ДОКУМЕНТАЦИЯ

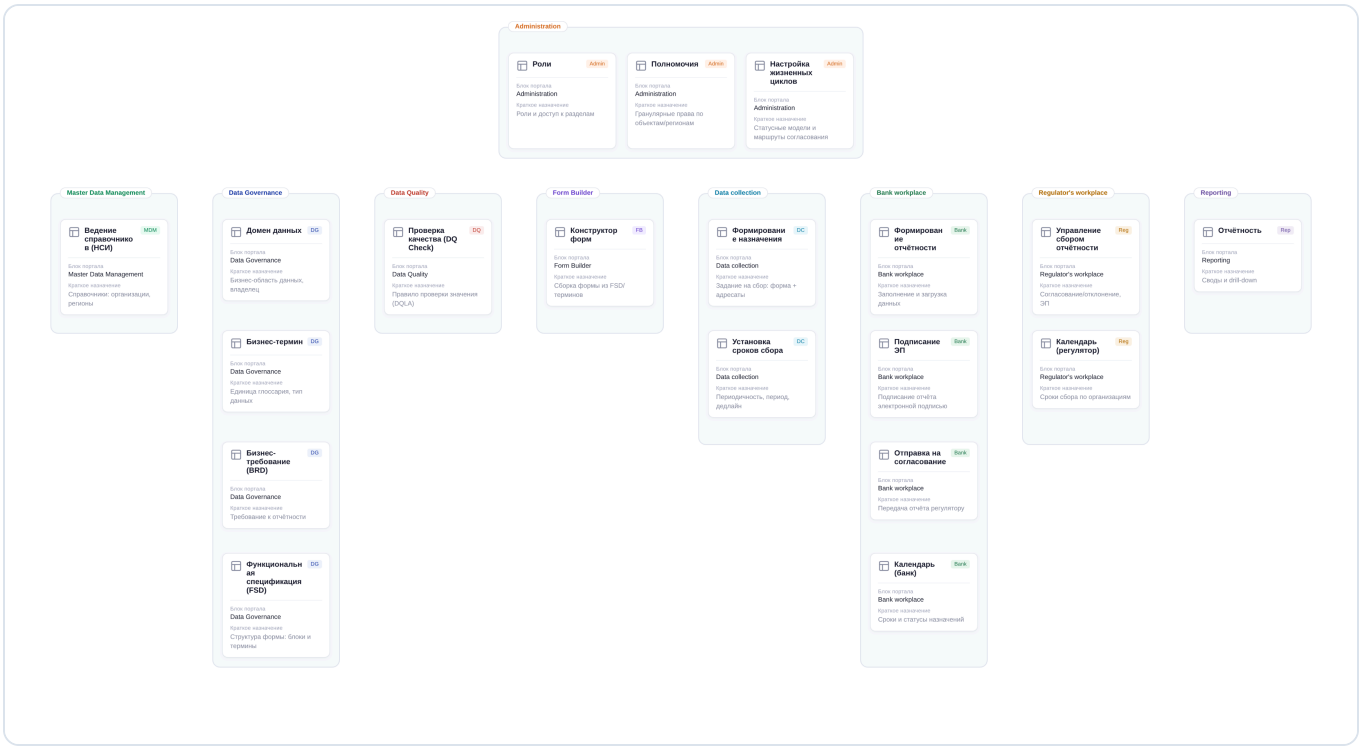
Регуляторный портал отчётности ЦБ РУз

Техническое описание платформы · июль 2026

ВЕРХНЕУРОВНЕВАЯ АРХИТЕКТУРА

Архитектура портала регулятора

Функциональные блоки портала и объекты внутри них. Интерактивная версия — на странице «Верхнеуровневая архитектура портала».



РЕГУЛЯТОРНЫЙ ПОРТАЛ ОТЧЁТНОСТИ ЦБ РУЗ

Обзор системы

Модульная платформа для управления метаданными (Data Governance) и сбора регуляторной отчётности.

3

функциональных модуля, объединённых сквозным путём из 12 этапов

11

контейнеров в docker-compose за одним nginx

4

базы PostgreSQL (по одной на backend)

~7

физических таблиц в ядре — вся модель через мета-объекты

Что это за система

Платформа для управления нормативно-справочной информацией (НСИ / метаданными) и сбора регуляторной отчётности с организаций (банки, небанковские, страховые) в адрес регулятора — **Центрального банка Республики Узбекистан**. Внутреннее имя проекта — `forms_dg`.

Три модуля

1 · Data Governance (DG)

Управление метаданными: домены, термины (глоссарий), FSD, BRD, справочники организаций. «Единый источник правды» для остальных модулей. Реализован как универсальная мета-модель «всё есть объект».

2 · Form Builder

Конструктор шаблонов отчётных форм из терминов/FSD. Генерирует JSON Schema формы, имеет собственное согласование шаблонов.

3 · Submission Engine

Сбор отчётности. Делится на **Task Manager** (регулятор: назначает сбор, согласует, консолидирует) и **Form Filler** (кабинет организации: заполняет, подписывает, отправляет).

Ключевой принцип — «всё есть объект»

Вместо отдельных таблиц под каждую сущность DG использует универсальную мета-модель: `meta_class` + `meta_object` + JSON-поле `properties`, валидируемое по JSON Schema класса. Домены, термины, FSD, BRD, DQLA, организации, регионы — это всё строки в одной таблице `meta_object`, различающиеся классом.

Сквозной поток данных

```
Домен → Термины → FSD → Шаблон формы (Form Builder)
      → Задание на сбор (Task Manager) → Заполнение (Form Filler)
      → Согласование + DQ-проверки (Task Manager) → Консолидация/аналитика
```

Как читать эту документацию

- Архитектура — топология, порты, маршрутизация, БД, dev/prod.
- Модули DG, Form Builder, Submission Engine — устройство каждого.
- Справочник: концепции, DQ Checks, Workflow, CJM.
- Приложения: развёртывание.

РАЗДЕЛ 2

Архитектура (реальная)

Топология из 11 контейнеров за одним nginx, 4 базы PostgreSQL, dev-режим. Ниже — стек, карта портов и маршрутизации.

2.1 · Технологический стек (по факту кода)

Слой	Технология
Backend (все 4 сервиса)	**FastAPI**, **SQLAlchemy 2.x**, **Pydantic v2**, Uvicorn
БД	**PostgreSQL 16** в Docker (в коде дефолт — SQLite, только для локального запуска без Docker)
Frontend (все)	**Vue 3** (`<script setup>` + TS), **Vite**, **vue-router 4**, **axios**
Граф Lineage	**Vue Flow** (`@vue-flow/core`)
Состояние фронта	простые `reactive()` / `ref()` стор-модули
UI	ручной CSS + inline SVG-иконки
Reverse proxy	**nginx 1.27** (path-based routing)
Excel	`openpyxl` (backend), `xlsx` (frontend Form Filler)
Auth	самописный **HS256 JWT**; тестовые токены без паролей

2.2 · Топология сервисов (docker-compose)

Единый хост, один `docker-compose.yml`. Каждый модуль = backend (FastAPI) + frontend (Vue/Vite), всё за одним nginx. Одна общая Postgres с 4 базами.

Сервис (DNS-имя)	Что это	Внутр. порт	БД
`postgres`	PostgreSQL 16-alpine (user/pw `forms/forms`)	5432 (публ.)	—
`dg-backend`	Data Governance API	8000	`forms_dg`
`dg-frontend`	Основной Vue-портал (`base=/demo/`)	5173	—
`admin-frontend`	Тот же фронт, режим админа (`/demo_admin/`)	5177	—
`dg-module-frontend`	Тот же фронт, DG-only (`/demo_dg/`)	5178	—
`form-builder-backend`	Конструктор форм API	8100	`form_builder`
`form-builder-frontend`	Конструктор форм UI (`/demo/form-builder/`)	5174	—
`task-manager-backend`	Сбор/согласование/DQ API	8200	`task_manager`
`task-manager-frontend`	Регуляторный UI (`/demo/submission/`)	5175	—
`form-filler-backend`	Кабинет организации API	8300	`form_filler`
`form-filler-frontend`	Кабинет организации UI (`/demo_org/`)	5176	—
`nginx`	Reverse proxy	80 (публ.)	—

Одна кодовая база — три инкарнации. `admin-frontend` и `dg-module-frontend` — это тот же каталог `frontend/`, собранный с разными флагами `VITE_*_MODULE` и base-path.

2.3 · Маршрутизация nginx

```
= /                              → статический landing.html
/demo/api/v1/                    → dg-backend:8000           (→ /api/v1/)
/demo/form-builder-api/api/v1/   → form-builder-backend:8100 (→ /api/v1/)
/demo/submission-api/api/v1/     → task-manager-backend:8200 (→ /api/v1/)
/demo_org/api/v1/                → form-filler-backend:8300  (→ /api/v1/)
/demo/form-builder/              → form-builder-frontend:5174
/demo/submission/                → task-manager-frontend:5175
/demo_org/                       → form-filler-frontend:5176
/demo_admin/                     → admin-frontend:5177
/demo_dg/                        → dg-module-frontend:5178
/demo/                           → dg-frontend:5173   (catch-all, последним)
```

Сервисы общаются по **DNS-именам сервисов** внутри compose-сети (например `http://dg-backend:8000/api/v1`). Браузер ходит на same-origin `/demo/...`, а nginx проксирует. HMR/WebSocket проброшены.

2.4 · Базы данных

- Все 4 backend-а в Docker переключены на Postgres через `*_DATABASE_URL`: `forms_dg`, `form_builder`, `task_manager`, `form_filler`.
- Базы создаются скриптом `docker/postgres-init/01-create-databases.sh` при первом старте.
- В коде дефолт каждого сервиса — **SQLite**; используется только при локальном запуске без Docker.
- Схема создаётся через `Base.metadata.create_all` при старте; миграции данных и схем выполняются скриптами в `scripts/`.

2.5 · Dev vs Prod

Стенд собран в **dev-режиме**: Vite dev-серверы, тривиальные пароли Postgres, только HTTP, разрешительный CORS. Исключение — `admin-frontend` и `dg-module-frontend`: собираются (`npm run build`) и работают через `vite preview` с `VITE_DISABLE_HMR=true`.

Обновление кода. Backend-сервисы собираются в образ (исходники копируются при сборке) и запускаются без `--reload` — изменения backend применяются пересборкой образа; Vite-фронтенды подхватывают правки на лету (HMR).

РАЗДЕЛ 3 · МОДУЛЬ 1

Data Governance

Ядро системы: универсальная мета-модель «всё есть объект», версионирование, статусы, аудит, граф зависимостей и ролевой доступ.

Корень: `backend/`. FastAPI, роутеры под `/api/v1` (`app/modules/data_governance/api.py` + `app/modules/auth_api.py`).

3.1 · Мета-модель (физические таблицы)

Всего ~7 таблиц (`app/db/models.py`). Никаких выделенных таблиц `domains` / `terms` / `fsd` нет — всё это классы и объекты.

Таблица	Назначение
`meta_class`	Реестр классов: `code` (уник.), `name`, `parent_class_id` (иерархия), `validation_schema` (JSON Schema), `ui_schema`, `is_abstract`, `is_system`.
`meta_object`	Экземпляр любой сущности: `class_id`, `code`, `name`, `properties` (JSON), `relations` (JSON), `version`, `parent_object_id`, `previous_version_id`, `status`. Уникальность `(class_id, code, version)`.
`meta_object_history`	Append-only снимки (`action`, `payload`, `actor`).
`meta_object_relation`	Нормализованные рёбра графа: `source`, `target`, `relation_type`, `properties.label`.
`meta_object_audit`	Аудит: `old_values` / `new_values`, `changed_by`, `reason`, `ip_address`, `user_agent`.
`meta_object_workflow`	Трекинг процессов: `process_instance_id`, `current_task`, `started_at` / `completed_at`.
`app_setting`	key/value JSON: конфиг сайдбара, ролей, пользователей, прав редактирования классов.

Все PK — строковые UUID. Поиск объектов выполняется через `LIKE` и фильтры.

3.2 · Классы объектов

Системные классы: `DOMAIN`, `TERM`, `FSD`, `BRD`, `DQLA`, `DQ_CHECK`, `ORGANIZATION`, `ORG_TYPE`, `REGION` и др. Каждый класс несёт свою JSON Schema для валидации `properties` его объектов.

3.3 · API (ключевое)

```
GET    /governance/summary                — метрики дашборда
GET    /classes            POST /classes    (ADMIN)
GET    /classes/{code}     PUT/DELETE /classes/{code}  (ADMIN)
POST   /objects                           — создать объект
GET    /objects/search                    — q, class_code, domain_id, status, limit,
GET    /objects/{id}                      — (?include_history)
PUT    /objects/{id}                      — (?create_new_version)
DELETE /objects/{id}                      — soft-archive (ADMIN)
POST   /objects/{id}/status               — переход статуса
GET    /objects/{id}/lineage              — direction, depth 1..10
GET    /domains/{id}/export               — только format=json (иначе 501)
```

Auth-роутер: `/auth/test-users`, `/auth/test-token/{username}`, админ-конфиги `/admin/roles`, `/admin/users`, `/admin/sidebar-config`, `/admin/class-edit-roles`, `/admin/form-builder-settings`.

3.4 · Валидация и «полнота»

Отдельного движка качества/health-score в DG **нет**. Есть:

1. **Schema-валидация при записи** (`_validate_object_properties`): обязательные поля, типы, ссылочная целостность для полей типа `meta_class`. Ошибки → HTTP 422 со списком `{field, message}`.
2. **Метрика покрытия доменов** в `get_summary`: по домену `domain_coverage` = число терминов vs `terms_plan_count` → процент описанности (виджет «Уровень описания доменов данных»).
3. `DQLA` / `DQ_CHECK` в DG — просто **классы-контейнеры**; логика исполнения живёт в Task Manager (см. DQ Checks).

3.5 · Lineage (граф зависимостей)

`get_lineage` — обход в ширину (BFS) с ограничением по `depth` и дедупликацией. Три источника рёбер:

- `parent_child` — из `meta_object.parent_object_id`.
- `property_ref` — из полей класса типа `meta_class`; для FSD термины извлекаются из `properties.blocks[].terms[].term_id` (label «Блок: {имя}»).
- `relation` — из таблицы `meta_object_relation`.

Рисуется на **Vue Flow**: колоночная раскладка (предки слева, потомки справа), цвет узла по классу, цвет ребра по типу связи, пошаговое раскрытие по 10 соседей, подсветка выбранного узла.

3.6 · Режимы сборки фронтенда

`router.ts` выбирает один из трёх наборов маршрутов на этапе сборки Vite:

- `VITE_ADMIN_MODULE=true` → **admin-модуль**: только `/classes`, `/admin/*`. Неадминов гвард разлогинивает.
- `VITE_DG_MODULE=true` → **DG-only**: `/`, `/objects`, `/approvals`, `/journey`, `/submission/dq-rules`.
- иначе → **полный портал**: всё выше + `/form-builder/*` и `/submission/*` (iframe-обёртки).

3.7 · Ролевая модель и доступ

- **Механизм:** самописный HS256 JWT. Пароля нет — токены выдаёт `/auth/test-token/{username}` (демо-режим).
- **Роли:** `VIEWER` `EDITOR` `APPROVER` `ADMIN` `SUPER_ADMIN` + модульные (`DOMAIN_EDITOR`, `TERM_EDITOR`, `FSD_EDITOR`, `DATA_MANAGER`, `SUBMISSION_*`…). `ROLE_SCOPES` мапит роли → скоупы (`dg:read`, `dg:editor`, `dg:approve`, `dg:admin`).
- **Два уровня контроля:** маршрутный `require_roles("ADMIN")` и класс-ориентированный (`editable_class_codes_for_roles` / `readable_class_codes_for_roles`). Поиск фильтруется по читаемым классам.
- **Кэш** (`core/cache.py`) — на Redis; включается заданием `redis_url`.

3.8 · Seed-данные

`scripts/seed_mock_data.py`: 3 системных класса (DOMAIN/TERM/FSD), 2 домена (`TAXATION`, `REPORTING`), ~1000 терминов, 1 FSD (`TAX_FORM_FSD`). Все объекты — со статусом `approved`. Пользователи/роли этим скриптом не сеются (они в `app_setting`).

РАЗДЕЛ 4 · МОДУЛЬ 2

Form Builder

Автономный конструктор шаблонов отчётных форм из терминов и FSD с генерацией JSON Schema и собственным согласованием.

Корень: `form_builder/`. Backend (8100) + Vue-фронт (5174, base `/demo/form-builder/`). Работает автономно, обращаясь к DG API за терминами/FSD.

4.1 · Данные (БД `form_builder`)

Таблица	Поля / назначение
`report_template`	Шаблон: `code`, `name`, `domain_id`, `version`, `status` (draft/review/approved/rejected), `settings` (JSON), `json_schema` (сгенерир.).
`form_section`	Секция: `order_index`, `is_repeatable`.
`form_field`	Поле: `source_type` (TERM/FSD/custom), `source_id`, `field_label`, `override_properties` (тип, ширина, режим привязки), `column_span` (1..12), `is_required`, `is_readonly`.
`form_layout`	Плейсхолдер раскладки.

4.2 · API

```
GET/POST /templates            GET/PUT /templates/{id}
POST/PUT/DELETE /templates/{id}/sections[/{sid}]
POST/PUT/DELETE /templates/{id}/fields[/{fid}]
GET  /templates/{id}/preview          — сгенерированная JSON Schema
POST /templates/{id}/submit-review    — на согласование
GET  /approvals/pending
POST /templates/{id}/approve | /reject
GET  /dg/domains | /dg/terms | /dg/fsd | /dg/organizations   (прокси в DG)
```

Интеграция с DG (`core/dg_client.py`): получает bearer-токен через DG `/auth/test-token/{username}`, тянет объекты через DG `/objects/search?class_code=…`. Палитра классов

настраивается (`palette_class_codes` , дефолт `["TERM","FSD"]`).

4.3 · Генерация схемы и версионирование

`_generate_schema()` перебирает поля по `order_index` , формирует JSON Schema (draft-07) + кастомный `ui:sections` . Схема **не** обращается заново к DG — строится из данных поля, зафиксированных в `form_field` при добавлении.

Версии — в `settings.versioning` (`approved_major.0.draft_patch`): согласование поднимает major; редактирование одобренного шаблона возвращает его в `draft` и поднимает patch; ведётся `version_history[]` .

4.4 · Фронт (сборка формы)

3-панельный конструктор: слева **палитра** DG-объектов (поиск + фильтр по классу), в центре **канвас** секций/полей с HTML5 drag-and-drop, внизу — живая JSON Schema.

> **FSD → секция.** Ключевая логика `addFsdAsSection()` : при перетаскивании FSD создаётся секция и по одному полю на каждый термин из `properties.blocks[].terms` (или легаси `properties.terms`). Отдельного backend-эндпоинта «сгенерировать из FSD» нет — это клиентская оркестрация.

Есть UI привязки **DQLA** к шаблону (сохраняется как `settings.dqla_ids`) — см. DQ Checks.

РАЗДЕЛ 5 · МОДУЛЬ 3

Submission Engine

Сбор отчётности: Task Manager на стороне регулятора и Form Filler в кабинете организации. Связаны только по HTTP.

Корень: `submission_engine/`. Две независимые связки FastAPI+Vue, обе зависят от DG (8000) и Form Builder (8100).

5.1 · Task Manager (регулятор) — 8200 / 5175

Назначение: создать задание на сбор, назначить организациям, мониторить, согласовать/отклонить отчёты, консолидировать, вести DQ-правила.

ДАННЫЕ (БД `TASK_MANAGER`)

Таблица	Назначение
`report_task`	Задание: `template_id` / `template_version` (согласованный шаблон FB), период, `deadline`, `target_system` (дефолт «FNS»), `status`.
`task_assignment`	Связь задание ↔ организация, свой `status`.
`task_target_config`	Таргетинг: `all` / `org_type` / `organization`.
`report_submission`	Присланный отчёт: `payload` (JSON), `status` (дефолт `review`), поля ревью.
`report_submission_snapshot`	Неизменяемая копия на каждую подачу.
`term_quality_rule`	**DQ-правило**, привязанное к термину DG (см. DQ Checks).

API: `/tasks`, `/tasks/assignments`, `/tasks/submissions`, `/tasks/submissions/{id}/decision`, `/tasks/submissions/{id}/export` (XLSX), `/monitoring/summary`, `/dq/term-rules` (CRUD), `/dq/term-rules/usage`, `/dq/submissions/{id}/checks`.

Страницы: Дашборд, Назначения, Организации, Мониторинг, Согласование, Консолидированный отчёт, Проверки качества, История слепков.

Доступ: декодирует JWT-подобный токен (base64, без проверки подписи) — роли + `submission_access` (view_all / region_ids / org_type_ids / organization_ids — региональная фильтрация).

5.2 · Form Filler (кабинет организации) — 8300 / 5176

Назначение: организация тянет свои задания, заполняет форму, подписывает электронной подписью, отправляет обратно в Task Manager.

ДАННЫЕ (БД `FORM_FILLER`)

Таблица	Назначение
`local_task`	Зеркало задания из Task Manager.
`local_instance`	Заполняемый отчёт: `form_data` (JSON), `completion_percentage`, `status` (жизненный цикл) + `local_status` (синхронизация), подпись.
`sync_queue`	Очередь офлайн-повторов при неудачной отправке.
`local_submission_snapshot`	Локальная копия отправленного.

API: `/auth/login` (выбор организации из справочника DG, без пароля), `/reports/my`, `/reports/{id}`, `/reports/{id}/dg-context`, `/filling/{id}/save`, `/signing/{id}/sign`, `/submission/{id}/send`, `/sync/tasks/pull`, `/sync/queue/process`.

Страницы: Вход, Мои отчёты, Заполнение отчёта, История. Импорт/экспорт таблиц через npm `xlsx`.

5.3 · Как связаны Task Manager ↔ Form Filler

Всё по HTTP. Form Filler хранит base-URL Task Manager.

- **Pull (TM→FF):** `sync_tasks` тянет `/tasks/assignments` + `/tasks`, фильтрует по организации, создаёт `local_task` + `local_instance`, синхронизирует статус.
- **Push (FF→TM):** `_push_submission_to_task_manager` шлёт `POST /tasks/submissions` с `payload`. Успех → снимок; ошибка → `sync_queue` на повтор.

- **Решение (TM→FF):** решение меняет записи только в TM; организация видит новый статус при следующем `sync/tasks/pull`.
- **Консолидация** — клиентская (`ConsolidatedReportView.vue`, XLSX в браузере). Отдельного backend-эндпоинта консолидации нет; на backend — только per-submission XLSX через `openpyxl`.

РАЗДЕЛ 6 · СПРАВОЧНИК

Ключевые концепции

Глоссарий элементов системы: Data Governance, Domain, Term, FSD, BRD, DQ Checks, Lineage, Workflow, CJM — с привязкой к реализации.

Определения ключевых элементов — не абстрактно, а «как это сделано в коде».

Data Governance (DG)

Управление метаданными предприятия: единый каталог доменов, терминов, спецификаций и правил, обеспечивающий, что все модули говорят на одном языке данных. Реализован как мета-модель «всё есть объект» (§3) с версионированием, статусами, аудитом и графом зависимостей.

Domain (Домен данных)

Бизнес-область данных («Налогообложение», «Кредитные продукты»). Класс `DOMAIN`. Имеет владельца и `terms_plan_count` (план по числу терминов, из которого считается метрика описанности). К домену привязываются термины.

Term (Бизнес-термин)

Единица бизнес-словаря (глоссария). Класс `TERM`. Привязан к домену, имеет тип данных, категорию чувствительности, владельца, тип (Термин/Атрибут) и, для атрибута, алгоритм расчёта. Термины — «кирпичи» для FSD, форм, DQ-правил и lineage.

FSD — Functional Specification Document

Функциональная спецификация формы. Класс `FSD`. Описывает набор **бизнес-блоков**, в каждом блоке — список терминов с ролью в форме (ПК — первичный ключ, Обязательное). «Контейнер» для последующей сборки формы в Form Builder.

- Хранение: `properties.blocks[]` — массив `{id, name, type: business|technical, terms: [{term_id, primary_key, required}]}`. Дополнительно — плоский `properties.terms[]` для совместимости.
- Ответственные лица: бизнес-владелец, технический владелец, системный аналитик, разработчик.

BRD — Business Rule Definition

Определение бизнес-правила. Класс `BRD` — правило типа validation / calculation / filter / trigger. Представлен как класс и тип узла в графе lineage (янтарный цвет). Практическую роль правил проверки на отчётности выполняют DQ Checks.

DQ Checks (проверки качества данных)

Проверки, применяемые к значениям отчёта. В UI и документах называются **DQLA** («проверки качества»). Полностью описаны в разделе DQ Checks.

Lineage (происхождение/зависимости данных)

Граф связей объекта: домен → термины → FSD → формы → DQLA/DQ_Check. Строится BFS-обходом трёх типов рёбер, рисуется на Vue Flow (§3.5).

Workflow (согласование)

Машина состояний жизненного цикла объектов/шаблонов/отчётов с ролевыми переходами — см. Workflow.

CJM — Customer Journey Map

Стартовая карта из 12 этапов пути пользователя, она же навигация демо-стенда — см. CJM.

РАЗДЕЛ 7 · СПРАВОЧНИК

DQ Checks — механизм проверок качества

Как устроены проверки качества (DQLA): описание в DG, привязка к форме в Form Builder, исполнение в Task Manager при согласовании отчёта.

Связка трёх уровней: **описание** проверок в DG → **привязка** к форме в Form Builder → **исполнение** в Task Manager при согласовании отчёта.

7.1 · Описание (в Data Governance)

- Класс `DQ_CHECK` — отдельная проверка (описание).
- Класс `DQLA` — набор/сборка проверок (контейнер), группирует `DQ_CHECK` и/или термины. Метаданные, без исполняемой логики в DG.

7.2 · Привязка (в Form Builder)

В конструкторе шаблона можно добавить DQLA к форме (блок «DQLA для отчётности»). Выбранные наборы сохраняются в `report_template.settings.dqla_ids` — это определяет, какие термины формы подлежат проверке.

7.3 · Исполнение (в Task Manager)

Реальная проверка — метод `evaluate_submission_quality(submission_id)` в `task_manager_service.py`:

1. Берутся активные правила `term_quality_rule` (`is_active=true`), сгруппированные по `term_id`.
2. Через `_allowed_term_ids_by_template_dqla` из `settings.dqla_ids` шаблона резолвится множество допустимых `term_id`. Если DQLA не заданы — правила применяются ко всем терминам.
3. Для каждого значения применяется `_apply_quality_rule` → `SubmissionQualityCheck {passed, message, rule_type}`. Табличные поля — `_evaluate_table_term_quality_checks`.

Типы правил

Тип (`rule_type`)	Параметры	Логика
`required`	—	значение не пустое (None / пустая строка / пустой список → провал)
`regex`	`pattern`	`re.fullmatch(pattern, str(value))`; пустое значение → провал
`number_range`	`min` и/или `max`	значение приводится к числу и сравнивается с границами

Валидация правил — `_validate_rule_payload` (regex компилируется, у number_range проверяется `min ≤ max`).

Где видно

- На согласовании отчёта (`/dq/submissions/{id}/checks`) — индикатор 🟢 зелёный / 🟡 жёлтый / 🔴 красный в очереди «Согласование отчётности».
- Управление правилами — страница «Проверки качества» (`DataQualityRulesPage`): CRUD + отчёт об использовании (`/dq/term-rules/usage`).

Итог. «DQ checks» реализованы как **term-level правила** (required/regex/number_range) в Task Manager, сгруппированные через DQLA-наборы из DG и привязанные к форме.

РАЗДЕЛ 8 · СПРАВОЧНИК

Workflow — машины состояний

Три независимые машины состояний: объекты Data Governance, шаблоны Form Builder и отчёты Submission Engine.

В системе **три независимые** машины состояний.

8.1 · Объекты Data Governance

Статусы: `draft`, `review`, `rejected`, `approved`, `deprecated`, `archived`.

```
draft      → review, archived
review     → rejected, approved
rejected   → review, archived
approved   → draft, deprecated, archived
deprecated → approved, archived
archived   → (терминальный)
```

Правила: `reject` требует комментарий; нельзя `archive` при наличии входящих связей; ролевой гейт `_assert_transition_roles` — `(draft→review): EDITOR/ADMIN`, `(review→approved): APPROVER/ADMIN`, архивация/депрекация — только ADMIN. Вход в `review` создаёт запись `meta_object_workflow`.

Важно. Любое редактирование согласованного объекта возвращает его в `draft` — нужно повторное согласование. Версионирование: `PUT ?create_new_version=true` разрешён только для `approved` — клонирует в новый `draft` с `version+1` и `previous_version_id`.

8.2 · Шаблоны Form Builder

```
draft → review → approved | rejected
```

Согласование поднимает major-версию; редактирование одобренного → откат в `draft` + patch.

8.3 · Отчёты Submission Engine (две оси)

ЖИЗНЕННЫЙ ЦИКЛ (`LOCAL_INSTANCE.STATUS` , FORM FILLER)

```
draft → in_progress → filled → signed → submitted → processing/review → accepted | rej
```

- `filled` при 100% заполнения, иначе `in_progress` ; редактирование блокируется после отправки.
- `signed` только из `filled` / `in_progress` ; `submitted` только из `signed` .
- `accepted` / `rejected` приходят обратно при следующей синхронизации.

СТОРОНА TASK MANAGER

- `report_submission.status` : `review` → `accepted` | `rejected` (через `/decision` ; `rejected` можно переподать, `accepted` — нет).
- `task_assignment.status` : `pending → submitted → accepted/rejected` .

ОСЬ СИНХРОНИЗАЦИИ (`LOCAL_STATUS`)

```
draft → ready_to_sync → synced
```

РАЗДЕЛ 9 · СПРАВОЧНИК

Customer Journey Map (CJM)

Сквозной путь пользователя из 12 этапов — от описания данных до сбора и консолидации отчётности. Он же навигация демо-стенда.

Стартовая страница `/demo/journey` — сетка 4×3 из 12 этапов; активные (оранжевые) блоки открывают нужный модуль под нужной ролью (автовыдача тестового токена).

1
USERS

2
Data Domains

3
Business Terms

4
BR

5
FSD

6
Building Form

7
Approve Form

8
Start Collection

9
Fill Collection

10
Approve Collection

11
Monitoring

12
Analysis

Этапы, роли и точки входа

Этап	Название	Роль	Точка входа	Действия
2	Data Domains	`domain_editor`	`/objects?class_code=DOMAIN`	создать/редактировать/на согласование
3	Business Terms	`term_editor`	`/objects?class_code=TERM`	мастер 4 шагов, привязка к домену
5	FSD	`fsd_editor`	`/objects?class_code=FSD`	блоки + пикер домен→термин
6	Building Form	`editor (FB)`	`/form-builder/`	drag-and-drop FSD/термины в секции
7	Approve Form	`approver (FB)`	`/form-builder/approvals`	открыть → согласовать/отклонить
8	Start Collection	`submission_editor`	`/submission/tasks`	создать задание → опубликовать
9	Fill Collection	`Организация`	`/demo_org/`	заполнить → подписать → отправить
10	Approve Collection	`submission_approver`	`/submission/approvals`	открыть (+DQLA) → согласовать/отклонить
12	Analysis	`submission_approver`	`/submission/consolidated`	сводные KPI, drill-down
+	Dashboard	`submission_approver`	`/submission/dashboard`	мониторинг статусов и дедлайнов

Сквозной поток данных

```
Домен → Термины → FSD → Шаблон формы (FB) → Задание на сбор (TM)
      → Заполнение (FF) → Согласование + DQ (TM) → Консолидация/аналитика (TM)
```

РАЗДЕЛ 10 · ПРИЛОЖЕНИЕ

Запуск и развёртывание

Как поднять весь стенд одной командой, запустить отдельный сервис локально и обновлять код на сервере.

Всё через Docker Compose

```
docker compose up --build
```

После старта nginx на `:80` раздаёт landing, а модули доступны по путям `/demo/`, `/demo/form-builder/`, `/demo/submission/`, `/demo_org/`, `/demo_admin/`, `/demo_dg/` (см. §2.3).

Локально по одному сервису (без Docker, на SQLite)

```
# Data Governance backend
cd backend && python -m venv .venv && source .venv/bin/activate
pip install -e . && uvicorn app.main:app --reload           # :8000
# DG frontend
cd frontend && npm install && npm run dev                    # :5173
```

Аналогично для `form_builder` (8100/5174), `submission_engine/task_manager` (8200/5175), `submission_engine/form_filler` (8300/5176).

Обновление кода на стенде

- **Frontend** (Vite dev): изменения подхватываются HMR (или пересборкой для admin/dg-module режимов).
- **Backend**: в Docker запущен **без** `--reload` → требуется пересборка образа (`docker compose up --build <service>`).

Демо-доступ

Все пользователи — на `/demo/login` (или автовыдача токена по клику в CJM `/demo/journey`). Пароли не требуются (тестовый режим).

Подключение к серверу-источнику

Копия проекта снята с сервера Tashkent. Пример подключения:

```
ssh -i ~/.ssh/tashkent.pem root@81.85.50.123
# либо через ~/.ssh/config:
ssh tashkent
```