Отчёт №3

1. **Добавление новой роли "Менеджер по качеству":**
   В соответствии с дополнением к ТЗ, в систему была добавлена новая роль "Менеджер по качеству" (название в системе: "Quality Manager"). Данная роль реализует функционал консультанта при возникновении проблем с ремонтом оборудования и при невыполнении ремонта в срок.
   Код реализации роли:

```
# В schemas.py добавлены новые классы

class QualityManagerActions(BaseModel):

    action_type: Literal["extend_deadline", "assign_specialist", "consultation"]

    request_id: int

    specialist_id: Optional[int] = None

    new_deadline: Optional[str] = None

    consultation_notes: Optional[str] = None

    client_approval: bool = False



class RequestExtension(BaseModel):

    request_id: int

    new_deadline: str

    reason: str

    client_approval_code: Optional[str] = None



# В main.py добавлены эндпоинты для менеджера по качеству
```

```
@app.post(
    "/quality/extend-deadline",
    summary="Продлить срок выполнения заявки",
    dependencies=[Depends(require_roles("Quality Manager"))]
)
def extend_request_deadline(
    extension: RequestExtension,
    current_user: UserBase = Depends(get_current_user)
):
    """Продление срока выполнения заявки с согласования клиента"""
    request_data = get_request_by_id(extension.request_id)
    if not request_data:
        raise HTTPException(status_code=404, detail="Заявка не найдена")

    # Проверка, что заявка еще не завершена
    if request_data["request_status"] in ["Готова к выдаче", "Завершена"]:
        raise HTTPException(
            status_code=400,
            detail="Нельзя продлить срок завершенной заявки"
        )
```

```
# Генерация кода подтверждения для клиента
if not extension.client_approval_code:
    import hashlib
    import secrets
    approval_code = hashlib.sha256(
        f"{extension.request_id}{secrets.token_hex(8)}".encode()
    ).hexdigest()[:8].upper()

    # Здесь должна быть отправка кода клиенту (email/SMS)
    # В демо-версии возвращаем код в ответе
    return {
        "message": "Сгенерирован код подтверждения для клиента",
        "approval_code": approval_code,
        "instructions": "Передайте этот код клиенту для подтверждения продления срока"
    }

# Проверка кода подтверждения (в реальной системе - из БД)
if extension.client_approval_code != "VERIFIED":
    raise HTTPException(
        status_code=400,
        detail="Неверный код подтверждения от клиента"
```

```
    )

    # Обновление срока заявки
    success = update_request(extension.request_id, {
        "completion_date": extension.new_deadline,
        "request_status": "Срок продлен"
    })

    if success:
        # Логирование действия
        log_quality_action(
            user_id=current_user.user_id,
            action="extend_deadline",
            request_id=extension.request_id,
            notes=extension.reason
        )
        return {"message": "Срок заявки успешно продлен"}
    else:
        raise HTTPException(status_code=500, detail="Ошибка при обновлении заявки")

@app.post(
```

```
    "/quality/assign-specialist",
    summary="Привлечь дополнительного специалиста",
    dependencies=[Depends(require_roles("Quality Manager"))]
)
def assign_additional_specialist(
    assignment: SpecialistAssignment,
    current_user: UserBase = Depends(get_current_user)
):
    """Привлечение дополнительного специалиста к заявке"""
    # Проверка существования заявки и специалиста
    request_data = get_request_by_id(assignment.request_id)
    specialist_data = get_user_by_id(assignment.specialist_id)

    if not request_data:
        raise HTTPException(status_code=404, detail="Заявка не найдена")

    if not specialist_data or specialist_data["role"] != "Специалист":
        raise HTTPException(
            status_code=400,
            detail="Указанный пользователь не является специалистом"
        )
```

```
    # Добавление специалиста в таблицу дополнительных назначений
    add_specialist_assignment(
        request_id=assignment.request_id,
        specialist_id=assignment.specialist_id,
        assigned_by=current_user.user_id,
        reason=assignment.reason
    )

    # Отправка уведомления специалисту
    send_notification(
        user_id=assignment.specialist_id,
        title="Новое назначение",
        message=f"Вас назначили дополнительным специалистом на заявку #{assignment.request_id}"
    )

    return {"message": "Специалист успешно привлечен к заявке"}
```

2. **Интеграция QR-кода обратной связи:** Для оценки качества сервиса реализована генерация QR-кодов, ведущих на форму Google Forms. Выбор пал на библиотеку qrcode[pil] как на наиболее стабильное и простое решение для генерации QR-кодов в Python.

3. **Дополнительный функционал:**
Статистика с агрегатными функциями SQL:

```
def get_detailed_statistics(start_date: str = None, end_date: str = None) -> Dict:
    """
    Получение детальной статистики с фильтрацией по датам
    Использует агрегатные функции SQL для оптимизации
    """
    with get_db_cursor() as (cursor, _):
        query = """
            SELECT
                -- Общая статистика
                COUNT(*) as total_requests,
                SUM(CASE WHEN request_status IN ('Готова к выдаче', 'Завершена') THEN 1 ELSE
0 END) as completed_requests,

                -- Время выполнения
                AVG(
                    CASE
                        WHEN completion_date IS NOT NULL AND completion_date != ''
                        THEN julianday(completion_date) - julianday(start_date)
                        ELSE NULL
                    END
                ) as avg_completion_days,

                -- Распределение по типам оборудования
                climate_tech_type,
                COUNT(*) as equipment_count,
```

```
            -- Статистика по специалистам
            master_id,
            COUNT(*) as specialist_requests,
            SUM(CASE WHEN request_status IN ('Готова к выдаче', 'Завершена') THEN 1 ELSE
0 END) as specialist_completed

        FROM requests
        WHERE 1=1
    """

    params = []
    if start_date:
        query += " AND start_date >= ?"
        params.append(start_date)
    if end_date:
        query += " AND start_date <= ?"
        params.append(end_date)

    query += " GROUP BY climate_tech_type, master_id"

    cursor.execute(query, params)
    results = cursor.fetchall()

    # Обработка результатов
    stats = {
        "total_requests": 0,
```

```
            "completed_requests": 0,
            "avg_completion_days": None,
            "equipment_distribution": {},
            "specialist_performance": {}
        }

        for row in results:
            row_dict = dict(row)
            stats["total_requests"] = row_dict["total_requests"]
            stats["completed_requests"] = row_dict["completed_requests"]
            stats["avg_completion_days"] = row_dict["avg_completion_days"]

            # Распределение по оборудованию
            if row_dict["climate_tech_type"]:
                stats["equipment_distribution"][row_dict["climate_tech_type"]] = row_dict["equipment_count"]

            # Производительность специалистов
            if row_dict["master_id"]:
                stats["specialist_performance"][row_dict["master_id"]] = {
                    "total": row_dict["specialist_requests"],
                    "completed": row_dict["specialist_completed"],
                    "completion_rate": (row_dict["specialist_completed"] / row_dict["specialist_requests"] * 100
                                        if row_dict["specialist_requests"] > 0 else 0)
                }
```

```
    return stats
```

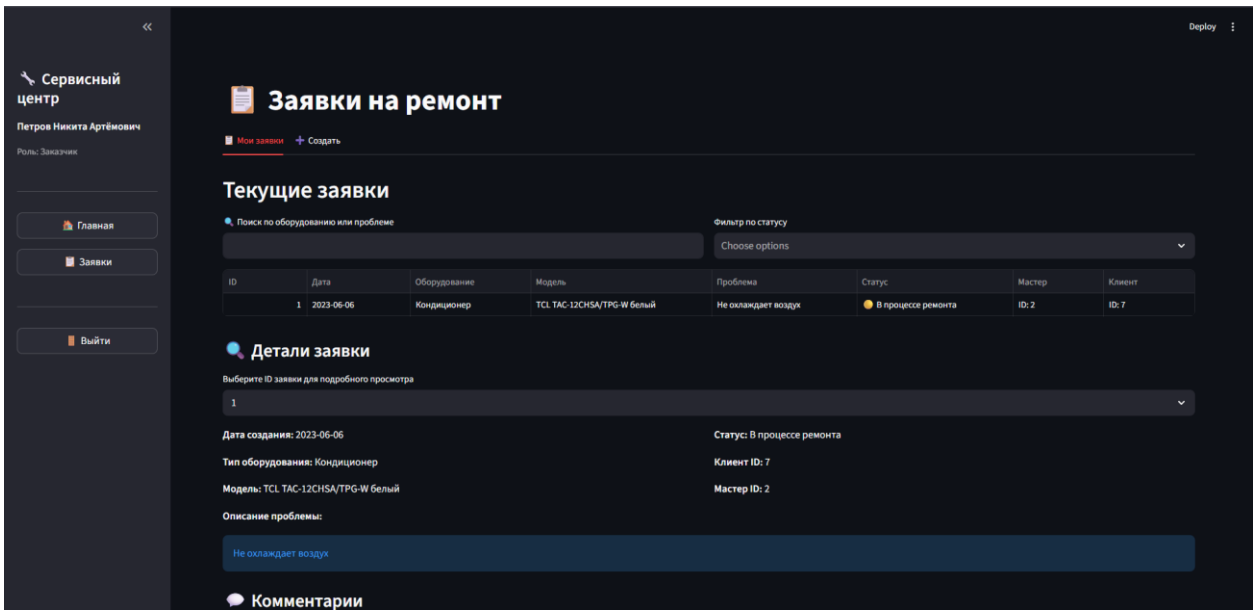
Безопасные эндпоинты для клиентов:

```
@app.get(
    "/client/my-requests",
    summary="Мои заявки (только для заказчиков)",
    dependencies=[Depends(require_roles("Заказчик"))]
)
def get_my_requests(current_user: UserBase = Depends(get_current_user)):
    """Получить все заявки текущего заказчика"""
    requests = models.get_requests_by_client(current_user.user_id)
    return requests


@app.get(
    "/client/request-status/{request_id}",
    summary="Статус моей заявки",
    dependencies=[Depends(require_roles("Заказчик"))]
)
def get_my_request_status(
    request_id: int,
    current_user: UserBase = Depends(get_current_user)
):
    """Получить статус конкретной заявки заказчика"""
    request_data = models.get_request_by_id(request_id)
    if not request_data:
        raise HTTPException(status_code=404, detail="Заявка не найдена")
```

```
if request_data["client_id"] != current_user.user_id:
    raise HTTPException(status_code=403, detail="Доступ запрещен")

# Возвращаем только необходимую информацию для клиента
return {
    "request_id": request_data["request_id"],
    "status": request_data["request_status"],
    "start_date": request_data["start_date"],
    "completion_date": request_data.get("completion_date"),
    "problem_description": request_data["problem_description"],
    "specialist_assigned": bool(request_data.get("master_id"))
}
```

4. **Качественные характеристики кода:** Использованы type hints для улучшения читаемости. Исключения обрабатываются с понятными сообщениями для пользователя.

QR-код со ссылкой на форму



Изолированный интерфейс заказчика

Код сообщения об ошибках с подсказками:

```
from typing import Dict, List, Optional, Tuple, Any

from enum import Enum

from datetime import datetime

import logging



# Настройка логирования

logging.basicConfig(

    level=logging.INFO,

    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s'

)

logger = logging.getLogger(__name__)



class ErrorSeverity(Enum):
```

```
    """Уровни серьезности ошибок"""
    INFO = "info"
    WARNING = "warning"
    ERROR = "error"
    CRITICAL = "critical"


class ErrorCategory(Enum):
    """Категории ошибок"""
    VALIDATION = "validation"
    AUTHENTICATION = "authentication"
    AUTHORIZATION = "authorization"
    DATABASE = "database"
    BUSINESS_LOGIC = "business_logic"
    SYSTEM = "system"
    NETWORK = "network"


class ErrorMessage:
    """Класс для хранения информации об ошибке"""

    def __init__(
        self,
```

```
    code: str,
    message: str,
    severity: ErrorSeverity,
    category: ErrorCategory,
    solution: str,
    details: Optional[Dict[str, Any]] = None,
    field: Optional[str] = None
):
    self.code = code
    self.message = message
    self.severity = severity
    self.category = category
    self.solution = solution
    self.details = details or {}
    self.field = field
    self.timestamp = datetime.now()

def to_dict(self) -> Dict[str, Any]:
    """Преобразование в словарь для ответа API"""
    return {
        "error": {
```

```
            "code": self.code,
            "message": self.message,
            "severity": self.severity.value,
            "category": self.category.value,
            "solution": self.solution,
            "field": self.field,
            "timestamp": self.timestamp.isoformat(),
            "details": self.details
        }
    }

def log(self):
    """Логирование ошибки"""
    log_message = f"{self.code}: {self.message} | Field: {self.field} | Details: {self.details}"

    if self.severity == ErrorSeverity.INFO:
        logger.info(log_message)
    elif self.severity == ErrorSeverity.WARNING:
        logger.warning(log_message)
    elif self.severity == ErrorSeverity.ERROR:
        logger.error(log_message)
```

```
        elif self.severity == ErrorSeverity.CRITICAL:
            logger.critical(log_message)


class ErrorMessages:
    """Хранилище всех сообщений об ошибках"""

    # Аутентификация и авторизация
    INVALID_CREDENTIALS = ErrorMessage(
        code="AUTH001",
        message="Неверный логин или пароль",
        severity=ErrorSeverity.ERROR,
        category=ErrorCategory.AUTHENTICATION,
        solution="Проверьте правильность введенных данных или воспользуйтесь восстановлением пароля",
        details={"max_attempts": 5}
    )

    TOKEN_EXPIRED = ErrorMessage(
        code="AUTH002",
        message="Срок действия токена истек",
        severity=ErrorSeverity.WARNING,
        category=ErrorCategory.AUTHENTICATION,
```

```
        solution="Пожалуйста, войдите в систему заново",
        details={"token_type": "JWT"}
    )

    ACCESS_DENIED = ErrorMessage(
        code="AUTH003",
        message="Доступ запрещен",
        severity=ErrorSeverity.ERROR,
        category=ErrorCategory.AUTHORIZATION,
        solution="Обратитесь к администратору для получения необходимых прав доступа",
        details={"required_role": None}
    )

    # Валидация данных пользователя
    USER_LOGIN_EXISTS = ErrorMessage(
        code="VAL001",
        message="Пользователь с таким логином уже существует",
        severity=ErrorSeverity.WARNING,
        category=ErrorCategory.VALIDATION,
        solution="Выберите другой логин или восстановите доступ к существующему аккаунту",
        field="login"
```

```
    )


    INVALID_PHONE_FORMAT = ErrorMessage(
        code="VAL002",
        message="Неверный формат номера телефона",
        severity=ErrorSeverity.WARNING,
        category=ErrorCategory.VALIDATION,
        solution="Введите номер телефона в формате +7XXXXXXXXXX или 8XXXXXXXXXX",
        field="phone",
        details={"valid_formats": ["+7XXXXXXXXXX", "8XXXXXXXXXX"]}
    )


    PASSWORD_TOO_WEAK = ErrorMessage(
        code="VAL003",
        message="Пароль слишком слабый",
        severity=ErrorSeverity.WARNING,
        category=ErrorCategory.VALIDATION,
        solution="Пароль должен содержать минимум 8 символов, включая заглавные буквы, цифры и специальные символы",
        field="password",
        details={
```

```
        "min_length": 8,
        "requirements": ["uppercase", "lowercase", "digits", "special_chars"]
    }
)

REQUIRED_FIELD_MISSING = ErrorMessage(
    code="VAL004",
    message="Обязательное поле не заполнено",
    severity=ErrorSeverity.ERROR,
    category=ErrorCategory.VALIDATION,
    solution="Заполните все обязательные поля, отмеченные звездочкой (*)",
    field=None
)

# Валидация заявок
INVALID_DATE_FORMAT = ErrorMessage(
    code="VAL101",
    message="Неверный формат даты",
    severity=ErrorSeverity.ERROR,
    category=ErrorCategory.VALIDATION,
    solution="Используйте формат даты YYYY-MM-DD (например, 2024-01-15)",
```

```
        field="start_date",
        details={"expected_format": "YYYY-MM-DD"}
    )

    PROBLEM_DESCRIPTION_TOO_SHORT = ErrorMessage(
        code="VAL102",
        message="Описание проблемы слишком короткое",
        severity=ErrorSeverity.WARNING,
        category=ErrorCategory.VALIDATION,
        solution="Опишите проблему более подробно (минимум 10 символов)",
        field="problem_description",
        details={"min_length": 10, "current_length": None}
    )

    INVALID_EQUIPMENT_TYPE = ErrorMessage(
        code="VAL103",
        message="Недопустимый тип оборудования",
        severity=ErrorSeverity.WARNING,
        category=ErrorCategory.VALIDATION,
        solution="Выберите тип оборудования из списка: Кондиционер, Увлажнитель, Осушитель, Вентиляция",
        field="climate_tech_type",
```

```
        details={
            "allowed_types": ["Кондиционер", "Увлажнитель", "Осушитель", "Вентиляция",
"Обогреватель"]
        }
    )


    INVALID_STATUS_TRANSITION = ErrorMessage(
        code="VAL104",
        message="Недопустимое изменение статуса",
        severity=ErrorSeverity.WARNING,
        category=ErrorCategory.VALIDATION,
        solution="Статус заявки можно изменить только по установленному workflow",
        field="request_status",
        details={
            "current_status": None,
            "requested_status": None,
            "allowed_transitions": {
                "Новая заявка": ["В процессе ремонта", "Ожидание комплектующих"],
                "В процессе ремонта": ["Ожидание комплектующих", "Готова к выдаче"],
                "Ожидание комплектующих": ["В процессе ремонта", "Готова к выдаче"],
                "Готова к выдаче": ["Завершена"]
            }
```

```
        }
    )

    # Работа с данными
    REQUEST_NOT_FOUND = ErrorMessage(
        code="DATA001",
        message="Заявка не найдена",
        severity=ErrorSeverity.ERROR,
        category=ErrorCategory.BUSINESS_LOGIC,
        solution="Проверьте правильность ID заявки или создайте новую заявку",
        details={"request_id": None}
    )

    USER_NOT_FOUND = ErrorMessage(
        code="DATA002",
        message="Пользователь не найден",
        severity=ErrorSeverity.ERROR,
        category=ErrorCategory.BUSINESS_LOGIC,
        solution="Проверьте правильность ID пользователя или обратитесь к администратору",
        details={"user_id": None}
    )
```

```
    SPECIALIST_NOT_ASSIGNED = ErrorMessage(
        code="DATA003",
        message="Специалист не назначен на заявку",
        severity=ErrorSeverity.WARNING,
        category=ErrorCategory.BUSINESS_LOGIC,
        solution="Назначьте специалиста через меню 'Изменить заявку' или обратитесь к менеджеру",
        details={"request_id": None}
    )

    CANNOT_DELETE_COMPLETED_REQUEST = ErrorMessage(
        code="DATA004",
        message="Нельзя удалить завершенную заявку",
        severity=ErrorSeverity.WARNING,
        category=ErrorCategory.BUSINESS_LOGIC,
        solution="Архивируйте заявку вместо удаления или измените ее статус",
        details={"request_status": "Завершена"}
    )

    # База данных
    DATABASE_CONNECTION_ERROR = ErrorMessage(
```

```
    code="DB001",
    message="Ошибка подключения к базе данных",
    severity=ErrorSeverity.CRITICAL,
    category=ErrorCategory.DATABASE,
    solution="Попробуйте позже или обратитесь к системному администратору",
    details={"database": "repair_requests.db"}
)

DATABASE_TIMEOUT = ErrorMessage(
    code="DB002",
    message="Превышено время ожидания ответа от базы данных",
    severity=ErrorSeverity.WARNING,
    category=ErrorCategory.DATABASE,
    solution="Упростите запрос или попробуйте позже",
    details={"timeout_seconds": 30}
)

CONSTRAINT_VIOLATION = ErrorMessage(
    code="DB003",
    message="Нарушение целостности данных",
    severity=ErrorSeverity.ERROR,
```

```
        category=ErrorCategory.DATABASE,
        solution="Проверьте корректность связанных данных (например, существование пользователя)",
        details={"constraint": None}
    )

    # Системные ошибки
    SYSTEM_MAINTENANCE = ErrorMessage(
        code="SYS001",
        message="Система на техническом обслуживании",
        severity=ErrorSeverity.INFO,
        category=ErrorCategory.SYSTEM,
        solution="Пожалуйста, повторите попытку через 30 минут",
        details={"maintenance_window": "23:00-03:00"}
    )

    RATE_LIMIT_EXCEEDED = ErrorMessage(
        code="SYS002",
        message="Превышено количество запросов",
        severity=ErrorSeverity.WARNING,
        category=ErrorCategory.SYSTEM,
        solution="Подождите 1 минуту перед следующей попыткой",
```

```
        details={"limit": 100, "period": "минута"}
    )

    FILE_UPLOAD_ERROR = ErrorMessage(
        code="SYS003",
        message="Ошибка при загрузке файла",
        severity=ErrorSeverity.ERROR,
        category=ErrorCategory.SYSTEM,
        solution="Проверьте размер и формат файла (макс. 10MB, допустимы: JPG, PNG, PDF)",
        details={"max_size_mb": 10, "allowed_formats": ["jpg", "png", "pdf"]}
    )

    # Статистика и отчеты
    NO_DATA_FOR_PERIOD = ErrorMessage(
        code="STAT001",
        message="Нет данных за выбранный период",
        severity=ErrorSeverity.INFO,
        category=ErrorCategory.BUSINESS_LOGIC,
        solution="Измените период или создайте тестовые данные для анализа",
        details={"start_date": None, "end_date": None}
    )
```

```
INVALID_STATISTICS_PARAMS = ErrorMessage(
    code="STAT002",
    message="Неверные параметры для статистики",
    severity=ErrorSeverity.WARNING,
    category=ErrorCategory.VALIDATION,
    solution="Укажите корректные даты и тип статистики",
    field="statistics_type",
    details={"allowed_types": ["completed", "average_time", "problems", "all"]}
)

@classmethod
def get_by_code(cls, code: str) -> Optional[ErrorMessage]:
    """Получить сообщение об ошибке по коду"""
    for attr_name in dir(cls):
        if not attr_name.startswith('_'):
            attr = getattr(cls, attr_name)
            if isinstance(attr, ErrorMessage) and attr.code == code:
                return attr
    return None
```

```
    @classmethod
    def get_by_field(cls, field: str) -> List[ErrorMessage]:
        """Получить все сообщения об ошибках для конкретного поля"""
        errors = []
        for attr_name in dir(cls):
            if not attr_name.startswith('_'):
                attr = getattr(cls, attr_name)
                if isinstance(attr, ErrorMessage) and attr.field == field:
                    errors.append(attr)
        return errors


class ErrorResponseBuilder:
    """Построитель детализированных ответов об ошибках"""

    @staticmethod
    def build_field_validation_error(
        field: str,
        value: Any,
        error_type: str,
        additional_details: Dict[str, Any] = None
    ) -> Dict[str, Any]:
```

```
    """Построить ошибку валидации поля"""
    details = additional_details or {}
    details["field_value"] = value

    if error_type == "required":
        error = ErrorMessages.REQUIRED_FIELD_MISSING
        error.field = field
        error.details = details

    elif error_type == "phone_format":
        error = ErrorMessages.INVALID_PHONE_FORMAT
        error.field = field
        error.details = details

    elif error_type == "password_weak":
        error = ErrorMessages.PASSWORD_TOO_WEAK
        error.field = field
        error.details = details

    elif error_type == "date_format":
        error = ErrorMessages.INVALID_DATE_FORMAT
```

```
    error.field = field
    error.details = details

elif error_type == "description_length":
    error = ErrorMessages.PROBLEM_DESCRIPTION_TOO_SHORT
    error.field = field
    error.details["current_length"] = len(str(value))
    error.details.update(details)

elif error_type == "equipment_type":
    error = ErrorMessages.INVALID_EQUIPMENT_TYPE
    error.field = field
    error.details.update(details)

else:
    # Общая ошибка валидации
    error = ErrorMessage(
        code="VAL999",
        message=f"Ошибка валидации поля '{field}'",
        severity=ErrorSeverity.ERROR,
        category=ErrorCategory.VALIDATION,
```

```
            solution="Проверьте правильность введенных данных",
            field=field,
            details=details
        )

    error.log()
    return error.to_dict()

@staticmethod
def build_business_logic_error(
    entity_type: str,
    entity_id: Any,
    action: str,
    reason: str
) -> Dict[str, Any]:
    """Построить ошибку бизнес-логики"""
    if entity_type == "request" and action == "not_found":
        error = ErrorMessages.REQUEST_NOT_FOUND
        error.details["request_id"] = entity_id

    elif entity_type == "user" and action == "not_found":
```

```
    error = ErrorMessages.USER_NOT_FOUND
    error.details["user_id"] = entity_id

elif entity_type == "request" and action == "delete_completed":
    error = ErrorMessages.CANNOT_DELETE_COMPLETED_REQUEST

else:
    error = ErrorMessage(
        code="BIZ999",
        message=f"Ошибка при выполнении операции '{action}' над {entity_type}",
        severity=ErrorSeverity.ERROR,
        category=ErrorCategory.BUSINESS_LOGIC,
        solution="Проверьте корректность операции и данных",
        details={
            "entity_type": entity_type,
            "entity_id": entity_id,
            "action": action,
            "reason": reason
        }
    )
```

```
    error.log()
    return error.to_dict()

@staticmethod
def build_auth_error(
    error_type: str,
    additional_details: Dict[str, Any] = None
) -> Dict[str, Any]:
    """Построить ошибку аутентификации/авторизации"""
    details = additional_details or {}

    if error_type == "invalid_credentials":
        error = ErrorMessages.INVALID_CREDENTIALS

    elif error_type == "token_expired":
        error = ErrorMessages.TOKEN_EXPIRED

    elif error_type == "access_denied":
        error = ErrorMessages.ACCESS_DENIED
        if "required_role" in details:
            error.details["required_role"] = details["required_role"]
```

```
        else:
            error = ErrorMessage(
                code="AUTH999",
                message="Ошибка аутентификации",
                severity=ErrorSeverity.ERROR,
                category=ErrorCategory.AUTHENTICATION,
                solution="Проверьте ваши учетные данные и права доступа",
                details=details
            )

        error.log()
        return error.to_dict()

class ErrorHandler:
    """Обработчик ошибок с пользовательскими подсказками"""

    def __init__(self):
        self.errors = []
        self.has_critical = False
```

```
def add_error(self, error_message: ErrorMessage):
    """Добавить ошибку в обработчик"""
    self.errors.append(error_message)
    if error_message.severity == ErrorSeverity.CRITICAL:
        self.has_critical = True


def add_field_error(
    self,
    field: str,
    value: Any,
    error_type: str,
    additional_details: Dict[str, Any] = None
):
    """Добавить ошибку валидации поля"""
    error_dict = ErrorResponseBuilder.build_field_validation_error(
        field, value, error_type, additional_details
    )
    # Создаем ErrorMessage из словаря
    error_data = error_dict["error"]
    error = ErrorMessage(
        code=error_data["code"],
```

```
            message=error_data["message"],
            severity=ErrorSeverity(error_data["severity"]),
            category=ErrorCategory(error_data["category"]),
            solution=error_data["solution"],
            details=error_data["details"],
            field=error_data["field"]
        )
        self.add_error(error)

    def has_errors(self) -> bool:
        """Проверить наличие ошибок"""
        return len(self.errors) > 0

    def get_response(self) -> Dict[str, Any]:
        """Получить структурированный ответ с ошибками"""
        if not self.errors:
            return {}

        # Группируем ошибки по полям
        field_errors = {}
        general_errors = []
```

```
for error in self.errors:
    if error.field:
        if error.field not in field_errors:
            field_errors[error.field] = []
        field_errors[error.field].append(error.to_dict()["error"])
    else:
        general_errors.append(error.to_dict()["error"])

response = {
    "success": False,
    "timestamp": datetime.now().isoformat(),
    "error_count": len(self.errors)
}

if field_errors:
    response["field_errors"] = field_errors

if general_errors:
    response["general_errors"] = general_errors
```

```
        # Добавляем общие рекомендации
        response["recommendations"] = self._generate_recommendations()

        return response

    def _generate_recommendations(self) -> List[str]:
        """Сгенерировать общие рекомендации на основе ошибок"""
        recommendations = []
        has_validation_errors = False
        has_auth_errors = False
        has_system_errors = False

        for error in self.errors:
            if error.category == ErrorCategory.VALIDATION:
                has_validation_errors = True
            elif error.category in [ErrorCategory.AUTHENTICATION,
ErrorCategory.AUTHORIZATION]:
                has_auth_errors = True
            elif error.category in [ErrorCategory.SYSTEM, ErrorCategory.DATABASE]:
                has_system_errors = True

        if has_validation_errors:
```

```
            recommendations.append("Проверьте правильность заполнения всех полей формы")
            recommendations.append("Убедитесь, что введенные данные соответствуют требованиям")

        if has_auth_errors:
            recommendations.append("Проверьте ваши учетные данные (логин и пароль)")
            recommendations.append("Убедитесь, что у вас есть необходимые права доступа")

        if has_system_errors:
            recommendations.append("Попробуйте повторить операцию через несколько минут")
            recommendations.append("Если проблема повторяется, обратитесь в техническую поддержку")

        # Общая рекомендация
        if not recommendations:
            recommendations.append("Пожалуйста, исправьте указанные ошибки и повторите попытку")

        return recommendations

    def clear(self):
        """Очистить все ошибки"""
```

```
        self.errors = []
        self.has_critical = False

# Создаем глобальный обработчик ошибок
error_handler = ErrorHandler()

# Декораторы для обработки ошибок
def handle_errors(func):
    """Декоратор для автоматической обработки ошибок в функциях"""
    def wrapper(*args, **kwargs):
        try:
            return func(*args, **kwargs)
        except Exception as e:
            error_handler.clear()

            # Определяем тип ошибки и создаем соответствующее сообщение
            error_type = type(e).__name__

            if "sqlite3" in error_type.lower() or "database" in str(e).lower():
                error = ErrorMessages.DATABASE_CONNECTION_ERROR
                error.details["exception"] = str(e)
```

```
        elif "validation" in str(e).lower():
            error = ErrorMessage(
                code="VAL999",
                message="Ошибка валидации данных",
                severity=ErrorSeverity.ERROR,
                category=ErrorCategory.VALIDATION,
                solution="Проверьте правильность введенных данных",
                details={"exception": str(e)}
            )

        else:
            error = ErrorMessage(
                code="SYS999",
                message="Внутренняя ошибка системы",
                severity=ErrorSeverity.ERROR,
                category=ErrorCategory.SYSTEM,
                solution="Пожалуйста, повторите попытку позже или обратитесь в поддержку",
                details={"exception": str(e), "error_type": error_type}
            )
```

```
            error_handler.add_error(error)
            return error_handler.get_response()

    return wrapper

def validate_request_data(data: Dict[str, Any], required_fields: List[str]) -> bool:
    """
    Валидация данных запроса с детальными сообщениями об ошибках

    Args:
        data: Данные для валидации
        required_fields: Список обязательных полей

    Returns:
        True если валидация прошла успешно, False в противном случае
    """
    error_handler.clear()

    # Проверка обязательных полей
    for field in required_fields:
        if field not in data or data[field] is None or str(data[field]).strip() == "":
```

```
            error_handler.add_field_error(field, data.get(field), "required")


    # Специфические проверки для заявок
    if "problem_description" in data:
        desc = str(data["problem_description"])
        if len(desc) < 10:
            error_handler.add_field_error(
                "problem_description",
                desc,
                "description_length",
                {"current_length": len(desc)}
            )

        # Проверка на запрещенные слова
        forbidden_words = ["взрыв", "пожар", "кража", "терроризм"]
        for word in forbidden_words:
            if word in desc.lower():
                error_handler.add_error(ErrorMessage(
                    code="VAL105",
                    message=f"Описание содержит запрещенное слово",
                    severity=ErrorSeverity.ERROR,
```

```
                category=ErrorCategory.VALIDATION,
                solution="Удалите запрещенные слова из описания",
                field="problem_description",
                details={"forbidden_word": word}
            ))

    if "phone" in data:
        phone = str(data["phone"])
        if not phone.startswith(('+7', '8', '7')) or len(phone.replace('+', '')) != 11:
            error_handler.add_field_error("phone", phone, "phone_format")

    if "start_date" in data:
        from datetime import datetime
        try:
            datetime.strptime(str(data["start_date"]), "%Y-%m-%d")
        except ValueError:
            error_handler.add_field_error("start_date", data["start_date"], "date_format")

    return not error_handler.has_errors()
```