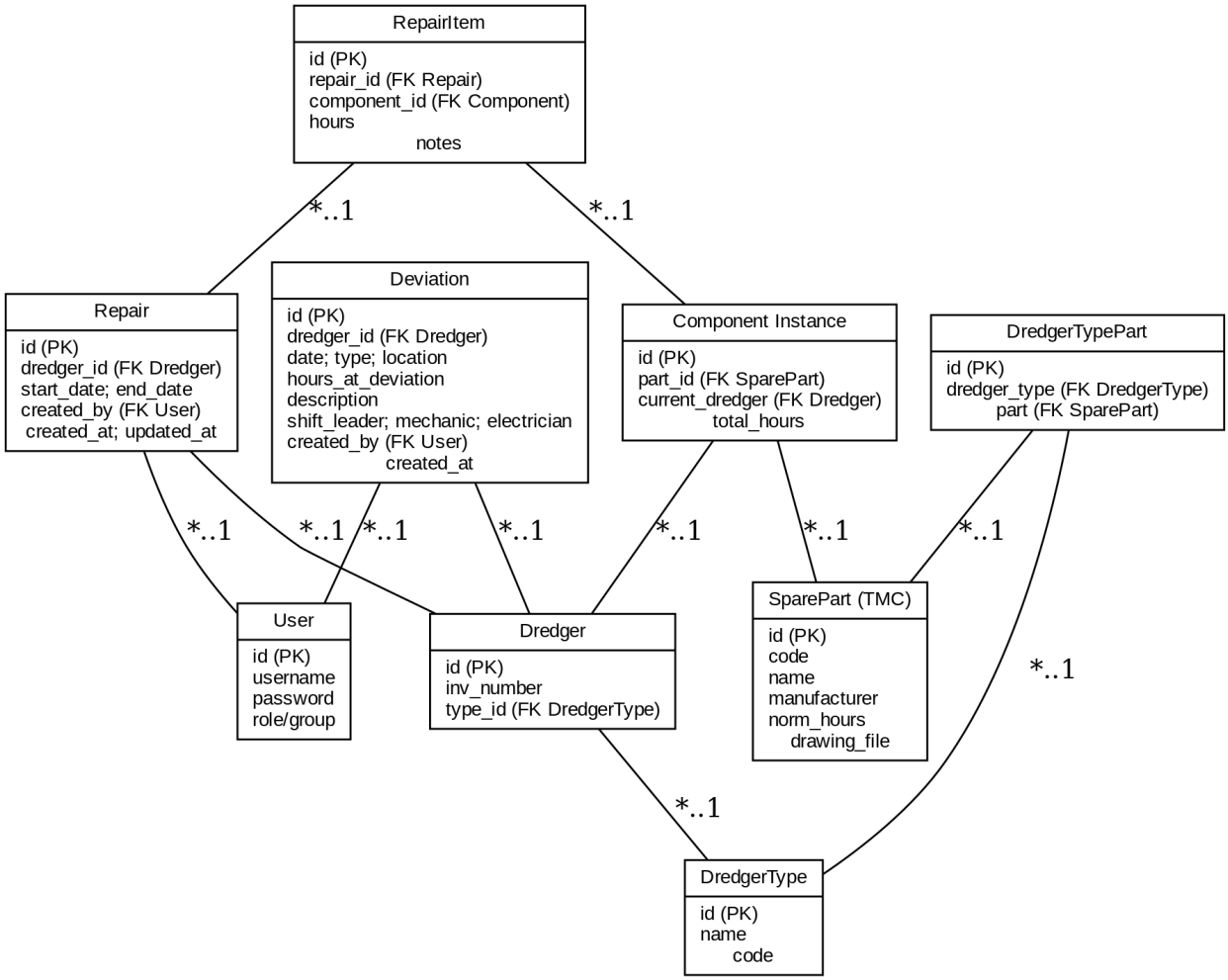

Архитектура базы данных

Рис. 1: ER-диаграмма основных таблиц системы электронного журнала.

Обзор сущностей и связей: База данных построена в соответствии с требованиями для отслеживания ремонтов землесосов, их агрегатов, отклонений в работе и справочника запчастей. На ER-диаграмме показаны ключевые таблицы и их связи. Ниже приводится описание каждой таблицы и отношений между ними:

- **User (Пользователи):** Хранит учетные записи пользователей системы с полями `username`, хэшированный `password` и роль/группа. Роли (администратор, оператор, инженер, просмотр) реализованы через группы Django: у каждой группы настроен набор прав, соответствующих

роли. Пользователи привязываются к группам, определяя тем самым их роль в системе.

- **DredgerType (Тип землесоса):** Справочник типов землесосов (например, LHD-49, 2ГрТ 8000/71, 2ГрТ 15000/85) с полями названия и кода типа. Каждый тип землесоса характеризуется фиксированным набором основных агрегатов.
- **SparePart (ТМЦ, запчасти):** Справочник запасных частей/агрегатов с полями: уникальный код ТМЦ, название, производитель, нормативный ресурс (допустимая наработка в часах) и файл чертежа. Информация о **нормативной наработке** хранится здесь для каждого типа агрегата. Чертежи хранятся как загруженные файлы (например, PDF или изображение схемы).
- **Dredger (Землесос):** Список конкретных землесосов в эксплуатации. Поля включают уникальный хозяйственный номер (инвентарный номер) и ссылку на `DredgerType`. Эта таблица фиксирует сами машины; каждый землесос принадлежит определенному типу, что позволяет при вводе данных подставлять шаблонный список его агрегатов.
- **DredgerTypePart (Комплектация типа):** Вспомогательная таблица, связывающая тип землесоса (`DredgerType`) и виды агрегатов (`SparePart`). Определяет, какие агрегаты (из справочника ТМЦ) входят в состав данного типа землесоса и подлежат учету наработки. Например, для типа LHD-49 может быть задан список из двигателя, насоса и т.д. Именно на основе этой связи при создании записи о ремонте будут подставляться шаблонные агрегаты.
- **Component Instance (Экземпляр агрегата):** Таблица, отражающая **конкретные экземпляры агрегатов** (узлов оборудования). Включает ссылку на тип агрегата (`part_id` → SparePart), текущее местонахождение (`current_dredger` → Dredger, может быть NULL если агрегат снят и находится на складе/в ремонте) и накопленную наработку в часах (`total_hours`). Каждый физический агрегат при установке на землесос или перемещении между ними отслеживается как отдельная сущность. Это позволяет вести историю использования одного и того же агрегата на разных машинах: при замене агрегата в одном землесосе и установке его на другой, запись

об экземпляре сохраняется, меняется только его поле текущего землесоса.

- **Repair (Ремонт):** Журнал ремонтных воздействий. Содержит записи о каждом ремонте землесоса с полями: ссылка на землесос (`dredger_id`), дата начала и окончания ремонта, а также служебные поля аудита – кем и когда создана/изменена запись. Пользователь, создавший запись, сохраняется в поле `created_by` (FK на User), дата создания/изменения – в полях `created_at`, `updated_at`. Таким образом фиксируется кто и когда вносил данные (без необходимости ручного ввода — значения проставляются автоматически).
- **RepairItem (Детали ремонта):** Связанная с ремонтом таблица, хранящая **список агрегатов, учтенных при ремонте**. Каждая запись включает: ссылку на родительский ремонт (`repair_id`), ссылку на экземпляр агрегата (`component_id` → Component Instance) и наработку этого агрегата в часах на момент ремонта. Также могут быть поля примечаний/описания замены. Поля производителя запчасти и чертежа определяются через связь с справочником SparePart: поскольку `Component Instance` ссылается на `SparePart`, мы знаем и тип агрегата, и его производителя, и можем при необходимости вывести чертеж. В ходе ввода данных оператор выбирает агрегаты из шаблонного списка для данного типа землесоса (DredgerTypePart) – система создает для каждого агрегата запись RepairItem. Пользователь указывает наработку на момент ремонта и при необходимости отмечает замену (например, в поле примечания). Если агрегат заменен новым, то старый экземпляр может быть отмечен как снятый (например, установить `current_dredger = NULL` у старого Component Instance), а для нового устанавливается `current_dredger =` этот землесос. Новому агрегату присваивается наработка 0 или небольшое значение, соответствующее испытаниям. Таким образом, **Component Instance позволяет прослеживать историю перемещений агрегата**: через разные RepairItem (и ремонты) можно увидеть, как агрегат «путешествовал» между землесосами, с указанием наработки и дат.
- **Deviation (Отклонение в работе):** Журнал простоев и сбоев. Поля: дата события, тип простоя (`type` – например, механический, электрический, технологический — удобно реализовать как ChoiceField с заранее

заданными значениями), участок (`location` – например, ПНС, ТВС, ШХ, тоже в виде перечня допустимых значений), ссылка на землесос (`dredger_id`), дата последнего ППР (планово-предупредительного ремонта) – можно хранить отдельно или вызывать из Repair, но по условию вводится вручную, наработка на момент отклонения (`hours_at_deviation`), описание обстоятельств, ФИО ответственных (начальник смены, механик, электрик). Также, как и в таблице ремонтов, у отклонений имеются поля аудита: кто и когда завел запись (создал/изменил). Записи в Deviations связаны с конкретным землесосом, но не с компонентами, так как описывают инцидент в целом.

Связи между таблицами реализованы через внешние ключи, как отражено на ER-диаграмме. Некоторые важные моменты:

- Каждый землесос (`Dredger`) относится к одному типу (`DredgerType`), связь многие-к-одному.
- Через таблицу DredgerTypePart задана связь многие-ко-многим между типами землесосов и видами агрегатов (SparePart), определяя комплектацию.
- **Экземпляры агрегатов** (Component Instance) связаны с типами запчастей (SparePart) как многие-к-одному (много экземпляров одного типа) и с землесосом как многие-к-одному (несколько агрегатов могут стоять на одном землесосе, а один агрегат в текущий момент может принадлежать максимум одному землесосу). Если агрегат снят, связь разрывается (`current_dredger` становится NULL).
- Repair (ремонт) связан с Dredger многие-к-одному (у землесоса много ремонтов), а RepairItem – с Repair (одному ремонту соответствует список агрегатов/работ). RepairItem ссылается на конкретный экземпляр агрегата (Component Instance), чтобы фиксировать наработку именно того агрегата, который участвовал в ремонте.
- Deviation (отклонение) ссылается на Dredger (у землесоса может быть несколько событий отклонений).
- Все основные таблицы ремонтов и отклонений имеют поля `created_by` (FK на User) для указания автора записи. Эта связь позволяет ввести ролевую модель контроля изменений и дальнейшее логирование.

Хранение файлов: Файлы чертежей, прикрепленные к запчастям, сохраняются на сервере (например, в директории `MEDIA_ROOT`, с URL префиксом `MEDIA_URL` для раздачи статических файлов). В настройках Django необходимо определить `MEDIA_ROOT` и `MEDIA_URL` и добавить конфигурацию URL для раздачи медиа-файлов в режиме отладки. Реализация хранения чертежей через FileField в модели SparePart позволяет загружать файлы через Django (при сохранении записи) и хранить путь в базе.

Django: модели и конечные точки API

Модели Django

Модели Django будут соответствовать описанным таблицам. На основе Django ORM создаются следующие модели:

- **User** – используем встроенную модель пользователя Django (или кастомную при необходимости). Роли реализуем через группы иPermissions: создаем группы "Администратор", "Оператор", "Инженер", "Наблюдатель" и настраиваем права доступа. Django не имеет встроенного понятия "роль", но позволяет объединять permisos в группы и присваивать группы пользователям. Таким образом, проверка ролей может сводиться к проверке принадлежности пользователя к той или иной группе.
- **DredgerType** – модель `DredgerType` с полями `name` (CharField) и `code` (CharField). Она потребуется, чтобы ссылаться из землесосов и определять состав агрегатов.
- **SparePart** – модель `SparePart` (ТМЦ) с полями: `code` (CharField, уникальный код запчасти), `name` (CharField, название), `manufacturer` (CharField), `norm_hours` (IntegerField, нормативный ресурс в часах) и `drawing_file` (FileField, хранит загружаемый файл чертежа). Эти поля отображают справочник запасных частей. Модель может также иметь поле `description` или `category` при необходимости, но согласно требованиям достаточно перечисленных. В админ-панели администратор сможет добавлять/редактировать записи справочника.

- **Dredger** – модель землесоса с полями: `inv_number` (CharField, хозяйственный номер, уникальный идентификатор машины) и внешний ключ `type` (ForeignKey на `DredgerType`, с опцией CASCADE). Также возможно поле местоположение или статус, если нужно, но требований на это нет. При создании нового землесоса через интерфейс можно будет указать его тип – это пригодится для шаблонного списка агрегатов.
- **ComponentInstance** – модель экземпляра агрегата, например `ComponentInstance`, с полями: внешний ключ `part` (ForeignKey на SparePart, CASCADE, тип агрегата), внешний ключ `current_dredger` (ForeignKey на Dredger, NULLABLE, текущий землесос, где агрегат установлен; может быть NULL, если агрегат снят), `total_hours` (IntegerField, суммарная наработка агрегата). Также можно добавить, если у агрегата есть серийный номер или уникальный идентификатор, соответствующее поле (например, `serial_number`). Эта модель позволит идентифицировать отдельный агрегат и отслеживать его ресурс. **Логика работы:** при замене агрегата в ремонте, если ставится новый узел, мы создаем новый экземпляр (ComponentInstance) с `total_hours = 0` и связываем с данным землесосом; у старого экземпляра обновляем `current_dredger = NULL`. Если агрегат переставлен на другой землесос, то при следующем ремонте его `current_dredger` сменится. Таким образом, по модели ComponentInstance можно получать историю (через ремонты) для каждого агрегата.
- **DredgerTypePart** – можно реализовать как модель (например, `CompatiblePart`) с связями ManyToMany: поле `dredger_type` (FK на DredgerType) и `part` (FK на SparePart). Однако Django позволяет упростить: в модели DredgerType можно завести ManyToManyField на SparePart с `through='DredgerTypePart'`. Это позволит через ORM получать `some_type.parts.all()` и `some_part.dredger_types.all()`. В Django REST API для получения шаблона агрегатов по типу землесоса можно либо сделать отдельный эндпоинт, либо включить в сериализацию DredgerType список связанных SparePart (через вложенный сериализатор или через дополнительный метод). Но основная задача этой связи – иметь возможность настроить через админку или специальный UI, какие агрегаты у какого типа должны учитываться.
- **Repair** – модель ремонта с полями: `dredger` (ForeignKey на Dredger, CASCADE), `start_date` (DateTimeField или DateField), `end_date`

(DateTimeField/DateField), а также служебные `created_at` (DateTimeField, auto_now_add) и `updated_at` (DateTimeField, auto_now). Для аудита – поля `created_by` и `updated_by` (ForeignKey на User, null=True, blank=True). Эти поля будут автоматически заполняться текущим пользователем при создании/редактировании записи. В Django REST Framework можно использовать `serializers.CurrentUserDefault()` или переопределить `perform_create` / `perform_update` метода ViewSet, чтобы проставлять пользователя. Фактически, при сохранении модель получит ID пользователя, что и позволит определить, кто внёс запись. Дополнительно, можно хранить `notes` (TextField) с общими примечаниями по ремонту.

- **RepairItem** – модель строк ремонта (можно назвать, например, `RepairEntry` или `RepairItem`). Поля: `repair` (ForeignKey на Repair, CASCADE), `component` (ForeignKey на ComponentInstance, CASCADE, указывает какой конкретно агрегат из справочника участвовал), `hours` (IntegerField, наработка данного агрегата на момент ремонта) и `notes` (CharField/TextField, примечание, например указание производителя замененной запчасти, если отличается, или номер чертежа, если нужен). Поскольку `component` уже связан с SparePart, отдельно хранить тип агрегата и производителя нет нужды – их можно получить через связи. Однако можно продублировать поле `part` (FK на SparePart) для удобства, если компонент может быть NULL (например, когда запчасть не установлена? Но в нашей логике компонент должен быть). В рамках реализации можно предусмотреть флаг, например `is_replaced` (BooleanField) – был ли агрегат заменён в ходе ремонта. Если да, то можно интерпретировать эту запись как "снят старый компонент (component ссылкой на старый) с такой-то наработкой и установлен новый", при этом появится ещё одна запись RepairItem для нового агрегата с нулевой наработкой. Однако для упрощения можно считать, что каждая запись RepairItem отражает состояние агрегата на момент ремонта (либо его замену, если в примечании указано).
  Шаблонные записи: при создании нового Repair через интерфейс, фронтенд после выбора землесоса (и, следовательно, его типа) может отправить запрос на получение списка SparePart для этого типа, и на основе них отрендерить форму – поля для ввода наработки по каждому агрегату. Пользователь заполнит часы и отметит, где были замены/детали. Затем будет отправлен один запрос, содержащий все эти данные

– Django может принять вложенный JSON и сохранить несколько RepairItem, связанных с одним Repair (можно использовать сериализатор с вложенным списком или отдельным вызовом API для добавления строк после создания ремонта). В любом случае, модель RepairItem поддерживает отношение один-ко-многим с Repair (у каждого ремонта множество записей-агрегатов).

- **Deviation** – модель отклонения (простоя) с полями: `dredger` (ForeignKey на Dredger), `date` (DateTimeField или DateField), `type` (CharField с ограниченным выбором – например, choices=[('mech','механический'), ...]), `location` (CharField с choices для участка), `last_ppr_date` (DateField, дата последнего ППР – если требуется сохранить, иначе можно вычислять, но по ТЗ вводится вручную), `hours_at_deviation` (IntegerField, наработка землесоса на момент отклонения – ее оператор вводит вручную), `description` (TextField, описание обстоятельств), `shift_leader` (CharField, ФИО начальника смены), `mechanic` (CharField, ФИО механика), `electrician` (CharField, ФИО электрика). Также поля аудита `created_at` (auto_now_add) и `created_by` (FK на User) для фиксации автора. Здесь, в отличие от RepairItem, мы не храним детали по компонентам – отклонение относится ко всему землесосу. Однако зная, какой агрегат стал причиной (из описания), инженер сможет сопоставить с компонентами при анализе.

Все модели регистрируются в Django admin для удобного управления справочниками (DredgerType, SparePart, Dredger) и просмотра журналов (Repair, Deviation).

Дополнительные моменты реализации моделей:

- В моделях SparePart и DredgerTypePart нужно обеспечить уникальность связей (можно через Meta unique_together (dredger_type, part), чтобы не было дублирования агрегата в составе типа).
- Для удобства может быть создан абстрактный класс-модель с полями аудита (`created_by`, `created_at`, `updated_by`, `updated_at`), от которого наследуются Repair и Deviation – это уменьшит дублирование кода.
- При сохранении моделей, связанных с ComponentInstance, стоит продумать логику обновления наработки: например, после каждого ремонта можно обновлять `total_hours` экземпляра агрегата значением из

RepairItem (если это не замена) или обнулять (если заменён новым). Возможно, `total_hours` агрегата можно рассчитывать суммарно как максимальное значение hours, зафиксированное в RepairItem, либо хранить и обновлять явным образом.

Основные API endpoints (Django REST)

Backend реализуется на Django с использованием Django REST Framework (DRF) для организации REST API. Каждый модуль требований (ремонты, отклонения, справочники) будет представлен набором конечных точек (endpoints). Ниже перечислены основные:

- **Аутентификация и авторизация:**
  - `POST /api/auth/login/` – проверка логина и пароля пользователя. При успешном входе возвращается JWT-токен (access + refresh) или токен сессии. Для автономности фронтенда выбор сделан в пользу JSON Web Token. Используем пакет Simple JWT для DRF, который предоставляет стандартные возможности JWT-аутентификации. После входа фронтенд получает access-токен и сохраняет его (например, в localStorage или cookie). Все дальнейшие запросы будут содержать заголовок `Authorization: Bearer <token>`. Также можно настроить refresh-токены для продления сессии. (В качестве альтернативы, можно использовать сессию и куки с CSRF, но при раздельном React-приложении JWT удобнее).
  - `POST /api/auth/refresh/` – получение нового access-токена по refresh-токену (стандартный эндпоинт SimpleJWT).
  - `GET /api/auth/user/` – получение информации о текущем пользователе и его ролях. Этот эндпоинт понадобится фронтенду после аутентификации, чтобы узнать права. Можно реализовать, например, возвращая username, роль (или список групп) и другие данные профиля.

Примечание по CORS: Поскольку фронтенд (React) будет обращаться к API, нужно включить Django CORS Headers и разрешить нужный источник. Как отмечено в руководствах, нужно установить пакет `django-cors-headers` и подключить его в настройки, прописав `CORS_ALLOWED_ORIGINS` (или

`CORS_ALLOW_ALL_ORIGINS=True` на время разработки). Это устранит проблемы с браузерным Same-Origin Policy.

- **Endpoints модуля ремонтов:**
  - `GET /api/repairs/` – получить список всех записей ремонтов. Поддерживает фильтрацию по землесосу, диапазону дат и др. (например, `GET /api/repairs/?dredger_id=5&start_date__gte=2025-01-01`). В DRF можно подключить DjangoFilterBackend, позволяющий легко фильтровать по полям модели. В ответе по каждому ремонту можно включить краткие данные (ID, даты, землесос, кто создал).
  - `GET /api/repairs/{id}/` – получить подробную информацию о конкретном ремонте, включая связанные RepairItem (список агрегатов с наработкой и примечаниями). Серилизатор может быть настроен с вложенным списком RepairItem.
  - `POST /api/repairs/` – создать новую запись о ремонте. Ожидает данные: `dredger` (ID землесоса), `start_date`, `end_date` и список `items` – массив объектов с полями `part` / `component` и `hours`, `notes`. Валидация: обязательны землесос и дата, по вложенным – hours должны быть >= 0 и не превышать нормативы (можно проверять по SparePart.norm_hours), также можно проверять что список агрегатов соответствует шаблону для данного типа (например, сравнивать множество part_ids со списком из DredgerTypePart). При сохранении: для каждого элемента либо обновляем существующий ComponentInstance (если не заменялся агрегат), либо создаем новый (если в примечании или поле указано, что замена). Решение: можно требовать от фронтенда передавать `component_id` если агрегат уже существующий, или `part_id` + флаг замены для нового – тогда бэкенд создаст ComponentInstance. Этот процесс можно также скрыть внутри логики ViewSet.
    - **Пример:** оператор выбирает землесос #5 типа LHD-49. Фронт делает `GET /api/dredgers/5/template/` – бэкенд возвращает список {part: "Двигатель", component: 12, current_hours: 5000, norm_hours:10000, ...} для каждого агрегата. Оператор вводит новые показания: двигатель 5200 ч, насос 3000 ч и т.д., отмечает что насос заменен (например, ставит галочку "замена" и выбирает из справочника новый насос производителя X). Фронт отправляет POST

/api/repairs/ с телом, содержащим repair и items. Бэкенд создает Repair, затем создает/обновляет ComponentInstance: старому насосу current_dredger=NULL, новому создается ComponentInstance с part=насос, current_dredger=5, total_hours=0; далее сохраняются RepairItem: один для двигателя (component_id = старый двигатель, hours=5200), один для насоса старого (component_id = старый насос, hours=3000, note="снят"), и один для насоса нового (component_id = новый насос, hours=0, note="установлен взамен")). В итоге история сохраняется. (Если такая детализация не требуется, можно упростить, но система спроектирована для отслеживания ресурса).

  - `PUT /api/repairs/{id}/` – обновить запись ремонта (например, откорректировать даты или примечания). Обычно оператор может создавать, а править – инженер или администратор. Права доступа контролируются (см. ниже). Обновление списка RepairItem можно разрешить (DRF умеет обновлять вложенные, но чтобы не усложнять, можно запретить менять items после создания, кроме как через интерфейс специально).
  - `DELETE /api/repairs/{id}/` – при необходимости, удаление записи (может разрешаться только администратору). Вместо физического удаления, можно пометить как удаленную или не предоставлять этот функционал вовсе, чтобы не терять данные.

- **Endpoints модуля отклонений:**
  - `GET /api/deviations/` – список всех отклонений (простоев/аварий). Поддерживает фильтры: по типу (`?type=mechanical`), по участку, по землесосу (`?dredger_id=5`), по дате (можно фильтровать по диапазону дат события). Фильтрация реализована аналогично через DjangoFilterBackend, что позволяет клиенту задавать параметры запросов. Например, фронтенд может сформировать URL с выбранными критериями.
  - `GET /api/deviations/{id}/` – детали конкретного отклонения (все поля, включая ФИО и описание, а также поля аудита – кто ввел).

- `POST /api/deviations/` – создание новой записи отклонения. Оператор заполняет форму на фронтенде (поля: дата, тип, участок, землесос (выбор из списка), наработка, ППР, описание, лица). Бэкенд проверяет обязательные поля, обеспечивает запись автора (из токена). После сохранения возвращается созданный объект или статус.
- `PUT /api/deviations/{id}/` – обновление (возможно для инженера или администратора, если нужно корректировать или дополнять информацию расследования причин).
- `DELETE /api/deviations/{id}/` – удаление (опять-таки, может разрешаться только особыми ролями).

- **Endpoints справочников:**
  - `GET /api/dredger-types/` – список типов землесосов.
  - `POST /api/dredger-types/` – добавить новый тип (админ/инженер).
  - `PUT /api/dredger-types/{id}/` – изменить (например, название).
  - `DELETE /api/dredger-types/{id}/` – удалить тип (если нет связанных землесосов).
  - Вложенный ресурс: `GET /api/dredger-types/{id}/parts/` – получить список запчастей, ассоциированных с этим типом (на основе DredgerTypePart). Можно реализовать и как поле в detail представлении типа.
  - `POST /api/dredger-types/{id}/parts/` – добавить новую связь (добавить агрегат в комплектацию типа). Тело запроса содержит `part_id`. Либо, альтернативно, сделать endpoint `POST /api/dredger-type-parts/` с указанием `dredger_type` и `part`.
  - `DELETE /api/dredger-types/{id}/parts/{part_id}/` – убрать агрегат из типа.
  - `GET /api/spare-parts/` – список всех запчастей (с фильтрацией, например по производителю или названию, если нужно).
  - `POST /api/spare-parts/` – добавить запись в справочник ТМЦ (админ либо инженер). Здесь реализуется загрузка файла чертежа: фронтенд отправляет multipart-formdata с полями `code, name, manufacturer, norm_hours` и файлом. DRF во view должен иметь соответствующий парсер (MultiPartParser) для обработки файла. При успешном сохранении

сервер вернет JSON новой записи, включая URL файла чертежа (например, `/media/drawings/abc123.pdf`).

- `GET /api/spare-parts/{id}/` – детали запчасти (можно возвращать URL на чертеж, который фронтенд откроет при необходимости).
- `PUT /api/spare-parts/{id}/` – обновить информацию (например, заменить файл чертежа или скорректировать норматив).
- `DELETE /api/spare-parts/{id}/` – удалить (если бизнес-логика позволяет, хотя лучше не удалять используемые запчасти).
- `GET /api/dredgers/` – список землесосов (с информацией о типе, можно включить текущие наработки основных агрегатов, если вычислять).
- `POST /api/dredgers/` – добавить новый землесос (админ).
- `PUT /api/dredgers/{id}/` – редактировать (например, сменить тип, если допущено, или обновить инвентарный номер).
- `DELETE /api/dredgers/{id}/` – удалить (при условии, что нет связанных ремонтов/отклонений, или реализовать каскадно – но тогда потеряем историю, поэтому, вероятно, удаление землесоса не будет использоваться, вместо этого – пометка "списан").
- **Специальные точки для шаблонов агрегатов и истории:**
  - `GET /api/dredgers/{id}/components/` – возвращает список текущих установленных агрегатов на землесосе (можно по таблице ComponentInstance фильтровать current_dredger=id). В ответе по каждому агрегату можно дать: тип (name), производитель, текущая наработка (total_hours) и норматив. Это позволит фронту показать, например, текущие счетчики.
  - `GET /api/dredgers/{id}/template/` – возвращает список компонентов, которые *должны быть* у данного землесоса по его типу, для заполнения формы ремонта. Например, даже если какого-то агрегата нет (NULL), все равно включить тип. Можно объединить с предыдущим: фактически, пройти по DredgerTypePart для типа землесоса, и для каждого SparePart попытаться найти ComponentInstance с current_dredger=id. Результат: список {part, component (может null, если новый будет), name, norm_hours,

current_hours (если установлен), manufacturer}. Фронтенд, получив этот список, отобразит поля ввода для каждого.

    - `GET /api/components/{id}/history/` – (опционально) получить историю перемещений и ремонтов конкретного агрегата. Можно собрать все RepairItem, где component_id = {id}, отсортировать по времени (Repair.start_date), и вернуть список: {repair_id, dredger_id, dates, hours, note}. Это для инженеров, чтобы быстро посмотреть судьбу узла.

- **Endpoint экспорта данных:** По требованию, необходимо экспортировать таблицы отклонений и ремонтов в Excel. Это можно сделать отдельными представлениями:
    - `GET /api/reports/repairs_excel/` – при вызове генерирует Excel-файл (.xlsx) со всеми записями ремонтов и возвращает его в ответе (Content-Type: application/vnd.openxmlformats-officedocument.spreadsheetml.sheet). В запрос можно передать параметры фильтрации, чтобы выгрузить, например, ремонты за определенный период или по конкретному землесосу. Формирование Excel можно реализовать с помощью библиотек **pandas + openpyxl** либо напрямую openpyxl. Например, Pandas позволяет конвертировать queryset в DataFrame и сразу в Excel. Такой подход (Django + Pandas) широко используется для экспорта данных в файлы Excel.
    - `GET /api/reports/deviations_excel/` – аналогично для отклонений.
    - Чтобы обеспечить безопасность, эти endpoints требуют авторизации (например, доступны инженеру или админу). Фронтенд по нажатию кнопки "Экспорт в Excel" будет открывать эти URL (в новом окне или через fetch + создание URL blob для скачивания).

Примечание: Все endpoints будут защищены соответствующими правами доступа. DRF позволяет на уровне ViewSet задать `permission_classes`. Например, для чтения справочников хватит IsAuthenticated, а для создания/удаления – кастомный пермишен, проверяющий роль. Можно написать собственные классы, например `IsAdmin` (проверяет `request.user.groups.filter(name="Администратор")`), `IsOperatorOrEngineer` и т.д., или задействовать модель Permissions (но в данном случае проще привязка к ролям). В сводке, **администратор** имеет доступ на

все CRUD операции всех разделов, **инженер** – на просмотр и частичное редактирование (например, может подтверждать/редактировать введенные оператором данные, управлять справочником ТМЦ), **оператор** – на создание новых записей ремонтов/отклонений и просмотр своих данных, **наблюдатель** – только чтение. Эти правила будут реализованы в настройке permission_classes на представлениях или внутри методов (например, `has_object_permission`).

Компоненты React и маршруты

Клиентская часть на React реализует удобный интерфейс для операторов и инженеров. Будут созданы отдельные компоненты для основных разделов приложения. Используется React + React Router для организации SPA (одностраничного приложения) с разными маршрутами. Ниже перечислены ключевые маршруты, компоненты и ограничения по ролям:

- **Компонент аутентификации**:
  - `/login` – страница логина (компонент LoginPage). Содержит форму ввода логина/пароля и отправляет запрос к `/api/auth/login/`. При успехе сохраняет JWT токен (например, используя Context API или Redux для состояния auth). После входа происходит перенаправление на главную панель.
  - Механизм хранит также информацию о роли пользователя (либо токен содержит роль в своих payload, либо фронт после входа делает запрос `/api/auth/user/` чтобы получить роль). Эта информация сохранится в глобальном состоянии (например, `currentUser.role`).
- **Главная страница / дашборд**:
  - `/` – может быть компонент Dashboard или просто перенаправление на список ремонтов. Dashboard для инженера/админа может показывать сводные показатели (например, сколько сейчас простаивает техники, сколько агрегатов требуют замены и пр.), но это скорее расширение. В простейшем случае главной страницей будет список журналов.
- **Раздел ремонтов:**

- `/repairs` – компонент `RepairsList` (список ремонтов). Отображает таблицу всех записей ремонтов с основными полями (землесос, даты, примечание, автор). Над таблицей – фильтры (по землесосу, по дате). Компонент при монтировании делает запрос GET `/api/repairs/` и сохраняет данные в состоянии. Реализована возможность фильтрации: либо путём обращения к API с параметрами, либо на стороне клиента (если данных немного). Предпочтительно делать фильтрацию на сервере для актуальности и облегчения клиента – например, при изменении поля фильтра компонент вызывает обновленный GET запрос.
    - Для оператора показываются только поля, доступные ему (возможно, без поля "created_by", т.к. он и так автор своих записей). Для инженера/админа – полнее информация.
    - **Доступ:** страница доступна для ролей оператор, инженер, админ, наблюдатель (т.е. всем аутентифицированным, но с разными возможностями внутри).
- `/repairs/new` – компонент `RepairForm` для создания нового ремонта. Доступен операторам (и инженерам). На странице сначала выбирается землесос (выпадающий список, загружается из `/api/dredgers/`). После выбора землесоса компонент делает запрос `/api/dredgers/{id}/template/` для получения списка агрегатов. Затем отображается форма: поля даты начала/окончания ремонта, и динамически список агрегатов с полями для ввода наработки и галочками/полями для замен. Например, для каждого агрегата: "Двигатель – текущая наработка ___ ч (норматив 10000) Заменен? Если да, выбрать запчасть". Флажок "Заменен" может, при установке, раскрывать дополнительный ввод – выбор производителя/кода новой запчасти (выпадающий список из SparePart по этому типу агрегата). Если агрегат заменен, компонент может занулить или отметить соответствие.
    - При сабмите форма собирает данные и отправляет POST `/api/repairs/` с требуемым JSON. После успешного ответа переадресует на список ремонтов или страницу этого ремонта.

        - **Доступ:** оператор и инженер (админ тоже может создать при необходимости).
    - `/repairs/{id}` – компонент `RepairDetail` (страница просмотра ремонта). Отображает подробные данные ремонта: землесос, даты, список агрегатов с их наработкой и примечаниями, а также кем и когда введено. Здесь инженер может проверить введенные оператором данные.
        - Возможно, **режим редактирования**: компонент `RepairForm` может использоваться и для редактирования, если роль пользователя позволяет. Например, инженер заметил ошибку и хочет поправить наработку или отметить замену – он нажимает "Редактировать", получает форму с уже заполненными значениями (для этого при нажатии можно загрузить данные GET `/api/repairs/{id}/`, или использовать те, что уже есть). После корректировки отправляет PUT запрос.
        - **Доступ:** оператор может просматривать свои введенные записи (возможно, ему не нужна отдельная страница, можно просто в списке показывать), инженер и админ – просматривать и редактировать. Наблюдатель – только просматривать.
- **Раздел отклонений:**
    - `/deviations` – компонент `DeviationsList` (список отклонений). Похожа на RepairsList: таблица всех простоев/аварий с колонками (дата, землесос, вид простоя, описание краткое, автор). Имеются фильтры: по типу простоя (выпадающий список), по землесосу, по периоду. Компонент делает GET `/api/deviations/` при загрузке. Фильтры вызывают перезагрузку списка с query params.
        - Для удобства можно реализовать быстрые фильтры: например, кнопки "Механические", "Электрические", "Все".
        - **Доступ:** см. ремонты (все роли, только чтение для наблюдателя).
    - `/deviations/new` – компонент `DeviationForm` для добавления нового отклонения. Содержит поля, соответствующие модели (включая выборы типа и участка, которые будут выпадающими списками). Поле

землесоса – выпадающий список или авто-комплит. Поля ФИО можно сделать текстовыми или тоже выпадающими, если заранее загружен список сотрудников (но в ТЗ об этом не сказано, видимо просто вручную вводят).

- Отправляет POST `/api/deviations/`. После создания возвращается к списку (либо показывает уведомление).
- **Доступ:** оператор, инженер (возможно, инженер тоже может завести, например, если сам фиксирует что-то).

- `/deviations/{id}` – компонент `DeviationDetail`. Отображает все подробности отклонения. Инженер может отредактировать (например, добавить результаты расследования причин в описание). Реализуется через тот же `DeviationForm` но в режиме edit, отправляя PUT.
  - **Доступ:** оператор может просматривать (свои или все, в зависимости от политики — скорее все, чтобы знать историю), инженер/админ – редактировать, наблюдатель – только просмотр.

- **Раздел справочников:**
  - `/dredgers` – компонент `DredgerList` для списка землесосов. Показывает таблицу: номер, тип, возможно текущее состояние (например, "в эксплуатации", если есть поле). Кнопки "Добавить землесос" (для админа).
  - `/dredgers/new` – компонент `DredgerForm` для добавления (поля: номер, выбор типа).
  - `/dredgers/{id}` – компонент `DredgerDetail`: информация о землесосе, включая, возможно, список текущих агрегатов и их наработку (вытягивается GET `/api/dredgers/{id}/components/`). Эта же страница может содержать например вкладки: "История ремонтов" и "История отклонений" – то есть отфильтрованные списки RepairList и DeviationList, ограниченные данным землесосом. Это позволит увидеть весь жизненный цикл конкретной машины.
    - **Доступ:** инженеры, админы – полный доступ, операторы/наблюдатели – просмотр.

- `/parts` – компонент `SparePartsList` (справочник ТМЦ). Таблица с колонками: код, название, производитель, норматив, [чертеж – можно значком ссылку на скачивание].
  - Можно добавить фильтр по названию/коду.
  - **Доступ:** только инженер и админ (операторам и наблюдателям вряд ли нужен справочник, разве что просмотр?). Вероятно, операторы не должны редактировать справочник, но могут просматривать для информации. Решение: сделать просмотр доступным всем залогиненным, но кнопки добавить/редактировать только у инженера и админа.
- `/parts/new` – компонент `SparePartForm` для добавления новой запчасти. Поля: код, название, производитель, норматив (число), файл (input type="file"). Реализуем загрузку файла: создаем объект FormData и включаем в него все поля, в том числе файл, отправляем методом POST на `/api/spare-parts/` с заголовком `Content-Type: multipart/form-data`. Бэкенд (DRF) распознает это с помощью парсеров и сохранит файл.
  - **Доступ:** инженер, админ.
- `/parts/{id}/edit` – компонент `SparePartForm` в режиме редактирования (например, заменить чертеж или поправить название).
- `/types` – компонент `DredgerTypeList` – список типов землесосов, с возможностью добавлять новый тип.
- `/types/{id}` – компонент `DredgerTypeDetail` – содержит название типа и список компонентов (агрегатов), относящихся к нему. Здесь можно предоставить UI для редактирования состава: например, таблица комплектующих с кнопкой удалить, и форма добавления (выпадающий список SparePart и кнопка "Добавить в состав").
  - **Доступ:** инженер/админ (редактирование), остальные – просмотр (если вообще нужна им эта страница).

- **Навигация и ограничение доступа:** В приложении будет настроен React Router с объявлением маршрутов. Например:

```
<Routes>
  <Route path="/login" element={<LoginPage />} />
  <Route element={<RequireAuth />}>  {/* компонент-обертка для прове
pki JWT и роли */}
    <Route path="/" element={<Dashboard />} />
    <Route path="/repairs" element={<RepairsList />} />
    <Route path="/repairs/new" element={<RepairForm />} />
    <Route path="/repairs/:id" element={<RepairDetail />} />
    ... (и т.д. для других маршрутов)
  </Route>
</Routes>
```

Компонент `RequireAuth` будет перехватывать попытки доступа к защищенным маршрутам. Он проверяет, что пользователь залогинен (есть токен) и при необходимости – что у него требуемая роль. Если условие не выполнено, переадресует на `/login` или выдает сообщение "нет доступа". Для реализации проверки роли можно либо задать разные RequireAuth для разных ролей, либо внутри компонента проверять `currentUser.role`. Например, для маршрута администратора можно сделать отдельный layout.

Кроме маршрутов, ограничения проявятся в самих компонентах. Например, компонент RepairsList рендерит кнопку "Добавить ремонт" только если `role === 'Оператор'` (или инженер/админ). А SparePartsList покажет кнопки редактирования только если роль имеет права. Таким образом, часть логики RBAC на фронтенде – это условный рендеринг UI в зависимости от роли. Однако полагаться только на фронт нельзя; бэкенд все равно проверяет права на каждом запросе. Фронтенд – лишь для удобства пользователя (не показывать лишнего).

Для сложных сценариев можно реализовать централизовано: например, создать компонент `HasPermission roles={['Admin','Engineer']}` который будет рендерить children только если у пользователя роль из списка. Либо получить перечень разрешений с бэкенда и проверять их. В простом случае достаточно проверки группы (ролей). Эта концепция соответствует советам: **"Define roles and permissions at the backend,**

после логина контролировать маршруты на фронтенде на основании этих permissions". Мы следуем этому принципу: все критичные проверки – на сервере, а на фронте – дублирующая защита UI.

Авторизация, логирование и загрузка файлов

Авторизация (AuthN) и распределение ролей (AuthZ): После того как пользователь вводит логин/пароль и получает JWT-токен, все запросы от его имени отправляются с этим токеном. Бэкенд на основе токена определяет `request.user` (DRF умеет декодировать JWT и получить ID пользователя). Далее, у каждого ViewSet/APIView установлены соответствующие `permission_classes`. Например, для просмотра списков может стоять `IsAuthenticated`, а для создания ремонта – кастомный пермишен `IsOperator` (который проверяет, что `request.user` в группе "Оператор" или "Инженер"). Для удаления – `IsAdmin` и т.д. Таким образом, **role-based access control** реализовано совместно средствами Django (группы/permissions) и DRF. Django REST Framework имеет удобный механизм PermissionClasses, позволяющий управлять доступом различных пользователей к разным частям API. Мы формируем простой подход: оператор может создавать новые записи, но не удалять; инженер может редактировать все записи и управлять справочниками; администратор – полные права, включая управление пользователями; наблюдатель – только чтение.

На фронтенде, как описано, React Router защищает маршруты компонентом RequireAuth. Кроме того, можно сделать, чтобы при попытке доступа к странице без нужных прав, пользователь перенаправлялся (например, если обычный оператор вручную введет URL админской страницы справочников, можно проверить роль и перенаправить). Часто это делают через обертки маршрутов или внутри компонентов с эффектом, проверяющим role.

Логирование действий пользователя: Каждое изменение данных (создание или редактирование записей о ремонте, простое, справочниках) фиксируется на сервере. Механизм следующий:

- **Через поля модели:** как упоминалось, модели Repair, Deviation и др. имеют поля `created_by`, `updated_by`, `created_at`, `updated_at`. Эти поля

автоматически проставляются текущим пользователем и временной меткой. Это позволяет всегда видеть в записях, кто их создал и когда, а также кто правил последним. Например, в Serializer можно использовать `CreatedBy = serializers.StringRelatedField(default=CurrentUserDefault())`, либо в ViewSet:

```
def perform_create(self, serializer):
    serializer.save(created_by=self.request.user)
def perform_update(self, serializer):
    serializer.save(updated_by=self.request.user)
```

После этого при каждом POST/PUT пользователь проставится.

- **Audit log (опционально):** Для более подробного журнала изменений можно использовать Django signals или готовые пакеты. Например, библиотека **django-auditlog** умеет автоматически сохранять историю изменений моделей (что изменилось, старые и новые значения). В рамках данного проекта можно это рассмотреть, если нужна полная прослеживаемость. Но требование ТЗ – фиксировать "кто и когда создал/изменил", что уже реализовано в полях. Эти данные можно отобразить во фронтенде (например, в RepairDetail показывать "Создано оператором Иванов И.И. 01.05.2025, последнее изменение инженером Петров П.П. 03.05.2025"). В админ-панели Django такие поля тоже отображаются. Дополнительно, можно настроить сохранение логов действий в текстовые файлы сервера (Django logging) или отправку уведомлений об изменениях, но это за рамками задачи.
- **Логирование входов/аутентификации:** Django автоматически логирует неуспешные попытки входа через AdminSite, но для API можно добавить ручной лог (например, запись "пользователь X вошел в систему в такое-то время", но в ТЗ это не просили).

Загрузка и хранение файлов чертежей:

- **На сервере (Django):** Модель SparePart содержит поле `drawing_file = models.FileField(upload_to="drawings/")`. В настройках определяем `MEDIA_ROOT` (путь на диске, где сохранять файлы) и `MEDIA_URL` (префикс для запросов к файлам). Например, `MEDIA_ROOT = /var/www/app/media`, `MEDIA_URL = /media/`. При разработке добавляем в `urls.py`: `urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)`

для отдачи файлов. В продакшене статическую раздачу можно поручить веб-серверу (NGINX).

- **На фронтенде (React):** Для загрузки файла используется `<input type="file">`. Когда пользователь выбирает файл (чертеж), компонент сохраняет его в состояние (например, `selectedFile`). При сабмите формы мы не можем отправлять JSON, содержащий файл напрямую, нужно использовать FormData. Создаем объект `const formData = new FormData();` и добавляем в него поля: `formData.append('code', code); ... formData.append('drawing_file', selectedFile);`. Запрос делаем через axios/fetch с заголовком `multipart/form-data` (причем браузер сам установит необходимый boundary). Django REST Framework ожидает, что файлы придут именно в формате multipart form-data, поэтому с фронта это обязательное условие. На стороне Django во ViewSet нужно указать `parser_classes = [MultiPartParser, FormParser]` чтобы парсить входящие данные. DRF Serializer для SparePart должен иметь поле `drawing_file = serializers.FileField(...)`. При успешном POST, сервер сохранит файл (например, в `media/drawings/`) и вернет остальные данные. Фронтенд может сразу отобразить, что файл прикреплен (например, иконку чертежа).
- **Просмотр/скачивание чертежа:** В SparePartList можно сделать иконку или ссылку "Чертеж". URL для скачивания – это `MEDIA_URL + path`, который приходит в поле `drawing_file` из API (например, `"drawing_file": "/media/drawings/engine123.pdf"`). При клике можно открыть в новой вкладке (если PDF, браузер покажет) или скачать.
- **Ограничения:** Можно контролировать размер файла и формат. Например, в модели ограничить upload_to по расширению (сделать свою валидацию в Serializer – разрешить только .pdf, .png и т.п.). Если файл слишком большой, DRF вернет ошибку (можно настроить MAX_UPLOAD_SIZE). Но обычно чертежи – это PDF или картинки умеренного размера.

Отображение и фильтрация истории наработки и отклонений

Для эффективного анализа данных инженерами, приложение предоставляет удобные интерфейсы фильтрации и представления истории.

История наработки агрегатов: Поскольку каждый важный агрегат имеет ограничение по моточасам, важно отслеживать его наработку и перемещения. Реализуются следующие подходы к отображению этой информации:

- **На уровне землесоса:** страница деталей землесоса содержит раздел "Агрегаты и наработка". Здесь выводится список всех агрегатов, установленных на данный момент, с их текущей наработкой и нормативом. Например:

  ```
  Двигатель – наработка 5200 ч из 10000 ч (52%), агрегат ID 1234, устан
  овлен 01.03.2025
  Насос – наработка 1200 ч из 8000 ч (15%), агрегат ID 5678, установлен
  10.04.2025
  ...
  ```

  Эти данные получаются из ComponentInstance (total_hours) и SparePart.norm_hours. Можно визуально подсвечивать агрегаты, ресурс которых близок к предельному (например, >80% нормативы). Информация об дате установки можно извлечь как дату последнего ремонта, в котором агрегат появился – при необходимости можно сохранить в ComponentInstance поле `installed_at`.

- **История замен по агрегатам:** В той же странице можно сделать для каждого агрегата кнопку "История". При нажатии – запрос к `/api/components/{id}/history/` и отображение модального окна или выпадающего списка с записями, где этот агрегат участвовал. История может выглядеть так:

  ```
  Агрегат ID 1234 (Двигатель):
  - 01.03.2025: установлен на землесос #5, наработка на момент устано
  вки 5000 ч.
  - 01.09.2025: снят с землесоса #5 при наработке 6000 ч.
  - 01.10.2025: установлен на землесос #7, наработка на момент установ
  ки 6000 ч.
  ...
  ```

Такие данные можно получить из записей RepairItem (где component_id = 1234). Таким образом инженер видит, как агрегат набирал часы и переходил. Если агрегат списан (достиг предела), историю тоже видно.

- **История ремонтов и отклонений по землесосу:** На странице землесоса можно также отобразить временную шкалу (timeline) или таблицу событий:
  - *Временная шкала:* события сортируются по дате, где ремонт и простой показываются в хронологическом порядке. Например:

    ```
    [10.02.2025] Ремонт: Замена насоса (наработка насоса 7500 ч)
    [15.03.2025] Отклонение: Механический простой, причина - вибрация насоса
    [01.06.2025] Ремонт: Плановый осмотр, без замен (двигатель 5400 ч, насос 1200 ч)
    [20.07.2025] Отклонение: Электрический – отказ генератора
    ```

    Это позволяет видеть связь: например, отклонение 15.03.2025 (вибрация насоса) вскоре после замены насоса в феврале. Такая визуализация поможет инженеру анализировать надежность.
  - *Таблично:* можно вывести две таблицы: "Ремонты землесоса #5" и "Отклонения землесоса #5", либо объединить в одну, добавив колонку "тип события". Фильтр по землесосу в RepairList и DeviationList фактически делает то же, но на отдельной странице можно сразу отфильтровать.
  - **Фильтрация по дате:** Полезно иметь возможность ограничить период, например, посмотреть историю за последний год. Фронтенд может предоставить date picker для "с" и "по". Эти параметры будут включаться в запросы (например, `?start_date__gte=2025-01-01&end_date__lte=2025-12-31`). Бэкенд через фильтры DRF отфильтрует результаты.

Фильтрация отклонений: В разделе "Отклонения" (простоя) предусмотрены фильтры:

- По типу простоя: выпадающий список с опциями (Все, Механический, Электрический, Технологический). Выбор сразу обновляет список, либо

по кнопке "Применить фильтр".

- По землесосу: выпадающий список всех землесосов (или поле автозаполнения, если их много). Можно, например, быстро выбрать конкретную машину и увидеть только ее простои.
- По участку: тоже выпадающий (ПНС, ТВС, ШХ, ...).
- По дате: два поля или готовый компонент "Date range".

  Компонент `DeviationsList` управляет состоянием фильтров и формирует запрос к API. Например, если выбран тип "механический" и землесос #5, он вызовет `GET /api/deviations/?type=mechanical&dredger_id=5`. DRF с DjangoFilterBackend умеет применять такие фильтры к QuerySet автоматически. Если нужна более гибкая логика (например, поиск по части описания), можно подключить SearchFilter или писать ручной фильтр. Но в наших требованиях фильтрация достаточно прямолинейна (точные совпадения по полям).

Фильтрация ремонтов: Аналогично, в списке ремонтов можно фильтровать:

- По землесосу,
- По дате (периоду),
- По наличию замен (например, показать только ремонты, где были заменены агрегаты – такого требования нет, но можно добавить чекбокс, а фильтровать по наличию RepairItem с примечанием или флагом замены).
- По типу ремонта, если классифицируются (тоже не указано, но можно маркировать плановый/внеплановый в примечании).

Взаимодействие фильтров: можно делать их независимыми или совмещать (все выбранные критерии применяются одновременно). Предусмотрена кнопка "Сбросить фильтры" для возврата ко всем данным.

Интерактивность и обновление данных: После добавления новой записи (ремонта или отклонения) она должна появляться в списке автоматически. Можно добиться этого несколькими путями:

- Обновлять состояние списков на фронтенде (например, после POST успешного, добавить запись в массив ремонтов в RepairsList состоянии, чтобы не делать лишний запрос).

- Либо перенаправлять на список и просто вызывать обновление (GET) заново.

  С учётом нечастого добавления, простой способ – перезагрузить список с сервера.

Отчеты и выгрузки: Для пользователя с правами (инженер/админ) интерфейс может предлагать экспорт таблиц. Например, на странице отклонений кнопка "Экспорт в Excel" – вызывает GET `/api/reports/deviations_excel/` (с теми же фильтрами, если применены). Браузер скачивает файл. Этот функционал реализуется, как отмечалось, через генерацию Excel на сервере, используя pandas/openpyxl. Фронтенд может просто открывать ссылку в новом окне (если токен в cookie) или получать blob через fetch (требуется добавить `responseType: 'blob'` для axios) и затем создавать ссылку для скачивания.

Визуальное отображение данных:

- В дополнение к таблицам, можно внедрить графические элементы. Например, диаграммы: ресурс агрегатов (проценты от нормативы) – в разделе землесоса, график простоев по месяцам – на dashboard. Эти элементы не были прямым требованием, но могут повысить наглядность. Реализовать их можно с помощью библиотек (Chart.js, Recharts) на основе данных API (можно сделать эндпоинт агрегированной статистики, но в простом случае фронт сам посчитает).
- Для большой истории (например, у землесоса эксплуатируемого 10 лет) – timeline удобно реализовать с прокруткой или пагинацией по годам. Но минимально можно использовать стандартный компонент списка с сортировкой.

Удобство для пользователя: короткие таблицы постранично (pagination) не так критичны, если данных сотни – React легко отобразит. Но DRF поддерживает пагинацию по умолчанию (например, PageNumberPagination). Можно ее включить: тогда фронтенд должен уметь подгружать страницы (навигация страницами или "Load more"). Для журналов на несколько сотен записей это не критично, но на будущее масштабирование учитывается.

Подводя итог, система обеспечивает детальное ведение журнала ремонтов и отклонений: **инженеры получают инструменты фильтрации и поиска** нужных данных (например, найти все случаи электрических отказов на

участке ТВС за последний квартал), а также **прослеживать историю агрегатов** и состояние техники. Операторы же получают понятный интерфейс ввода данных с минимизацией ошибок (шаблоны агрегатов, выпадающие справочники, подсказки по нормативам). Все данные хранятся локально в SQLite, но архитектура позволяет перейти на более мощную СУБД при необходимости без изменения кода (Django ORM абстрагирует хранение).