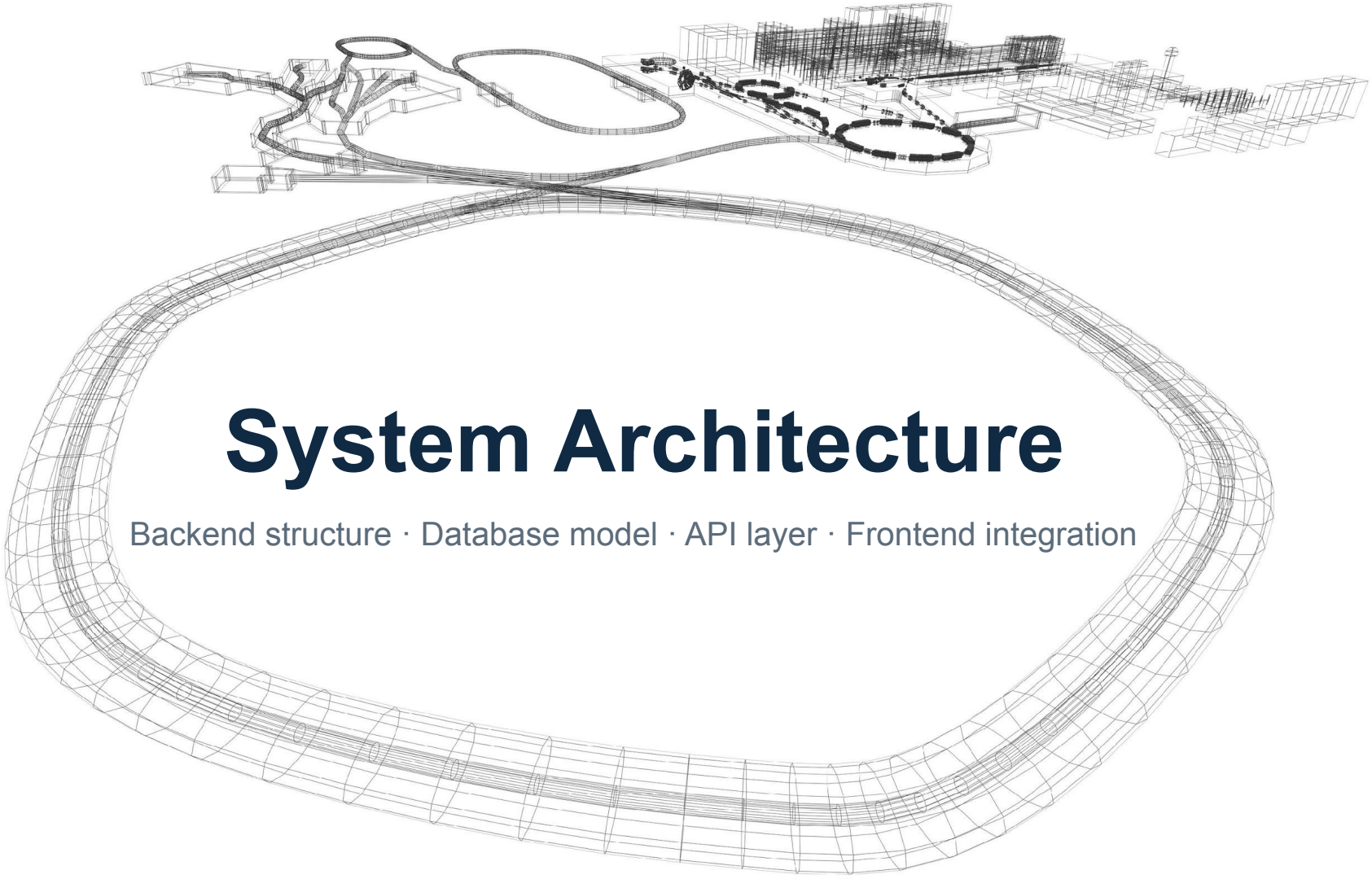


System Architecture

Backend structure · Database model · API layer · Frontend integration

Ivan Knežević, GSI Helmholtzzentrum für Schwerionenforschung

What is NAPMIX?

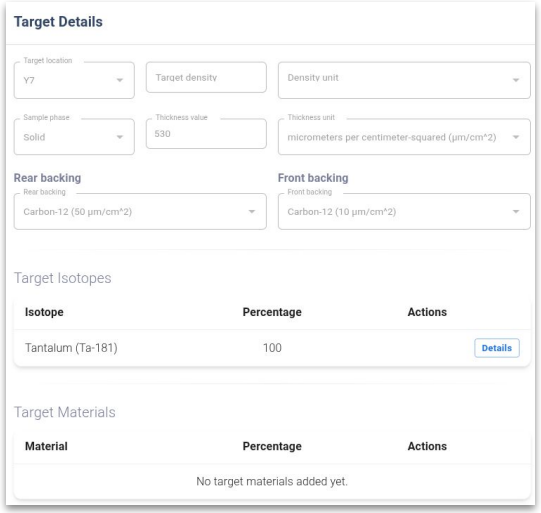
Metadata infrastructure for NAP Physics

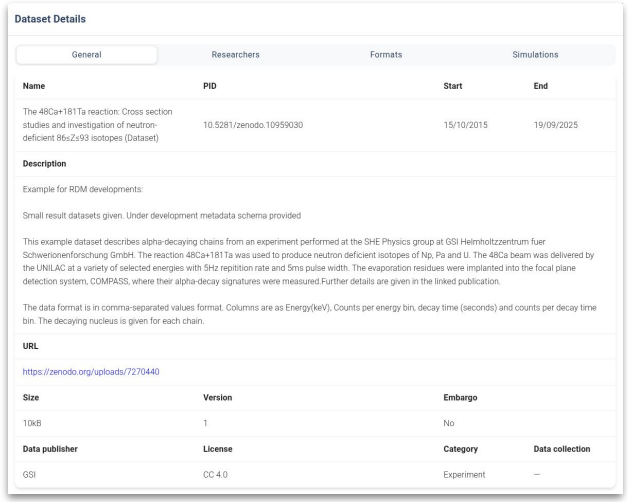
- NAPMIX is a metadata platform built around a **project-hub structure.**
- It connects project-level context with datasets, reactions, projectiles, targets, infrastructures, researchers, and related metadata.
- **Goal**: Capture richer experimental context and make it FAIR, reusable, and exportable.
- **Implementation**: Django + React architecture with a normalized relational schema.

> Project is the **umbrella entity**. This is why our strategy is project-rooted rather than dataset-rooted.

NAPMIX

Nuclear, Astro, and Particle Metadata Integration for eXperiments





Ivan Knežević, GSI Helmholtzzentrum für Schwerionenforschung; **Dr Quentin Fable**, Grand Accélérateur National d'Ions Lourds

What this session should make clear

1 **Backend structure**

Where Django models, serializers, views, permissions and utilities sit.

2 **Database model**

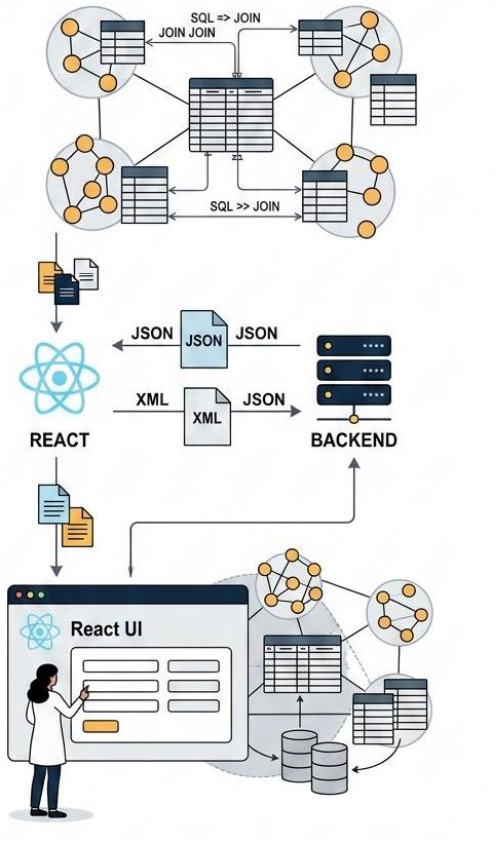
Why the schema is relational, nodal and cluster-based.

3 **API layer**

How React talks to the backend and how metadata is exposed.

4 **Frontend integration**

How users curate complex metadata without seeing database complexity.



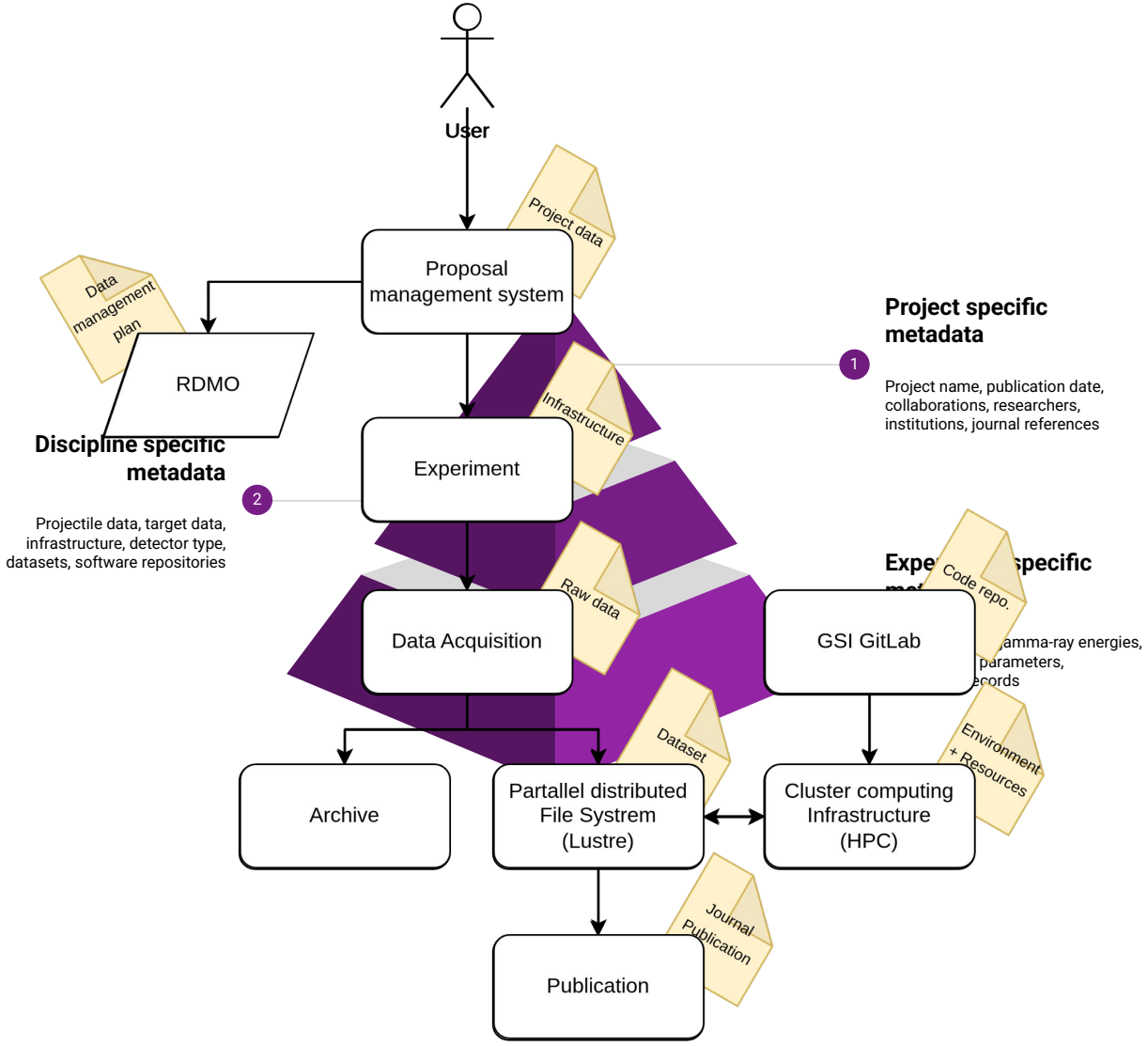
Architecture starts from the metadata problem

The problem

Experimental metadata is spread across files, spreadsheets, local conventions and repositories. The architecture has to make metadata structured, reusable and exportable without forcing every community into a single rigid workflow.

The design response

NAPMIX uses a shared common layer, domain-specific clusters, controlled vocabulary tables and serialisation/mapping logic so that the same curated metadata can serve local work, publication and harvesting.



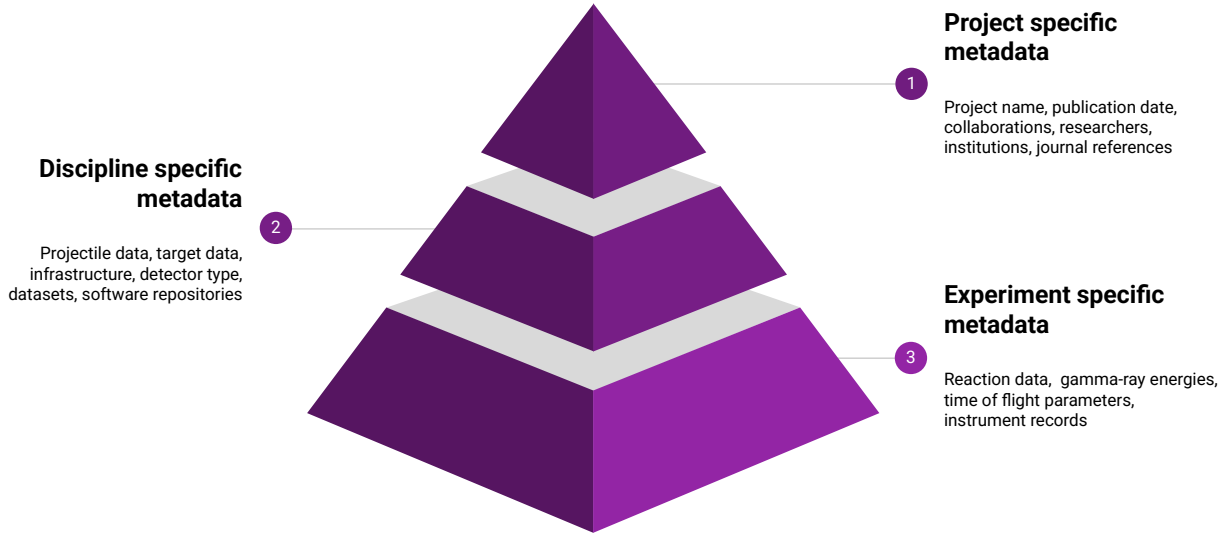
Architecture starts from the metadata problem

The problem

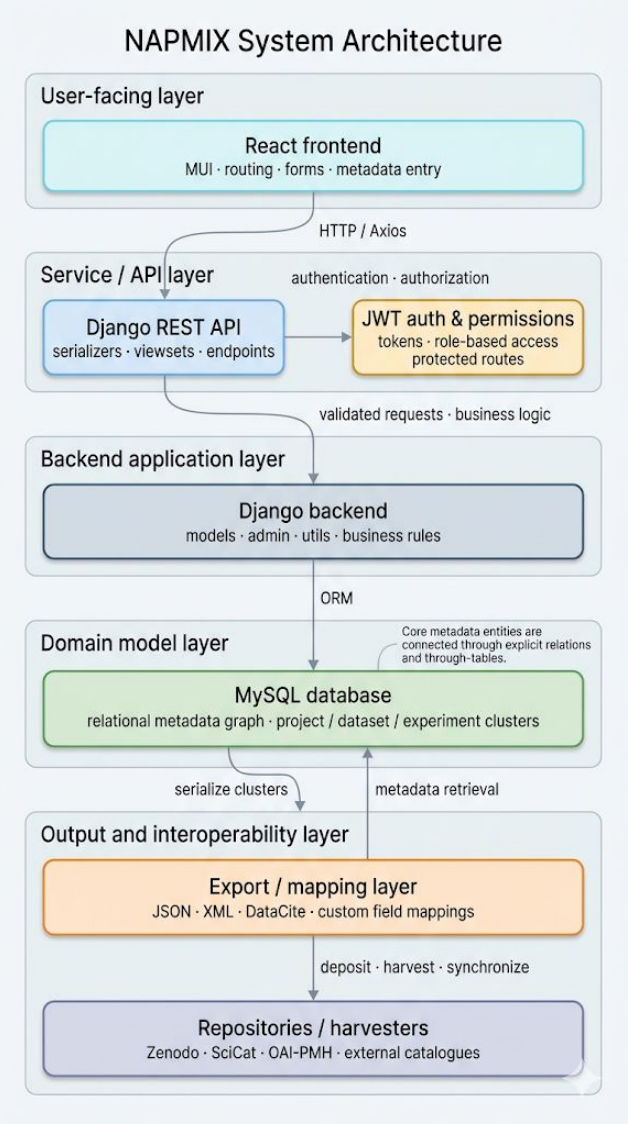
Experimental metadata is spread across files, spreadsheets, local conventions and repositories. The architecture has to make metadata structured, reusable and exportable without forcing every community into a single rigid workflow.

The design response

NAPMIX uses a shared common layer, domain-specific clusters, controlled vocabulary tables and serialisation/mapping logic so that the same curated metadata can serve local work, publication and harvesting.



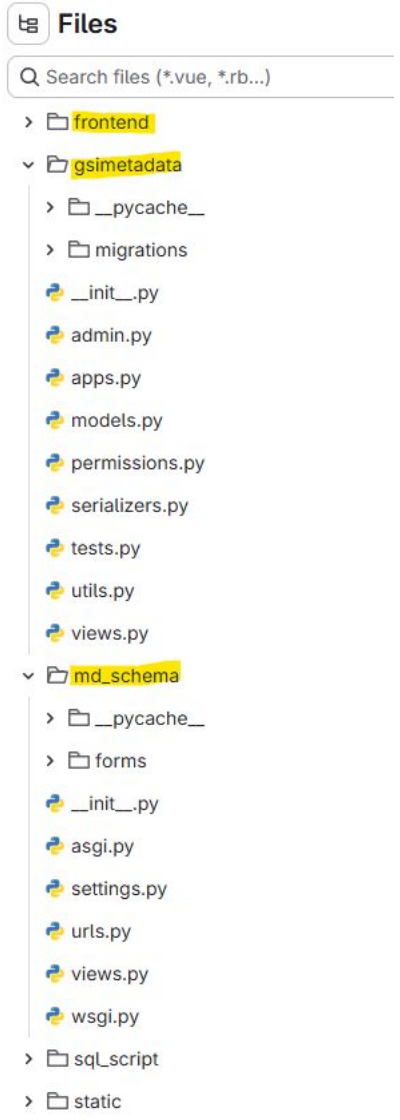
System architecture overview



- **React frontend** provides metadata entry, browsing, and editing interfaces.
- **Django REST API** exposes backend functionality through structured endpoints.
- **JWT authentication** manages secure access and user permissions.
- **Django backend** implements models, validation logic, admin tools, and export utilities.
- **MySQL database** stores the relational metadata graph across project, dataset, and experiment clusters.
- **Export layer** maps internal metadata into JSON, XML, DataCite, and repository-ready formats.

The architecture separates **user interaction, business logic, data persistence, and interoperability** into clear layers.

Repository and runtime structure



frontend/

React application: routing, pages, MUI components, Axios API services, JWT token handling and user-facing metadata management screens.

md_schema/

Django project configuration: settings, URL routing, WSGI/ASGI entry points and cross-cutting project setup.

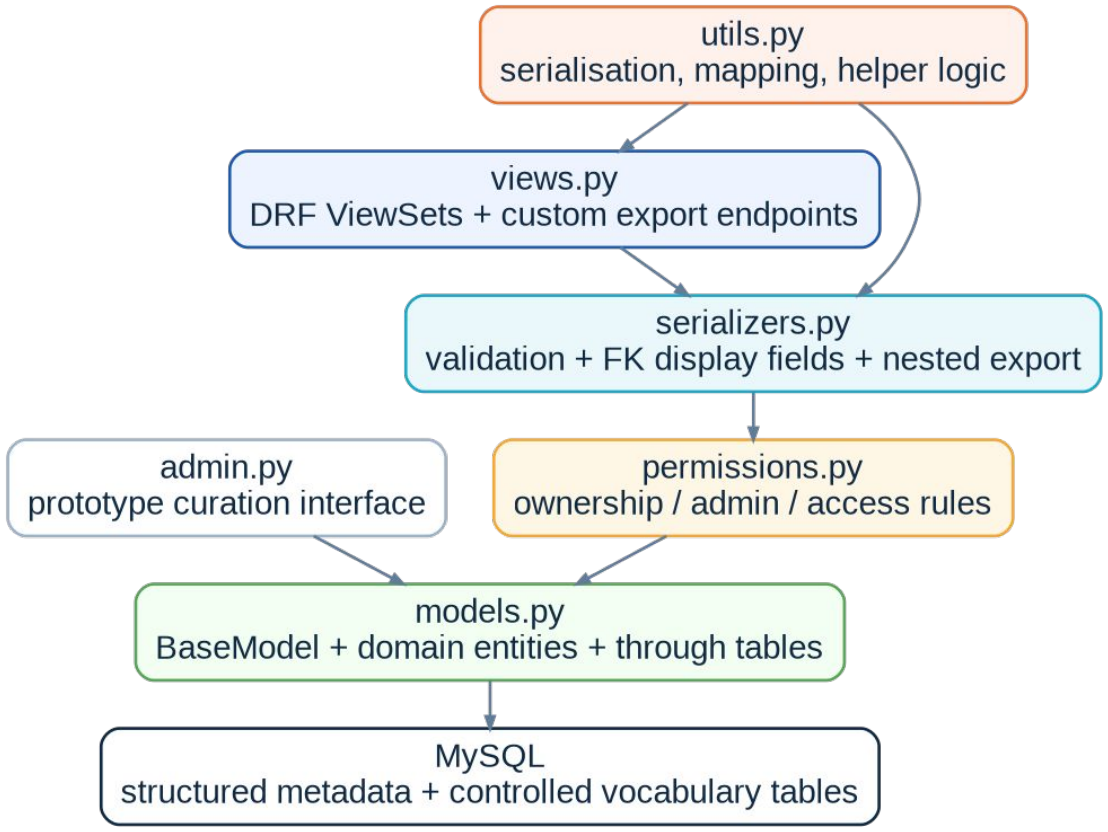
gsimetadata/

Django app containing the domain model, admin configuration, serializers, API viewsets, permissions and export utilities.

Docker + MySQL

The deployment path combines a Django/Gunicorn web service and a MySQL database service, with configuration through environment files.

Backend structure

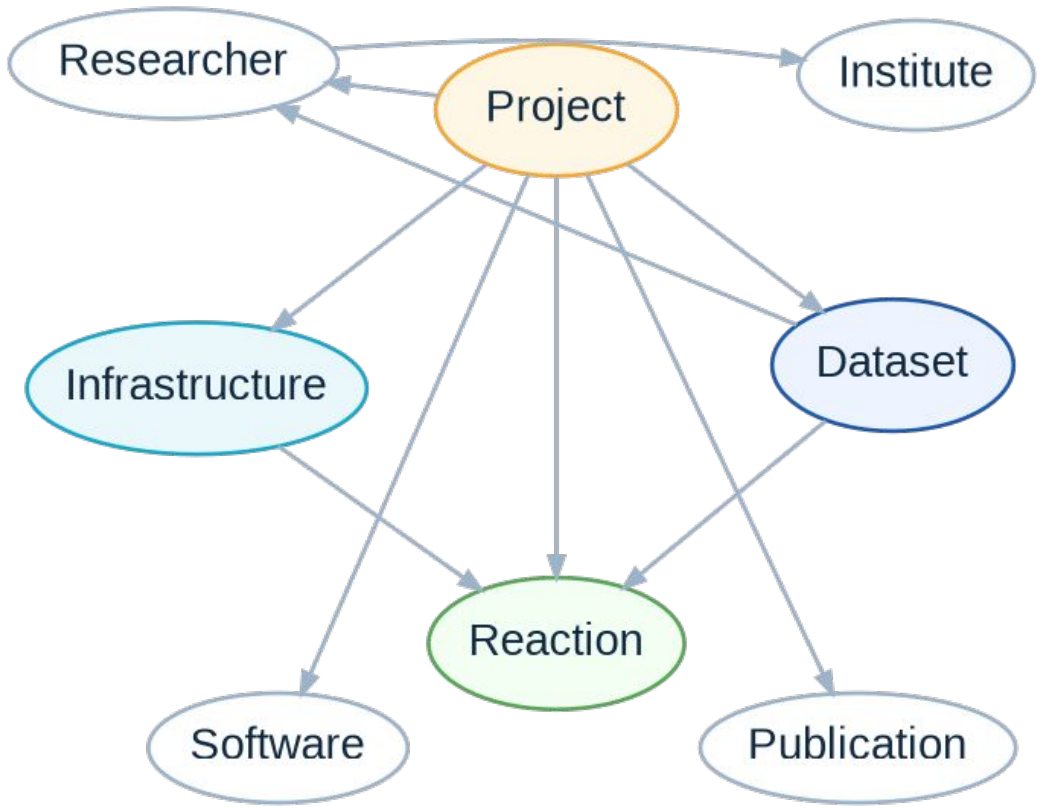


- **models.py** defines the persistent metadata graph and controlled vocabularies.
- **serializers.py** turns relational objects into API payloads and export structures.
- **views.py** exposes CRUD and custom metadata export endpoints through DRF.
- **permissions.py** keeps ownership/admin rules outside of model logic.
- **admin.py** remains useful as a curation and debugging interface.

```
class BaseModel(models.Model):
    created_by = ForeignKey(User, ...)
    updated_at = DateTimeField(auto_now=True)
    is_deleted = BooleanField(default=False)
    deleted_by = ForeignKey(User, null=True, ...)

    def delete(self, user=None):
        # frontend/API soft delete
        ...
```

Database model: nodal clusters



Common layer

Project, researcher, institute, infrastructure, software and publication metadata are modelled once and reused across NAP communities.

Dataset layer

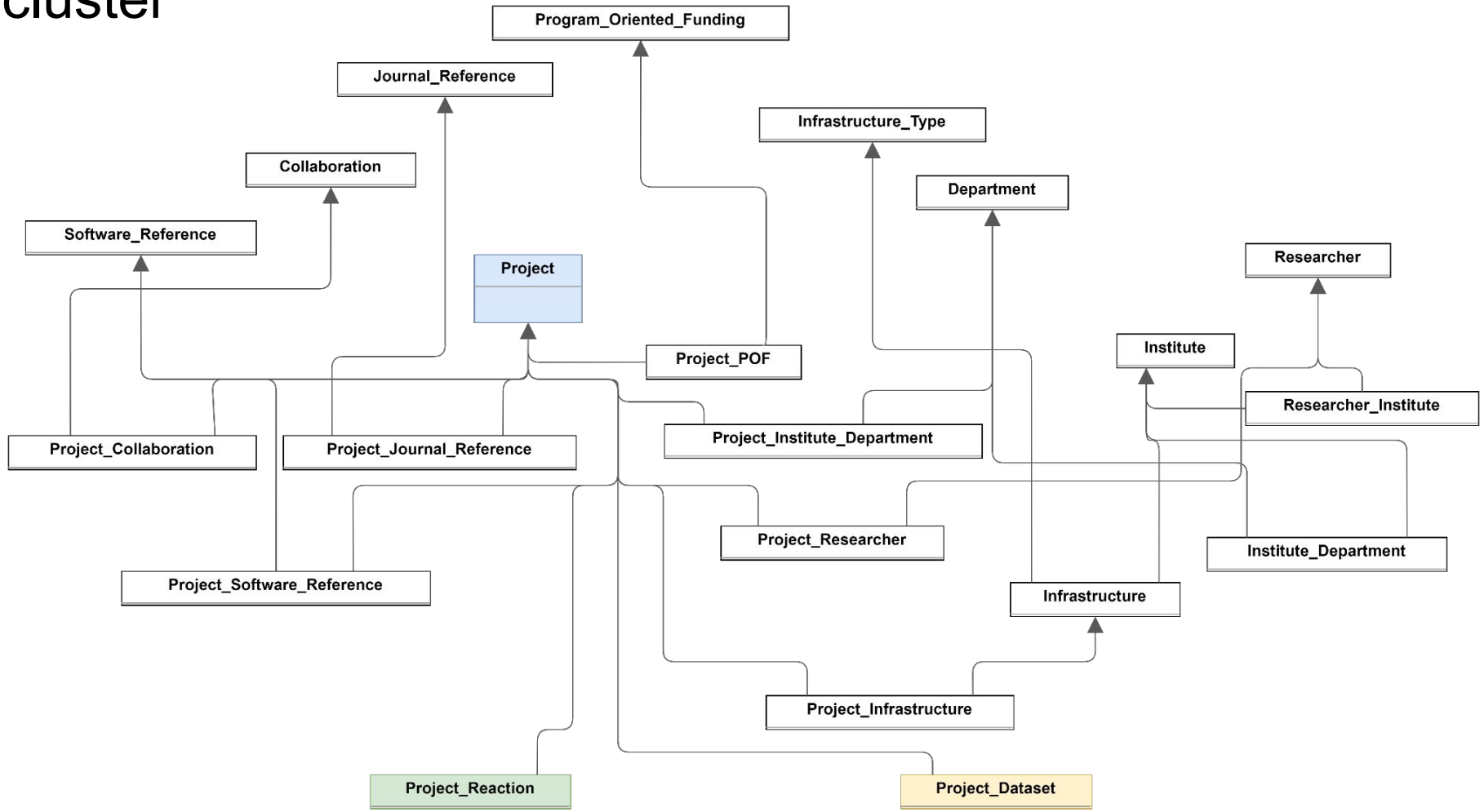
Datasets act as FAIR publication objects linked to formats, licences, researchers, data collections and reactions.

Experiment layer

Reaction and related physics entities capture domain-specific metadata without breaking the shared project and dataset layers.

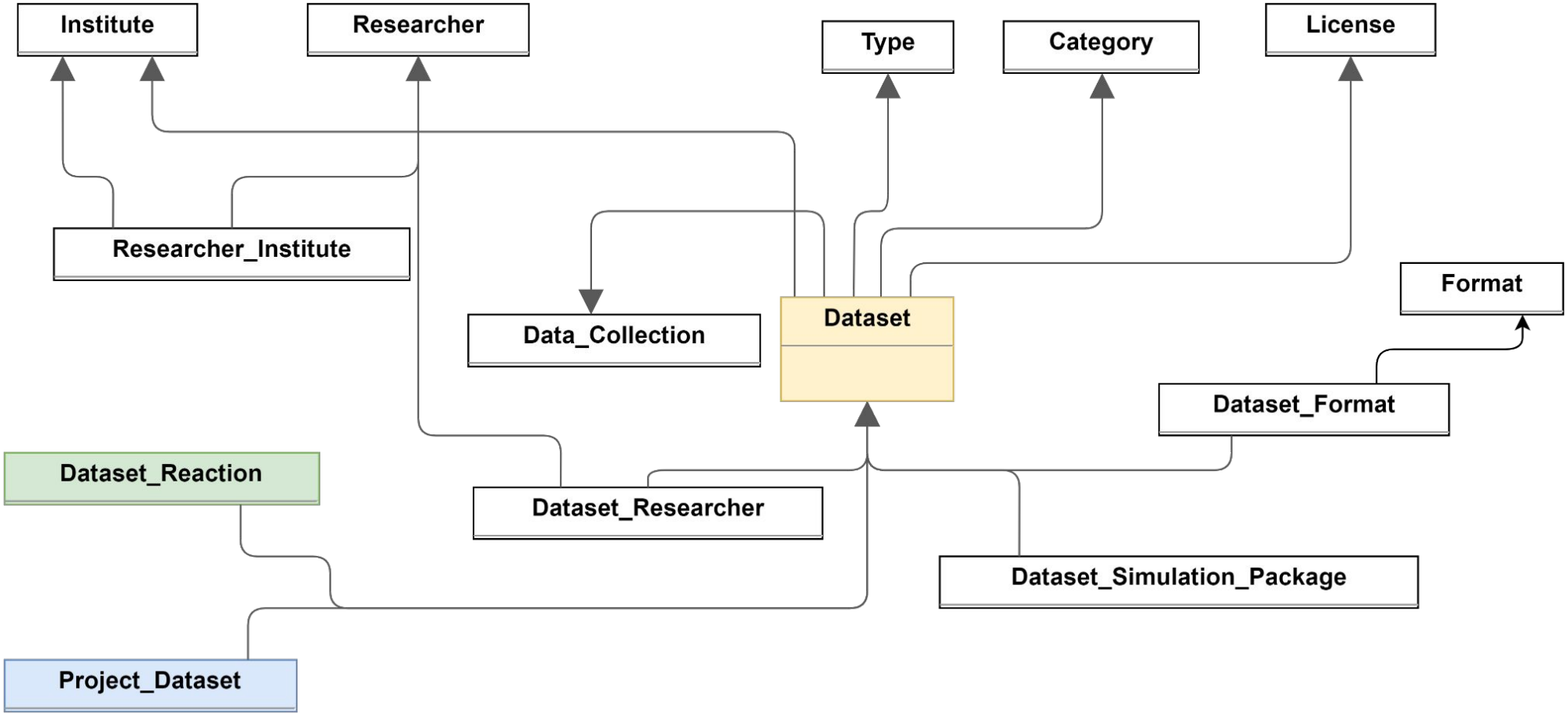
The Database Schema Design

Project cluster



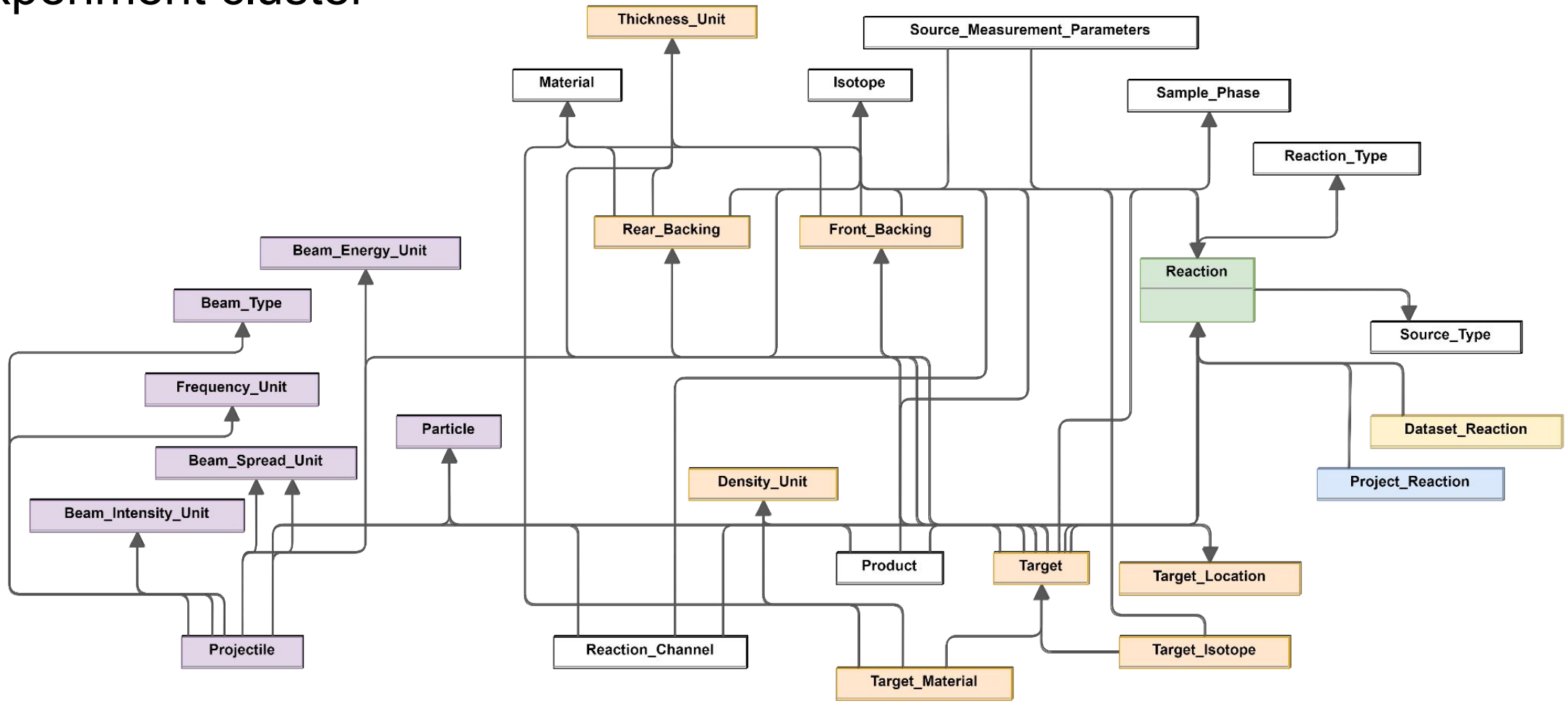
The Database Schema Design

Dataset cluster

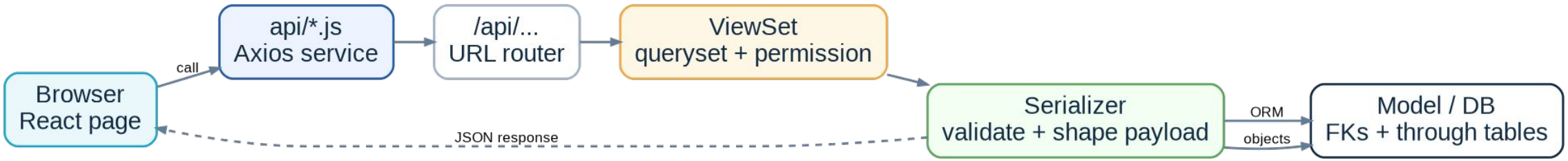


The Database Schema Design

Experiment cluster



API layer



CRUD endpoints

Most major models are exposed through REST endpoints so frontend management pages can list, create, update, soft-delete and inspect records.

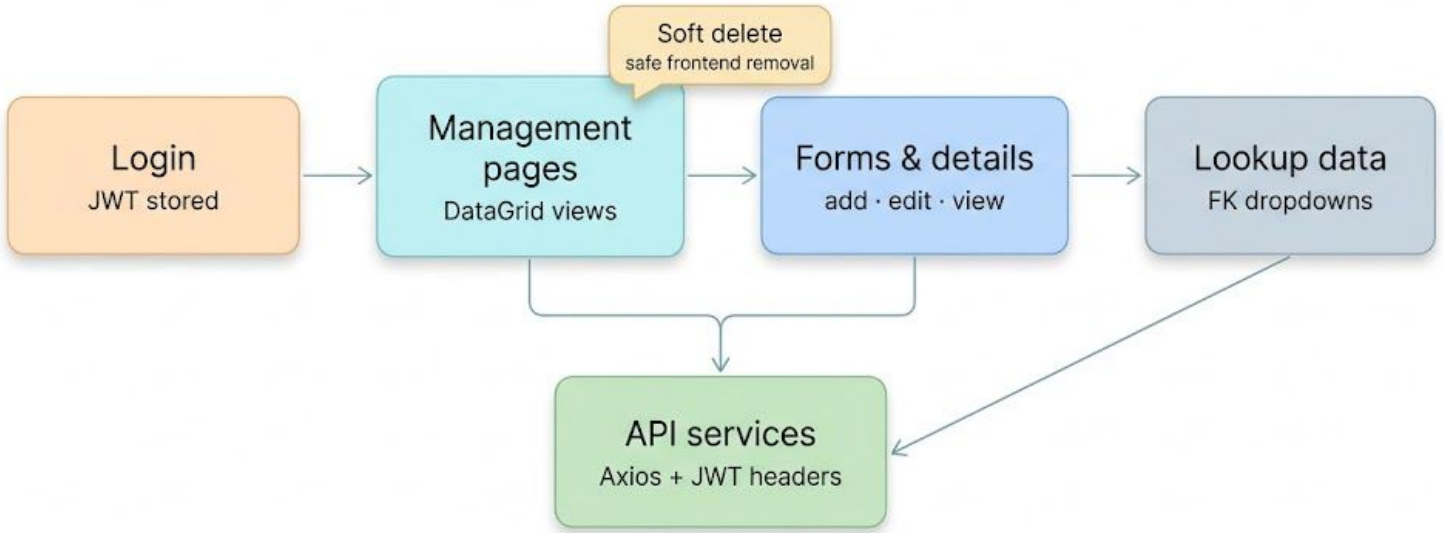
Payload shaping

Serializers expose both IDs for writes and readable labels for tables/details, reducing frontend guesswork around foreign keys.

Custom exports

Special endpoints collect connected records and serialise them into JSON/XML/DataCite-oriented outputs for FAIR workflows.

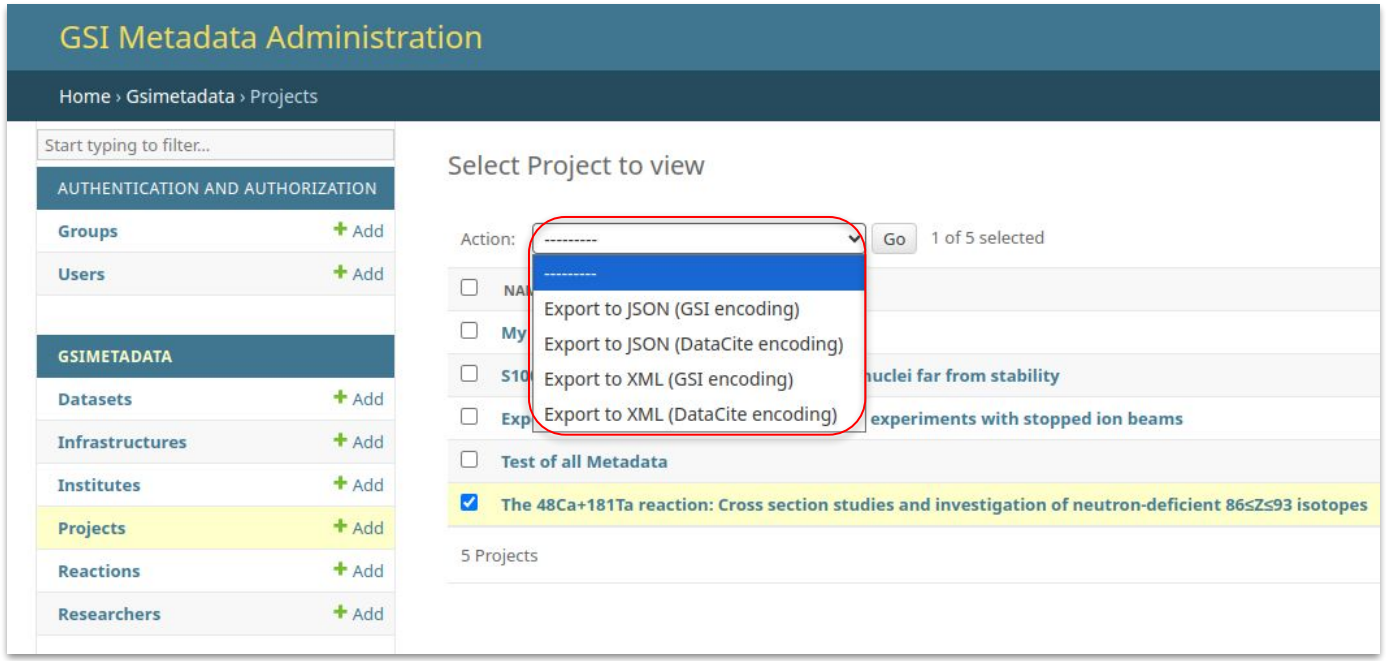
Frontend integration



- MUI DataGrid pages give users a management view of each entity type.
- Add/edit/detail modals hide relational complexity behind grouped forms and vertical tabs.
- Axios services centralise API calls and JWT headers.
- Foreign-key dropdowns are populated from API lookups and can support dependent selections.
- Soft delete keeps frontend operations safe while preserving admin-level recovery/debugging.

Metadata export and interoperability

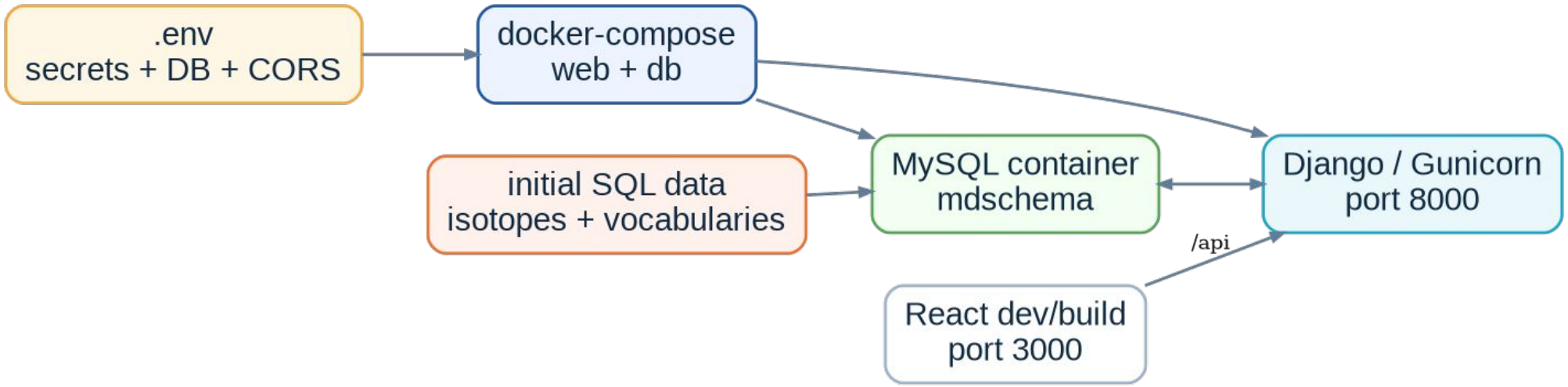
Select project or dataset → Collect clusters Project · Dataset · Reaction → Map attributes internal → target schema → Serialize output JSON · XML · DataCite → Publish / harvest repository · DOI · OAI-PMH



JSON schema

```
"reactions": [
    {
        "name": "208Pb + 9Be",
        "reaction_type": "Fragmentation",
        "source_type": "Accelerator",
        "projectiles": [
            {
                "isotope": "Lead (Pb-208)",
                "percentage": 100.0,
                "beamEnergy": 1050.0,
                "beamEnergyUnit": "Megaelectronvolt/nucleon (MeV/u)",
                "frequency_type": "Pulsed",
                "beamIntensity": 1500000000.0,
                "beamIntensityUnit": "Particles per spill (pps)",
                "beam_destination": "HFS",
                "beam_type": "Stable"
            }
```

Deployment and local installation model




```
docker-compose up --build
python3 manage.py createsuperuser
mysql -u ${DB_USER} -p${DB_PASSWORD} ${DB_NAME}
< mdschema_script.sql
npm install && npm start
```

- Configuration is externalised in backend and frontend .env files.
- Backend runs as a web service on port 8000; React consumes /api from port 3000 in local development.
- Initial data import covers controlled vocabularies and large reference tables such as isotopes.
- This setup is understandable for partner institutes and suitable for custom local deployments.

NAPMIX is a metadata service architecture

not just a database schema.

Backend
Reusable Django service with model, API, permissions and export logic.

Database
Clustered relational graph for shared and domain-specific metadata.

API
Stable layer between curation UI and metadata model.

Frontend
Guided React/MUI interface for entering complex relations safely.

Interoperability
JSON/XML/DataCite-ready export path towards repositories and harvesters.