
Orefox KMS - Detailed Application Overview

What is Orefox KMS?

Orefox KMS (Knowledge Management System) is a proof-of-concept web application designed for the mining and

exploration industry. It serves as a centralized platform for managing:

- Documents (PDFs of exploration reports, geological surveys, technical documents)
- Geospatial datasets (mining projects, exploration tenements, drillholes, prospects)
- Metadata and tagging for intelligent search and retrieval
- Geospatial intelligence for location-based queries (e.g., "find all documents within 5km of this tenement")

The application demonstrates how geological/exploration data can be stored, tagged, and accessed via a clean web

interface with spatial query capabilities.

Technology Stack

Backend

- Django 5.x - Python web framework providing ORM, migrations, authentication, and admin interface
- GeoDjango - Django's geospatial extension for handling geographic data (polygons, points)
- PostgreSQL 16 + PostGIS 3.4 - Relational database with spatial extensions for geospatial queries
- MinIO - S3-compatible object storage for file uploads (PDFs, documents)
- django-storages + boto3 - Integration between Django and S3-compatible storage

Frontend

- HTMX - Lightweight JavaScript library for dynamic page updates without heavy frameworks
- Tailwind CSS - Utility-first CSS framework loaded via CDN (no build step required)
- Vanilla JavaScript - Minimal progressive enhancement (drag-and-drop file uploads)

Infrastructure

- Docker + Docker Compose - Containerized development environment
- Python 3.11 - Application runtime

Directory Structure Breakdown

```
IFB398-T07-MINDS/
│
├── config/                # Django project configuration
│   ├── settings.py        # Main settings (database, storage, middleware)
│   ├── urls.py            # Root URL configuration
│   ├── wsgi.py            # WSGI server entry point
│   └── asgi.py            # ASGI server entry point (async)
│
├── core/                  # Main Django application
│   ├── models.py          # Database models (Organisation, Process, Document, etc.)
│   ├── views.py           # Request handlers (dashboard, upload, documents)
│   ├── urls.py            # URL routing for core app
│   ├── admin.py           # Django admin customization
│   ├── forms.py           # Form definitions (DocumentForm)
│   ├── utils.py           # Utility functions (SHA-256 file hashing)
│   ├── apps.py            # App configuration
│   └── migrations/        # Database migration files
```

```
│   └── 0001_initial.py     # Initial schema creation
│
├── templates/              # HTML templates
│   ├── base.html           # Base template with Tailwind CSS
│   └── core/
│       ├── dashboard.html      # Main dashboard with metrics and sidebar
│       ├── upload.html         # Document upload page with drag-and-drop
│       └── projects.html       # Projects listing page
│
├── infra/                  # Infrastructure configuration
│   └── web/
│       ├── Dockerfile          # Docker image for Django app
│       └── requirements.txt    # Python dependencies
│
├── media/                  # User-uploaded files (local development fallback)
│   └── docs/               # Document storage directory
│
├── docker-compose.yml          # Multi-container orchestration
├── manage.py               # Django management script
├── seed-db.py              # Database seeding script for test data
├── pyproject.toml          # Python project metadata and dependencies
├── .env                    # Environment variables (secrets, config)
└── README.md               # Project documentation
```

Core Components Deep Dive

1. Data Models (core/models.py)

The application follows a hierarchical domain model with automatic validation:

Organisation

- Represents a mining/exploration company
- Fields: id, name, mode (EXPLORATION or MINING), created_at, updated_at
- Root entity - all other models reference this

Process (may be renamed to "Campaign")

- Represents a project or mining operation under an Organisation
- Fields: id, name, organisation, mode (PROJECT or OPERATION), commodity, geom (MultiPolygon), timestamps
- Contains geospatial boundary (MultiPolygonField with SRID 4326)
- Used for spatial queries (e.g., "which documents are within this project area?")

Prospect, Tenement, Drillhole

- Domain-specific entities for exploration activities
- All link to both Organisation and Process
- Basic structure currently, ready for expansion

Document

- Stores file metadata and handles uploads to MinIO
- Fields: id, title, file, organisation, process, tags (ArrayField), timestamp, doc_type, confidentiality,

checksum_sha256, created_by, timestamps

- Key feature: SHA-256 checksum prevents duplicate uploads
- File storage: Uses MinIO via S3 API (not filesystem)

Model Mixins (Advanced Pattern)

ValidatedChoiceModel = ChoiceValidationMixin + AutoCleanMixin

- ChoiceValidationMixin: Automatically validates choice fields against allowed values

- AutoCleanMixin: Runs full_clean() before every save

- Used by Organisation and Process models for robust data validation

---

2. Views & Business Logic (core/views.py)

Dashboard (dashboard view)

- Main landing page after login

- Displays metrics cards: project count, document count, prospect/drillhole/tenement counts

- Shows 8 most recent documents

- Uses dynamic model loading (_get_model() helper) to work even if models don't exist yet

Document Upload (upload_doc view)

- Handles GET (display form) and POST (process upload)

- Duplicate detection workflow:

  a. Compute SHA-256 checksum of uploaded file (via sha256_file())

  b. Check if any document has matching checksum

  c. If duplicate found: reject upload with error message

  d. If unique: save to MinIO and create database record

- Form rendered with drag-and-drop enhancement (vanilla JS in template)

Other Views

- documents: Document library with search and pagination

- prospects, drillholes, tenements: Placeholder pages (models defined but minimal implementation)

- ai_insights: Placeholder for future AI features

- map_view: Placeholder for geospatial map interface

- healthcheck: JSON endpoint for container health monitoring

---

3. Django Admin (core/admin.py)

Customized admin interface:

```
@djadmin.register(Process)
class ProcessAdmin(GISModelAdmin):  # Uses GeoDjango map widget
    list_display = ("name", "mode", "commodity")
```

- ProcessAdmin: Uses GISModelAdmin to enable interactive map editing for polygon boundaries

- DocumentAdmin: Standard admin for document metadata

- Access at http://localhost:8000/admin

---

4. URL Routing

Root URLs (config/urls.py)

```
urlpatterns = [
    path("admin/", admin.site.urls),     # Django admin
    path("", include("core.urls")),      # Core app URLs
]
```

Core App URLs (core/urls.py)

```
urlpatterns = [
    path("", views.dashboard, name="dashboard"),
```

```
    path("upload/", views.upload_doc, name="upload"),
    path("documents/", views.documents, name="documents"),
    path("prospects/", views.prospects, name="prospects"),
    path("drillholes/", views.drillholes, name="drillholes"),
    path("tenements/", views.tenements, name="tenements"),
    path("ai-insights/", views.ai_insights, name="ai_insights"),
]
```

---

5. Templates

Base Template (templates/base.html)

- Minimal layout with Tailwind CSS loaded from CDN
- Defines {% block content %} for child templates
- No JavaScript build step required

Dashboard (templates/core/dashboard.html)

- Sidebar navigation with links to all modules
- Five metric cards showing counts
- Recent documents table
- User info section with admin panel link

Upload Page (templates/core/upload.html)

- Django form with CSRF protection
- Drag-and-drop zone (vanilla JS enhancement)
- Real-time file size display
- Recent uploads list (last 20)
- Inline JavaScript for progressive enhancement (no external dependencies)

---

6. File Utilities (core/utils.py)

SHA-256 File Hashing

```
def sha256_file(django_file) -> str:
    """Compute SHA-256 checksum of uploaded file in chunks"""
    h = hashlib.sha256()
    for chunk in iter(lambda: django_file.read(8192), b""):
        h.update(chunk)
    return h.hexdigest()
```

- Reads file in 8KB chunks (memory efficient for large files)
- Used for duplicate detection before saving

---

7. Database Configuration (config/settings.py)

PostgreSQL + PostGIS

```
DATABASES = {
    "default": {
        "ENGINE": "django.contrib.gis.db.backends.postgis",  # GeoDjango
        "NAME": env("DB_NAME"),
        "USER": env("DB_USER"),
        "PASSWORD": env("DB_PASS"),
        "HOST": env("DB_HOST"),
        "PORT": env("DB_PORT"),
    }
}
```

MinIO Storage

DEFAULT_FILE_STORAGE = "storages.backends.s3boto3.S3Boto3Storage"

AWS_S3_ENDPOINT_URL = f"http://{env('MINIO_ENDPOINT')}"

AWS_STORAGE_BUCKET_NAME = env("MINIO_BUCKET")

- All file uploads go to MinIO (S3-compatible object storage)

- Database only stores file metadata and references

---

8. Docker Infrastructure

docker-compose.yml

Orchestrates 4 services:

1. db (PostgreSQL + PostGIS)
  - Image: postgis/postgis:16-3.4
  - Healthcheck: pg_isready
  - Persistent volume: db_data
  - Port: 5432
2. minio (Object storage)
  - Image: minio/minio:latest
  - Console port: 9001
  - API port: 9000
  - Persistent volume: minio_data
3. create-bucket (One-time setup)
  - Creates MinIO bucket on startup
  - Sets download permissions
  - Exits after completion
4. web (Django application)
  - Built from infra/web/Dockerfile

- Runs migrations on startup
- Port: 8000
- Volume-mounted source code (hot reload in development)

Dockerfile (infra/web/Dockerfile)

```
FROM python:3.11-slim

# Install GDAL, GEOS, PROJ for GeoDjango
RUN apt-get update && apt-get install -y \
  gdal-bin libgdal-dev \
  proj-bin libproj-dev \
  libgeos-dev

# Install Python dependencies
COPY requirements.txt .
RUN pip install -r requirements.txt

# Run Django development server
CMD ["bash", "-lc", "python manage.py migrate && python manage.py runserver 0.0.0.0:8000"]
```

Key dependencies:
- GDAL (Geospatial Data Abstraction Library) - handles geospatial raster/vector data
- GEOS (Geometry Engine) - performs spatial operations
- PROJ (Cartographic Projections Library) - coordinate transformations

---
9. Environment Configuration (.env)

All configuration is externalized:

```
# Django
DJANGO_DEBUG=1
SECRET_KEY=your-long-random-string

# PostgreSQL
POSTGRES_DB=orefox
POSTGRES_USER=postgres
POSTGRES_PASSWORD=postgres
DB_HOST=db     # Docker service name
DB_PORT=5432

# MinIO
MINIO_ROOT_USER=minio
MINIO_ROOT_PASSWORD=minio12345
MINIO_BUCKET=documents
MINIO_ENDPOINT=minio:9000  # Docker service name
MINIO_USE_SSL=0
```

Never commit secrets to Git - .gitignore excludes .env

---

10. Database Seeding (seed-db.py)

Development utility for generating test data:

- Uses Faker library to generate random data
- Creates Organizations and Documents with realistic names
- Inserts directly via psycopg2 (bypasses Django ORM)
- Note: Requires local psycopg2 or run via nix-shell

Application Flow

Startup Sequence

1. Docker Compose starts services in dependency order:
 - PostgreSQL starts and runs healthcheck
 - MinIO starts
 - Bucket creation service runs and exits
 - Django web app starts (waits for db health + bucket creation)
2. Django initialization:
 - Loads environment variables from .env
 - Connects to PostgreSQL
 - Runs pending migrations (python manage.py migrate)
 - Starts development server on port 8000

Document Upload Flow

1. User navigates to /upload/
2. Selects/drags file into upload zone
3. JavaScript displays file name and size
4. User fills in metadata (title, project, doc_type, etc.)
5. Form submits to upload_doc view
6. Django reads uploaded file and computes SHA-256
7. Checks database for existing document with same checksum
8. If unique:
 - Saves file to MinIO (via S3 API)
 - Creates Document record in PostgreSQL
 - Redirects to upload page with success
9. If duplicate:
 - Re-renders form with error message
 - File is NOT saved

Geospatial Query Example

```
# Find all processes within 5km of a point
from django.contrib.gis.measure import D
from django.contrib.gis.geos import Point

point = Point(-115.8, 37.2, srid=4326)  # Longitude, Latitude
nearby = Process.objects.filter(geom__distance_lte=(point, D(km=5)))
```

Key Design Decisions

1. Container-Based Development

- Why? Eliminates "works on my machine" issues
- Benefit: No local Python/PostgreSQL/MinIO installation needed
- Tradeoff: Slightly slower startup than native development

2. GeoDjango + PostGIS

- Why? Mining/exploration is inherently spatial
- Benefit: Native support for spatial queries (distance, contains, intersects)
- Use case: "Find all documents related to projects within 10km of this tenement"

3. MinIO vs. Filesystem

- Why? S3-compatible storage is production-ready
- Benefit: Easy migration to AWS S3 or other cloud storage
- Local development: MinIO runs in Docker, no cloud account needed

4. HTMX + Tailwind (No Build Step)

- Why? Proof-of-concept doesn't need complex frontend

- Benefit: Fast iteration, no webpack/vite configuration

- Tradeoff: Less suitable for complex single-page apps

5. SHA-256 Deduplication

- Why? Prevent duplicate document uploads

- Implementation: Hash computed before save, checked against existing records

- Benefit: Saves storage space and prevents data duplication

6. UUID Primary Keys

- Why? Better for distributed systems and data merging

- Benefit: No ID collisions when importing from multiple sources

- Tradeoff: Slightly larger index size than integers

What Each File Is Responsible For

File	Responsibility
manage.py	Django management CLI entry point
config/settings.py	Django configuration (database, storage, middleware, installed apps)
config/urls.py	Root URL routing (admin + core app)
core/models.py	Database schema definitions (Organisation, Process, Document, etc.)
core/views.py	Request handlers and business logic (dashboard, upload, search)
core/admin.py	Django admin customization (GIS map widget for Process)
core/forms.py	Form definitions for user input (DocumentForm)

| core/utils.py | Utility functions (SHA-256 hashing, commented-out metaclass experiments) |

| core/urls.py | URL routing for core app endpoints |

| templates/base.html | HTML base layout with Tailwind CSS |

| templates/core/dashboard.html | Main dashboard UI with metrics and sidebar |

| templates/core/upload.html | Document upload page with drag-and-drop |

| docker-compose.yml | Multi-container orchestration (db, minio, web) |

| infra/web/Dockerfile | Django container image definition |

| infra/web/requirements.txt | Python dependencies (Django, GeoDjango, boto3, etc.) |

| seed-db.py | Database seeding script for test data |

| .env | Environment variables (secrets, config) |

Current State & Future Directions

What's Working

✅ Docker Compose environment with PostgreSQL + PostGIS + MinIO ✅ Django admin with GIS map editing ✅ Document upload with SHA-256 deduplication ✅ Basic dashboard with metrics ✅ Data models for exploration domain

What's Incomplete

⚠️ Prospect/Drillhole/Tenement pages are placeholders ⚠️ No geospatial map viewer in UI (only in admin) ⚠️ AI insights page is a placeholder ⚠️ No user authentication/authorization (uses Django's default) ⚠️ Document search is basic (no full-text search or advanced filters)

Potential Enhancements

- Map Interface: Add Leaflet.js for public-facing map visualization

- Spatial Search: "Find documents near this location" feature

- AI Integration: Document summarization, entity extraction

- Advanced Search: Full-text search, tag filtering, date ranges

- File Preview: PDF viewer in browser

- Bulk Import: CSV/Shapefile import for tenements/drillholes

This is a well-structured proof-of-concept that demonstrates the core value proposition of a geospatial knowledge

management system for the mining industry. The architecture is clean, the code follows Django best practices, and

the container-based setup makes it easy to develop and deploy.
