Панель администратора

- Техническая документация: веб-админка системы согласований

Техническая документация: веб-админка системы согласований

Оглавление

---

1. Структура репозитория (package map)

```
rest_admin_service/
├── cmd/
│   └── app.go                    — точка входа, инициализация роутера и сервера
├── docs/
│   ├── docs.go, swagger.json, swagger.yaml  — сгенерированная Swagger-документация
├── handlers/
│   ├── admin/                    — CRUD администраторов
│   ├── admin_actions/            — журнал действий администраторов (аудит-лог)
│   ├── approval_steps/           — шаги согласования заявок
│   ├── approval_tasks/           — задачи исполнителей по заявкам
```

```
│   ├── approvalextras/           — вложенные сущности заявки (доп. согласующие, комментарии, наблюдатели, задачи, запрос отзыва)
│   ├── approvals/
│   │   ├── handlers.go           — CRUD заявок
│   │   ├── files.go               — файлы заявок
│   │   ├── reports.go             — выгрузка отчётов
│   ├── auth/                     — аутентификация администратора
│   ├── dashboard/                — сводная статистика
│   ├── departments/               — CRUD отделов
│   ├── flow_nodes/
│   │   ├── handler.go             — CRUD узлов дерева согласования
│   │   ├── preview.go             — предпросмотр дерева
│   │   ├── router.go              — регистрация маршрутов
│   │   ├── utils.go               — bumpFlowVersion и вспомогательные функции
│   ├── middleware.go              — JWTVerifier, AccessCodeVerifier
│   ├── notification/               — рассылка уведомлений
│   ├── permission/                 — CRUD прав доступа
│   ├── profile/                    — профиль текущего администратора
│   ├── register.go                 — обёртка хендлеров, единый формат ответа
│   ├── roles/                      — CRUD ролей
│   ├── system_monitoring/          — статус systemd-юнитов, БД, Redis
│   ├── topic_categories/           — CRUD категорий тем
│   ├── topic_executors/            — CRUD исполнителей по темам
│   ├── topics/                     — CRUD тем согласования
│   ├── user_hierarchy/             — CRUD иерархии согласующих
│   ├── user_roles/                 — назначение ролей пользователям
│   └── users/                      — список и карточка пользователей
├── types/
│   ├── module.go                   — InternalProviderControl (Config, Redis, Database, TGBotClient)
│   └── req.go                      — LimitRequest, Response
├── go.mod, go.sum
└── version
```

Все пакеты `handlers/*` следуют единому паттерну: структура `<Name>Reg` с полем `ipc *typesm.InternalProviderControl`, метод `Register<Name>Routes(r chi.Router, ipc *typesm.InternalProviderControl) error`, набор `Handler`-методов с сигнатурой `func(w http.ResponseWriter, r *http.Request) (interface{}, error)`.

2. Точка входа и инициализация сервиса

`cmd/app.go` — единственная точка запуска сервиса.

Последовательность инициализации (`initService`):

1. `getClientsExternal()` — загружает конфигурацию через `configcore.ConfigLoadOptions` с флагами `Redis`, `Database`, `Telegram`, `Secrets.AuthJWT.Admin`, `Secrets.AdminAccessCode`, `Server.RESTAdminService`.
2. `initInternalProvider()` — собирает `typesm.InternalProviderControl` (Config, RedisClient, Database, TGBotClient) из внешних клиентов.
3. `initBaseApiRouter()` — создаёт `chi.Mux`, подключает middleware, регистрирует маршруты через `registerRoutes()`.
4. `startRestApiServer()` — поднимает `http.Server` на порту из `Config.Server.RESTAdminService.PortRest`.

Таймауты сервера: `ReadTimeout`, `WriteTimeout`, `IdleTimeout` — по 15 минут каждый. Лимит запросов: 15 запросов в секунду на IP (`httprate.LimitByIP`).

Порядок глобальных middleware в `initBaseApiRouter`:

1. `cors.Handler` — allowed origins из конфига, методы GET/POST/PUT/DELETE/PATCH/OPTIONS, заголовки `Accept, Authorization, Content-Type, ApiKey, x-device-id`.
2. `middleware.RealIP`
3. `middleware.Recoverer`
4. `httprate.LimitByIP(15, 1s)`

Перед регистрацией остальных маршрутов проверяется `ipc.Database != nil` — при отсутствии подключения к БД роутер не создаётся, сервис не запускается.

Swagger UI подключён отдельно от общей цепочки middleware по пути `/swagger/*` (без авторизации).

3. Слой авторизации

Каждый защищённый маршрут проходит две независимые проверки, применяемые как middleware группы `r.Route(...)`: сначала `AccessCodeVerifier`, затем `JWTVerifier`. Порядок фиксирован во всех пакетах.

3.1. AccessCodeVerifier — проверка клиента (`handlers.AccessCodeVerifier`)

Назначение — верифицировать, что запрос пришёл от легитимного фронтенда (HMAC-подпись), независимо от того, авторизован ли конкретный пользователь.

Требуемые заголовки:

- `ApiKey` — HMAC-подпись.
- `X-Timestamp` — unix-время запроса.
- `x-device-id` — идентификатор устройства (используется в этом middleware только для логирования, не для проверки).

Шаги проверки:

1. Конфигурация должна содержать `Secrets.ApiPublicAdmin` и `Secrets.ApiPrivateAdmin` — иначе 500.
2. Оба заголовка (`ApiKey`, `X-Timestamp`) обязательны — иначе 401.
3. `X-Timestamp` парсится как Unix-время.
4. Время округляется до минуты (и текущее, и полученное).
5. Разрешённое окно — не более 3 минут в прошлом от округлённого текущего момента, и не в будущем — иначе 401 (`Timestamp expired` / `Timestamp in future`).
6. Ожидаемая подпись вычисляется как `GenerateHMACSignature(ApiPublicAdmin, timestampStr, ApiPrivateAdmin)` и сравнивается с `ApiKey` через `HmacEquals` (защита от timing-атак) — при несовпадении 401 `Unauthorized client`.
7. При успехе публичный ключ кладётся в контекст (`securecore.AccessContextKey`).

3.2. JWTVerifier — проверка администратора (`handlers.JWTVerifier`)

Назначение — определить, какой именно администратор выполняет запрос, и подтвердить, что его сессия и устройство актуальны.

Требуемые заголовки:

- `Authorization: Bearer <JWT>`
- `x-device-id`

Шаги проверки:

1. `Secrets.AuthJWT.AdminSecret` должен быть задан — иначе 500.
2. Заголовок `Authorization` обязателен, формат строго `Bearer <token>` — иначе 401.

3. `x-device-id` обязателен — иначе 401 `Unauthorized device`.
4. Redis-клиент должен быть инициализирован — иначе 500.
5. Токен верифицируется (`securecore.VerifyToken`), из claims извлекается `user_id`.
6. Администратор ищется в БД по ID с фильтром `IsBlocked = false`; отсутствие записи или блокировка — 401.
7. В Redis по ключу `{RedisPathUserAuthSessions}:admin:{userID}` ищется JSON-сессия (`typescore.UsersSessions`) — отсутствие ключа означает, что сессия истекла или отозвана — 401 `User session not found. Please authorize again`.
8. Сессия должна содержать переданный `x-device-id` в списке `AuthorizedDevices` — иначе 401 `Unauthorized device` (устройство было деавторизовано, например, при смене пароля или ручном отзыве сессий).
9. При успехе `user_id` кладётся в контекст (`securecore.IDContextKey`) и используется далее хендлерами через `handlers.GetIDFromContext(ctx)` — в частности, для записи автора действия в аудит-лог (`admin_actions`).

> Инвариант: сессия администратора хранится в Redis, а не только подписывается в JWT — это позволяет принудительно отзывать доступ (блокировка, смена пароля, logout всех устройств) без ожидания истечения токена.

4. Универсальная обёртка хендлеров (register.go)

Все хендлеры имеют единую сигнатуру `func(w http.ResponseWriter, r *http.Request) (interface{}, error)` и регистрируются через `handlers.RegisterRoute(r, method, path, handlerFunc)`, которая оборачивает их в `WrapHandlerF`.

4.1. WrapHandlerF

1. Оборачивает `http.ResponseWriter` в `responseWriterWrapper`, предотвращающий повторный вызов `WriteHeader` (защита от паники при двойной записи статуса).
2. Вызывает хендлер, получает `(payload, err)`.
3. Если `payload` имеет тип `*ExcelFileResponse` и `err == nil` — отдаёт бинарный файл через `serveExcelFile` (используется для выгрузки отчётов и файлов заявок), минуя JSON-сериализацию.
4. Иначе — сериализует ответ в JSON через `respondWithJSON`.

4.2. Формат JSON-ответа

Единая структура `types.Response`:

```
{
  "total_count": 0,
  "count": 0,
  "error": "текст ошибки",
  "data": {}
}
```

При ошибке хендлера ответ всегда `500 Internal Server Error` с телом `ErrorResponse{ErrorCode, ErrorDescription}` — коды ошибок доменного уровня (400/403/404) в текущей реализации `respondWithJSON` не различаются; хендлеры возвращают `errors.New(...)`, транслируемый в 500 независимо от смысловой причины отказа. Это единообразное, но не RESTful поведение стоит учитывать при интеграции фронтенда — различать причины ошибок можно только по тексту `error_description`.

4.3. ExcelFileResponse

Специальный тип для потоковой отдачи бинарных файлов (выгрузка отчётов по заявкам, скачивание вложений):

```
type ExcelFileResponse struct {
    Data     []byte
    FileName string
}
```

`serveExcelFile` выставляет заголовки `Content-Type: application/vnd.openxmlformats-officedocument.spreadsheetml.sheet`, `Content-Disposition: attachment; filename="..."`, `Content-Length`, `Cache-Control: no-cache, no-store, must-revalidate`.

5. Каталог REST-пакетов

Все пакеты используют единый паттерн: `<Name>Reg{ipc}`, `Register<Name>Routes`, middleware-цепочка `AccessCodeVerifier → JWTVerifier`, CRUD-хендлеры через `handlers.RegisterRoute`. Ниже — назначение каждого пакета и один характерный маршрут.

Пакет	Назначение	Пример маршрута

`admin`	CRUD администраторов панели	`GET/POST/PUT/DELETE /api/admin`
`admin_actions`	Журнал действий администраторов (аудит-лог), пишется другими пакетами через `LogAdminActionWithMetadata`	`GET /api/admin-logs`, `DELETE /api/admin-logs/{id}`
`approval_steps`	Шаги согласования конкретной заявки	`GET /api/approval-steps/by-approval/{approval_id}`
`approval_tasks`	Задачи исполнителей, включая файлы, прикреплённые к задаче	`GET /api/approval-tasks/{id}/files/{file_id}/download` (скачивание)
`approvalextras`	Вложенные сущности заявки: доп. согласующие, комментарии, наблюдатели, запрос на отзыв, задачи — все смонтированы на `/api/approvals/{id}/...` через `r.Group`, а не вложенный `r.Route` (см. §22.2 бот-документации — та же проблема 64-байтного лимита здесь неактуальна, но паттерн монтирования идентичен)	`GET /api/approvals/{id}/additional-approvers`
`approvals`	CRUD заявок, файлы заявки, отчёты (просмотр и скачивание Excel)	`GET /api/approvals/report/download`
`auth`	Авторизация администратора (логин/пароль, обмен токена), публичный `/api/time` для синхронизации часов клиента	`POST /api/auth/login`
`dashboard`	Агрегированная статистика: счётчики, разбивка по категориям, узкие места по согласующим, отсутствующие сегодня сотрудники, предупреждения о некорректной настройке	`GET /api/dashboard/stats`
`departments`	CRUD отделов	`GET /api/departments`
`flow_nodes`	CRUD узлов дерева согласования, сохранение дерева целиком, перемещение/дублирование узла, предпросмотр, ручная перезагрузка бота	`POST /api/flow-nodes/category/{category_id}/tree`
`notification`	Массовая рассылка сообщений через Telegram-бота по фильтрам получателей	`POST /api/notifications`
`permission`	CRUD прав доступа (роль × категория/тема × действие)	`GET /api/permissions/by-role/{role_id}`
`profile`	Просмотр/изменение профиля текущего администратора	`GET/PUT /api/admin/profile`
`roles`	CRUD ролей	`GET /api/roles`

`system_monitoring`	Статус systemd-юнитов, БД, Redis	`GET /api/monitoring`
`topic_categories`	CRUD категорий тем	`GET /api/categories`
`topic_executors`	CRUD исполнителей по темам	`GET /api/topic-executors/by-topic/{topic_id}`
`topics`	CRUD тем согласования	`GET /api/topics`
`user_hierarchy`	CRUD иерархии согласующих	`GET /api/hierarchies`
`user_roles`	Назначение/снятие ролей пользователю	—
`users`	Список и карточка пользователей Telegram-бота (не администраторов)	`GET /api/users/{id}`

Все списковые (`GetXListHandler`) эндпоинты возвращают `types.Response{data, count, total_count}` и, как правило, поддерживают фильтрацию и пагинацию через `typescore.ListDbOptions`. Мутирующие операции (создание/изменение/удаление значимых сущностей — администраторы, отделы, роли, права, темы, категории, пользователи) фиксируются в аудит-логе через `admin_actions.LogAdminActionWithMetadata` с указанием исполнителя, типа действия, целевой сущности и снапшотом изменённых полей.

---

6. Веб-панель

6.1. Стек и библиотеки

Категория	Библиотека	Назначение
Фреймворк	React 18 + TypeScript, сборка Vite	SPA
Роутинг	`react-router-dom` (`BrowserRouter`, `Routes`)	клиентская маршрутизация, защищённые роуты
UI-кит	`@mantine/core`, `@mantine/hooks`, `@mantine/form`, `@mantine/notifications`, `@mantine/modals`, `@mantine/dates`	компоненты интерфейса, формы, тосты, модалки, выбор дат
Работа с сервером	`@tanstack/react-query`	кэш, инвалидация, `useQuery`/`useMutation` во всех `features/*/hooks`
HTTP-клиент	`axios` (обёрнут в `src/api/client.ts`)	единая точка вызовов REST API
Подпись запросов	`crypto-js` (`HmacSHA256`)	генерация `ApiKey`-заголовка на клиенте, зеркалит `AccessCodeVerifier` бэкенда

Категория	Библиотека	Назначение
Даты	`dayjs` (+ плагины `utc`, `timezone`)	форматирование и расчёт дат
Графики	`recharts`	графики на дашборде
Drag-and-drop	`@dnd-kit/core`, `@dnd-kit/sortable`, `@dnd-kit/utilities`	сортировка узлов дерева согласования (`SortableFlowTree`)
Прочее	`crypto.randomUUID()` (браузерный API)	генерация `device_id`

6.2. Структура репозитория

```
src/
├── App.tsx                    — дерево маршрутов приложения
├── main.tsx                   — точка входа: провайдеры (Router, QueryClient,
MantineProvider, ModalsProvider, Notifications, ErrorBoundary)
├── theme.ts                   — тема Mantine
├── api/                       — по одному файлу на REST-ресурс, тонкие обёртки над apiClient
│   ├── client.ts               — настройка axios: HMAC-подпись, JWT, device_id,
синхронизация времени
│   ├── auth.ts, time.ts, dashboard.ts, users.ts, departments.ts, roles.ts,
│   │   permissions.ts, admins.ts, adminLogs.ts, categories.ts, topics.ts,
│   │   topicExecutors.ts, hierarchies.ts, flowNodes.ts, approvals.ts,
│   │   approvalTasks.ts, notifications.ts, userRoles.ts
├── features/                  — бизнес-логика по доменам, отделена от страниц
│   └── <domain>/
│       ├── hooks/              — useQuery/useMutation-хуки конкретного домена
│       ├── components/         — доменные компоненты, переиспользуемые между страницами
(опционально)
│       └── utils.ts            — доменные хелперы (опционально)
│   Домены: adminLogs, admins, approvalTasks, approvals, categories,
│   departments, flowNodes, hierarchies, notifications, permissions,
│   roles, topicExecutors, topics, users
├── pages/                     — страницы, привязанные к маршрутам
│   └── <domain>/
│       ├── <Domain>Page.tsx
│       └── components/         — модалки и виджеты, специфичные для конкретной страницы
├── components/                — общие сквозные компоненты
│   ├── Layout/ (Header, MainLayout, Sidebar)
│   ├── DiagnosticErrorBoundary.tsx
```

```
│   ├── CopyRolesModal.tsx, TopicGapsModal.tsx, UserEditModal.tsx
├── hooks/
│   └── useAuth.ts              — состояние аутентификации на уровне приложения
├── types/
│   └── api.ts                  — полная модель данных (TypeScript-интерфейсы)
├── utils/
│   └── date.ts
└── vite-env.d.ts
```

6.3. Страницы (маршруты)

Маршрутизация определена в `App.tsx`. Все страницы, кроме `/login`, обёрнуты в `ProtectedRoute` (редирект на `/login`, если `useAuth().isAuthenticated === false`) и общий `MainLayout` (шапка + боковое меню).

Маршрут	Страница	Назначение
`/login`	`Login`	форма входа по логину/паролю
`/`	`Placeholder`	заглушка главной страницы («в разработке»)
`/notifications`	`NotificationsPage`	отправка и история Telegram-рассылок
`/approval-dash`	`Dashboard`	сводная статистика по заявкам, графики, узкие места, предупреждения о настройке
`/nodes`	`FlowTreePage`	редактор дерева согласования по категориям
`/guide/flow-trees`	`FlowTreeGuidePage`	справочная страница по работе с деревом согласования
`/users`	`UsersPage`	список сотрудников Telegram-бота
`/departments`	`DepartmentsPage`	CRUD отделов
`/roles`	`RolesPage`	CRUD ролей
`/approvals`	`ApprovalsPage`	список и карточка заявок, просмотр файлов
`/permissions`	`PermissionsPage`	CRUD прав доступа роль/категория/тема
`/categories`	`CategoriesPage`	CRUD категорий тем
`/topics`	`TopicsPage`	CRUD тем согласования
`/executors`	`TopicExecutorsPage`	назначение исполнителей задач по темам

Маршрут	Страница	Назначение
`/hierarchies`	`HierarchiesPage`	иерархия согласующих, визуализация и дублирование цепочек
`/settings`	`SettingsPage`	настройки
`/admins`	`AdminsPage`	CRUD администраторов панели
`/logs`	`AdminLogsPage`	просмотр аудит-лога действий администраторов
`/approval-tasks`	`ApprovalTasksPage`	задачи исполнителей по заявкам

6.4. Полная модель данных (`src/types/api.ts`)

TypeScript-интерфейсы зеркалят Go-структуры бэкенда (см. §2 бот-документации), в snake_case, соответствующем JSON-ответам API.

User

```
interface User {
  id?: number; telegram_id?: number; username?: string; full_name: string
  language_code?: string; department_id?: number
  is_blocked?: boolean; is_fired?: boolean; blocked_at?: string
  last_login?: string; created_at?: string; updated_at?: string
  department?: Department; user_roles?: UserRole[]
}
```

Department

```
interface Department {
  id?: number; name: string; code: string; description?: string
  is_active?: boolean; created_at?: string; updated_at?: string
}
```

Role

```
interface Role {
  id?: number; name: string; code: string
  is_default?: boolean; is_director?: boolean; is_approver?: boolean
  is_controller?: boolean; is_observer?: boolean; is_executor?: boolean
```

```
  description?: string; created_at?: string; updated_at?: string
}
```

Флаги `is_director/is_approver/is_controller/is_observer/is_executor` определяют функциональную роль в процессе согласования — используются, в частности, при подборе получателей уведомлений (`TargetFilters.role_type_flags`) и первого согласующего по роли (`first_approver_role_code` в `ApprovalTopic`).

UserRole — связка пользователь↔роль: `{ id?, user_id, role_id, created_at?, updated_at?, role?: Role }`.

Admin — учётная запись администратора панели:

```
interface Admin {
  id?: number; login: string; name?: string
  is_root?: boolean; is_observer?: boolean
  telegram_id?: number; telegram_user?: User
  created_at?: string; last_login?: string; last_ip_login?: string
  is_blocked?: boolean; password?: string
}
```

`is_root` — суперадминистратор; `telegram_id`/`telegram_user` — привязка учётки администратора к пользователю Telegram-бота (для уведомлений и связи ролей).

ApprovalTopicCategory — категория тем: `{ id?, name, code, description?, is_active?, created_at?, updated_at?, topics?: ApprovalTopic[] }`.

ApprovalTopic — тема согласования (полное соответствие Go-структуре `ApprovalTopic` из документации бота, §2.2):

```
interface ApprovalTopic {
  id?: number; category_id?: number; name: string; code: string
  approval_mode?: string          // sequential | any_of
  duration_type?: string; time_type?: string; payment_type?: string
  template?: string; first_approver_role_code?: string
  allows_file?: boolean; allow_approver_selection?: boolean
  allow_observer_selection?: boolean; allow_delegation_selection?: boolean
  require_approver_comment?: boolean; requires_file?: boolean
  require_observer_selection?: boolean; is_active?: boolean
  instructions?: string; created_at?: string; updated_at?: string
  category?: ApprovalTopicCategory
}
```

Approval — заявка:

```
interface Approval {
  id?: number; initiator_id?: number; topic_id?: number; department_id?: number
  content?: string; status?: string; metadata?: string
  revoked_by_id?: number; revoked_at?: string
  revision_comment?: string; revision_by_id?: number; revision_at?: string; revision_count?:
number
  created_at?: string; updated_at?: string
  files?: ApprovalFile[]; steps?: ApprovalStep[]
  initiator?: User; topic?: ApprovalTopic; department?: Department; revoked_by?: User
}
```

ApprovalStep — шаг согласования: `{ id?, approval_id?, approver_id?, step_order?, status?, comment?, approver_name_snapshot?, approver_department_snapshot?, acted_at?, returned_at?, created_at?, updated_at?, approver?: User }`.

ApprovalFile — файл заявки или задачи: `{ id?, approval_id?, task_id?, file_base64?, file_name?, attached_by_id?, created_at? }`.

ApprovalTask — задача исполнителя: `{ id?, approval_id?, executor_id?, task_text?, status?, step_order?, require_file?, topic_executor_id?, completed_at?, created_at?, updated_at?, approval?: Approval, executor?: User, files?: ApprovalFile[] }`.

ApprovalTopicExecutor — правило назначения исполнителя по теме: `{ id?, topic_id?, user_id?, step_order?, require_file?, task_text?, is_active?, target_system?, created_at?, updated_at?, topic?: ApprovalTopic, user?: User }`.

ApprovalFlowNode — узел дерева согласования (полное соответствие `ApprovalFlowNode` из документации бота, §2.3):

```
interface ApprovalFlowNode {
  id?: number; topic_id?: number; parent_id?: number | null
  key: string; category_id: number
  message_template?: string | null; code_template?: string | null
  label: string
  data_key?: string | null; data_value?: string | null
  is_input?: boolean; input_key?: string | null; input_hint?: string | null
  is_multi_select?: boolean; multi_select_key?: string | null; multi_select_value?: string |
null
  topic_code?: string | null; sort_order?: number
  created_at?: string; updated_at?: string
  children?: ApprovalFlowNode[]
```

```
}
```

Permission — право доступа: `{ id?, role_id?, topic_category_id?: number | null, topic_id?: number | null, action?, created_at?, updated_at?, role?: Role, topic_category?: ApprovalTopicCategory, topic?: ApprovalTopic }`. Взаимоисключающая область действия (категория / тема / глобально) идентична паттерну `UserHierarchy` бэкенда.

UserHierarchy — правило иерархии согласующих: `{ id?, subordinate_id?, approver_id?, department_id?, topic_category_id?: number | null, topic_id?: number | null, priority?, target_system?: string | null, created_at?, updated_at?, subordinate?: User, approver?: User, department?: Department }`.

Notification / NotificationRecipient / TargetFilters — рассылка уведомлений:

```
interface TargetFilters {
  user_ids?: number[]; role_ids?: number[]
  role_type_flags?: string[]   // director | approver | observer | executor | controller
  department_ids?: number[]
}

interface Notification {
  id?: number; title: string; message_html: string
  photo_base64?: string; photo_file_name?: string
  target_filters?: string      // сериализованный JSON TargetFilters
  status?: string               // pending | sending | completed | failed
  total_count?: number; sent_count?: number; failed_count?: number
  created_by_id?: number; created_at?: string; updated_at?: string; sent_at?: string
  recipients?: NotificationRecipient[]
}

interface NotificationRecipient {
  id?: number; notification_id?: number; user_id?: number
  status?: string                // pending | sent | failed
  error_message?: string; telegram_message_id?: number
  sent_at?: string; created_at?: string; user?: User
}
```

Служебные типы: `ApiResponse<T> = { data: T; total_count?: number }`, `LoginRequest = { login, password }`, `LoginResponse = { token }`.

6.5. Управление состоянием (State Management)

Отдельного глобального стора (Redux/Zustand/MobX) в проекте нет. Состояние разделено по назначению:

1. **Серверное состояние (данные API)** — полностью на `@tanstack/react-query`. Каждый домен в `features/<domain>/hooks/use<Domain>.ts` инкапсулирует `useQuery` (чтение списков/карточек) и `useMutation` (создание/изменение/удаление) с автоматической инвалидацией кэша (`queryClient.invalidateQueries`) после успешной мутации и уведомлением через `@mantine/notifications`. Глобальный `QueryClient` создаётся один раз в `main.tsx` с `refetchOnWindowFocus: false` и `retry: 1`.
2. **Состояние аутентификации** — локальный хук `useAuth()` (`src/hooks/useAuth.ts`) на `useState`/`useEffect`, без контекста верхнего уровня: при каждом вызове хука состояние (`user`, `loading`) создаётся заново на уровне компонента, использующего `useAuth`. Источник истины для факта авторизации — JWT в `localStorage.token`; профиль подгружается через `authApi.getProfile()` при монтировании, если токен присутствует.
3. **Постоянное хранение на клиенте** — `localStorage`: `token` (JWT), `device_id` (UUID устройства, генерируется один раз через `crypto.randomUUID()` и переиспользуется), `user` (очищается при logout).
4. **Локальное UI-состояние** — стандартный `useState`/`useForm` (`@mantine/form`) внутри страниц и модалок: значения форм, открытость модалок, выбранные строки таблиц и т. п. — не выносится выше компонента, которому принадлежит.
5. **Кросс-компонентные уведомления и модалки** — управляются через провайдеры Mantine (`ModalsProvider`, `Notifications`), подключённые глобально в `main.tsx`; вызываются императивно (`notifications.show(...)`, `modals.open(...)`) без промежуточного стора.

6.6. API-клиент (`src/api/client.ts`)

Единый `axios`-инстанс с `baseURL: '/api'`. Request-интерцептор на каждый запрос:

1. Добавляет `Authorization: Bearer <token>` из `localStorage`, если токен есть.
2. Вычисляет `X-Timestamp` с поправкой на смещение серверного времени (`serverTimeOffset`), полученное через `GET /api/time` при загрузке модуля и далее каждые 5 минут.
3. Генерирует `ApiKey` — `HMAC-SHA256(publicKey + ":" + timestamp, privateKey)`, где ключи берутся из `VITE_API_PUBLIC_KEY` / `VITE_API_PRIVATE_KEY` (`.env`) — зеркалит проверку `AccessCodeVerifier` на бэкенде.
4. Добавляет `x-device-id` — постоянный UUID устройства из `localStorage`.

Response-интерцептор:

- при `401` с телом `"Timestamp expired\n"` — единожды (`_retry`) синхронизирует время с сервером, пересчитывает подпись и повторяет запрос;
- при любом другом `401` — очищает `token` из `localStorage` и делает `window.location.href = '/login'` (жёсткий редирект, минуя роутер).

Каждый файл в `src/api/*.ts` — тонкая типизированная обёртка над этим клиентом для конкретного ресурса (один файл на REST-пакет бэкенда).