ChatGPT

Банк ЛФМШ – Техническое задание

Обзор проекта

Банк ЛФМШ – это внутренняя система учета виртуальных валют и достижений в лагере ЛФМШ. Проект представляет собой переработку существующего банка на новом технологическом стеке (FastAPI вместо Django) с чистой архитектурой REST API без лишнего boilerplate-кода. Система позволит педагогическому составу управлять балансами “пионеров” (участников лагеря), отслеживать их успехи (счетчики посещений занятий и мероприятий) и применять различные транзакции (начисления, штрафы, переводы и т.д.).

Основные цели переработки: - Перенести функциональность на **FastAPI** (Python) для бэкенда, сохранив существующий API, чтобы фронтенд (React + Vite) мог бесшовно взаимодействовать с новым сервером. - Использовать **JWT-аутентификацию** для защиты API. - Обеспечить **чистую структуру кода** и единый подход к конфигурации (все константы и параметры – в одном конфигурационном файле). - Минимизировать ручной труд и ускорить разработку (MVP за ~3 дня, полный релиз за ~1 неделю), в том числе за счет повторного использования готовых решений (Docker, ORM, JWT-библиотеки, и др.). - **Docker-compose** будет использоваться для контейнеризации: в состав входят Nginx (reverse proxy), backend (FastAPI), frontend (React) и база данных Postgres – это облегчит развертывание и тестирование системы.

Уже реализована часть фронтенда, которая ожидает определенные эндпоинты (согласно файлу `api.ts`), поэтому важно соблюсти формат данных и маршруты API.

Архитектура и стек технологий

Проект развертывается в виде микросервисов под управлением **Docker Compose**: - **Backend**: приложение на FastAPI (Python 3.x). Оно предоставляет REST API на порту 8000 (например, базовый URL `http://backend:8000/api/`). В коде будут использоваться Pydantic-модели для валидации запросов/ответов и SQLAlchemy (или аналог) для работы с PostgreSQL. - **Frontend**: одностраничное React-приложение (Vite) – обеспечивает пользовательский интерфейс. Фронтенд обращается к backend API (например, все запросы на `/api/...` проксируются). - **Nginx**: выполняет роль обратного прокси. Он раздает статику фронтенда и перенаправляет запросы к API на FastAPI-сервис. Например, запросы к `http://<host>/api/` → FastAPI, все остальное → React. - **Database**: PostgreSQL для хранения данных (пользователи, транзакции, счетчики и т.д.). Данные хранятся в Docker volume, чтобы сохраняться между перезапусками.

Конфигурация (пароли, ключи) задается через `.env` файлы, подключаемые к контейнерам: - **JWT секрет** для подписи токенов, - креденшелы к базе данных, - прочие настройки (например, размер ежедневного налога, штрафов и т.д., могут храниться в отдельном config-файле внутри приложения).

Модели данных и структуры

Ниже описаны основные сущности системы и их атрибуты, вытекающие из функциональных требований:

Пользователь (Account)

Пользователь представляет аккаунт участника системы банка. Сущность пользователя включает: - **Идентификатор**: уникальный `id` (целое число, PK). - **Имя пользователя (username)**: уникальный логин для входа. - **Пароль**: хранится в захешированном виде (например, алгоритм bcrypt). Аутентификация выполняется с помощью JWT-токенов, выдаваемых при вводе корректных учетных данных. - **ФИО / имя**: реальные имя и фамилия (для отображения). В API часто используется поле `name` – для полного имени. - **Роль**: тип пользователя. Возможны роли: - **Пионер** (участник лагеря) – основной тип. Имеет личный счет и счетчики достижений. - **Педсостав** (сотрудник/вожатый) – имеет административные права в системе. - **Менеджер банка** – особый подтип педсостава с максимальными правами (может выполнять операции управления системой). *Примечание:* Менеджер банка, по сути, является сотрудником с расширенными полномочиями (может создавать/удалять учетные записи, менять настройки системы и др.). В базе можно хранить флаг `is_staff` (педсостав) и дополнительный флаг `is_manager` либо единое поле `role` с перечислимым типом. - **Статус**: активен или неактивен. Неактивный аккаунт не может входить в систему и осуществлять операции (используется для блокировки либо архивирования пользователей, например, если участник выбыл). - **Отряд (party)**: номер отряда/группы, к которому принадлежит пионер. Указывается для пионеров (целое число, например 1…N). У педсостава и менеджера может быть `null` или 0 (не принадлежит отряду). - **Баланс**: количество основной валюты на счету пользователя. В системе используется внутренняя валюта — условные *«баксы»* (обозначается символом @). Баланс хранится числом (целое или дробное значение, скорее всего `DECIMAL` с двумя знаками после запятой для валюты). У педсостава баланс может не использоваться (может оставаться 0, так как педсостав обычно не оперирует личными средствами, но для унификации поле присутствует). У пионеров баланс изменяется транзакциями. - **Сертификаты**: второй тип ценности на счету. Сертификаты начисляются в особых случаях (например, награда за достижения) и хранятся отдельно от основных “баксов”. Можно реализовать как отдельное поле `certificates` на модели пользователя либо как часть баланса с типом. В текущем API-интерфейсе сертификаты упоминаются в транзакциях (поле `certs`), поэтому уместно хранить их отдельно. - **Счетчики (counters)**: показатели посещения/успехов пионера. Включают: - Количество **лабораторных** работ (например, `lab`), - Количество **лекций** посещенных (`lec`), - Количество **семинаров** (`sem`), - Количество **факультативов** (`fac`).

В системе эти счетчики отражают участие ребенка в образовательных активностях. Они увеличиваются при проведении соответствующих типов транзакций (см. ниже “Посещение факультатива” и т.п.). Для удобства в API они предоставляются как массив объектов `{ counter_name, value, max_value }`. Здесь `value` – текущее значение счетчика, а `max_value` – плановое или требуемое значение (норма) для данного вида активности. **Пример:** если от пионера ожидается посетить 3 семинара за смену, то для счетчика семинаров `value` может быть 2, `max_value` = 3, и на основе этого считается возможный штраф за недобор. Все нормативы (max_value для счетчиков) задаются глобально в конфиге и могут зависеть от контекста (например, от продолжительности смены или класса/отряда пионера). - **Ожидаемый штраф (expected_penalty)**: вычисляемое поле, актуальное только для пионеров. Это прогнозируемая сумма штрафа, которую пионер получит *к конечной дате* при текущем состоянии счетчиков. Например, если у ребенка не

хватает определенных посещений или есть пропуски, система заранее покажет, сколько с него спишут в образовательный штраф. Это поле не хранится в БД, а рассчитывается “на лету” при запросе `GET /users/me/` или `/users/{id}/` на основе текущих данных и настроек штрафов.

- **Аватар**: фото профиля пользователя. Для ускорения разработки мы храним аватар как Base64-строку прямо в базе (тип текст или binary). Таким образом, при запросе данных пользователя можно возвращать аватар закодированным Base64 (например, JSON-поле `avatar`), и фронтенд сможет его отображать. В будущем можно заменить на хранение файлов (например, в облаке или в файловой системе), но для прототипа Base64 удобен (минимум инфраструктуры). Размер аватара может быть ограничен (например, не более 1-2 МБ в эквиваленте Base64).

Права и возможности пользователей согласно роли:

- **Пионер**:
- Может просматривать свой баланс и счетчики (через `users/me/`).
- Получает средства и штрафы от педсостава, может **инициировать P2P-транзакции** (перевод другому пионеру), но такие транзакции требуют подтверждения педсоставом.
- Не может напрямую изменять свой баланс (кроме как запрашивать перевод) или просматривать данные других пионеров (кроме, возможно, ограниченной информации, например, членов своего отряда – это можно добавить позже, не в MVP).

- Состоит в определенном отряде (для учета соревнований между отрядами, групповой статистики и т.д.).

- **Педсостав (вожатые, преподаватели)**:

- Имеют право **совершать любые транзакции** с любыми счетами пионеров: начислять или списывать “баксы” и сертификаты, а также изменять счетчики (например, отметить посещение или пропуск занятия).
- Может просматривать список всех пионеров и их показатели, а также историю **всех транзакций** в лагере.
- Может подтверждать или отклонять P2P-транзакции, инициированные пионерами.
- Может отменять транзакции (см. раздел про транзакции).

- Может просматривать **общую статистику** по лагерю (например, суммарный бюджет, средний баланс, статистику посещений и штрафов).

- **Менеджер банка**:

- Обладает всеми правами педсостава, плюс дополнительными административными возможностями:
- Управление пользователями: создание новых аккаунтов, изменение ролей (например, назначить педсостав или нового менеджера), активация/деактивация пользователей.
- Управление глобальными настройками: изменение значений констант (размер налога, штрафов, нормативов по счетчикам и пр.) без правки кода – в идеале через конфигурационный файл или административный интерфейс.

- Выполнение системных операций: например, запуск ежедневного налога вручную (если это запускается не автоматически), генерация отчетов, наполнение базы мок-данными в режиме отладки и т.д.

Транзакция (Transaction)

Транзакция отражает любое изменение баланса и/или счетчиков пользователя. Это центральная сущность, записывающая операции в банке. Структура транзакции: - **ID транзакции**: уникальный идентификатор (PK). - **Автор (author)**: пользователь, создавший транзакцию. Обычно это сотрудник педсостава или менеджер. Если транзакция типа P2P инициирована пионером, то автором будет сам пионер, но такая транзакция будет иметь статус “ожидает подтверждения” (pending) до одобрения. - **Тип транзакции (type)**: категория или причина транзакции. Примеры типов: - **Начисление** – общее положительное начисление “баксов” (бонус, приз и т.д.). - **Штраф** – вычет “баксов” (наказание за проступок, пропуск и т.п.). - **Помощь педсоставу** – поощрение за помощь воспитателям (может быть начислением определенной суммы или, например, сертификата). - **P2P перевод** – перевод средств от одного пионера другому. Требует одобрения: создается как pending, после одобрения педсоставом средства списываются у отправителя и зачисляются получателю. - **Ежедневный налог** – специальный тип транзакции, который системно списывает фиксированную сумму со всех пионеров ежедневно. - **Образовательный штраф** – штраф за невыбранные/непосещенные обязательные активности к контрольной дате (например, в конце смены). - **Пропуск лекции** – штраф за пропущенную лекцию. Особенность: сумма штрафа увеличивается с каждым повторным пропуском (см. Логика штрафов ниже). - **Спортивное достижение** – начисление за спортивные успехи (например, получение сертификатов или баллов без валюты, либо наоборот – педсостав может выдать наградные “баксы” за победу в соревновании). - **Посещение факультатива** – особый тип транзакции без денежных изменений: используется педсоставом, чтобы отметить посещение мероприятия (например, факультатив, семинар). Такая транзакция **не дает денег**, но увеличивает соответствующий счетчик (в данном случае счетчик `fac` – факультативы). Аналогично могут быть транзакции "посещение лекции/семинара/лаборатории" для учета посещаемости (в MVP можно начать с факультативов как примера).

Тип транзакции сохраняется строкой или перечислимым типом (Enum) в базе. Это позволит фильтровать и обрабатывать их по-разному в бизнес-логике.

- **Описание (description)**: текстовое описание операции, вводимое автором. Например, “Штраф за опоздание на зарядку” или “Награда за победу в конкурсе”. Если транзакция массовая (см. ниже), описание общее для всех получателей.
- **Статус (status)**: состояние транзакции. Возможные статусы:
- **Completed (завершена)** – транзакция проведена и все балансы обновлены.
- **Pending (ожидает)** – транзакция создана, но ожидает подтверждения. Применимо к P2P переводам (до одобрения) или потенциально к каким-то штрафам, применяемым по событию (например, штрафы можно создавать заранее и применять позднее – но в нашем случае, вероятно, только P2P).
- **Canceled (отменена)** – транзакция была отменена и не повлияла на баланс *либо* ее эффект откачен. Отменять может педсостав/менеджер (например, если ошибочно начислили не тому или неверную сумму). **Важно:** при отмене уже завершенной транзакции нужно решить, как поступать: либо запрещаем отмену после проведения (отмена возможна только пока pending), либо при отмене выполняем обратную операцию (например, сторнирующий отрицательный транзакция). В рамках MVP планируем: P2P можно отклонить (не провести) – тогда статус

“canceled” и баланс не менялся; для обычных транзакций педсостав может отменить только сразу же, пока информация свежая (либо выполнить отдельной транзакцией компенсацию). Мы введем отмену как возможность пометить запись отмененной, а в бизнес-логике решить, как она влияет на баланс (возможно, будем хранить флаг и не учитывать отмененные транзакции в итоговых суммах).

- **Rejected (отклонена)** – опционально, можем выделить отдельный статус для явно отклоненных P2P-запросов, чтобы отличать от просто отмененных по ошибке. Но допускается и объединение с “Canceled”.

- **Дата и время создания (date_created)**: метка времени, когда транзакция была создана (и проведена, если завершена). Хранится в UTC в БД, фронтенд может отображать в локальном времени лагеря. Формат ISO 8601 в API.

- **Получатели (receivers)**: список объектов получателей и деталей начислений. Одна транзакция может затрагивать **нескольких получателей одновременно**. Например, в интерфейсе предусмотрено совершение массовой транзакции сразу нескольким пионерам: педагог выбирает группу пионеров, пишет одно общее описание, но каждому может назначить свою сумму. Модель данных должна это поддерживать:
- Для реализации можно использовать отдельную таблицу `TransactionRecipient` со связью многие-к-одному к `Transaction`. Поля `TransactionRecipient`: ссылка на пользователя-получателя, сумма `bucks` (изменение основного баланса), сумма `certs` (изменение сертификатов), изменение счетчиков `lab/lec/sem/fac` (целые числа, обычно 0 или 1 в случае отметки посещения, либо в случаях штрафов за пропуски – тоже 1 увеличение счетчика пропусков).
- В JSON представлении (как на фронтенде) каждый получатель выглядит как:

```
{
  "username": "<логин получателя>",
  "bucks": <число>,
  "certs": <число>,
  "lab": <число>,
  "lec": <число>,
  "sem": <число>,
  "fac": <число>
}
```

Здесь `bucks` и `certs` могут быть положительными (начисление) или отрицательными (штраф). Поля счетчиков `lab/lec/sem/fac` обычно используются для транзакций посещаемости: например, транзакция типа "посещение факультатива" будет иметь `fac: 1` (то есть увеличить счетчик факультативов на 1) и `bucks: 0`. Для штрафа за пропуск лекции, наоборот, может быть `bucks: -X` и одновременно `lec: 1` в смысле увеличение счетчика пропущенных лекций (если мы ведем учет пропусков как отдельный счетчик) – хотя можно реализовать пропуски и просто через штраф без отдельного счетчика, но лучше учесть для формулы.

В базе храним транзакцию и связанных получателей. При выполнении транзакции: - Балансы всех указанных получателей увеличиваются или уменьшаются на соответствующее значение `bucks` и `certs`. - Счетчики пользователей изменяются согласно полям `lab/lec/sem/fac`. (Эти счетчики также отражаются в модели пользователя, поэтому нужно или сразу обновлять поля в таблице пользователей, или хранить каждое посещение как отдельную запись – компромисс между производительностью и простотой: в прототипе можно обновлять агрегаты прямо в пользователе для простоты). - Если транзакция **pending**, то изменения применяются только после смены статуса на “Completed” (например, по одобрению). - Если транзакция **отменена/отклонена**, изменения не применяются (или откатываются).

Дополнительные структуры и настройки

- **JWT-токены**: Аутентификация по JWT. При входе (эндпоинт `/auth/jwt/create/`) пользователь получает **access token** (например, на 1 час) и **refresh token**. Refresh токен хранится у клиента (в localStorage, как видно из фронтенда) и используется для получения нового access-токена по эндпоинту `/auth/jwt/refresh/`. Мы будем использовать стандартный подход: access-токен (в заголовке Authorization: Bearer) обязателен для всех защищенных запросов. При 401 ошибке (истек токен) фронтенд автоматически пытается обновить токен. В случае недействительности refresh-токена – требуется повторный вход. Реализация: библиотека SimpleJWT (если бы был Django) или аналоги для FastAPI (например, `fastapi-jwt-auth` или собственная реализация с PyJWT). Главное – сохранить совместимыми маршруты и формат ответов:
- POST `/auth/jwt/create/` – принимает `username` и `password`, возвращает JSON с полями `access` и `refresh`.
- POST `/auth/jwt/refresh/` – принимает `refresh` токен, возвращает новый `access`.
- (Возможно `/auth/jwt/verify/` для проверки токена – не обязательно для MVP).
- **Конфигурационный файл**: Все важные константы системы будут вынесены в отдельный модуль, например, `config.py` или `settings.py`. Это включает:
- Размер ежедневного налога (фиксированная сумма списания в день с каждого пионера).
- Даты контрольных событий (например, дата финального образовательного штрафа, возможно середина смены “экватор” – если требуется).
- Нормативы по счетчикам: сколько минимум лекций/семинаров/лабораторий/факультативов должен посетить пионер к концу смены (и, возможно, к середине) чтобы избежать штрафа.
- Размеры штрафов:
    - Базовая сумма штрафа за один пропуск лекции и шаг увеличения (для прогрессии).
    - Сумма штрафа за недобранные занятия (например, за каждую не посещенную обязательную активность – алгоритм расчета, см. далее).
    - Сумма ежедневного налога.
    - Прочие константы (может быть, стартовый баланс, размер вознаграждений за типовые достижения и т.д.).
- Эти константы менеджер банка сможет менять (пока что – вручную правкой файла или через админ-интерфейс, если успеем сделать).

Функциональные требования и бизнес-логика

В данной секции описаны ключевые функции системы и правила, которым она следует:

- **JWT авторизация**: как описано выше, все запросы к защищенным ресурсам требуют валидного JWT. Реализуем middleware в FastAPI, проверяющий токен, и endpoints для получения/обновления токенов. Без JWT доступ разрешен только на страницу логина и, возможно, статические ресурсы. Период жизни access-токена небольшой, refresh-токена – более длительный (например, несколько дней).

- **Управление аккаунтами**:

- Регистрация новых пользователей не осуществляется самими пионерами – аккаунты создаются централизованно (например, менеджером банка до начала смены загружается список участников). Возможно, для ускорения, мы сначала занесем пользователей вручную в БД или миграцией. Предусмотрим, что менеджер банка может добавить пользователя через отдельный вызов API (не в MVP, но структура ролей готова).
- Активировать/деактивировать аккаунт: менеджер может поменять статус (например, если ребенок выбыл, пометить неактивным – тогда он не сможет залогиниться).
- Просмотр списка пользователей: эндпоинт GET `/users/` возвращает список всех пользователей (поля: id, username, name, party, staff, balance) – используется педсоставом для обзора. Пионерам этот список, скорее всего, не показывается (или можно ограничить на бэке по роли).
- Просмотр профиля пользователя: GET `/users/{id}/` – детальная информация о пользователе (в структуре UserData: включая счетчики, expected_penalty и т.д.). Педсостав может получить данные любого пионера, пионер – только свой (будет проверка: если пользователь запрашивает не свой id и он не staff – возвращать отказ).

- Просмотр собственного профиля: GET `/users/me/` – возвращает детальные данные текущего залогиненного пользователя.

- **Транзакции (операции)**:

- **Создание транзакции**: основной метод POST `/transactions/create/`. Данные запроса соответствуют интерфейсу `TransactionCreate` из фронтенда:

```
{
  "type": "<тип>",
  "description": "<описание>",
  "recipients": [
    { "id": <user_id>, "amount": <число> },
    ... (несколько получателей)
  ]
}
```

Здесь `amount` подразумевает сумму в **баксах** (для сертификатов и счетчиков отдельных полей нет в запросе MVP, поэтому:

- Если нужно начислить сертификаты или изменить счетчики без денег, это будет определяться полем `type`: например, если `type = "fac_attend"` (посещение факультатива), то бэкенд интерпретирует запрос как: сумма `amount` игнорируется или равна 0, и вместо этого увеличит счетчик `fac` у указанных пользователей.
- Для простоты на стороне фронтенда пока каждый такой особый тип может просто отправлять amount = 0.) **Логика обработки:**
- Бэкенд проверяет право: только педсостав или менеджер могут выполнять произвольные транзакции. Пионер, если вызывает этот метод (теоретически для P2P), то система должна определить, что это P2P: например, если автор не staff, значит это запрос на P2P перевод. В таком случае:
  - Игнорируем поле `type` присланное от пионера? Логично, пионер не должен сам определять тип – скорее фронт при P2P будет ставить `type = "p2p"`. Мы поддержим тип "p2p".
  - Создаем транзакцию со статусом “Pending” и без немедленного изменения балансов. Необходим механизм уведомить педсостав об ожидающем переводе (например, педсостав просто увидит эту транзакцию в общем списке или отдельном фильтре).
- Если автор – педсостав, транзакция проводится сразу (статус “Completed”), и все указанные изменения применяются мгновенно.
- Для каждого получателя создаются записи изменения баланса/счетчиков. Балансы пользователей обновляются в той же операции (с помощью транзакции базы данных, чтобы все либо прошло, либо нет).
- Возвращается объект `Transaction` с присвоенным ID и заполненными полями, включая статус.

- **Массовые транзакции**: Как указано, один запрос может содержать несколько получателей. Например, вожатый выбрал 5 пионеров и указал, кому сколько начислить за выполнение задания – будет создан один `Transaction` с 5 записями-получателями. В итоге у каждого из 5 обновится баланс на свою сумму, а в истории это отобразится как одна запись (с возможностью в UI раскрыть список получателей).

- **Отмена транзакции**:

  - Для транзакций, ожидающих подтверждения (pending P2P), отмена означает **отклонить запрос**. Это действие выполняет педсостав/менеджер. Нужен отдельный эндпоинт, например, `POST /transactions/{id}/cancel/` или общий PATCH на `/transactions/{id}/` с изменением статуса. При отмене pending-транзакции просто помечается отмененной (или удаляется, но лучше хранить для истории), баланс не трогается.
  - Для транзакций, уже **проведенных по ошибке**, прямой отмены мы делать не будем (сложно автоматом откатывать финансы, чтобы не нарушить историю). Вместо этого, если нужно сторнировать операцию, педсостав может создать новую обратную транзакцию (например, если случайно начислили лишнее, сделать штраф на ту же сумму с комментарием отмены). Однако, мы предусмотрим возможность пометить

любую транзакцию как “Canceled” для исключения из статистики и явного обозначения, что она не должна была случиться. В реализации: возможно, разрешим отмену сразу после создания в короткий промежуток (до того, как другой пользователь совершит зависимые действия). **Итого**: в ТЗ укажем отмену, но в MVP ограничимся отменой pending и, при необходимости, добавим флаг отмены для completed-транзакций без автоматического перерасчета баланса.

- **Просмотр транзакций**: GET `/transactions/` – возвращает список транзакций. Формат каждого объекта соответствует интерфейсу `Transaction` на фронтенде: включает id, author, description, type, status, date_created, receivers[...].

    - Если вызывающий – педсостав или менеджер, возвращаются **все транзакции** лагеря (или, возможно, можно реализовать фильтрацию, но базово – все, отсортированные по дате, свежие первые).
    - Если вызывающий – пионер, возвращаем только транзакции, где он является получателем или автором, чтобы он видел свою финансовую историю. (В MVP можно упростить и выдавать всем всё, полагаясь на то, что UI для пионера просто не содержит общей ведомости; но лучше добавить проверку роли для безопасности).
    - Необходимо поддержать пагинацию или хотя бы ограничение по количеству, если транзакций много, но на уровне MVP можно вернуть списком (позже улучшить).

- **Просмотр статистики**: GET `/statistics/` – сводная статистика по системе, доступная педсоставу/менеджеру. В интерфейсе `Statistics` предусмотрены поля `avg_balance` (средний баланс) и `total_balance` (суммарный баланс) – скорее всего рассчитываются по всем пионерам:

    - `total_balance` = сумма всех `balance` у пользователей-пионеров.
    - `avg_balance` = средний баланс на пионера.
    - Можно дополнительно в будущем расширить статистику (например, количество активных пионеров, суммарные выданные штрафы и награды и т.п.), но пока достаточно этих показателей.
    - Пионеру этот эндпоинт не нужен (можно вернуть 403, или не отображать).

- **Ежедневный налог**: Каждый день с *каждого* пионера списывается определенная сумма “налога”. Например, условно 1 @ в день (настраивается). Реализация:

- Этот процесс должен происходить автоматически раз в сутки. Возможные подходы:
    - Крон-задача на стороне сервера (например, запуск фонового потока или использование библиотеки APScheduler внутри FastAPI, который будет каждый день выполнять функцию начисления штрафов).
    - Либо внешний скрипт/cron (развернутый на сервере хостинга) дергающий специальный endpoint.
    - Проще всего, для начала, сделать отдельный endpoint (доступный только менеджеру) `POST /transactions/daily_tax/` который сразу создает транзакцию типа "ежедневный налог" на всех пионеров (массово). Менеджер банка или cron-job будет вызывать его ежедневно. В дальнейшем можно автоматизировать полностью.
- При выполнении налоговой транзакции: создается запись Transaction с type "tax" (или "daily_tax"), author = система (можно от имени менеджера), description например "Ежедневный

налог", и receivers – список всех активных пионеров с суммой `-N` @ (минус N, N – размер налога). Таким образом, это массовая транзакция. **Важно:** если у кого-то на балансе меньше, чем налог, баланс станет отрицательным (это возможно, система не запрещает – долг пионера).

- Все суммы и логика налога сконфигурированы (N – константа, время списания – вне кода, если cron).

- **Образовательный штраф (за недобор)**: В конце смены (или в другую установленную дату) пионерам, не выполнившим образовательные нормативы, начисляется штраф. Логика (пример из требований):

- Проверяются счетчики каждого пионера: сколько семинаров, факультативов и др. он посетил. Если каких-то показателей не хватает до требуемого минимума, рассчитывается штраф.
- Алгоритм штрафа: может быть суммарный штраф за все недобранные активности. В предыдущей версии, судя по коду, штраф рассчитывался с прогрессивной шкалой за каждое неотработанное обязательное занятие. Упрощенно можно задать: за каждый недостающий семинар/факультатив – фиксированный штраф X @, за недостающую лабораторную – Y @ и т.д., или даже сложнее (например, 1-й пропуск – 5@, 2-й – 6@, 3-й – 7@...). В конфиге эти формулы заложим.
- Этот штраф, скорее всего, накладывается *два раза*: один промежуточный (на середине смены, “экваторе”) – может быть, если лагерь длинный, и финальный – в конце. В ТЗ сказано “по достижении определенной даты” (ед.ч.), можно подразумевать финальный день. Но мы заложим возможность и промежуточного контроля.

- Реализация: аналогично налогу, можно иметь endpoint или автоматический запуск в определенный день:

    - Например, `POST /transactions/final_penalty/` (только менеджер или cron) – проходит по всем пионерам, для каждого считает штраф на основе формулы и, если штраф > 0, создает транзакцию типа "educational_fine" на этого пионера (списание). Возможно, всех штрафников объединить в одну транзакцию **не получится**, так как у каждого своя сумма – хотя интерфейс позволяет разные суммы для разных получателей под одним описанием. В принципе, можно сделать одну массовую транзакцию “Образовательный штраф” с перечислением всех пионеров и их сумм штрафа. Это облегчит отмену, например (одной транзакцией отменить нельзя, правда, разве что пометить).
    - В `expected_penalty` для каждого пионера мы заранее считаем эту сумму (например, за финальный штраф) по текущим данным. Так что формула будет реализована в виде функции, использующей текущие счетчики и константы нормативов. **Пример подхода**:
    - Требуется X семинаров, Y факультативов, Z лабораторных работ.
    - Не хватает: `d_sem = max(0, X - sem)`, `d_fac = max(0, Y - fac)`, `d_lab = max(0, Z - lab)`.
    - Штраф может считаться как: `d_sem * A + d_fac * B + d_lab * C` (где A, B, C – штраф за единицу недобора каждого вида).
    - Или более сложная прогрессия: например, первый пропуск обяз. занятия – 5@, за каждый следующий на 5@ больше (5, 10, 15...). Тогда фактически: штраф = 5 * n + 5 * (0+1+...+(n-1)) = 5n + 5*(n-1)*n/2, где n = (d_sem + d_fac + d_lab) недобранных занятий.

Старый алгоритм (из Django версии) именно так и делал для обязательных занятий: суммировал приращения [1] [2].
    - Мы упростим: можно сделать линейно увеличивающийся штраф за каждый пропуск обязательной активности (семинар/факультатив) – например, 1-й пропуск 5@, 2-й 10@, 3-й 15@... Это стимулирует участников выполнять нормы.
    - Так или иначе, к моменту вызова финального штрафа, у каждого пионера expected_penalty примерно равен тому, что будет списано.

- **Автоштраф за пропуск лекции**:

- Когда пионер пропускает лекцию (мероприятие, отмеченное как обязательное), ему выписывается штраф, причем сумма штрафа увеличивается с каждым новым пропуском у данного пионера. То есть, штрафы за пропуски имеют прогрессивную шкалу: например, первый пропуск – 1@, второй – 2@, третий – 3@ (арифметическая прогрессия с разницей 1), или с другим шагом согласно константе.

- Реализация: предполагается, что педсостав отмечает факт пропуска лекции, возможно через специальную транзакцию. Например, тип "lecture_miss". Когда вожатый создаёт транзакцию пропуска:

    - Она будет содержать `bucks: -P` (штрафная сумма) и, возможно, мы ведем счетчик пропущенных лекций (в старой системе это делалось, но можно и вывести из количества таких транзакций).
    - Сумма P вычисляется динамически на основе уже имеющихся штрафов у этого пионера: если до сих пор было `n` штрафов за пропуски, новый штраф = базовый + n*шаг. В старой системе, судя по коду, формула: `amount = LECTURE_PENALTY_INITIAL + (count_missed) * LECTURE_PENALTY_STEP` [3]. То есть, если шаг = 1@, начальный = 1@, то штрафы будут 1, 2, 3, ...
    - Как реализовать: при создании транзакции типа "lecture_miss", backend будет подсчитывать, сколько у данного пользователя *уже есть* проведенных транзакций этого типа (или хранить счетчик пропусков как часть модели пользователя). Проще хранить счетчик `missed_lectures` и обновлять его, но тогда и для штрафа можно заранее иметь формулу. Однако, надежнее считать на лету: `n = число транзакций type=lecture_miss у пользователя (со статусом Completed)`. Новый штраф = initial + n*step. Затем создается новая транзакция с рассчитанной суммой, и (опционально) счетчик пропусков увеличивается (или просто по транзакциям будет +1).
    - В итоге, каждая новая транзакция пропуска будет дороже предыдущей для того же пользователя.
    - Все константы (initial, step) – в конфиге.

- **Проверка возможности посещения мероприятия**:

- В лагере мероприятия проходят в определенные **слоты времени** (1, 2, 3 – условно утро/день/вечер). Требование: пионер не может одновременно присутствовать в двух разных активностях одного блока. Например, в 1 слот не может посетить и семинар, и факультатив; должен выбрать одно.
- Следовательно, система должна предотвращать ситуацию двойного засчитывания посещения в одном слоте.

- Реализация: нужно учитывать **расписание событий**. В MVP можно сделать упрощенно: в рамках одного типа это не возникает (нельзя два семинара одновременно — таких ситуаций нет), а вот конфликт "семинар vs факультатив" возможен если считаются разными типами. Это значит, при регистрации посещения надо проверить, не было ли у пользователя уже посещения другого типа в тот же слот.
- Как знать слот? Два подхода:
    - Добавить поле “slot” (1,2,3) в транзакцию посещения/пропуска. Например, транзакция "fac_attend" будет содержать информацию, в какой блок было посещено. Но тогда нужно от фронтенда получать этот номер блока. Пока в API такого поля нет.
    - Либо решать на уровне UI (не позволять отмечать несовместимые вещи).
    - Так как фронтенд пока не передает слот, возможно, стоит на первое время заложить, что **педсостав сам не внесет конфликтующих транзакций**. В будущем:
    - Добавим поле `time_slot` в модель транзакции или отдельную модель `Event` с расписанием.
    - Тогда проверка: при создании транзакции посещения, если у пользователя уже есть посещение/пропуск в тот же `time_slot` и дату, вернуть ошибку.

- В ТЗ отметим эту логику как требование, но реализация может быть отложена до полного релиза, когда будет время учесть в API. Возможно, к полной версии добавим эту проверку.

- **Начисления за спортивные достижения**:

- Предусматривается тип транзакций для спортивных успехов. По сути, это частный случай **начисления** или **награждения сертификатом**. Вероятно, в лагере за спортивные победы дают сертификаты (как почетные очки) вместо валюты.
- Реализация: транзакция типа "sport" или "achievement" – может начислять **сертификаты** (`certs` поле) вместо или вместе с валютой. Например, 0 @, но +1 сертификат, с описанием "Победа в футболе".
- Педсостав создаёт такую транзакцию для конкретного пионера или команды (можно и массово на команду, если все участники получают).

- Особой логики (как штрафы) тут нет, кроме того, что она влияет на другой баланс. Сертификаты могут отображаться отдельно (в UI, вероятно, просто как часть данных пользователя).

- **Посещение факультатива (и других занятий)**:

- Описано выше: это транзакции, которые **не меняют баланс**, а лишь повышают счетчики. В системе такие транзакции нужны для учета посещаемости и расчета образовательных штрафов.
- Педсостав создаёт, например, транзакцию типа "fac_attend" (посещение факультатива) с указанием пионеров (можно сразу весь список посетивших занятие, каждому поставить amount: 0). Бэкенд, видя тип "fac_attend", понимает, что нужно увеличить `fac` для каждого получателя на 1. Аналогично, можем предусмотреть типы "sem_attend" (семинар), "lab_pass" (сдана лабораторная работа), etc. Эти типы можно обрабатывать индивидуально.
- Таким образом, **счетчики** обновляются только через подобные транзакции (или штрафные транзакции, которые тоже могут влиять, напр. пропуск – может отдельно учитываться).

Педсостав в любой момент может посмотреть счетчики пионеров, а система – посчитать их отставание.

- Заметим, что **часть счетчиков может относиться к посещению (attend), а часть к сдаче зачётов (pass)**. Например, в старой системе были `lab_pass` (сдана лаба) отдельно от посещения. Мы можем не детализировать до такого уровня в MVP – считать, что любой "lab" транзакция означает выполнение лабораторной работы.

- **Гибкая настройка констант**: Все перечисленные параметры (суммы налогов, штрафов, нормы посещений, прогрессии штрафов, список типов транзакций и их поведение и т.д.) должны быть легко меняемы **в одном месте**. Будет создан файл (например, `settings/constants.py`), где определим все необходимые константы:

- `DAILY_TAX_AMOUNT = ...`,
- `LECTURE_MISS_PENALTY_INITIAL = ...`, `LECTURE_MISS_PENALTY_STEP = ...`,
- `REQUIRED_SEM = ...`, `REQUIRED_FAC = ...`, `REQUIRED_LAB = ...` (мб словарь по возрастам/отрядам, если нужно),
- `FINE_PER_MISSING_SEM = ...`, etc.
- Также там можно определить перечисление типов транзакций или хотя бы их строки, чтобы использовать консистентно (например, `TRANSACTION_TYPES = { "fine": "Штраф", "tax": "Ежедневный налог", ... }` если потребуется локализация, но для логики достаточно констант).
- При изменении этих значений менеджер перезапускает сервер (в простейшем случае) либо, если реализуем веб-интерфейс настроек, они применяются динамически.

API эндпоинты

Ниже описаны основные маршруты API и их назначение. Все URL подразумевают префикс `/api/` (например, `/api/users/me/`), как указано в `API_URL` фронтенда. Формат данных — JSON. Authorization: Bearer JWT требуется, если не указано иное.

Аутентификация: - `POST /auth/jwt/create/` – **Вход (логин)**. Принимает в теле JSON с `username` и `password`. Возвращает `access` и `refresh` токены. При неверных данных – 401 Unauthorized. - `POST /auth/jwt/refresh/` – **Обновление access-токена**. Принимает `refresh` (старый refresh-токен). Возвращает новый `access` токен. Если refresh недействителен или просрочен – 401, потребуется логин. - *(Опционально)* `POST /auth/jwt/verify/` – может быть реализовано для проверки валидности токена (не обязательная часть, зависит от выбранной JWT библиотеки).

Пользователи: - `GET /users/me/` – **Данные текущего пользователя**. Возвращает объект **UserData**:

```
{
  "username": "alexey",
  "name": "Алексей Иванов",
  "staff": false,
  "balance": 120.5,
```

```
  "expected_penalty": 15,
  "counters": [
     {"counter_name": "lec", "value": 5, "max_value": 8},
     {"counter_name": "sem", "value": 3, "max_value": 5},
     ...
  ],
  "avatar": "<base64String>"
}
```

Здесь `staff` = false означает пионер (true – педсостав), `expected_penalty` – рассчитан, `counters` – массив всех счетчиков. Педсоставу также может вернуться пустой или незначимый expected_penalty и counters (либо счетчики не нужны педсоставу). - `GET /users/` – **Список пользователей**. Возвращает массив объектов **UserListItem** для каждого пользователя:

```
[
  {
    "id": 10,
    "username": "petrov",
    "name": "Петров И.",
    "party": 3,
    "staff": false,
    "balance": 50.0
  },
  ...
]
```

Педсостав и менеджер получают всех пользователей (по умолчанию отсортировано по имени или id). Можно добавить фильтр по отряду или имя, но не обязательно сейчас. Пионеру вызов этого эндпоинта может быть запрещен (вернем 403), либо возвращать ограниченный список (например, только сам пионер и его отряд) – лучше запретить для упрощения. - `GET /users/{id}/` – **Просмотр пользователя по ID**. Возвращает объект **UserData** (аналогично /me). Педсостав может просматривать любые id пионеров (и, возможно, педсостав тоже, хотя особой нужды нет). Пионер может запрашивать только свой id (если попробует чужой – вернем 403 Forbidden). - **(Административные, для менеджера)**: - `POST /users/` – **Создание нового пользователя**. Будет использоваться менеджером банка, чтобы добавить учетку. В теле – данные пользователя (имя, логин, пароль, роль, отряд и т.п.). Пароль можно также генерировать. *Для MVP можно отложить, так как список пользователей может быть загружен миграцией.* Но в техническом задании упоминаем как будущее расширение. - `PUT/PATCH /users/{id}/` – **Обновление пользователя**. Например, смена статуса активен/нет, смена роли (повышение до педсостава) или правка имени. Доступно менеджеру. Можно реализовать частично (скажем, только статус менять) в MVP, остальное потом. - `DELETE /users/{id}/` – **Удаление пользователя**. Возможно не нужно удалять (чтобы не терять данные транзакций), лучше деактивация. Но на всякий случай пропишем как потенциальную функцию для менеджера (удаление без удаления связанной истории, если с каскадом – нельзя, надо чистить транзакции).

Транзакции и счета: - `GET /transactions/` – **Список транзакций**. Возвращает массив объектов **Transaction** (согласно описанной выше структуре) в обратном хронологическом порядке. - Поля каждого Transaction: - id, author (например, "petrov" или имя пользователя автора транзакции), - description, type, status, date_created, - receivers: список {username, bucks, certs, lab, lec, sem, fac} по каждому затронутому пользователю. - Примеры: - Транзакция штрафа: `"type": "fine", "description": "Штраф за опоздание", receivers: [ { "username": "ivanov", "bucks": -5, "certs": 0, "lab": 0, "lec": 0, "sem": 0, "fac": 0 } ]`. - Транзакция посещения: `"type": "fac_attend", "description": "Посещение факультатива 12.07", receivers: [ { "username": "petrov", "bucks": 0, "certs": 0, "lab": 0, "lec": 0, "sem": 0, "fac": 1 } ]`. - Массовая транзакция: `"type": "general", "description": "Награда за конкурс", receivers: [ {"username": "sidorov", "bucks": 10, ...}, {"username": "ivanov", "bucks": 5, ...} ]`. - Доступ: педсостав/менеджер видят все. Пионер – только свои (будет фильтрация на сервере по receiver/author). - В будущем можно добавить параметры запроса для фильтрации (по типу, дате, пользователю и т.д.), но базово – полный список.

- `POST /transactions/create/` – **Создание новой транзакции**. Принимает JSON как в примере выше (`TransactionCreate`). Обработка подробно описана ранее:

- Проверяются права:

  - Если вызывающий – педсостав/менеджер:
  - Транзакция выполняется сразу (статус “Completed”).
  - Проход по каждому recipient: обновление баланса (balance += amount) и сертификатов (certificates += ... если будем поддерживать `certs` в запросе в будущем), обновление счетчиков (если тип транзакции подразумевает изменение счетчиков).
  - Сохранение Transaction и связей в БД.
  - Возврат объекта транзакции (как бы результат, включая сгенерированный id, статус Completed, и рассчитанные поля receivers).
  - Если вызывающий – пионер (т.е. type должно быть "p2p", recipients вероятно 1 получатель):
  - Создается Transaction со статусом “Pending”.
  - **Важно:** Нужно определить, кто будет second party в P2P. Вероятно, p2p подразумевает: один пионер (отправитель) переводит часть своего баланса другому пионеру (получателю). Однако, в нашей модели Transaction, все receivers – те, кому *зачисляются* средства. Отправитель не фигурирует как receiver, кроме если мы хотим отразить списание у него.
  - Поэтому для P2P, сделаем так: когда пионер A хочет перевести X @ пионеру B:
    - Он вызывает /create с type = "p2p", recipients = [{id: B, amount: X}], description, author = A (через токен определяется).
    - Бэкенд сохранит Transaction: author=A, receiver=B, bucks = +X для B, **но** статус Pending и никаких немедленных действий.
    - Баланс A не уменьшается сразу – это важно. Нужно дождаться одобрения.
    - Когда педсостав увидит эту транзакцию (в списке она будет видна как pending от A к B) и решит одобрить:
    - Должен быть endpoint, например, `POST /transactions/{id}/approve/`. Этот вызов сделает менеджер/педсостав:
      - Бэкенд, найдя транзакцию, проверит что она pending и type=p2p.

        - Затем выполним перевод: уменьшим баланс автора (A) на X, увеличим баланс получателя (B) на X.
        - Обновим статус транзакции на Completed.
        - Возможно, добавим в Transaction.receivers и самого отправителя как участника (для истории), либо просто по тому, что author и type=p2p, фронт будет понимать, что автор потерял X. (В текущей схеме receivers не содержат отправителя; можем отразить списание отправителя как отдельный receiver с отрицательным bucks. Но тогда B как понять? *Вариант:* в массив receivers для p2p будет два записа: {username: B, bucks: +X}, {username: A, bucks: -X}. Это позволит увидеть обе стороны. Однако, фронтенд структура не указывала возможность отрицательных bucks? Она позволяет, они просто число, так что можно. Либо хранить только получателей, а отправителя понять через author. Но лучше на будущее добавить и отправителя в receivers с отрицательной суммой, для консистентности и упрощения истории).
    - Также педсостав может **отклонить** перевод: для этого endpoint /cancel (или /reject). Тогда транзакция получает статус Canceled, и никакие изменения балансов не происходят вообще.
  - Эти нюансы P2P можно реализовать сразу во Phase 2 разработки.

• Типы транзакций и их обработка:

  - Если type = "fine" или "general" (штраф или начисление) – просто списываем или начисляем `amount` на баланс.
  - Если type указывает на счетчики (например, "fac_attend", "sem_attend", "lab_pass", "lecture_miss"):
  - Выполняем соответствующее изменение счетчиков. Деньги обычно 0, кроме "lecture_miss" где есть штраф.
  - Возможно, для "lecture_miss" игнорируем присланный `amount` и сами рассчитываем штраф, переопределяем amount перед применением.
  - Важно заранее договориться с фронтендом: либо фронт отправляет 0, а бэк вычисляет, либо фронт умеет запрашивать расчет? Скорее, фронт попроще – отправит 0, бэк сам подставит нужную сумму и вернет итоговую транзакцию с правильными данными.
  - Если type = "tax" (ежедневный налог) – ожидается, что recipients охватывают всех пионеров с отрицательной суммой. Вероятно, этот тип будет вызываться только системно, а не через UI вручную, но API на всякий случай поддержит.

• **Дополнительные эндпоинты (планируемые):**

• `POST /transactions/{id}/cancel/` – отмена/отклонение транзакции. Как описано, если транзакция pending (например, p2p), то просто меняет статус на Canceled. Если уже completed (нежелательно, но если реализуем, то надо сделать компенсирующее действие – в MVP можно не делать).
• `POST /transactions/{id}/approve/` – подтверждение P2P-транзакции. Применяет изменения (списывает/начисляет деньги) и ставит статус Completed.
• `POST /transactions/daily_tax/` – инициировать ежедневный налог (менеджер либо cron). Создает транзакцию типа "tax" массово.

- `POST /transactions/final_fine/` – инициировать образовательные штрафы в установленный день (менеджер либо cron). Создает соответствующие транзакции (массово или по отдельности).
- В случае реализации системы заявок и ачивок (см. будущее) появятся эндпоинты:
    - `/requests/` – CRUD для заявок (например, ребенок отправляет POST /requests/ {type: "sport_inventory", ...} – создание заявки; педагоги GET /requests/ – список и POST /requests/{id}/complete – выполнить или т.п.).
    - `/achievements/` – возможно GET для получения списка достижений, POST для добавления (менеджером) и т.п. Пока задел, подробности не прорабатываем.

Все новые или административные эндпоинты будут защищены ролями (например, с помощью dependency в FastAPI, проверяющей `current_user.role`).

План разработки по этапам

Учитывая сжатые сроки (3 дня на прототип, неделя на полный функционал), разобьем работу на этапы, чтобы как можно скорее получить MVP – минимально работоспособную версию – и затем нарастить остальное:

Этап 1: MVP (первые 3 дня)

Цель: обеспечить базовые возможности системы – авторизацию, просмотр пользователей, просмотр/создание транзакций – чтобы фронтенд, реализованный под старый API, мог работать без ошибок. MVP будет сфокусирован на функционале финансовых транзакций и аккаунтов.

- **Настройка проекта**: Создать каркас FastAPI-приложения. Настроить Dockerfile и docker-compose:
- Образ FastAPI (Python) + Uvicorn.
- Образ PostgreSQL, применение начальной миграции или SQL для создания таблиц.
- Образ Nginx для проксирования (настроить конфиг: например, `location /api/ -> fastapi:8000`, остальные -> статика React).
- Поднять контейнер React (образ собирается из Vite проекта).
- Проверить, что фронтенд статика доступна, и запросы /api/* доходят до приложения.
- **Модель БД**: Описать модели SQLAlchemy (или Pydantic + Tortoise) для `User`, `Transaction`, `TransactionRecipient`. Выполнить миграцию.
- User: поля как описаны (в MVP можно опустить avatar, expected_penalty – не хранится, counters можно реализовать как отдельную таблицу `Counters` или как поля в User: `lab_count, lec_count, sem_count, fac_count` + возможно `missed_lectures`).
- Transaction: поля id, author_id (ForeignKey -> User), type, description, status, date_created.
- TransactionRecipient: поля transaction_id -> Transaction, user_id -> User, bucks, certs, lab_delta, lec_delta, sem_delta, fac_delta.
- Изначально можно создать минимум полей: только то, что нужно для финансового MVP (например, certs и counters изменения можно пока не использовать активно, но создать столбцы с дефолтом 0, чтобы интерфейс не ломался при парсинге JSON).
- **JWT аутентификация**: подключить библиотеку (например, `fastapi_jwt_auth` или реализовать вручную):

- Реализовать `/auth/jwt/create/`: проверить пользователя в БД (будет заранее парой записей – например, тестовый педсостав и пара пионеров – или использовать встроенную аутентификацию, если использовать FastAPI Users). Выдать JWT с нужными полями (идентификатор и признак роли).
- `/auth/jwt/refresh/`: проверяет refresh-токен (можно хранить refresh-токены в базе, или использовать неподписанный refresh-секрет – упрощенно можно JWT refresh тоже, но тогда на сервере не нужно хранить).
- Убедиться, что фронтенд получает токены и сохраняет.
- Добавить зависимость в FastAPI для защищенных маршрутов, которая декодирует JWT, достает пользователя и кладет в `current_user`.
- **Эндпоинты пользователей**: `/users/me/`, `/users/`, `/users/{id}/`:
- Реализовать контроллеры:
  - `/me/` возвращает текущего пользователя (из токена) – найти его в базе, сформировать `UserData`. Здесь нужно подсчитать `expected_penalty` (пока можно заглушкой 0 либо легкой функцией если счетчики заведены) и отдать список счетчиков (если нет отдельной таблицы, то собрать из полей).
  - `/users/` – вернуть список всех пользователей (UserListItem: id, username, name, party, staff, balance). Ограничить доступ: только staff/manager.
  - `/users/{id}/` – вернуть подробности по указанному пользователю (если автор запроса staff/manager **или** id совпадает с текущим пионером).
- Проверить через Swagger или Postman, что возвращаются данные в ожидаемом формате (с точными именами полей, как в `api.ts` интерфейсах).
- **Эндпоинты транзакций**: `/transactions/`, `/transactions/create/`:
- `/transactions/` – выборка транзакций из базы (с джойном recipients). Собрать в нужный формат JSON. Для MVP можно не реализовывать фильтрацию по пользователю на уровне SQL, а просто:
  - Если current_user.staff или manager: вернуть все.
  - Если current_user – пионер: отфильтровать список в Python по условию (user.id равен author_id или присутствует в recipients). (В дальнейшем лучше на уровне запроса).
- `/transactions/create/` – обработать по схеме:
  - Проверить JWT, взять current_user.
  - Если current_user.staff/manager:
  - Принять JSON, валидировать (через Pydantic-модель TransactionCreate).
  - Создать Transaction (status = "Completed").
  - Для каждого recipient:
    - Найти User по id, обновить его баланс += amount.
    - Если потребуется, обновить certificate и счетчики:
    - В MVP, можно отложить логику счетчиков до Этапа 2, но если успеваем – включить частично. Например, если type = "fac_attend", то value.amount, скорее всего, будет 0; тогда мы знаем, что надо user.fac_count += 1.
    - Если type = "lecture_miss", не доверять amount из запроса: вычислить штраф как описано (нужно знать уже сколько было пропусков или штрафов – можно пока просто взять count транзакций lecture_miss у этого user, или отдельный счетчик missed_lectures).
    - Обновить нужные поля пользователя.
    - Создать запись TransactionRecipient с указанными дельтами.
  - Сохранить все в рамках одной DB-транзакции.

    - Вернуть объект Transaction (возможно, удобнее сформировать по базе или на основе того же, что вставили).
    - Если current_user – пионер:
    - Обработать как P2P: установить status = "Pending".
    - Ограничить: recipients должен быть длиной 1 (один получатель) и amount <= текущий баланс отправителя, возможно, чтобы он не отправил больше чем имеет. Но можно и не строго (в долг дать не логично, лучше ограничить).
    - Сохранить Transaction (пока без изменения балансов).
    - Вернуть объект (статус Pending). Педсостав на UI ничего не делает автоматически, но мы предоставим возможность через API утвердить.
    - В MVP можно разрешить p2p, но если не успеваем интерфейс подтверждения, то либо не давать пионеру делать (UI можно скрыть кнопку для пионера) или все P2P автоматически считать pending навсегда до вмешательства вне системы.
- Включить базовые проверки: существование пользователей, не нулевая сумма (кроме типов посещения), и т.д.
- **Примечание:** транзакции типа “налог” и “финальный штраф” в MVP вручную не создаются – они будут реализованы на этапе 2 (но если захотим протестировать, можем через /create с соответствующим type от имени менеджера и указав всех).
- **Инициализация данных (мок)**: Уже на этапе 1 целесообразно иметь несколько тестовых записей:
- Внести через скрипт или миграцию:
    - 1 менеджера (например, username: admin, пароль: admin, role: manager),
    - 1-2 педсостава,
    - 3-5 пионеров с разными отрядами, начальными балансами и счетчиками.
    - Пару примерных транзакций (например, начисление, штраф) для проверки списка.
- Это поможет сразу проверить фронтенд без необходимости создавать через UI.
- Сделать настройку: если приложение запущено в режиме DEBUG (определяется env переменной), при старте выполнить заполнение тестовыми данными (но так, чтобы не дублировать при каждом запуске; можно проверять пустая ли база). Этот **debug-mode с мок-данными** включаем на dev-стенде, а на продакшене – выключен.
- **Тестирование MVP**: Проверить вручную через **Swagger UI** (`/docs`) или **Postman**:
- Получение токена (логин менеджера/педагога, также проверить, что пионер не может получить токен с чужими данными).
- Доступ к /users/me (для разных ролей).
- Список пользователей (под педсоставом и под пионером – пионеру должно отказать).
- Создание транзакции:
    - От педсостава – напр. начисление 5@ одному пионеру.
    - Проверить, что баланс обновился, /transactions/ отдает эту транзакцию, /users/{id} показывает новый баланс.
    - Попробовать массовую транзакцию – 2-3 получателя, разные суммы.
    - P2P: залогиниться пионером, попробовать перевести кому-то деньги. Убедиться, что транзакция создается pending, балансы не поменялись.
- Исправить баги, если найдутся, в пределах времени.

К концу 3-го дня ожидается работающая основа: фронтенд может получать данные и создавать простые транзакции, JWT авторизация работает.

Этап 2: Расширенная функциональность (день 4-7, до полного релиза)

Цель: реализовать все специальные правила и функции, предусмотренные ТЗ, довести систему до полного соответствия требованиям.

Основные задачи этапа: - **Реализация бизнес-логики штрафов и автоматических операций:** - **Ежедневный налог:** внедрить автоматизацию. Варианты: - Настроить планировщик (APScheduler) внутри FastAPI, который каждый день в заданное время вызывает функцию сбора налога. Это потребует, чтобы приложение было постоянно запущено (что ок) и мы учтем часовой пояс лагеря. Этот способ прост, но в Docker-контейнере может быть нюанс с время жизни – но если контейнер постоянный, нормально. - Либо реализовать cron в хост-системе (если deploy позволяет). - Для надежности на случай перезапуска, можно делать проверку: если последний налог был начислен более 24ч назад – начислить при старте. - В рамках недели, возможно, проще реализовать через APScheduler. - **Образовательный штраф (финальный и/или промежуточный):** - Определить дату(-ы) контрольных точек. Можно в конфиге указать `FINAL_FINE_DATE` (и `EQUATOR_DATE` если нужно). - Когда текущее время превышает эту дату и штрафы еще не начислялись – автоматически начислить. Здесь тоже уместно использовать планировщик: запланировать задачу на конкретное время. - Реализовать функцию расчета штрафа: опираясь на counters и константы, вычислить сумму для каждого. Создать транзакции (либо одну общую, либо индивидуальные – решим; один общий может быть очень большим с десятками получателей, но это ок). - Проставить expected_penalty = 0 у всех (т.к. после начисления его уже как бы нет, или он может пересчитываться в ноль). Либо expected_penalty всегда можно считать по формуле до конца смены, а после финального штрафа он уже не актуален. - Если требуется промежуточный штраф (экватор): аналогично, но, возможно, половинные нормативы. - **Штраф за пропуски (лекции):** - Доделать логику, оставленную на MVP: - В счетчике пользователя можно добавить поле `missed_lectures` или просто вычислять как число транзакций lecture_miss. Лучше добавить, чтобы не пересчитывать каждый раз и для expected_penalty. Если добавим – инкрементировать при каждом штрафе за лекцию. - Модифицировать создание транзакции "lecture_miss" так, чтобы вычислять amount на основе либо этого поля, либо запроса к транзакциям. Тестировать: если один пионер получает несколько таких штрафов, что сумма растет правильно. - Отображение: в counters пионера, возможно, стоит показать количество пропущенных лекций (можно завести отдельный счетчик или включить в counters как, например, counter_name: "lec_missed"). Либо не обязательно, так как штрафы и так видны. - **Учёт слотов времени:** - Если фронтенд к этому времени будет готов передавать `slot` при отметке посещения, обновим модель Transaction (добавим поле slot опционально). - Добавим проверку: при создании транзакции типа *_attend, если указано slot (и, возможно, дата – если не по умолчанию берем текущую дату): - Запросить из БД, есть ли у данного пользователя другая транзакция attend (или miss) с тем же slot и текущей датой. Если да – отклонить с ошибкой. - Если фронтенд не передает slot, можно пока зашить, например, что факультативы – слот 3, семинары – слот 2, лекции – слот 1 (если соответствует расписанию), но это негибко. Вероятно, лучше ждать поддержки UI. - В ТЗ достаточно указать, что система должна предотвратить одновременное добавление двух посещений несовместимых мероприятий; реализация – либо сейчас с некоторыми допущениями, либо в ближайшем будущем.

- **Отладка ролей и прав доступа:**
- Убедиться, что педсостав не может создавать/удалять менеджеров (только существующий менеджер мог бы).

- Проверить, что пионер не может вызвать admin-функции и т.п. Добавить нужные декораторы/Depends.

- Добавить менеджеру возможность управлять пользователями (если не успели в MVP) – хотя бы активацию/деактивацию:

    - Endpoint PATCH /users/{id}/activate или универсальный PATCH /users/{id} с полем is_active.
    - Тестирование этого.

- **Расширение API по необходимости фронтенда:**

- Если фронтэнду понадобится какой-то дополнительный маршрут (например, список транзакций конкретного пионера или фильтрация) – можно добавить. Но судя по `api.ts`, основных хватает.

- Возможно, добавить `/transactions/{id}/` GET (детально одну транзакцию). Не было прямого требования, но иногда нужно для обновления или если хотим отдельную страницу транзакции. Пока можно не делать.

- **Тестирование и фиксация**:

- К этому времени, интегрировать фронтенд с новым API и проверить весь функционал в браузере:
    - Авторизация: вход педагога, вход пионера.
    - UI просмотра списков, профилей.
    - Создание разных типов транзакций из интерфейса (возможно, фронтенд пока умеет только обычные начисления/штрафы и p2p; специфичные типа "факультатив" возможно нет кнопки – их могут добавить, тогда протестировать).
    - Проверить daily tax и финальный штраф в действии: можно вручную вызвать (например, через Swagger/Postman) и посмотреть, правильно ли обновляются балансы.
    - Отладить все обнаруженные проблемы (например, ошибки сериализации, время ответа, concurrency issues).
- Написать несколько **автоматизированных тестов** (если время):
    - Юнит-тесты для функций расчета штрафов (например, функция вычисления `expected_penalty` – проверить на разных комбинациях счетчиков).
    - Тест API (с использованием TestClient FastAPI): имитация логина, получение токена, вызов защищенных маршрутов, проверка статусов и JSON-структуры. Особое внимание P2P flow (создание под пионером, одобрение под педсоставом).
- Улучшить документацию: заполнить описание эндпоинтов (FastAPI позволяет в декораторах писать help, оно пойдет в Swagger UI). Добавить перечисления возможных типов транзакций и статусов, чтобы разработчикам фронта было понятно.

К концу недели система должна включать весь функционал, указанный в требованиях, и быть достаточно протестированной. Первоначальные данные (мок) могут быть заменены на реальные списки пользователей.

Тестирование и отладочный режим

Важная часть – обеспечить возможность быстро проверить систему с тестовыми данными: - **Debug mode с мок-данными**: В структуру проекта будет включен режим, когда при запуске на локальной машине или в специальном контейнере, база **автоматически заполняется тестовыми записями**: - Например, 10 пионеров с разными счетами, 2 педагога, 1 менеджер. - Несколько транзакций: пара штрафов, пара начислений, один pending P2P. - Это позволит фронтенд-разработчикам сразу увидеть рабочий интерфейс без ручного ввода данных. - Реализация: проверять переменную среды, например `DEBUG_SEED_DATA=true`. Если да и таблицы пустые – выполнить скрипт добавления. Либо подключить Alembic migration, которая помечена как “dev seed”. - **Важно:** исключить выполнение на боевом сервере – то есть по умолчанию в продакшн-сборке этот флаг выкл. - **Postman-коллекция / Swagger**: Сгенерировать Postman-коллекцию для всех эндпоинтов (или хотя бы описать в документации). FastAPI автоматически предоставляет Swagger UI и ReDoc – это уже даст возможность тестировать вручную. - **Логирование и ошибки**: Включить подробное логирование на время разработки. Если что-то падает (500 ошибка), FastAPI в debug выдает трассировку. В продакшене – настроить логирование в файл. Также сделать обработку возможных исключений (например, 404, 403) с понятными сообщениями (на русском или английском – в лагере можно и на русском, но API стандарт обычно англ; хотя для внутренней системы можно и русские тексты ошибок). - **Юнит-тесты**: По возможности написать несколько тестов, как упомянуто, особенно для вычислительных аспектов: - Функция вычисления прогрессирующего штрафа за пропуски – правильность последовательности. - Функция расчета educational fine – при разных входных (много не хватило vs все выполнено). - Проверка, что нельзя создать две посещения в один слот (если реализовано). - **Интеграционный тест**: при наличии времени – написать сценарий: 1. Логин под менеджером, 2. создать нового пионера, 3. логин под пионером, 4. отправить P2P запрос, 5. логин под педсоставом, одобрить запрос, 6. проверить балансы – всё через API.

- **Прочее**: Убедиться, что после перезапуска контейнеров данные сохраняются (Volume для Postgres настроен). Проверить в Docker-Compose, что зависимости (бд) поднимаются до app.

Резерв на будущее (планы развития)

В ТЗ заложены некоторые расширения функционала, которые не входят в текущую итерацию, но которые следует учитывать архитектурно: - **Система заявок по ролям**: возможность пользователям с определенной ролью принимать запросы от других. Пример, указанный заказчиком: - Роль "физрук" (спортивный инструктор) – ему пионеры могут отправлять заявки на выдачу спортинвентаря. - Другие возможные: "лавка" (заявка на покупку чего-то), "медик" (заявка в медпункт) – это потенциально. - Архитектура: добавить сущность **Request** с полями: id, type (например, "sport_inventory"), author (пионер), approver_role ("phys_instructor"), description, status (new/approved/rejected), date_created, date_resolved. - Добавить соответствующие роли пользователей (физрук и т.п., возможно как отдельное поле role or group membership). - Эндпоинты: пионер создает POST /requests/ (доступно типы, которые ему разрешены), пользователь с ролью approver_role делает GET /requests?type=sport_inventory (увидит список), и POST /requests/{id}/approve. - Пока это только план, но при разработке моделей и аутентификации мы будем учитывать, что роли могут быть не только {pioneer, staff, manager}, но и дополнительные (или использовать гибкую модель групп). - **Ачивки (достижения)**: - Дети могут получать достижения/награды (бейджи, условные “ачивки”). Это нефинансовый показатель, но приятный функционал. - Реализация: сущность **Achievement** (например, id, название, описание, возможно категория, и эмблемка), и связь многие-ко-многим с

User (какие ачивки у кого). - Можно позволить педсоставу вручную назначать ачивки (выбирает ачивку и пользователя – добавляется связь). Или автоматически, если определенные условия выполнены (например, все счетчики выполнены – “Отличник”). - Эндпоинт: GET /achievements/{user_id} – список достижений пользователя; менеджер POST /achievements/assign/. - В текущей архитектуре это почти независимый модуль, можно добавить без затрагивания остального.

- **Улучшение интерфейсов и UX**: хотя это больше про фронтенд, но может потребовать бэкенд-изменений:
- Например, возможность **фильтровать** или **пагинировать** списки (пользователей, транзакций) – если данных много. Добавим query-параметры в API (например, `?party=3` для пользователей, `?type=fine` для транзакций, `?limit=50&offset=0` для пагинации).

- **Отчеты и экспорт**: менеджер банка, возможно, захочет выгрузить таблицы (балансы, штрафы) в Excel/PDF. Это вне текущего объема, но несложно потом добавить endpoint, генерирующий CSV/PDF.

- **Безопасность**: в будущем стоит реализовать более строгие проверки:

- Ограничить число запросов (rate limiting) на критичные эндпоинты (например, чтобы пионер не мог спамить P2P заявками).
- Добавить капчу или доп. верификацию при входе (если будет нужно).
- Вести audit log (отдельно от транзакций) для действий админов (создание пользователей и т.п.).

- Резервное копирование БД перед финальными операциями, чтобы откатить если что.

- **Производительность**: текущий объем данных невелик (лагерь – десятки или сотни пользователей, тысячи транзакций максимум), FastAPI + Postgres справятся легко. Если масштабировать: можно подумать о кешировании статистики (avg, total) вместо вычисления на лету, оптимизировать запросы. В Docker можно позже добавить Redis для кеша или Celery для тяжелых задач (генерация отчетов, например).

Архитектура, заложенная на этапах 1-2, уже учитывает возможность добавления этих компонентов.

Итого, техническое задание охватывает описание API, структуры данных и ключевых механизмов банковской системы ЛФМШ. Проект будет реализован поэтапно, начиная с MVP-функционала (аккаунты, транзакции, JWT) и заканчивая внедрением всей бизнес-логики (автоштрафы, налог, проверки) в срок ~1 недели. Тестирование (как автоматическое, так и вручную через отладочный режим) включено в план, что позволит обеспечить надежность прототипа и готовность к дальнейшему расширению функционала.

---

1 2 3 Account.py
https://github.com/idutvuk/lfmsh_bank/blob/ab51d3140c6f3cbf5ae076d025218080f04e318f/django-app/bank/models/Account.py