись с id {detector_camera_id} не найдена.")

    detector_camera_result = {c.name: getattr(detector_camera, c.name) for c in
detector_camera.__table__.columns}

    query =
select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.element_id
== detector_camera_id,
                                       db_models.AdditionalInfo.category ==
db_models.AdditionalInfoCategory.DETECTOR_CAMERA
                                       )

    additional_data = await session.execute(query)

    if additional_data:
        additional_data = additional_data.scalars().all()
        detector_camera_result['additional_info'] = [
            {c.name: getattr(add_data, c.name) for c in
add_data.__table__.columns}
            for add_data in additional_data]

    return detector_camera_result


async def delete_detector_camera(detector_camera_id: str):
    """Удаление записи 'Датчики, детекторы, видеокамеры' по ее id"""

    async with db.AsyncSession() as session:
```

```
        async with session.begin():
            query = select(db_models.DetectorCamera).where(
                db_models.DetectorCamera.id == detector_camera_id)
            result = await session.execute(query)
            detector_camera_to_delete = result.scalar_one_or_none()

            additional_data_query = select(db_models.AdditionalInfo).where(
                db_models.AdditionalInfo.element_id == detector_camera_id)
            additional_data_result = await session.execute(additional_data_query)
            additional_data_to_delete = additional_data_result.scalars().all()

            if not detector_camera_to_delete:
                raise ValueError(f"Запись с id {detector_camera_id} не найдена.")

            await session.delete(detector_camera_to_delete)
            if additional_data_to_delete:
                for item_to_delete in additional_data_to_delete:
                    await session.delete(item_to_delete)
            await session.commit()
            return True


async def delete_detectors_cameras_for_junction(junction_id: str):
    """Удаление всех записей 'Датчики, детекторы, видеокамеры'
привязанных к осмотру"""

    async with db.AsyncSession() as session:
        async with session.begin():
            query = select(db_models.DetectorCamera).where(
```

```
            db_models.DetectorCamera.junction_id == junction_id)
        result = await session.execute(query)
        detectors_cameras_to_delete = result.scalars().all()

        for detector_camera in detectors_cameras_to_delete:
            detector_camera_id = detector_camera.id

            await session.delete(detector_camera)

            additional_info_query = select(db_models.AdditionalInfo).where(
                db_models.AdditionalInfo.element_id == detector_camera_id)
            additional_info_result = await session.execute(additional_info_query)
            additional_info_to_delete = additional_info_result.scalars().all()

            await asyncio.gather(*(session.delete(item) for item in
additional_info_to_delete))

        await session.commit()
        return True


async def update_detector_camera(detector_camera_id, **kwargs):
    async with db.AsyncSession() as session:
        async with session.begin():
            query =
select(db_models.DetectorCamera).where(db_models.DetectorCamera.id ==
detector_camera_id)
            result = await session.execute(query)
            detector_camera_to_update = result.scalar_one_or_none()
```

```
        if not detector_camera_to_update:
            raise ValueError(f"Запись с id {detector_camera_id} не найдена.")

        state_fields = {
            'functionality_state',
            'broken_body_state',
            'dirt_state',
            'fastening_reliability_state',
        }

        updates = {}
        for field, value in kwargs.items():
            if value:
                updates[field] = await get_object_state(value) if field in state_fields
else value

        for attr, val in updates.items():
            setattr(detector_camera_to_update, attr, val)

        if "additional_info" in kwargs and kwargs["additional_info"]:
            for item in kwargs["additional_info"]:
                additional_id = item.get('id')
                additional_name = item.get('name')
                additional_field_name = item.get('field_name')
                additional_field_state = item.get('field_state')
                additional_field_description = item.get('field_description')

                additional_info_query = select(db_models.AdditionalInfo).where(
```

```
                db_models.AdditionalInfo.id == additional_id)
            additional_info_result = await
session.execute(additional_info_query)
            additional_info_update =
additional_info_result.scalar_one_or_none()

            if additional_name:
                additional_info_update.name = additional_name
            if additional_field_name:
                additional_info_update.field_name = additional_field_name
            if additional_field_state:
                additional_info_update.field_state = await
get_object_state(additional_field_state)
            if additional_field_description:
                additional_info_update.additional_field_description =
additional_field_description

        await session.commit()
        return True


async def add_tsodd(tsodd_id, junction_id, name,
            inscriptions_advertisement_state,
inscriptions_advertisement_description,
            dirt_state, dirt_description,
            reflective_film_state, reflective_film_description,
            position_state, position_description,
            fastening_reliability_state, fastening_reliability_description,
            additional_info
```

```
            ):
    if not tsodd_id:
        tsodd_id: str = await cuid2_generator()

    inscriptions_advertisement_state = await
get_object_state(inscriptions_advertisement_state)
    dirt_state = await get_object_state(dirt_state)
    reflective_film_state = await get_object_state(reflective_film_state)
    position_state = await get_object_state(position_state)
    fastening_reliability_state = await get_object_state(fastening_reliability_state)

    new_tsodd = db_models.TSODD(
        id=tsodd_id,
        junction_id=junction_id,
        name=name,
        inscriptions_advertisement_state=inscriptions_advertisement_state,

inscriptions_advertisement_description=inscriptions_advertisement_description,
        dirt_state=dirt_state,
        dirt_description=dirt_description,
        reflective_film_state=reflective_film_state,
        reflective_film_description=reflective_film_description,
        position_state=position_state,
        position_description=position_description,
        fastening_reliability_state=fastening_reliability_state,
        fastening_reliability_description=fastening_reliability_description,
    )

    async with db.AsyncSession() as session:
```

```
session.add(new_tsodd)
await session.commit()
await session.refresh(new_tsodd)

if additional_info:
    for item in additional_info:
        additional_id: str = await cuid2_generator()
        junction_id: str = junction_id
        element_id = new_tsodd.id
        name: str = item.name
        category = db_models.AdditionalInfoCategory.TSODD
        field_name: str = item.field_name

        field_state = await get_object_state(item.field_state)

        field_description = item.field_description

        new_additional_info = db_models.AdditionalInfo(
            id=additional_id,
            junction_id=junction_id,
            element_id=element_id,
            name=name,
            category=category,
            field_name=field_name,
            field_state=field_state,
            field_description=field_description,
        )

        session.add(new_additional_info)
```

```
            await session.commit()

    return new_tsodd


async def get_tsodd_for_junction(junction_id: str):
    """Получить все записи 'ТСОДД - технические средства организации дорожного движения' для осмотра"""

    async with db.AsyncSession() as session:
        query = select(db_models.TSODD).where(db_models.TSODD.junction_id == junction_id)
        result = await session.execute(query)
        tsodd_devices = result.scalars().all()

        tsodd_result = (
            [{c.name: getattr(device, c.name) for c in device.__table__.columns} for device in
             tsodd_devices])

        for device in tsodd_result:
            device_id = device.get('id')
            query = select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.element_id == device_id,
                                                           db_models.AdditionalInfo.category == db_models.AdditionalInfoCategory.TSODD
                                                           )
```

```
            additional_data = await session.execute(query)

            if additional_data:
                additional_data = additional_data.scalars().all()
                device['additional_info'] = [
                    {c.name: getattr(add_data, c.name) for c in
add_data.__table__.columns}
                    for add_data in additional_data]

        return tsodd_result



async def get_tsodd(tsodd_id: str):
    """Получить объект осмотра по его id"""

    async with db.AsyncSession() as session:
        query = select(db_models.TSODD).where(db_models.TSODD.id ==
tsodd_id)
        result = await session.execute(query)
        tsodd = result.scalar_one_or_none()

        if not tsodd:
            raise ValueError(f"Запись с id {tsodd_id} не найдена.")

        tsodd_result = {c.name: getattr(tsodd, c.name) for c in
tsodd.__table__.columns}

        tsodd_id = tsodd_result.get('id')
```

```
        query =
select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.element_id
== tsodd_id,
                                               db_models.AdditionalInfo.category ==
db_models.AdditionalInfoCategory.TSODD
                                               )

        additional_data = await session.execute(query)

        if additional_data:
            additional_data = additional_data.scalars().all()
            tsodd_result['additional_info'] = [
                {c.name: getattr(add_data, c.name) for c in
add_data.__table__.columns}
                for add_data in additional_data]

        return tsodd_result



async def delete_tsodd(tsodd_id: str):
    """Удаление записи 'ТСОДД - технические средства организации
дорожного движения' по ее id"""

    async with db.AsyncSession() as session:
        async with session.begin():
            query = select(db_models.TSODD).where(db_models.TSODD.id ==
tsodd_id)
            result = await session.execute(query)
            accessible_device_to_delete = result.scalar_one_or_none()
```

```
            additional_data_query = select(db_models.AdditionalInfo).where(
                db_models.AdditionalInfo.element_id == tsodd_id)
            additional_data_result = await session.execute(additional_data_query)
            additional_data_to_delete = additional_data_result.scalars().all()

            if not accessible_device_to_delete:
                raise ValueError(f"Запись с id {tsodd_id} не найдена.")

            await session.delete(accessible_device_to_delete)
            if additional_data_to_delete:
                for item_to_delete in additional_data_to_delete:
                    await session.delete(item_to_delete)
            await session.commit()
            return True


async def delete_tsodd_for_junction(junction_id: str):
    """Удаление всех записей 'ТСОДД - технические средства организации дорожного движения' привязанных к осмотру"""

    async with db.AsyncSession() as session:
        async with session.begin():
            query = select(db_models.TSODD).where(db_models.TSODD.junction_id == junction_id)
            result = await session.execute(query)
            tsodd_to_delete = result.scalars().all()
```

```
        for tsodd_device in tsodd_to_delete:
            tsodd_id = tsodd_device.id

            await session.delete(tsodd_device)

            additional_info_query = select(db_models.AdditionalInfo).where(
                db_models.AdditionalInfo.element_id == tsodd_id)
            additional_info_result = await session.execute(additional_info_query)
            additional_info_to_delete = additional_info_result.scalars().all()

            await asyncio.gather(*(session.delete(item) for item in
additional_info_to_delete))

        await session.commit()
        return True


async def update_tsodd(tsodd_id, **kwargs):
    async with db.AsyncSession() as session:
        async with session.begin():
            query = select(db_models.TSODD).where(db_models.TSODD.id ==
tsodd_id)
            result = await session.execute(query)
            tsodd_to_update = result.scalar_one_or_none()

            if not tsodd_to_update:
                raise ValueError(f"Запись с id {tsodd_id} не найдена.")

            state_fields = {
```

```
            'inscriptions_advertisement_state',
            'dirt_state',
            'reflective_film_state',
            'position_state',
            'fastening_reliability_state',
        }

        updates = {}
        for field, value in kwargs.items():
            if value:
                updates[field] = await get_object_state(value) if field in state_fields
else value

        for attr, val in updates.items():
            setattr(tsodd_to_update, attr, val)

        if "additional_info" in kwargs and kwargs["additional_info"]:
            for item in kwargs["additional_info"]:
                additional_id = item.get('id')
                additional_name = item.get('name')
                additional_field_name = item.get('field_name')
                additional_field_state = item.get('field_state')
                additional_field_description = item.get('field_description')

                additional_info_query = select(db_models.AdditionalInfo).where(
                    db_models.AdditionalInfo.id == additional_id)
                additional_info_result = await
session.execute(additional_info_query)
```

```
            additional_info_update =
additional_info_result.scalar_one_or_none()

            if additional_name:
                additional_info_update.name = additional_name
            if additional_field_name:
                additional_info_update.field_name = additional_field_name
            if additional_field_state:
                additional_info_update.field_state = await
get_object_state(additional_field_state)
            if additional_field_description:
                additional_info_update.additional_field_description =
additional_field_description

        await session.commit()
        return True


async def add_light_section(light_section_id, junction_id, name,
                            functionality_state, functionality_description,
                            mode_state, mode_description,
                            indicator_state, indicator_description,
                            inscriptions_advertisement_state,
inscriptions_advertisement_description,
                            dirt_state, dirt_description,
                            design_position_state, design_position_description,
                            fastening_reliability_state, fastening_reliability_description,
                            additional_info
                            ):
```

```
    if not light_section_id:
        light_section_id: str = await cuid2_generator()

    functionality_state = await get_object_state(functionality_state)
    mode_state = await get_object_state(mode_state)
    indicator_state = await get_object_state(indicator_state)
    inscriptions_advertisement_state = await
get_object_state(inscriptions_advertisement_state)
    dirt_state = await get_object_state(dirt_state)
    design_position_state = await get_object_state(design_position_state)
    fastening_reliability_state = await get_object_state(fastening_reliability_state)

    new_light_section = db_models.LightSections(
        id=light_section_id,
        junction_id=junction_id,
        name=name,
        functionality_state=functionality_state,
        functionality_description=functionality_description,
        mode_state=mode_state,
        mode_description=mode_description,
        indicator_state=indicator_state,
        indicator_description=indicator_description,
        inscriptions_advertisement_state=inscriptions_advertisement_state,

inscriptions_advertisement_description=inscriptions_advertisement_description,
        dirt_state=dirt_state,
        dirt_description=dirt_description,
        design_position_state=design_position_state,
        design_position_description=design_position_description,
```

```
    fastening_reliability_state=fastening_reliability_state,
    fastening_reliability_description=fastening_reliability_description,
)

async with db.AsyncSession() as session:
    session.add(new_light_section)
    await session.commit()
    await session.refresh(new_light_section)

    if additional_info:
        for item in additional_info:
            additional_id: str = await cuid2_generator()
            junction_id: str = junction_id
            element_id = new_light_section.id
            name: str = item.name
            category = db_models.AdditionalInfoCategory.LIGHT_SECTIONS
            field_name: str = item.field_name

            field_state = await get_object_state(item.field_state)

            field_description = item.field_description

            new_additional_info = db_models.AdditionalInfo(
                id=additional_id,
                junction_id=junction_id,
                element_id=element_id,
                name=name,
                category=category,
                field_name=field_name,
```

```
                field_state=field_state,
                field_description=field_description,
            )

            session.add(new_additional_info)
        await session.commit()

    return new_light_section


async def get_light_sections_for_junction(junction_id: str):
    """Получить все записи 'Информационные световые секции' для осмотра"""

    async with db.AsyncSession() as session:
        query = select(db_models.LightSections).where(db_models.LightSections.junction_id == junction_id)
        result = await session.execute(query)
        light_sections_devices = result.scalars().all()

        light_sections_result = (
            [{c.name: getattr(device, c.name) for c in device.__table__.columns} for device in
             light_sections_devices])

        for light_section in light_sections_result:
            light_section_id = light_section.get('id')
```

```
            query =
select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.element_id
== light_section_id,
                                                db_models.AdditionalInfo.category ==
db_models.AdditionalInfoCategory.LIGHT_SECTIONS
                                                )

            additional_data = await session.execute(query)

            if additional_data:
                additional_data = additional_data.scalars().all()
                light_section['additional_info'] = [
                    {c.name: getattr(add_data, c.name) for c in
add_data.__table__.columns}
                    for add_data in additional_data]

        return light_sections_result



async def get_light_section(light_section_id: str):
    """Получить запись 'Информационные световые секции' осмотра по ее
id"""

    async with db.AsyncSession() as session:
        query = select(db_models.LightSections).where(db_models.LightSections.id
== light_section_id)
        result = await session.execute(query)
        light_section = result.scalar_one_or_none()
```

```
    if not light_section:
        raise ValueError(f"Запись с id {light_section_id} не найдена.")

    light_section_result = {c.name: getattr(light_section, c.name) for c in
light_section.__table__.columns}

    query =
select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.element_id
== light_section_id,
                                           db_models.AdditionalInfo.category ==
db_models.AdditionalInfoCategory.LIGHT_SECTIONS
                                           )

    additional_data = await session.execute(query)

    if additional_data:
        additional_data = additional_data.scalars().all()
        light_section_result['additional_info'] = [
            {c.name: getattr(add_data, c.name) for c in
add_data.__table__.columns}
            for add_data in additional_data]

    return light_section_result


async def delete_light_section(light_section_id: str):
    """Удаление записи 'Информационные световые секции' по ее id"""

    async with db.AsyncSession() as session:
```

```
        async with session.begin():
            query =
select(db_models.LightSections).where(db_models.LightSections.id ==
light_section_id)
            result = await session.execute(query)
            light_section_to_delete = result.scalar_one_or_none()

            additional_data_query = select(db_models.AdditionalInfo).where(
                db_models.AdditionalInfo.element_id == light_section_id)
            additional_data_result = await session.execute(additional_data_query)
            additional_data_to_delete = additional_data_result.scalars().all()

            if not light_section_to_delete:
                raise ValueError(f"Запись с id {light_section_id} не найдена.")

            await session.delete(light_section_to_delete)
            if additional_data_to_delete:
                for item_to_delete in additional_data_to_delete:
                    await session.delete(item_to_delete)
            await session.commit()
            return True


async def delete_light_sections_for_junction(junction_id: str):
    """Удаление всех записей 'Информационные световые секции'
привязанных к осмотру"""

    async with db.AsyncSession() as session:
        async with session.begin():
```

```
        query =
select(db_models.LightSections).where(db_models.LightSections.junction_id ==
junction_id)
        result = await session.execute(query)
        light_sections_to_delete = result.scalars().all()

        for light_section in light_sections_to_delete:
            light_section_id = light_section.id

            await session.delete(light_section)

            additional_info_query = select(db_models.AdditionalInfo).where(
                db_models.AdditionalInfo.element_id == light_section_id)
            additional_info_result = await session.execute(additional_info_query)
            additional_info_to_delete = additional_info_result.scalars().all()

            await asyncio.gather(*(session.delete(item) for item in
additional_info_to_delete))

        await session.commit()
        return True



async def update_light_section(light_section_id, **kwargs):
    async with db.AsyncSession() as session:
        async with session.begin():
            query =
select(db_models.LightSections).where(db_models.LightSections.id ==
light_section_id)
```

```
        result = await session.execute(query)
        light_section_to_update = result.scalar_one_or_none()

        if not light_section_to_update:
            raise ValueError(f"Запись с id {light_section_id} не найдена.")

        state_fields = {
            'functionality_state',
            'mode_state',
            'indicator_state',
            'inscriptions_advertisement_state',
            'dirt_state',
            'design_position_state',
            'fastening_reliability_state',
        }

        updates = {}
        for field, value in kwargs.items():
            if value:
                updates[field] = await get_object_state(value) if field in state_fields
else value

        for attr, val in updates.items():
            setattr(light_section_to_update, attr, val)

        if "additional_info" in kwargs and kwargs["additional_info"]:
            for item in kwargs["additional_info"]:
                additional_id = item.get('id')
                additional_name = item.get('name')
```

```
            additional_field_name = item.get('field_name')
            additional_field_state = item.get('field_state')
            additional_field_description = item.get('field_description')

            additional_info_query = select(db_models.AdditionalInfo).where(
                db_models.AdditionalInfo.id == additional_id)
            additional_info_result = await
session.execute(additional_info_query)
            additional_info_update =
additional_info_result.scalar_one_or_none()

            if additional_name:
                additional_info_update.name = additional_name
            if additional_field_name:
                additional_info_update.field_name = additional_field_name
            if additional_field_state:
                additional_info_update.field_state = await
get_object_state(additional_field_state)
            if additional_field_description:
                additional_info_update.additional_field_description =
additional_field_description

        await session.commit()
        return True


async def add_additional_info(additional_info_id, name, junction_id, element_id,
category,
                              field_name, field_state, field_description,
```

```
                    ):
    if not additional_info_id:
        additional_info_id: str = await cuid2_generator()

    field_state = await get_object_state(field_state)
    category = db_models.AdditionalInfoCategory(category).name

    new_additional_info = db_models.AdditionalInfo(
        id=additional_info_id,
        name=name,
        junction_id=junction_id,
        element_id=element_id,
        category=category,
        field_name=field_name,
        field_state=field_state,
        field_description=field_description,
    )

    async with db.AsyncSession() as session:
        session.add(new_additional_info)
        await session.commit()
        await session.refresh(new_additional_info)

    return new_additional_info


async def get_additional_info_for_junction(junction_id: str):
    """Получение всех записей "Дополнительный пункт осмотра"' для
осмотра"""
```

```
    async with db.AsyncSession() as session:
        query =
select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.junction_id
== junction_id)
        result = await session.execute(query)
        additional_info = result.scalars().all()

        additional_info_result = (
            [{c.name: getattr(data, c.name) for c in data.__table__.columns} for data
in additional_info]
        )

        return additional_info_result


async def get_additional_info(additional_info_id: str):
    """Получить объект осмотра по его id"""

    async with db.AsyncSession() as session:
        query =
select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.id ==
additional_info_id)
        result = await session.execute(query)
        additional_info = result.scalar_one_or_none()

        if not additional_info:
            raise ValueError(f"Запись с id {additional_info_id} не найдена.")
```

```
        additional_info_result = {c.name: getattr(additional_info, c.name) for c in
additional_info.__table__.columns}

        return additional_info_result



async def delete_additional_info(additional_info_id: str):
    """Удаление записи 'Дополнительный пункт осмотра' по ее id"""

    async with db.AsyncSession() as session:
        async with session.begin():
            query =
select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.id ==
additional_info_id)
            result = await session.execute(query)
            additional_info_to_delete = result.scalar_one_or_none()

            if not additional_info_to_delete:
                raise ValueError(f"Запись с id {additional_info_id} не найдена.")

            await session.delete(additional_info_to_delete)
            await session.commit()
            return True



async def delete_additional_info_for_junction(junction_id: str):
    """Удаление всех записей 'Дополнительный пункт осмотра' привязанных
к осмотру"""
```

```
    async with db.AsyncSession() as session:
        async with session.begin():
            query =
select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.junction_id
== junction_id)
            result = await session.execute(query)
            additional_info_to_delete = result.scalars().all()

            for additional_info in additional_info_to_delete:
                await session.delete(additional_info)

            await session.commit()
            return True



async def update_additional_info(additional_info_id, **kwargs):
    async with db.AsyncSession() as session:
        async with session.begin():
            query =
select(db_models.AdditionalInfo).where(db_models.AdditionalInfo.id ==
additional_info_id)
            result = await session.execute(query)
            additional_info_to_update = result.scalar_one_or_none()

            if not additional_info_to_update:
                raise ValueError(f"Запись с id {additional_info_id} не найдена.")

            additional_name = kwargs.get('name')
            additional_field_name = kwargs.get('field_name')
```

```
        additional_field_state = kwargs.get('field_state')
        additional_field_description = kwargs.get('field_description')

        if additional_name:
            additional_info_to_update.name = additional_name
        if additional_field_name:
            additional_info_to_update.field_name = additional_field_name
        if additional_field_state:
            additional_info_to_update.field_state = await
get_object_state(additional_field_state)
        if additional_field_description:
            additional_info_to_update.additional_field_description =
additional_field_description

        await session.commit()
        return True



async def add_file(junction_id, file_name):
    file_id: str = await cuid2_generator()

    new_file = db_models.Files(
        id=file_id,
        junction_id=junction_id,
        link=file_name,
    )

    async with db.AsyncSession() as session:
        session.add(new_file)
```

```
        await session.commit()
        await session.refresh(new_file)
        return new_file


async def add_or_update_file(junction_id, file_name):
    async with db.AsyncSession() as session:
        query = select(db_models.Files).where(db_models.Files.junction_id ==
junction_id).where(
            db_models.Files.link == file_name)
        result = await session.execute(query)

        file = result.scalar_one_or_none()

        if not file:
            # если файл не сохранен ранее, сохраняем
            file_id: str = await cuid2_generator()

            new_file = db_models.Files(
                id=file_id,
                junction_id=junction_id,
                link=file_name,
            )

            async with db.AsyncSession() as session:
                session.add(new_file)
                await session.commit()
                await session.refresh(new_file)
                return new_file
```

```
        else:
            # если файл сохранен ранее, обновляем updated_at
            file.updated_at = datetime.now()

            await session.commit()
            return file


async def get_files_for_junction(junction_id: str):
    """Получить список файлов для осмотра"""

    async with db.AsyncSession() as session:
        query = select(db_models.Files).where(db_models.Files.junction_id ==
junction_id)
        result = await session.execute(query)
        files = result.scalars().all()

        files_result = ([{c.name: getattr(file, c.name) for c in
file.__table__.columns} for file in files])

        return files_result


async def get_file_db_data(file_id: str):
    """Получить данные о файле"""

    async with db.AsyncSession() as session:
        query = select(db_models.Files).where(db_models.Files.id == file_id)
```

```
        result = await session.execute(query)
        file = result.scalar_one_or_none()

        if not file:
            raise ValueError(f"Запись с id {file} не найдена.")

        file_result = {c.name: getattr(file, c.name) for c in file.__table__.columns}

        return file_result


async def delete_file(file_id: str):
    """Удаление файла по его id"""

    async with db.AsyncSession() as session:
        async with session.begin():
            query = select(db_models.Files).where(db_models.Files.id == file_id)
            result = await session.execute(query)
            file_to_delete = result.scalar_one_or_none()

            if not file_to_delete:
                raise ValueError(f"Запись с id {file_id} не найдена.")

            file_name = file_to_delete.link
            await session.delete(file_to_delete)
            await session.commit()
            return file_name
```

```
async def delete_files_for_junction(junction_id: str):
    """Удаление всех файлов привязанных к осмотру"""

    async with db.AsyncSession() as session:
        async with session.begin():
            query = select(db_models.Files).where(db_models.Files.junction_id ==
junction_id)
            result = await session.execute(query)
            files_to_delete = result.scalars().all()

            files_names_list = list()  # список наименований удаляемых файлов

            for file in files_to_delete:
                file_name = file.link
                files_names_list.append(file_name)
                await session.delete(file)

            await session.commit()
            return files_names_list
```

7. Скрипт для реализации вспомогательных функций

Скрипт util.py, являющийся неотъемлемой частью текста программы, предназначен для реализации вспомогательных функций, используемых в различных частях системы. Включает функции для генерации идентификаторов, работы с состояниями объектов и взаимодействия с хранилищем файлов S3.

Входные данные: параметры для генерации идентификаторов, состояния объектов, файлы для сохранения.

Выходные данные: сгенерированные идентификаторы, обработанные состояния, результаты операций с файлами.

```
from cuid2 import cuid_wrapper
from typing import Callable
from api.models import CheckObjectState
from connectors.s3 import s3_client
from settings import settings
from botocore.exceptions import ClientError
from io import BytesIO


async def cuid2_generator() -> str:
    """Генерация cuid2"""

    cuid_generator: Callable[[], str] = cuid_wrapper()
    id_pk: str = cuid_generator()
    return id_pk


async def get_object_state(state: str) -> str | None:
    """Получение значения состояния объекта из enum"""

    try:
        field_state = CheckObjectState(state)
        field_name = field_state.name
        if field_name == 'null':
            return None
        return field_name
    except ValueError:
        return None
```

```
async def save_s3_file(file):
    """Сохранение файла в S3"""

    file_name = file.filename

    try:
        # проверка существования ранее добавленного файла с таким названием в корзину
        is_exist = s3_client.head_object(Bucket=settings.s3_bucket, Key=file_name)

        if is_exist['ResponseMetadata']['HTTPStatusCode'] == 200:
            raise FileExistsError('Файл с таким именем уже существует')
    except ClientError:
        # если файл отсутствует в корзине, сохраняем
        s3_client.upload_fileobj(file.file, settings.s3_bucket, file_name)

    return True

async def save_or_update_s3_file(file, file_name: str = None):
    """Сохранение файла в S3, либо обновление если файл существует """

    s3_client.upload_fileobj(file, settings.s3_bucket, file_name)

    return True

async def get_s3_file(file_name: str):
    """Получение файла из S3"""
```

```
    file_stream = BytesIO()

    s3_client.download_fileobj(settings.s3_bucket, file_name, file_stream)

    file_stream.seek(0)

    return file_stream

async def delete_s3_file(file_name: str):
    """Удаление файла в S3"""

    s3_client.delete_object(Bucket=settings.s3_bucket, Key=file_name)
```

8. Скрипт для реализации вспомогательных функций

Скрипт files.py, являющийся неотъемлемой частью текста программы, предназначен для обработки API-запросов, связанных с файлами. Реализует функциональность загрузки, получения и удаления файлов, связанных с осмотрами.

Входные данные: файлы для загрузки, идентификаторы файлов и осмотров.

Выходные данные: результаты операций с файлами, метаданные файлов, содержимое файлов.

```
from api.schemas import GenericErrorResponse, GenericResponse

from fastapi import APIRouter, UploadFile, File, Form, HTTPException
from fastapi.responses import StreamingResponse
import api.db_requests as db_requests
import api.models as pydantic_models
from api.util import save_s3_file, get_s3_file, delete_s3_file
```

```
from urllib.parse import quote


files_router = APIRouter(prefix="/api", tags=["files_api"])



@files_router.post('/upload_file',
                   summary='Добавление файла для осмотра',
                   responses={200: {"model": GenericResponse},
                              400: {"model": GenericErrorResponse}, }
                   )
async def files(junction_id: str = Form(...), file: UploadFile = File(...)):
    """
    Добавление файла для осмотра по его id
    - Все добавляемые файлы должны иметь уникальные имена.
    """

    try:
        file_name = file.filename

        if await save_s3_file(file):
            new_file = await db_requests.add_file(junction_id, file_name)
            if new_file:
                return GenericResponse(message='added', record_id=new_file.id)
    except Exception as e:
        return GenericErrorResponse(message='error', error=str(e))



@files_router.get('/get_files_list',
                  summary='Получение списка файлов для осмотра',
```

```
            responses={200: {"model": pydantic_models.FilesResponse},
                    400: {"model": GenericErrorResponse}, }
            )
async def files(junction_id: str):
    try:
        files_list = await db_requests.get_files_for_junction(junction_id)

        return pydantic_models.FilesResponse(files=files_list)
    except Exception as e:
        return GenericErrorResponse(message='error', error=str(e))



@files_router.get('/get_files_data',
            summary='Получение данных о файле для осмотра по его id',
            responses={200: {"model": pydantic_models.FilesSchema},
                    400: {"model": GenericErrorResponse}, }
            )
async def get_file_data(file_id: str):
    try:
        file = await db_requests.get_file_db_data(file_id)

        return pydantic_models.FilesSchema(**file)
    except Exception as e:
        return GenericErrorResponse(message='error', error=str(e))



@files_router.get('/get_file',
            summary='Получение файла по его id',
            responses={200: {
```

```
            "content": {
                "application/octet-stream": {
                    "schema": {
                        "type": "string",
                        "format": "binary"
                    }
                }
            },
            "description": "Файл в binary формате"
        },
            400: {"model": GenericErrorResponse}, }
        )
async def get_file(file_id: str):
    try:
        file_data = await db_requests.get_file_db_data(file_id)
        if file_data:
            file_name = file_data.get('link')
            file_stream = await get_s3_file(file_name)

            if file_stream:
                content_type = "application/octet-stream"

                encoded_filename = quote(file_name)

                return StreamingResponse(file_stream, media_type=content_type, headers={
                    "Content-Disposition": f"attachment; filename={encoded_filename}"
                })
            else:
```

```
            raise HTTPException(status_code=404, detail="Файл отсутствует в хранилище")
    except Exception as e:
        return GenericErrorResponse(message='error', error=str(e))


@files_router.delete('/delete_file',
                     summary='Удаление файла по его id',
                     responses={200: {"model": GenericResponse},
                                400: {"model": GenericErrorResponse}, }
                     )
async def delete_file(request: pydantic_models.ElementById):
    try:
        file_id = request.id

        if file_name := await db_requests.delete_file(file_id):
            await delete_s3_file(file_name)

            return GenericResponse(message='deleted', record_id=file_id)
    except Exception as e:
        return GenericErrorResponse(message='error', error=str(e))


@files_router.delete('/delete_all_files',
                     summary='Удаление всех файлов по junction_id',
                     responses={200: {"model": GenericResponse},
                                400: {"model": GenericErrorResponse}, }
                     )
async def delete_all_files(request: pydantic_models.ElementById):
```

```
    try:
        junction_id = request.id
        if file_names := await db_requests.delete_files_for_junction(junction_id):
            for file_name in file_names:
                await delete_s3_file(file_name)


            return GenericResponse(message='deleted', record_id=junction_id)
    except Exception as e:
        return GenericErrorResponse(message='error', error=str(e))
```

9. Скрипт для обработки API-запросов

Скрипт junction.py, являющийся неотъемлемой частью текста программы, предназначен для обработки API-запросов, связанных с объектами осмотра (перекрестками). Реализует CRUD-операции для управления данными об осмотрах.

Входные данные: параметры запросов API, данные для создания и обновления записей. Выходные данные: результаты выполнения операций, данные об объектах осмотра.

```
from api.schemas import GenericErrorResponse, GenericResponse
from fastapi import APIRouter, Query
import api.db_requests as db_requests
import api.models as pydantic_models
from typing import Literal

junction_router = APIRouter(prefix="/api", tags=["junction_api"])
```

```
@junction_router.post('/junctions',
                      summary='Создание объекта осмотра',
                      responses={200: {"model": GenericResponse},
                                 400: {"model": GenericErrorResponse}, }
                      )
async def junctions(request: pydantic_models.JunctionsCreateSchema):
    """
    <details>
    <summary>**Описание значений полей:**</summary>
    - **kind_id**: str - "Вид перекрестка (id из junctions_kind)"
    - **region**: str - "Наименование региона (субъекта РФ)"
    - **locality**: str - "Наименование населенного пункта"
    - **address**: str - "Адрес расположения"
    - **longitude**: float - "Долгота"
    - **latitude**: float - "Широта"
    - **status**: bool - "Узел оборудованный/не оборудованный"
    - **number**: str - "Номер документа"
    - **project**: str - "Наименование проекта"
    - **inspection_status**: enum - "Статус осмотра"
    - **inspection_at**: datetime -"Дата осмотра"
    - **executor_name**: str - "ФИО пользователя"
    - **executor_position**: str -"Должность пользователя"
    - **comment**: str - "Комментарий"
    </details>

    **Возможные значения для поля 'inspection_status':**
    - "Not inspected"
    - "Inspected"
    """
```

```
    try:
        junction_id = request.id
        kind_id = request.kind_id
        region = request.region
        locality = request.locality
        address = request.address
        longitude = request.longitude
        latitude = request.latitude
        status = request.status
        number = request.number
        project = request.project
        inspection_status = request.inspection_status
        inspection_at = request.inspection_at
        executor_name = request.executor_name
        executor_position = request.executor_position
        comment = request.comment

        new_junction = await db_requests.add_junctions(junction_id, kind_id, address,
longitude, latitude, status, number,
                                                       inspection_status, inspection_at,
                                                       executor_name, executor_position, comment,
                                                       region, locality, project)
        if new_junction:
            return GenericResponse(message='added', record_id=new_junction.id)
    except Exception as e:
        return GenericErrorResponse(message='error', error=str(e))
```

```
@junction_router.get('/junctions',
                summary='Получение объектов осмотра',
                responses={200: {"model": pydantic_models.JunctionsResponse},
                        400: {"model": GenericErrorResponse}, }
                )
async def junctions(
    sort_by: Literal['executor_name',
    'executor_position',
    'inspection_status',
    'address',
    'number',
    'locality',
    'inspection_at',
    'created_at'] = Query(default="created_at", description="Поле для сортировки"),
    sort_direction: Literal['asc', 'desc'] = Query(default="desc", description="Направление сортировки"),
    search_by_number: str = Query(default=None, description="Поиск по номеру"),
    search_by_address: str = Query(default=None, description="Поиск по адресу"),
    search_by_executor_name: str = Query(default=None, description="Поиск по имени исполнителя"),
    search_by_created_at: str = Query(default=None, description="Поиск по дате создания"),
    search_by_locality: str = Query(default=None, description="Поиск по населенному пункту"),
):
  try:
    junctions_list = await db_requests.get_all_junctions(sort_by, sort_direction,
                                                         search_by_number,
                                                         search_by_address,
```

```
                                search_by_executor_name,
                                search_by_created_at,
                                search_by_locality,
                                )

        return pydantic_models.JunctionsResponse(junctions=junctions_list)
    except Exception as e:
        return GenericErrorResponse(message='error', error=str(e))


@junction_router.get('/junction',
                     summary='Получение объекта осмотра по его id',
                     responses={200: {"model": pydantic_models.JunctionsSchema},
                                400: {"model": GenericErrorResponse}, }
                     )
async def junctions(junction_id: str):
    try:
        junction = await db_requests.get_junction(junction_id)

        return pydantic_models.JunctionsSchema(**junction)
    except Exception as e:
        return GenericErrorResponse(message='error', error=str(e))


@junction_router.delete('/junctions',
                        summary='Удаление объекта осмотра',
                        responses={200: {"model": GenericResponse},
                                   400: {"model": GenericErrorResponse}, }
                        )
```

```
async def junctions(request: pydantic_models.JunctionDelete):
    try:
        junction_id = request.id
        if await db_requests.delete_junction(junction_id):
            return GenericResponse(message='deleted', record_id=junction_id)
    except Exception as e:
        return GenericErrorResponse(message='error', error=str(e))



@junction_router.patch('/junctions',
                       summary='Обновление объекта осмотра',
                       responses={200: {"model": GenericResponse},
                                  400: {"model": GenericErrorResponse}, }
                       )
async def junctions(request: pydantic_models.JunctionsUpdateSchema):
    """
    <details>
    <summary>**Описание значений полей:**</summary>
    - **id**: str - "Идентификатор (CUID2)"
    - **kind_id**: str - "Вид перекрестка (id из junctions_kind)"
    - **region**: str - "Наименование региона (субъекта РФ)"
    - **locality**: str - "Наименование населенного пункта"
    - **address**: str - "Адрес расположения"
    - **longitude**: float - "Долгота"
    - **latitude**: float - "Широта"
    - **status**: bool - "Узел оборудованный/не оборудованный"
    - **number**: str - "Номер документа"
    - **project**: str - "Наименование проекта"
    - **inspection_status**: enum - "Статус осмотра"
```

- **inspection_at**: datetime -"Дата осмотра"
- **executor_name**: str - "ФИО пользователя"
- **executor_position**: str -"Должность пользователя"
- **comment**: str - "Комментарий"
</details>

Возможные значения для поля 'inspection_status':
- "Not inspected"
- "Inspected"
"""

```
try:
    junction_id = request.id
    kind_id = request.kind_id
    region = request.region
    locality = request.locality
    address = request.address
    longitude = request.longitude
    latitude = request.latitude
    status = request.status
    number = request.number
    project = request.project
    inspection_status = request.inspection_status
    inspection_at = request.inspection_at
    executor_name = request.executor_name
    executor_position = request.executor_position
    comment = request.comment
```

```
    if await db_requests.update_junction(junction_id, kind_id, address, longitude, latitude,
                                         status, number, inspection_status,
                                         inspection_at, executor_name, executor_position,
                                         comment, region, locality, project):
        return GenericResponse(message='updated', record_id=junction_id)
    except Exception as e:
        return GenericErrorResponse(message='error', error=str(e))
```

10. Скрипт для обработки API-запросов

Скрипт main_router.py, являющийся неотъемлемой частью текста программы, предназначен для обработки основных API-запросов, включая проверку состояния сервиса. Содержит базовые эндпоинты для мониторинга работоспособности системы.

Входные данные: параметры запросов API.

Выходные данные: результаты проверки состояния сервиса.

```
from api.schemas import GenericErrorResponse, GenericResponse
from fastapi import APIRouter

main_router = APIRouter(prefix="/api", tags=["main_api"])


@main_router.get('/health', description='Проверка состояния запуска сервиса',
                 responses={200: {"model": GenericResponse},
                            400: {"model": GenericErrorResponse},
                            404: {"model": GenericErrorResponse}})
async def health_check():
```

"""Эндпоинт для проверки состояния сервиса внутри docker compose health check"""

return GenericResponse(message='ok')

11. Скрипт для обработки API-запросов, связанных с видами перекрестков.

Скрипт junction_kind.py, являющийся неотъемлемой частью текста программы, предназначен для обработки API-запросов, связанных с видами перекрестков. Реализует CRUD-операции для управления справочником видов перекрестков.

Входные данные: параметры запросов API, данные для создания и обновления записей. Выходные данные: результаты выполнения операций, данные о видах перекрестков.

Перечень принятых сокращений

Сокращения	Обозначения
СПО	– специальное программное обеспечение.
КЛ СО	– контрольные листы свветофорных объектов

Лист регистрации изменений									
Номера листов (страниц)					Всего листов (страниц) в докум.	№ докуме н-та	Входящий № сопроводит. докум. и дата	Под п.	Дата
Изм.	изменен-ных	заменен-ных	новых	аннули-рованных					
