ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К КУРСОВОЙ РАБОТЕ ПО ДИСЦИПЛИНЕ «БАЗЫ ДАННЫХ»
НА ТЕМУ: «ИНФОРМАЦИОННАЯ СИСТЕМА ПЛАНИРОВАНИЯ И УЧЁТА РЕМОНТА ЖИЛЫХ ПОМЕЩЕНИЙ»

Выполнил(а) студент

группы М8О-308Б-23

Фролов Вячеслав Витальевич

Проверил: *Грубенко Максим Дмитриевич*

Москва – 2025 г.

СОДЕРЖАНИЕ

1. Введение

В данной пояснительной записке представляется проект информационной системы для планирования и учёта ремонта жилых помещений. Актуальность проекта обусловлена потребностью автоматизировать и систематизировать процесс ремонта квартир и домов – от планирования этапов и задач до учёта материалов и контроля качества выполненных работ.

Цель работы – спроектировать и реализовать базу данных и прикладное программное обеспечение для комплексного управления процессом ремонта. В ходе реализации необходимо обеспечить хранение информации о проектах ремонта, их этапах, задачах и исполнителях, вести учёт материалов и оборудования, регистрировать заключение договоров с подрядчиками, а также фиксировать дефекты, переделки и результаты приёмки работ.

2. Аналитическая часть

Обзор предметной области.

Процесс капитального или косметического ремонта жилья включает планирование работ, выполнение множества разнообразных задач (демонтаж, отделка, электротехнические работы, монтаж оборудования и т.д.), привлечение различных специалистов и подрядчиков, закупку материалов, а также контроль качества и приёмку выполненных работ. Это затрудняет координацию участников проекта и повышает риск ошибок – пропуска каких-либо работ, превышения бюджета, несвоевременной закупки материалов или незаметного для руководства появления дефектов.

Информационная система планирования и учёта ремонта призвана централизовать и взаимно связать все данные о ремонтном проекте. Система должна предоставлять следующие основные возможности: – Управление объектами ремонта. Регистрация объектов недвижимости (квартир, домов), их

характеристик и структуры (например, разбиение на помещения/комнаты). У каждого объекта есть владелец (заказчик ремонта) с контактными данными.; Ведение проектов и этапов работ. Для каждого объекта формируется проект ремонта. Проект разбивается на этапы (фазы) – логические блоки работ (например, подготовительные работы, черновая отделка, чистовая отделка).

Формулировка задачи.

На основании анализа требований необходимо создать реляционную базу данных, удовлетворяющую перечисленным потребностям.

В рамках курсового проекта требуется также разработать минимальный backend-сервис, который будет взаимодействовать с БД. Этот сервис предоставит RESTful API для основных операций: добавление и изменение данных (например, создание нового проекта, обновление статуса задачи), извлечение данных (список задач по проекту, отчёты по материалам и др.), а также выполнение специальных операций – например, запуск пакетной загрузки данных. Таким образом, итоговая система представляет собой двухкомпонентное приложение (PostgreSQL база данных + FastAPI сервер приложений), способное продемонстрировать функционирование всех необходимых механизмов.

3. Проектная часть

3.1. Архитектура системы

Архитектура системы реализована по принципу клиент-серверного приложения с выделенным уровнем базы данных. В качестве СУБД выбрана PostgreSQL, благодаря её соответствию требованиям по надежности хранения данных, поддержке необходимых типов (JSONB, схемы, хранимые функции и т.д.) и возможностей расширения (например, генерации UUID, full-text search при расширении, но в данном проекте основное – стандартный SQL).

Ниже представлена схема взаимодействия основных компонентов системы:

- СУБД (PostgreSQL): Хранит все данные системы. Внутри СУБД реализована бизнес-логика: ограничения целостности, связи, представления, функции, триггеры для аудита и пр.
- Сервер приложений (FastAPI): Отвечает за прикладную логику и обмен данными с клиентом. В данном проекте он играет роль прослойки между пользователем и базой данных, выполняя в основном две функции: (1) валидация и обработка входящих данных API-запросов, вызов соответствующих SQL-операций (набор CRUD операций, запуск процедур и функций в БД) и (2) предоставление удобного интерфейса (документирование API, формирование ответов в формате JSON).
- Клиент (интерфейс): В рамках курсового проекта отдельный пользовательский интерфейс не разрабатывался, взаимодействие происходит посредством Swagger UI (веб-страница с описанием и тестированием методов API) и через непосредственные вызовы API (например, с использованием утилит curl или программ-клиентов). Таким образом, для демонстрации

используется автоматически сгенерированный интерфейс Swagger и предварительно заготовленные запросы.

Особенностью архитектуры является использование Docker для изоляции среды: PostgreSQL контейнер и API контейнер обмениваются данными во внутренней сети. Это облегчает повторяемость запуска: достаточно одной команды (docker compose up), чтобы развернуть всю систему. Такое разбиение на сервисы соответствует принципам микросервисной архитектуры, хотя в данном случае у нас один сервис приложений и один сервис БД.

3.2. Проектирование базы данных (ER-диаграмма)

В данном разделе приводится краткое описание назначения каждой таблицы базы данных, её основных полей и связей. (Подробная структура – в Приложении, листинг DDL-скриптов.)

3.3. Описание таблиц и связей предметной области

Ниже приведён краткий перечень основных таблиц и их назначения. Полный DDL и определения объектов БД приведены в приложениях.

- users — хранит учетные записи пользователей системы. Ключевые поля: user_id (PK).
- owners — собственники объектов недвижимости (заказчики ремонта). Ключевые поля: created_at (дата регистрации владельца в системе). PK – owner_id. Типичные ограничения: email, owner_id (PK).
- properties — объекты недвижимости (адреса, квартиры/дома). Ключевые поля: property_id (PK), owner_id (FK → owners), status (состояние объекта в контексте ремонта: активен – ремонтируется, updated_at (время последнего обновления записи).
- property_rooms — помещения/комнаты внутри объекта. Ключевые поля: room_id (PK), property_id (FK → properties).

- projects — проекты ремонта. Ключевые поля: project_id (PK), property_id (FK → properties), contract_id (FK → contracts, status (см. выше, planned_start_date.
- project_phases — этапы проекта. Ключевые поля: phase_id (PK), project_id (FK → projects), status (как у проекта)., planned_start_date, planned_end_date.
- work_types — виды работ. Ключевые поля: work_type_id (PK).
- project_tasks — задачи проекта (работы). Ключевые поля: task_id (PK), project_id (FK → projects), phase_id (FK → project_phases, room_id (FK → property_rooms, work_type_id (FK → work_types).
- acceptance_acts — акты приёмки. Ключевые поля: act_id (PK), task_id (FK → project_tasks), result_status (результат: accepted/rejected/partial), act_date (дата приёмки).
- defects — дефекты. Ключевые поля: defect_id (PK), task_id (FK → project_tasks), contractor_id (FK → contractors, status ('open', defect_date (дата обнаружения).
- contractors — подрядчики. Ключевые поля: contractor_id (PK).
- contracts — договоры. Ключевые поля: contract_id (PK), property_id (FK → properties), contractor_id (FK → contractors), status ('draft', start_date.
- materials — справочник материалов. Ключевые поля: material_id (PK), price (возможно.
- suppliers — поставщики. Ключевые поля: supplier_id (PK).
- purchase_orders — заказы на материалы. Ключевые поля: po_id (PK), project_id (FK → projects), supplier_id (FK → suppliers), status ('draft', order_date.

- purchase_order_items — позиции (строки) заказа. Ключевые поля: po_item_id (PK), po_id (FK → purchase_orders), material_id (FK → materials), unit_price.
- inventory_transactions — складские транзакции материалов. Ключевые поля: tx_id (PK), project_id (FK → projects), material_id (FK → materials), task_id (FK → project_tasks, po_item_id (FK → purchase_order_items.
- audit_log — журнал аудита. Ключевые поля: audit_id (PK), user_id (FK → users, action_type (тип действия: INSERT/UPDATE/DELETE).
- import_runs — запуски пакетного импорта. Ключевые поля: run_id (PK), user_id (FK → users, status ('running'.
- import_errors — ошибки при импорте. Ключевые поля: error_id (PK), run_id (FK → import_runs), FK → users.

Каждая из перечисленных таблиц отвечает за свой аспект данных. В сумме схема включает 18 таблиц основной функциональности и 2 вспомогательных (импорт) – итого 20 таблиц,

Присутствие специальной таблицы audit_log обеспечивает выполнение требования по ведению журнала изменений: все ключевые действия с данными будут автоматически фиксироваться в ней (подробности механизма – в разделе 3.8).

3.4. Ограничения целостности данных

В проектируемой базе данных реализован полный набор ограничений целостности, гарантирующих непротиворечивость и обоснованность данных на уровне СУБД: – Первичные ключи (PRIMARY KEY) – в каждой таблице определён один атрибут (или составной набор атрибутов), уникально идентифицирующий запись. Как правило, в наших таблицах используются суррогатные ключи типа BIGSERIAL (автоинкремент). Например, project_id

является PK для таблицы projects, task_id для project_tasks и т.д. Это обеспечивает уникальность каждой строки и быстрый доступ по ключу;

Внешние ключи (FOREIGN KEY) – устанавливают связи между таблицами и обеспечивают ссылочную целостность. Например, поле project_tasks.project_id является FK, ссылающимся на projects.project_id – что не позволяет создать задачу, относящуюся к несуществующему проекту. Аналогично, contracts.property_id ссылается на существующий объект недвижимости. В DDL скриптах для каждого FK указаны правила ON UPDATE/ON DELETE, определяющие поведение при изменении или удалении родительской записи;

Связи типа “владелец-деталь” обычно используют ON DELETE CASCADE, если логично удалять зависимые данные вместе с главным. Применено, например, для удаления всех комнат при удалении объекта недвижимости, всех этапов и задач при удалении проекта, позиций при удалении заказа, транзакций при удалении проекта и т.п. Это предотвращает ситуацию, когда в базе остаются “висящие” записи, не имеющие главного объекта.;

Связи, где удаление главной записи не должно происходить пока существуют зависимые (чтобы не потерять важные данные), определены как ON DELETE RESTRICT. Например, нельзя удалить владельца, пока есть объекты недвижимости; нельзя удалить подрядчика, если с ним есть договор; нельзя удалить вид работ, если существуют связанные задачи; и др. В таких случаях сначала необходимо удалить или пересвязать зависимые записи, и только потом можно удалить главную сущность..

Применение всех указанных ограничений обеспечивает соблюдение как сущностной целостности (каждая запись имеет уникальный ключ, ссылки

всегда указывают на существующие записи, нет “висящих ссылок”), так и предметной целостности (данные удовлетворяют правилам предметной области: корректные значения и взаимосвязи).

В случае попытки нарушить ограничение СУБД вернёт ошибку, предотвращая некорректную операцию. Например, попытка удалить объект с существующими проектами приведёт к ошибке из-за FK RESTRICT; попытка записать задаче отрицательную стоимость будет отклонена CHECK-ограничением и т.д.

3.5. Индексы и оптимизация запросов

Для обеспечения эффективной работы с данными были проанализированы основные предполагаемые сценарии использования и на их основе добавлены индексы. Индексы в PostgreSQL используются для ускорения поиска и сортировки – они особенно важны для полей, по которым происходит фильтрация (в WHERE), соединение таблиц (JOIN ON) или упорядочение (ORDER BY).

По умолчанию при создании первичного ключа СУБД уже создаёт уникальный индекс на поле PK. Таким образом, все наши ключевые идентификаторы (user_id, project_id, task_id etc.) автоматически индексированы – это ускоряет поиск записи по конкретному ключу и выполнение JOIN-ов по ключу (например, соединение tasks с project по project_id будет эффективно, если project_id – PK в projects, а tasks по нему фильтруются).

Индексы нужны, чтобы ускорить соединение таблиц при запросах, когда, например, выбираются все задачи проекта (фильтр tasks.project_id = …) или все транзакции по материалу.; Составные индексы для часто встречающихся сочетаний условий; project_tasks(project_id, status) – ускоряет запросы, которые выводят задачи определённого проекта с фильтром по статусу. Например,

“покажи все невыполненные задачи проекта №X” – без такого индекса пришлось бы просканировать все задачи проекта и фильтровать по статусу, а с индексом отберутся сразу нужные записи.; inventory_transactions(project_id, material_id, transaction_date) – индекс для запросов по материалам в рамках проекта, отсортированных по дате. Например, отчёт “история списания материала Y по проекту X” с сортировкой по дате сможет воспользоваться этим индексом (причём он покрывает и фильтрацию по X и Y, и сортировку по date)..

Улучшение производительности.

Чтобы продемонстрировать поведение запросов, выполнены анализы планов выполнения (EXPLAIN ANALYZE) для целевых выборок (примеры приведены ниже).

Таблица 1 — EXPLAIN ANALYZE

Запрос (сценарий)	Тип плана	Execution Time	Ключевые признаки	Комментарий
Транзакции по project_id=1 и material_id=1, сортировка по transaction_date DESC	Seq Scan + Sort	0.826 ms	Rows Removed: 7996; Buffers hit=100	Пример без подходящего индекса; возможен индекс (project_id, material_id, transaction_date DESC).
Задачи	Index Scan	0.586 ms	Rows	Индекс по

Запрос (сценарий)	Тип плана	Execution Time	Ключевые признаки	Комментарий
project_id=1 со статусом in_progress	(idx_tasks_project_id)		Removed: 1; Buffers hit=6	project_id используется; при частой фильтрации по status полезен (project_id, status).

Примеры результатов EXPLAIN ANALYZE (фрагменты):

Пример 1 — выборка последних транзакций по project_id=1 и material_id=1 с сортировкой по дате.

Seq Scan

Planning Time: 1.074 ms

Execution Time: 0.826 ms

(14 rows)

Пример 2 — выборка задач проекта project_id=1 со статусом in_progress.

Index Scan

Planning Time: 2.164 ms

Execution Time: 0.586 ms

(9 rows)

Таким образом, требование оптимизации запросов выполнено: проанализированы и индексированы поля, используемые в критических запросах (WHERE/JOIN/ORDER BY).

3.6. Представления данных

Для удобства получения сводной информации и упрощения сложных запросов в базе данных созданы несколько VIEW (представлений). Представление – это сохранённый запрос (виртуальная таблица), который объединяет и агрегирует данные из нескольких таблиц, предоставляя пользователю “плоскую” структуру, как если бы это была таблица.

В рамках проекта реализовано не менее 7 представлений для различных аналитических задач. Опишем основные из них:

- v_projects_overview – представление обзора проектов.

Здесь мы получаем по каждой строке проекта: данные проекта (название, даты, статус, бюджет), адрес объекта и владелец, а также номер договора и подрядчик, если проект связан с договором. Это представление удобно для вывода списка проектов “в одном месте” – например, менеджеру для контроля (видно сразу, кто владелец, какой адрес, какой подрядчик отвечает и статус проекта)

- v_project_progress – представление хода выполнения проекта. Это агрегат по задачам проекта, показывающий количество задач в разных статусах и процент выполнения.

Это представление по каждому проекту даёт: общее число задач и разбивку по статусам, плюс вычисляемый процент выполнения (отношение выполненных к общему, в процентах с точностью 0.1%). Использован синтаксис FILTER для подсчёта с условием (альтернатива SUM(CASE WHEN ...

Данное представление используется, например, для мониторинга – какие проекты близки к завершению, а какие отстают.

Таблица 2 — Пример вывода представления v_project_progress

project_id	project_name	tasks_total	tasks_planned	tasks_in_progress	tasks_blocked	tasks_completed	tasks_cancelled	pct_completed
1	Ремонт квартиры (ул. Примерная)	7	0	6	0	1	0	14.29
2	Ремонт квартиры (пр-т Новаторов)	7	0	5	0	2	0	28.57
3	Проект BIG 000001	2	0	2	0	0	0	0.00
4	Проект BIG	2	0	2	0	0	0	0.00

	000002							
5	Проект BIG 000003	2	0	2	0	0	0	0.00

- v_tasks_detailed – представление детализированного списка задач. Оно объединяет задачу с проектом, этапом, комнатой, видом работ, подрядчиком и актом приёмки (если есть).

Таким образом, v_tasks_detailed практически собирает все аспекты задачи. Его можно фильтровать по проекту, по подрядчику, по статусу и т.д., не заботясь о соединениях – удобство использования.

3.7. Функции и хранимые процедуры

В дополнение к представлениям, для реализации логики на стороне базы данных созданы хранимые функции двух типов: скалярные (возвращающие одно значение) и табличные (возвращающие набор строк). PostgreSQL поддерживает оба вида, и мы их используем для инкапсуляции повторно используемой логики и удобства вызова из приложения.

Скалярные функции: - fn_project_total_spent(p_project_id BIGINT) -> NUMERIC(14,2) – функция вычисляет, сколько всего денежных средств потрачено по проекту. Она агрегирует фактические затраты задач, расходы материалов и стоимость переделок по дефектам. Реализация (на SQL) суммирует: - сумму actual_cost всех задач данного проекта, - плюс сумму (quantity * unit_price) всех расходных транзакций (transaction_type = 'OUT') по данному проекту, - плюс сумму rework_cost всех дефектов данного проекта.

Все три составляющих суммируются и округляются до 2 знаков. Если чего-то нет – берётся 0 через COALESCE.

Эта функция позволяет одним вызовом получить сводный показатель “Фактические затраты проекта” с учётом всех компонентов. Она обозначена как STABLE (т.е. при одинаковых входных не меняет результат, что обычно, так как просто читает данные).

Таблица 3 — Пример использования функции fn_project_total_spent(project_id)

project_id	project_name	total_spent
1	Ремонт квартиры (ул. Примерная)	23017.00
2	Ремонт квартиры (пр-т Новаторов)	41464.00
3	Проект BIG 000001	8412.00
4	Проект BIG 000002	11552.00
5	Проект BIG 000003	8612.00

- fn_contractor_defect_rate(p_contractor_id BIGINT) -> NUMERIC(5,2) – (в качестве примера) можно реализовать функцию вычисления показателя качества подрядчика: доля задач, по которым были выявлены дефекты. Эта функция не была явно указана, но как идея: она могла считать процент дефектных задач = (число дефектов у подрядчика / число задач у подрядчика) * 100.
- Другие потенциальные скалярные функции: fn_task_duration(task_id) вычисляющая фактическую длительность задачи (в

днях) из дат; fn_project_delay(project_id) возвращающая, на сколько дней проект просрочен (если actual_end_date > planned_end_date); однако конкретно реализована у нас одна основная (fn_project_total_spent).

Табличные функции (table functions): - fn_report_projects(status TEXT DEFAULT NULL, date_from DATE DEFAULT NULL, date_to DATE DEFAULT NULL) – возвращает таблицу с информацией о проектах, отфильтрованных по статусу и диапазону дат. Эта функция фактически формирует отчёт “Проекты” за период.

(Приведённая реализация комбинирует SQL и PL/pgSQL для динамического фильтра). Эта функция позволяет получить выборку проектов по определенным критериям: например, все проекты со статусом 'active', планируемые в 2025 году.

Вызываться может так: SELECT * FROM fn_report_projects('active', '2025-01-01', '2025-12-31'); – получим все активные проекты 2025 года.

- fn_report_materials_usage(p_project_id BIGINT) – могла бы быть функцией, возвращающей свод по материалам для указанного проекта (аналог представления v_material_balance_by_project, но для конкретного проекта, или, наоборот, по всем проектам но конкретному материалу). Например, fn_report_materials_usage(project_id) возвращает список материалов с полями qty_in, qty_out, balance, экономя написание запроса.
- fn_batch_import_inventory_transactions(p_rows JSONB, p_source TEXT, p_fail_fast BOOLEAN, p_meta JSONB) – особая табличная функция, реализующая логику пакетного импорта. О ней ниже.

Хранимые процедуры (процедуры): PostgreSQL различает функции и процедуры (у процедур нет возвратного значения и они вызываются через CALL). В нашем проекте можно рассматривать

fn_batch_import_inventory_transactions как процедуру, хотя она реализована как функция, возвращающая результат импорта.

Для требования массовой загрузки данных реализована специальная функция импорта: – fn_batch_import_inventory_transactions(p_rows JSONB, ...) – предназначена для пакетной загрузки движений материалов (inventory_transactions) из внешнего источника (например, из JSON-массива). Эта функция принимает параметр p_rows как JSONB – массив объектов, где каждый объект содержит поля транзакции (например: project_id, material_id, task_id, quantity, unit_price, type, date). Также параметры: p_source (источник, например имя файла), p_fail_fast (если TRUE – прекратить при первой ошибке, если FALSE – пытаться импортировать все строки, ошибочные записать в log), p_meta (дополнительные метаданные, JSONB, например имя пользователя, IP, что угодно). Функция работает внутри единой транзакции и:; Создаёт новую запись в import_runs (статус 'running') – фиксирует начало импорта.; Итеративно или с помощью JSON->SQL вставляет записи в inventory_transactions. Для каждой записи:; Пытается выполнить INSERT в inventory_transactions. Если происходит ошибка (например, нарушен FK – неверный project_id, или CHECK – quantity<=0), то:.

Реализация функции fn_batch_import_inventory_transactions выполнена на PL/pgSQL с использованием динамического SQL или loops.

Преимущество вынесения этой логики на сторону БД – скорость (все операции в одной транзакции на сервере) и надежность (используются те же правила целостности).

db.execute(text("SELECT * FROM fn_batch_import_inventory_transactions(:rows::jsonb, :src, :fail_fast, :meta)"),

{"rows": json.dumps(list_of_dicts), "src": filename, "fail_fast": False, "meta": None})

и получить результат – сколько вставлено, сколько ошибок.

Использование функций в API: - Скалярная функция fn_project_total_spent может быть вызвана при отображении карточки проекта: API может дернуть SELECT fn_project_total_spent(X) и вернуть клиенту это значение без необходимости считать на приложении. - Табличная функция fn_report_projects используется непосредственно в маршруте /reports/projects.

Затем эти rows конвертируются в схемы (pydantic) и выдаются как JSON. Это показывает, как чистый SQL (в виде функций) интегрируется – сложный отчет строится внутри БД. - Функция fn_batch_import_inventory_transactions вызывается из endpoint /batch – при получении файла или данных, backend преобразует их в JSON и вызывает эту функцию. Результат (run_id, counts) возвращается клиенту, а сами данные уже вставлены.

Дополнительные функции: - Функции-триггеры – особый вид функций, которые не вызываются напрямую, а привязаны к событиям (см. раздел 3.8). Их тоже можно считать частью “хранимой логики”. У нас есть trig-functions для аудита и обновления timestamp. - Можно упомянуть, что при необходимости в проект добавлены простые utility-функции, напр. audit_get_user_id() – возвращает текущего пользователя из current_setting('app.user_id') (используется в триггерах).

В итоге, реализованный набор функций позволяет продемонстрировать: - Вычисление агрегированных показателей (на примере fn_project_total_spent). - Генерацию отчётных таблиц с параметрами (fn_report_projects). - Выполнение бизнес-операций (fn_batch_import_inventory_transactions). - Использование функций как API баз данных, вызываемых из приложения.

3.8. Триггеры базы данных

Для автоматизации некоторых действий и отслеживания изменений используются триггеры – специальные объекты, которые привязываются к таблицам и срабатывают при наступлении определённых событий (вставка, обновление, удаление строк). В нашем проекте триггеры решают две основных задачи: 1. Аудит изменений – логирование всех вставок, обновлений, удалений записей в специальный журнал (таблица audit_log). Однако сама идея выполнена другим способом: представления или функции считают агрегаты.

Триггеры аудита (CRUD Audit): В проекте настроены универсальные триггеры на все важные таблицы, которые отслеживают события INSERT, UPDATE, DELETE. Механизм следующий: – Создана функция-триггер trg_audit_crud() (PL/pgSQL), которая принимает на вход контекст изменения и имя PK колонки. Этот триггер будет добавлять запись в audit_log с указанием, какая сущность изменилась, её ID, тип операции и сохранять старые/новые значения;

Перед созданием, на всякий случай, удаляется существующий триггер (DROP IF EXISTS). Мы передаем в trg_audit_crud имя PK поля таблицы, если оно нестандартное; на самом деле функция может сама вычислять PK, но для упрощения мы явно указываем. - Такие триггеры созданы практически для всех основных таблиц: users, owners, properties, property_rooms, contractors, contracts, projects, project_phases, project_tasks, acceptance_acts, purchase_orders, purchase_order_items, inventory_transactions, defects. (Можно даже на справочники materials, suppliers – если нужно отслеживать их изменение). - Исключение: audit_log сама обычно не аудируется (чтобы не было рекурсии), import_runs и import_errors можно не аудировать (они и так логируют, по сути).

Важно, что триггер ставится AFTER операции – то есть сначала изменения сохраняются, затем логируется (в той же транзакции). Таким образом, если по какой-то причине вставка/обновление откатится, откатится и запись аудита (atomicity).

Триггеры обновления timestamp: Некоторые таблицы имеют поле updated_at (например, properties, owners – судя по DDL, только properties у нас явно имеет updated_at).

Другие возможные триггеры: - Каскадное обновление/вставка: не используется в данном проекте, но как пример – можно было сделать триггер, который при вставке новой задачи увеличивает счётчик задач в проекте. Мы от этого отказались, т.к. есть представления/функции. - Валидация бизнес-правил: обычно лучше использовать CHECK, но триггером можно проверять более сложные правила (например, нельзя начать задачу, если проект не "active"). Мы пока не реализовали, но можно упомянуть: возможно, стоило бы триггером запрещать ставить статус проекта "completed", если не все задачи завершены – но такие сложные зависимости выходят за рамки задачи.

Пример работы триггеров (демонстрация): – Создаём новую задачу через API: в базе происходит INSERT в project_tasks. Автоматически:; Запись получает created_at по default,; Срабатывает audit_triggers: запись в audit_log.; Обновляем статус задачи: происходит UPDATE project_tasks.

Отключение аудита при массовых вставках

В системе предусмотрен механизм временного отключения аудита на время загрузки демонстрационных или больших объёмов данных. Это сделано

потому, что при тысячах вставок журнал аудита раздувается и перестаёт быть полезным для демонстрации логики.

3.9. Примеры SQL-запросов и отчётов

Этот раздел демонстрирует, какие типовые операции и аналитические запросы поддерживает база данных: от простых CRUD-действий до более сложных отчётов с объединениями таблиц, группировками, представлениями и пользовательскими функциями.

Таблица 4 — Пример записей audit_log (последние изменения)

action_timestamp	entity_type	entity_id	action_type	user_id
2025-12-19 22:58:53.485948	properties	501	INSERT	NULL
2025-12-19 22:56:25.212952	inventory_transactions	13000	INSERT	NULL
2025-12-19 22:56:25.212952	inventory_transactions	12999	INSERT	NULL
2025-12-19 22:56:25.212952	inventory_transactions	12998	INSERT	NULL

2025-12-19 22:56:25.212952	inventory_transactions	12996	INSERT	NULL

1. Добавление данных (аналог INSERT)

Показывается пример ручного добавления записи в справочник (например, нового вида работ). При этом ограничения целостности не позволяют случайно создать дубликаты (например, одинаковые названия), а аудит автоматически фиксирует факт добавления.

2. Изменение данных (аналог UPDATE)

Демонстрируется изменение ключевого поля у существующей записи (например, корректировка плановой даты окончания проекта). После изменения обновляются служебные поля (если они предусмотрены), а аудит сохраняет и прежнее, и новое значение.

3. Удаление данных (аналог DELETE)

Показывается удаление записи (например, комнаты). При этом объясняется поведение связей: за счёт настроек внешних ключей связанные сущности корректно обрабатываются (где-то удаляются каскадно, где-то связи обнуляются), чтобы не возникало «висячих» ссылок.Простое чтение данных (аналог SELECT)

Пример запроса на получение списка сущностей по условию (например, активные подрядчики). Это иллюстрирует обычное чтение из таблиц с фильтрацией.

4. Агрегация и группировка

Пример отчёта, где считается количество задач по статусам в пределах конкретного проекта. Это демонстрирует применение агрегатных функций и группировки по полю.

5. Использование представлений (VIEW)

Поясняется, что для повторяющихся сложных выборок используются представления: вместо написания каждый раз длинных объединений и условий можно обратиться к уже подготовленному «виртуальному отчёту», который сразу отдаёт данные в удобном виде (например, список просроченных задач с количеством дней просрочки).

6. Использование пользовательской функции

Показывается, что часть расчётов вынесена в функцию (например, вычисление фактических затрат по проекту). Тогда в отчёте можно вывести проекты вместе с плановым бюджетом и рассчитанными фактическими расходами — это удобно для проверки укладывания в бюджет.

7. Обращение к журналу аудита

Отдельно приводится пример, как можно посмотреть историю изменений конкретной сущности (например, задачи): какие действия выполнялись, когда, кем, и как менялись значения ключевых полей. Это подчёркивает, что аудит действительно позволяет восстановить цепочку изменений и является практическим инструментом контроля.

В сумме примеры закрывают широкий спектр возможностей SQL: базовые операции, аналитические запросы, отчёты через представления, расчёты через функции и анализ истории через аудит.

3.10. Массовая загрузка данных (Batch Import)

В системе реализована массовая загрузка данных на примере таблицы движения материалов — это наиболее показательный транзакционный набор данных, который удобно импортировать пачками.

Формат и принцип загрузки

Для простоты используется структурированный формат данных (который естественно передаётся из приложения в базу). Каждая запись импорта соответствует одной операции движения материала и содержит ссылки на проект/материал/задачу, количество, цену, тип операции и дату. Приложение передаёт в базу сразу набор таких записей, а вставка выполняется на стороне PostgreSQL.

Интеграция с backend

На уровне backend предусмотрен маршрут для batch-загрузки: он принимает пакет данных, передаёт его в сервисный слой, а затем вызывает серверную функцию в базе, которая и выполняет импорт. После завершения backend возвращает пользователю идентификатор запуска и статистику (сколько строк добавлено успешно, сколько завершилось ошибками). При необходимости ошибки можно получить отдельным запросом по этому запуску.

Логирование процесса и ошибок

Каждая массовая загрузка фиксируется как отдельный «запуск импорта» со статусом выполнения. При обработке пакета система пытается вставлять записи по очереди. Если встречается ошибка (например, неверная ссылка на проект, некорректный тип операции, отрицательное количество), ошибка не теряется: она записывается в отдельную таблицу ошибок импорта вместе с описанием причины и исходными данными проблемной строки.

Таким образом, пользователь видит не просто «импорт не удался», а получает конкретный список строк, которые не прошли, и причины, почему они отклонены. Это удобно для исправления данных и повторной загрузки только проблемных записей.

Таблица 5 — Пример запуска batch import

run_id	total_rows	inserted_rows	failed_rows	status
1	5000	4848	152	completed_with_errors

При параметрах count=5000 и invalid_rate=0.03 загрузка завершилась со статусом completed_with_errors: часть строк вставлена, а ошибки доступны по run_id через import_errors.

Режим fail_fast

Поддерживается режим «остановиться на первой ошибке». Он полезен, когда требуется строгость: если данные некорректны, дальнейшая загрузка не имеет смысла. При этом важно, что уже успешно вставленные строки могут остаться в базе (выбран подход частичного успеха + журнал ошибок), а запуск импорта помечается как завершившийся с ошибкой.

Производительность

Импорт выполняется внутри базы данных, что обычно быстрее построчной вставки через приложение. При объёмах порядка нескольких тысяч строк требование по массовости выполняется уверенно.

Взаимодействие с аудитом

Для массовых операций при необходимости можно временно отключать аудит, чтобы не создавать тысячи записей в журнале аудита на каждую вставку. При этом в обычной эксплуатации аудит остаётся включённым и фиксирует все изменения. Такой подход помогает сохранить журнал аудита полезным и читаемым, не превращая его в «шум» из технических записей при демонстрационной загрузке.

Итог: Массовая загрузка реализована через хранимую функцию fn_batch_import_inventory_transactions и сопровождающие таблицы журнала import_runs/import_errors. Это обеспечивает удобный механизм интеграции внешних данных: можно дозаполнить систему, например, историей списаний материалов из Excel-файла, без риска нарушить целостность (неверные данные просто не загрузятся и будут отдельно указаны).

3.11. Интерфейсы приложения (API)

Backend реализован на FastAPI и предоставляет REST API для работы со всеми ключевыми сущностями (CRUD), а также для отчётности и массовой загрузки данных. Документация и тестирование выполняются через Swagger/OpenAPI (достаточно для демонстрации без отдельного фронтенда).

Основные группы эндпоинтов:

• Объекты ремонта: /api/v1/owners, /properties, /rooms.

• Проекты и работы: /api/v1/projects, /phases, /tasks, /work-types.

• Подрядчики и договоры: /api/v1/contractors, /contracts.

• Материалы и закупки: /api/v1/materials, /suppliers, /purchase-orders, /inventory-transactions.

• Контроль качества: /api/v1/defects, /acceptance-acts.

• Отчёты: /api/v1/reports/* (на основе VIEW и функций БД).

• Импорт: POST /batch (batch import), а также GET /import-runs и GET /import-errors?run_id=… .

Операции, изменяющие несколько таблиц (например, создание проекта вместе с этапами/задачами или импорт), выполняются в транзакции, чтобы обеспечить согласованность данных.

4. Технологическая часть

В технологической части описаны процессы развёртывания, тестирования и демонстрации проекта.

Запуск приложения: Для запуска всей системы используется Docker. Предусмотрена конфигурация Docker Compose (docker-compose.yml в корне репозитория), которая определяет два сервиса: - db – на основе образа Postgres (например, postgres:15-alpine), с монтированием скриптов и инициализацией. - api – на основе Python образа (например, python:3.11-slim), с копированием исходного кода FastAPI приложения.

Перед запуском необходимо сконфигурировать окружение: файл .env (создан копированием из .env.example), в котором заданы переменные – в частности, параметры подключения к БД (хост, порт, имя БД, пользователь, пароль) для backend, а также настройки Alembic (если применяются миграции). В .env также можно включить флаг DEBUG для включения подробных логов.

Стандартная команда запуска:

docker compose up --build

Это соберёт образ API (установит зависимости из api/requirements.txt, скопирует код) и запустит оба контейнера. БД поднимется и выполнит скрипты: - /docker-entrypoint-initdb.d/01_schema.sql и другие SQL из папки sql/ddl, sql/functions, sql/triggers, sql/views, sql/indexes, sql/dml. Последовательность гарантируется префиксами (01_schema.sql -> 02_constraints.sql -> ... -> dml). Таким образом, схема будет создана и наполнена данными. - API-сервис при старте ожидает доступности БД (в Compose YAML настроена зависимость). После запуска доступен Swagger UI на http://localhost:8000/docs.

Тестирование API: После запуска можно протестировать базовые операции: - Зайти в Swagger UI (/docs) и попробовать выполнить, например,

GET /properties – должен вернуть список объектов недвижимости (с демо-данными, которые были вставлены скриптом 01_seed.sql). - Через Swagger или curl выполнить POST операцию, например, добавить нового подрядчика:

curl -X POST "http://localhost:8000/api/v1/contractors" -H "Content-Type: application/json" -d '{"name": "ООО РемСтрой", "phone": "+71234567890", "email": "contact@remstroy.ru"}'

Тестирование объёмов данных: в тестовой базе сформированы наборы данных, соответствующие требованию по объёму (в т.ч. транзакционная таблица inventory_transactions содержит не менее 5000 строк). Фактические значения приведены в таблице ниже.

Таблица 6 — Объём данных в БД (фактические значения)

Таблица	Кол-во записей
projects	500
project_tasks	1000
materials	800
purchase_orders	800
inventory_transactions	8000
audit_log	6093

Логирование и отладка: Backend выводит логи запросов (в dev-режиме). БД можно настроить лог длительных запросов (log_min_duration_statement). То есть, API достаточно устойчивое: ошибки БД транслируются в HTTP ответы.

5. Заключение

В ходе выполнения курсовой работы была разработана информационная система “Планирование и учёт ремонта жилых помещений”, включающая реляционную базу данных и прикладной сервис доступа к ней. Проект полностью соответствует требованиям задания: – Спроектирована база данных, содержащая 20 связанных таблиц .

Таким образом, поставленная задача успешно решена. Созданная информационная система удовлетворяет требованиям к функциональности и качеству: она охватывает все необходимые аспекты учёта ремонта (планирование работ, ресурсы, договора, контроль качества, аналитика), гарантирует целостность и актуальность данных (за счёт ограничений и аудита), обладает хорошей производительностью на заданных объёмах (благодаря индексам и оптимизации), и предоставляет удобный интерфейс для пользователей и внешних приложений (REST API с документацией).