SDL Developer Guide

Version latest, 2026-05-01

Developer Guide

Welcome to the SDL Developer Guide. This documentation is designed for developers, system administrators, and technical personnel responsible for installing, configuring, deploying, and maintaining SDL (SOF Data Layer) systems..

Purpose of This Guide

This developer guide focuses on **installing, configuring, and deploying** SDL platforms. If you're an end-user looking to operate an already deployed SDL system, please refer to the [R]DP User Guide.

What is SDL?

SDL is a modular, container-native data platform that enables organizations to collect, process, fuse, query, and disseminate mission-critical data — from cloud data centers to disconnected tactical edge nodes. The platform is decentralized by design, capable of running on tactical edge, on-prem, or cloud services, and can be deployed in 100% air-gapped environments.

SDL federates data across distributed nodes while maintaining data governance. A single software baseline adapts its runtime behavior to the deployment environment.

For a comprehensive architectural overview, see Platform Architecture.

Who This Guide Is For

- **Developers**: Software engineers building custom integrations and data pipelines
- **System Administrators**: Personnel responsible for deploying and maintaining SDL
- **DevOps Engineers**: Staff managing CI/CD pipelines and infrastructure
- **Data Engineers**: Technical staff designing data flows and transformations
- **Technical Architects**: Decision-makers evaluating and designing SDL deployments

Key Platform Capabilities

Streaming & Transformation

Real-time event streaming and transformation hub supporting 14+ tactical data formats. Hub-and-spoke architecture with bidirectional format conversion. Learn more about Data Pipelines →

Federated Query & Analytics

Federated SQL engine, virtual knowledge graph (VKG), and real-time analytics. "Zero ETL" — data stays where it lives. Learn more about Federated SQL →

Security & Governance

Policy engine with row-level and column-level data obfuscation, classification markings, and multi-enclave evaluation. Data-policy-as-code deployed alongside the platform. Learn more about Security →

Getting Started

SDL is packaged as a collection of Helm charts located in `$RDP_HOME/charts`. Environment-specific overrides live in `$RDP_HOME/overrides/` (e.g., `sdl-dev`, `dev-minimal`).

Quick Start

```
# Clone the repo and set RDP_HOME
git clone https://github.com/raft-tech/sdl.git
export RDP_HOME=$(pwd)/rdp

# Provide credentials for the GitHub container registry
export GHP_USERNAME=your_github_username
export GHP_SECRET=your_github_personal_access_token

# Create a local Kind cluster
${RDP_HOME}/scripts/rdp_create_kind.sh

# Deploy SDL (use dev-minimal for a lighter footprint)
${RDP_HOME}/scripts/rdp_deploy.sh -e dev-minimal
```

For prerequisite CLI tools (kubectl, helm, kind, jq, yq) and detailed environment setup, see Tools & Environment.

Two Paths for Data Flow

SDL's architecture provides **principled flexibility**: enforcement where it creates value and flexibility where it enables mission.

Path 1: Data Model Path

Feeds from tactical systems (TAK, GCCS-J, TRAX, and others) pass through configurable transformers that map source data to a common data model. Organizations choose one of two options:

Bring Your Own Data Model (BYODM)

> Organizations can replace the Warfighting Data Model with their own model. The platform is unopinionated about which model is used — it enforces whatever model the organization configures (e.g., OMS/UCI or other domain-specific models).

This is an **organization-level choice**, not a per-deployment choice. When an organization selects a

data model, all deployments across that enterprise use the same model, ensuring complete internal consistency while preserving flexibility across different organizations.

Path 2: Native Transformation Path

Data can flow through transformers without being forced into the data model. This is not a gap — it is a **deliberate design decision** that enables critical interoperability scenarios that forced conformity would break:

- **Coalition interoperability**: Format bridges between allied systems with different standards
- **Legacy system integration**: Connecting systems that cannot be modified to speak new protocols
- **Exploratory data**: Ingesting new data sources for analysis before committing to model representation
- **High-fidelity passthrough**: Preserving source format when downstream systems require it

Forced conformity to a single model is what creates stovepipes, because it prevents systems from interoperating with partners who use different models. Flexibility is not the enemy of interoperability — it is what enables interoperability in a heterogeneous coalition environment.

Semantic Data Layer

Both paths integrate with SDL's semantic layer through Ontology-Based Data Access (OBDA):

- **Path 1 data**: OBDA maps modeled entities directly to BFO/CCO ontology terms
- **Path 2 data**: OBDA applies virtual ontology projections, interpreting native format fields as ontology concepts without requiring transformation

This means an analyst can issue a single SPARQL query that returns data from both paths — despite different sources and incompatible schemas. The analyst doesn't need to know which path the data took; they query the ontology and OBDA handles the rest.

Learn more about Transformation Pipelines →

Working with Data

DataSources

The SDL Catalog uses DataSource to curate its data connections with the world. DataSources require an Enablement to set up the data connection information (e.g., URL, access key, configuration, etc.). This step is typically done by an admin or a user with an `enablement` role **prior** to a data user interacting with SDL.

DataSets

DataSets in SDL are related to DataSources. DataSets may include: * Streaming topics * Files (Excel,

CSV, Parquet) * Full-motion video (FMV) * Database tables * API endpoints

A DataSet may be local or remote, with appropriate access controls applied.

Capability Areas

Detailed documentation for each SDL capability area:

Capability	Documentation
Event Streaming	Event Streaming — Real-time data streaming backbone
Object Storage	Object Storage — S3-compatible storage with lifecycle policies
Federated SQL	Federated SQL Engine — Distributed query across heterogeneous sources
Virtual Knowledge Graph	RDF & OBDA — Ontology-based data access and SPARQL queries
Operational Monitoring	Operational Monitoring — Platform health and performance metrics
Data Science Notebooks	Data Science Notebooks — Interactive analysis environment
Data Pipelines	Transformation Pipelines — Format conversion and enrichment
Security & Governance	Security — Policy engine, classification, access control
Federation	Federation — Cross-node data exchange and governance

Development Workflows

API Development

- RESTful API standards and patterns
- gRPC service integration
- WebSocket support for real-time data
- API versioning and documentation

Data Pipeline Development

- Transformer development guide
- Testing frameworks for data quality
- Performance optimization techniques
- Error handling and retry strategies

Security Implementation

- Authentication flows with identity provider
- Authorization with policy engine

- Data classification implementation
- Audit logging configuration

Support and Resources

- **User Documentation**: [R]DP User Guide — End-user operational guidance
- **Examples**: [R]DP Examples — Demonstration scripts and use cases
- **API Reference**: Comprehensive API documentation for all services

Next Steps

1. **Review Architecture** — Understand the platform architecture and capability layers
2. **Set Up Development Environment** — Follow installation guides for your target environment
3. **Explore Examples** — Review practical implementations and demo scripts

Platform Architecture

SDL is a layered, modular platform designed for tactical and enterprise data operations. This page provides a high-level architectural overview with pointers to detailed documentation for each capability area.

Architecture Overview

The platform is organized into horizontal layers, each responsible for a distinct set of capabilities. Layers communicate through well-defined interfaces, enabling independent deployment, scaling, and evolution.

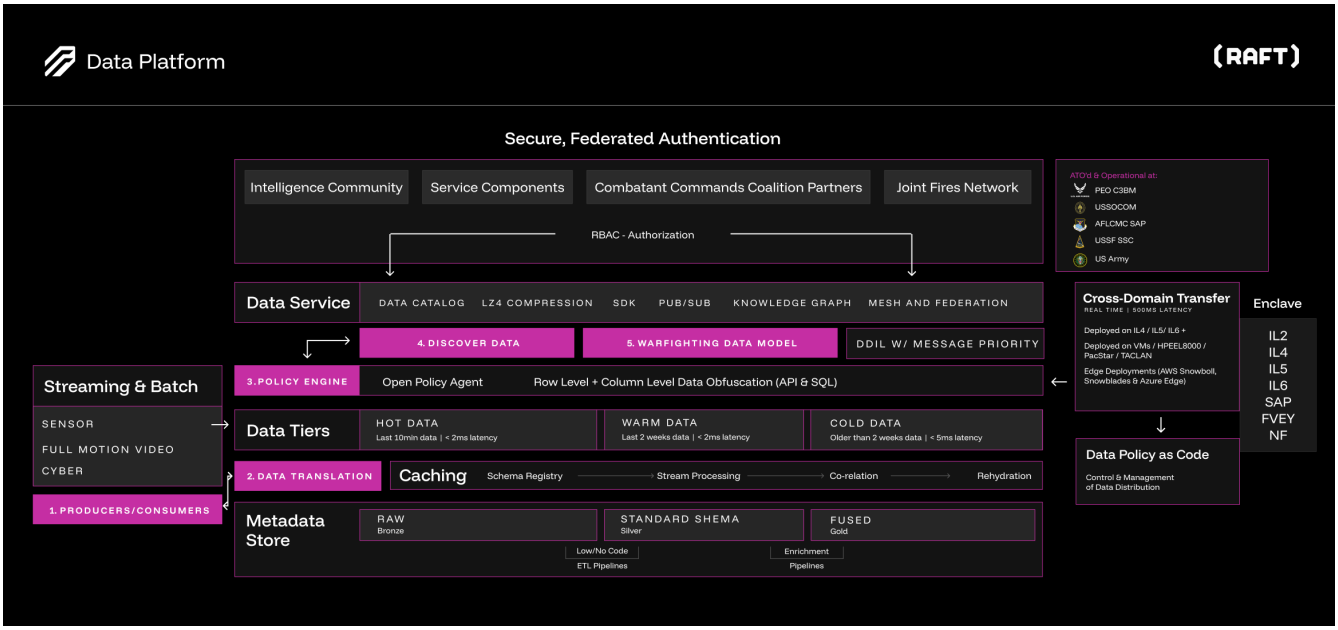


Figure 1. Platform Architecture

Architecture Layers

Ingestion Layer

The ingestion layer connects SDL to external data sources. It supports connectors for event streaming, gRPC, TAK servers, TCP, PostGIS, and other protocols.

The layer is format-agnostic: 14+ tactical data formats (including CoT, AIS, ADS-B, Link-16, GCCS-J, SIGINT, and others) are converted to the canonical data model through configurable transformers. This decouples the platform from source-specific formats, enabling new sources to be added without modifying downstream components.

Processing Layer

The processing layer handles data transformation, enrichment, and schema validation. It provides the following integration patterns:

Native Transformation Path

Data flows through transformers without being forced into the data model. This supports coalition interoperability, legacy system integration, exploratory data ingestion, and high-fidelity passthrough where downstream systems require the source format.

Both paths are processed through the transformation hub, which manages pipeline orchestration, error handling, and monitoring.

Query and Analytics Layer

The query and analytics layer provides multiple access patterns for consuming data:

- **Federated SQL** — A distributed SQL engine that queries across heterogeneous data sources (relational databases, object storage, event streams) through a single SQL interface.
- **Virtual Knowledge Graph (VKG)** — Ontology-Based Data Access (OBDA) maps data to BFO/CCO ontology terms, enabling SPARQL queries across data from both the Data Model Path and Native Transformation Path.
- **Real-time analytics** — Low-latency analytics on streaming data for operational dashboards and alerts.
- **Data visualization** — Interactive dashboards for operational monitoring, data exploration, and reporting.

Storage Layer

The storage layer implements a tiered architecture for cost-effective, performance-optimized data retention:

- **Hot tier** — Low-latency access for active operational data. Recent entities, tasks, and streaming updates.
- **Warm tier** — Balanced latency and cost for recent historical data. Queryable through federated SQL.
- **Cold tier** — Object storage (S3-compatible) for long-term retention and archival.

The metadata lake follows a Bronze-Silver-Gold pattern:

- **Bronze** — Raw ingested data in source format.
- **Silver** — Cleaned, validated, and schema-conformant data.
- **Gold** — Enriched, correlated, and analytics-ready data products.

Security and Governance Layer

Security and governance are enforced across all other layers:

- **Policy engine** — Attribute-based access control (ABAC) with fine-grained policy evaluation.
- **Classification markings** — ISM-format security markings on every entity and task.
- **Obfuscation** — Row-level and column-level data obfuscation based on the requester’s clearance

and need-to-know.

- **Audit logging** — Complete data lineage and access tracking for compliance and after-action review.

API Layer

The API layer provides unified access to all platform capabilities:

- **REST API** — JSON over HTTP for web integrations, scripting, and ad-hoc queries.
- **gRPC API** — High-performance protobuf-based access with bidirectional streaming.
- **SDKs** — Client libraries in Go, Java, and Python with built-in connection management, retry logic, and TLS configuration.

Capability-to-Documentation Mapping

Use the following table to navigate from a capability area to its detailed documentation:

Capability	Documentation
Event Streaming	Event Streaming
Object Storage	Object Storage
Federated SQL	Federated SQL Engine
Operational Monitoring	Operational Monitoring
Data Science Notebooks	JupyterHub
Identity and Access Management	Identity Management
Data Catalog	Data Catalog
Data Pipelines	Transformation Pipelines
GeoServer	GeoServer

Hierarchical Mesh-Federation Architecture

SDL deployments typically follow a hierarchical architecture that mirrors military echelon structures.

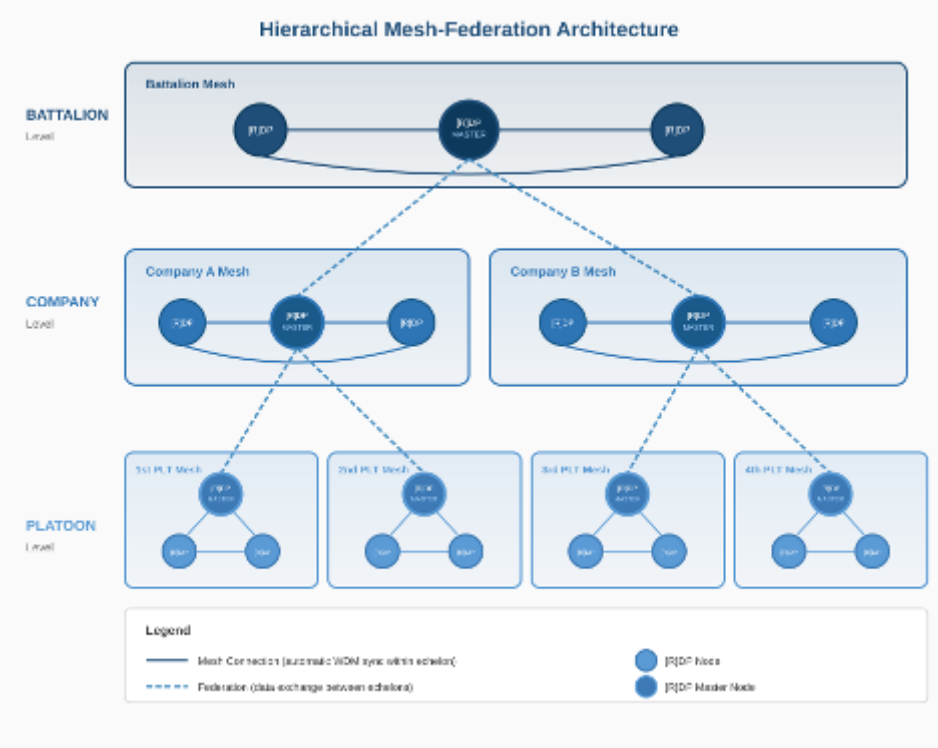


Figure 2. Hierarchical Architecture

Echelon Hierarchy

The architecture supports multi-echelon deployments where each level operates as an autonomous SDL instance:

Platoon

Edge-deployed nodes with constrained compute and bandwidth. Nodes within a platoon sync bidirectionally to maintain a shared operational picture. Platoon nodes push upward to the Company echelon through unidirectional edges.

Company

Aggregates data from multiple platoon nodes. Company-level nodes maintain the combined picture from all subordinate platoons. Company nodes push upward to Battalion through unidirectional edges.

Battalion and Above

Enterprise-level aggregation with full compute and bandwidth capacity. Receives data from all subordinate echelons and provides the widest operational picture. May also push data downward (e.g., tasking, intelligence products) through directed edges.

Mesh Within, Federation Between

Each echelon level runs its own internal mesh:

- **Within an echelon**: Nodes sync bidirectionally, forming a fully connected mesh. Any node's update propagates to all other nodes at the same echelon level.
- **Between echelons**: Directed (typically unidirectional) edges connect echelons. Data flows upward for aggregation and selectively downward for tasking and dissemination.

This architecture provides resilience at each level — if the inter-echelon link goes down, nodes within an echelon continue to operate and sync with each other. When the link recovers, delta sync brings the higher echelon back up to date.

Bidirectional vs. Unidirectional Edges

Edge Type	Usage
Bidirectional	Within an echelon, between peers at the same classification level. Full convergence — both nodes have the same data.
Unidirectional (push up)	From lower echelon to higher echelon. Data flows from edge to enterprise for aggregation.
Unidirectional (push down)	From higher echelon to lower echelon. Tasking, intelligence products, and reference data flow from enterprise to edge.
Cross-domain	Between classification levels. Data flows through cross-domain guards with XSD validation. Typically unidirectional from lower to higher classification.

Related Topics

- Data Pipelines — Transformation and ingestion pipeline details
- Security — Policy engine, ABAC, and classification markings

API

The SDL API is rooted at `https://localhost/api`.

Scalar is also available from the main nav bar in the SDL UI, and documents all the available APIs.

The REST API is divided into three main categories.

- **Fabric Services** (`/api/v1/*`) are platform services hosted/provided by SDL.
- **Proxied Services** (`/api/proxy/*`) are proxies to external services (see api:proxy/index.adoc).
- **Adaptive API** (`/api/adaptive/*`) provides on-the-fly API extensibility via ConfigMaps (see Adaptive API).

See Endpoints for full listing.

OpenAPI

To retrieve the OpenAPI specification for a particular service, you can either download it from the Scalar API reference or you can `curl` for it at the base of the service API.

For example, to download the spec for the `/api/v2/catalog` API:

```
curl https://localhost/api/v2/catalog
```

This will return you the YAML representation of the spec.

Authentication

All SDL REST endpoints support the following authentication mechanisms:

- `Basic` (username/password)

  ```
  curl -H "Authorization: Basic $(echo -n 'user:password' | base64)"
  ```

- `Bearer` tokens (JSON Web Tokens)

  ```
  curl -H "Authorization: Bearer $TOKEN"
  ```

- `OAuth2` single-sign on authorization flows (Scalar only)

In all cases, your credentials must authenticate with the SDL Keycloak (which can federate with other Keycloak instances).

The SDL API also provides test endpoints for testing your connection and credentials.

Endpoints

Endpoint	Description
`/api/id`	General identification of a cluster.
`/api/health`	Get health status of a cluster.
`/api/test`	Test auth credentials and basic connectivity.
Platform Services	
`/api/proxy`	Proxy to external services, with payload caching.
`/api/v1/auth`	Auth utilities (get a token, etc).
`/api/v1/courier`	Drop data into SDL.
`/api/v1/kafka`	Connect to Kafka.
`/api/v1/s3`	S3 object storage.
Data Services	
`/api/v2/catalog`	Data catalog.
`/api/v1/lakehouse`	Data lake APIs.

Swagger

Adaptive API

The Adaptive API Gateway enables on-the-fly API extensibility in airgapped environments. Deploy new APIs via ConfigMaps without rebuilding Docker images.

Overview

The `/api/adaptive` endpoint provides a dynamic API gateway that supports:

- **Auto-generation** of working endpoints from OpenAPI 3.0 specifications
- **Hot-reload** of new APIs via ConfigMaps (no pod restart required)
- **Custom Python handler injection** for business logic
- **Integration** with SDL Swagger UI
- **Full CRUD support** (GET, POST, PUT, DELETE, PATCH)

How It Works

The Adaptive API Gateway uses a two-phase approach:

Phase 1: OpenAPI Spec (Auto-Generated Endpoints)

Create an OpenAPI 3.0 specification and deploy it as a ConfigMap. The gateway automatically generates working endpoints that return example data from the spec.

Use cases:

- API design and prototyping
- Frontend development (mock backend)
- Contract-first development
- API documentation

Phase 2: Custom Python Handlers (Real Business Logic)

Write Python FastAPI handlers and deploy them as ConfigMaps. Custom handlers override auto-generated routes with real business logic.

Use cases:

- Database integration
- External API calls
- Business logic validation
- Production implementations

Authentication

Authentication is handled by the **SDL API Gateway**, not by the Adaptive API itself. The Adaptive API requires sdl-api to be deployed as an upstream dependency.

All requests to `/api/adaptive/*` are routed through the SDL API Gateway, which enforces authentication using:

- **Basic Auth** (`df_basic`): Username/password authentication
- **Bearer JWT** (`df_bearer_jwt`): JWT tokens from Keycloak

```
# Using Basic Auth
curl -u username:password https://localhost/api/adaptive/my-api/endpoint

# Using Bearer Token
curl -H "Authorization: Bearer $TOKEN" https://localhost/api/adaptive/my-api/endpoint
```

Swagger UI Authentication

1. Open https://localhost/api/adaptive/swagger-ui/
2. Click the **Authorize** button (lock icon)
3. Choose authentication method:
   - **df_basic**: Enter username and password
   - **df_bearer_jwt**: Enter JWT token from Keycloak
4. Click **Authorize**

5. All "Try it out" requests will include authentication

The security schemes in Swagger UI are for documentation purposes. Actual authentication enforcement happens at the SDL API Gateway layer.

Direct Access

When accessing the Adaptive API service directly (bypassing the SDL API Gateway), no authentication is enforced. This is intended for:

- Kubernetes health probes
- Internal service-to-service communication
- Development and testing environments

For production deployments, always access through the SDL API Gateway.

Quick Start

See Quick Start for a complete example of deploying a Library API with both auto-generated and custom endpoints.

Gateway Endpoints

Documentation

- `GET /api/adaptive/swagger-ui/` - Interactive Swagger UI documentation
- `GET /api/adaptive/openapi.json` - OpenAPI 3.0 schema

Management

- `GET /api/adaptive/` - Gateway information and endpoint listing
- `GET /api/adaptive/actuator/health/liveness` - Kubernetes liveness probe
- `GET /api/adaptive/actuator/health/readiness` - Kubernetes readiness probe
- `GET /api/adaptive/actuator/info` - Application information (requires auth)
- `POST /api/adaptive/actuator/refresh` - Hot-reload routes and handlers (requires auth)

Custom APIs

All APIs you deploy will be mounted under:

- `/api/adaptive/<api-name>/*` - Your custom API endpoints

Example: If you deploy a `library-api`, endpoints will be at `/api/adaptive/library-api/books`, etc.

Adding New APIs

Option 1: OpenAPI Spec Only (Auto-Generated)

Perfect for API design, prototyping, and frontend development.

1. Create your OpenAPI 3.0 specification
2. Deploy as ConfigMap
3. Gateway auto-generates working endpoints
4. Returns example data from spec

```
# Create ConfigMap with your spec
kubectl get configmap df-adaptive-api-sample-specs -n svraft \
  -o jsonpath='{.data.sample-crud\.openapi\.yaml}' > /tmp/existing-spec.yaml

kubectl create configmap df-adaptive-api-sample-specs \
  --from-file=sample-crud.openapi.yaml=/tmp/existing-spec.yaml \
  --from-file=my-api.openapi.yaml=/tmp/my-api.openapi.yaml \
  -n svraft --dry-run=client -o yaml | kubectl apply -f -

# Restart deployment
kubectl rollout restart deployment df-adaptive-api -n svraft
```

Option 2: Custom Python Handler (Real Logic)

For production implementations with real business logic.

1. Write Python FastAPI handler
2. Deploy as ConfigMap
3. Handlers override auto-generated routes
4. Full access to FastAPI features

```
# my_handler.py
from fastapi import APIRouter, HTTPException

router = APIRouter(tags=["My API"])

@router.get("/items")
async def list_items():
    return {"items": [{"id": 1, "name": "Item 1"}]}

@router.get("/items/{item_id}")
async def get_item(item_id: int):
    if item_id == 1:
        return {"id": 1, "name": "Item 1"}
    raise HTTPException(status_code=404, detail="Not found")
```

```
# Deploy handler
```

```
kubectl create configmap df-adaptive-api-handlers \
  --from-file=my_handler.py=/tmp/my_handler.py \
  -n svraft --dry-run=client -o yaml | kubectl apply -f -

# Restart deployment
kubectl rollout restart deployment df-adaptive-api -n svraft
```

Your API will be available at `/api/adaptive/my` (filename `my_handler.py` → mount path `/my`).

Path Mapping Convention

Handler filenames determine the API mount path:

Handler Filename	Mount Path	Example Endpoint
`library-api_handler.py`	`/library-api`	`/api/adaptive/library-api/books`
`weather_handler.py`	`/weather`	`/api/adaptive/weather/seattle`
`products_handler.py`	`/products`	`/api/adaptive/products/123`

Rules:

- `_handler` suffix is removed
- Underscores (`_`) are converted to hyphens (`-`)
- Routes in handler are relative to mount path

Hot Reload

Reload routes without pod restart:

```
curl -u username:password -X POST https://localhost/api/adaptive/actuator/refresh
```

Returns:

```
{
  "status": "success",
  "message": "Routes reloaded successfully",
  "handlers": ["library-api_handler.py"],
  "specs": ["sample-crud.openapi.yaml"]
}
```

ConfigMap changes require `kubectl rollout restart` to remount updated volumes. Hot-reload only works for changes to already-mounted files.

Configuration

Environment variables (set in Helm values):

Variable	Description	Default
`AUTH_ENABLED`	Enable authentication enforcement	`true`
`KEYCLOAK_URL`	Keycloak server URL	`http://sdl.local/auth`
`KEYCLOAK_REALM`	Keycloak realm name	`data-fabric`
`SPECS_DIR`	Directory containing OpenAPI specs	`/app/specs`
`HANDLERS_DIR`	Directory containing custom handlers	`/app/handlers`
`RELOAD_ON_CHANGE`	Enable hot-reload	`true`

Examples

See Quick Start for a complete walkthrough including:

- Creating OpenAPI specifications
- Deploying to Kubernetes
- Testing auto-generated endpoints
- Adding custom Python handlers
- Authentication

Troubleshooting

Specs Not Loading

Check ConfigMaps and logs:

```
# View specs ConfigMap
kubectl get configmap df-adaptive-api-sample-specs -n svraft -o yaml

# Check logs for loading errors
kubectl logs -l app.kubernetes.io/name=df-adaptive-api -n svraft -c api --tail=100
```

Handler Errors

Check Python syntax and imports:

```
# View handler logs
kubectl logs -l app.kubernetes.io/name=df-adaptive-api -n svraft -c api -f
```

```
# Test handler syntax locally
python -m py_compile my_handler.py
```

Authentication Failures

Verify credentials and check if auth is enabled:

```
# Check if auth is enabled
kubectl get deployment df-adaptive-api -n svraft \
  -o
jsonpath='{.spec.template.spec.containers[0].env[?(@.name=="AUTH_ENABLED")].value}'

# Disable auth for testing
kubectl set env deployment/df-adaptive-api -n svraft AUTH_ENABLED=false
kubectl rollout restart deployment df-adaptive-api -n svraft
```

Route Conflicts (Spec vs Handler)

If both an OpenAPI spec and custom handler define the same routes, the spec-generated route may take precedence.

Solution: Remove the OpenAPI spec from the ConfigMap when deploying a custom handler:

```
# Keep only other specs, remove conflicting one
kubectl get configmap df-adaptive-api-sample-specs -n svraft \
  -o jsonpath='{.data.sample-crud\.openapi\.yaml}' > /tmp/sample-crud.yaml

kubectl create configmap df-adaptive-api-sample-specs \
  --from-file=sample-crud.openapi.yaml=/tmp/sample-crud.yaml \
  -n svraft --dry-run=client -o yaml | kubectl apply -f -

kubectl rollout restart deployment df-adaptive-api -n svraft
```

Quick Start

This guide walks through deploying a complete Library API using the Adaptive API Gateway.

Overview

We'll build a Library API that manages books and borrowing operations through six steps:

1. Create OpenAPI specification
2. Deploy spec to Kubernetes
3. Test auto-generated endpoints
4. Create custom Python handler
5. Deploy handler to Kubernetes

6. Test custom endpoints with real business logic

Step 1: Create OpenAPI Spec

Create an OpenAPI 3.0 specification defining your API:

```
cat > /tmp/library-api.openapi.yaml <<'EOF'
openapi: 3.0.3
info:
  title: Library API
  description: API for managing library books and borrowing operations
  version: 1.0.0

servers:
  - url: /api/adaptive/library-api
    description: SDL environment

tags:
  - name: Books
    description: Book catalog management

paths:
  /books:
    get:
      tags:
        - Books
      summary: List all books
      operationId: listBooks
      parameters:
        - name: available
          in: query
          required: false
          schema:
            type: boolean
      responses:
        '200':
          description: Successful response
          content:
            application/json:
              example:
                books:
                  - id: 1
                    title: "The Python Handbook"
                    author: "Jane Developer"
                    isbn: "978-1234567890"
                    available: true
                count: 1

    post:
      tags:
```

```
        - Books
      summary: Add a new book
      operationId: createBook
      requestBody:
        required: true
        content:
          application/json:
            example:
              title: "New Book"
              author: "Author Name"
              isbn: "978-1234567890"
      responses:
        '201':
          description: Book created successfully
          content:
            application/json:
              example:
                id: 2
                title: "New Book"
                author: "Author Name"
                isbn: "978-1234567890"
                available: true

  /books/{bookId}:
    get:
      tags:
        - Books
      summary: Get book by ID
      operationId: getBook
      parameters:
        - name: bookId
          in: path
          required: true
          schema:
            type: integer
      responses:
        '200':
          description: Successful response
          content:
            application/json:
              example:
                id: 1
                title: "The Python Handbook"
                author: "Jane Developer"
                isbn: "978-1234567890"
                available: true
        '404':
          description: Book not found
EOF
```

Step 2: Deploy OpenAPI Spec

Add your spec to the ConfigMap and deploy:

```
# Get existing specs to preserve them
kubectl get configmap df-adaptive-api-sample-specs -n svraft \
  -o jsonpath='{.data.sample-crud\.openapi\.yaml}' > /tmp/sample-crud.openapi.yaml

# Update ConfigMap with both specs
kubectl create configmap df-adaptive-api-sample-specs \
  --from-file=sample-crud.openapi.yaml=/tmp/sample-crud.openapi.yaml \
  --from-file=library-api.openapi.yaml=/tmp/library-api.openapi.yaml \
  -n svraft --dry-run=client -o yaml | kubectl apply -f -

# Restart deployment
kubectl rollout restart deployment df-adaptive-api -n svraft

# Wait for pod to be ready
kubectl wait --for=condition=ready pod -l app=df-adaptive-api -n svraft --timeout=60s
```

Step 3: Test Auto-Generated Endpoints

Test the endpoints that auto-generate responses from the OpenAPI spec:

```
# List all books (returns example data from spec)
curl -u username:password https://localhost/api/adaptive/library-api/books

# Get a specific book
curl -u username:password https://localhost/api/adaptive/library-api/books/1

# Create a book (echoes request with example data)
curl -u username:password -X POST https://localhost/api/adaptive/library-api/books \
  -H "Content-Type: application/json" \
  -d '{"title":"New Book","author":"Author Name","isbn":"978-1234567890"}'
```

```
{
  "books": [
    {
      "id": 1,
      "title": "The Python Handbook",
      "author": "Jane Developer",
      "isbn": "978-1234567890",
      "available": true,
      "publishedYear": 2024,
      "addedAt": "2024-01-15T10:00:00Z"
    },
    {
      "id": 2,
```

```
      "title": "FastAPI Patterns",
      "author": "John Coder",
      "isbn": "978-0987654321",
      "available": true,
      "publishedYear": 2023,
      "addedAt": "2023-06-20T14:30:00Z"
    }
  ],
  "count": 2,
  "timestamp": "2025-11-26T14:00:00Z"
}
```

At this stage, all endpoints return the example data defined in your OpenAPI spec. No Python code required yet.

Step 4: Create Custom Handler

Write a Python FastAPI handler with real business logic:

```
cat > /tmp/library-api_handler.py <<'EOF'
"""
Library API Custom Handler - Real business logic implementation
"""
from fastapi import APIRouter, HTTPException, Query, Request
from datetime import datetime
from typing import Optional, Dict

router = APIRouter(tags=["Library API"])

# In-memory storage (replace with database in production)
books_db: Dict[int, dict] = {
    1: {
        "id": 1,
        "title": "The Python Handbook",
        "author": "Jane Developer",
        "isbn": "978-1234567890",
        "available": True,
        "publishedYear": 2024
    }
}
next_book_id = 2

@router.get("/books")
async def list_books(
    available: Optional[bool] = Query(None)
):
    """List all books with optional filtering"""
    books = list(books_db.values())

    if available is not None:
```

```
        books = [b for b in books if b["available"] == available]

    return {"books": books, "count": len(books)}

@router.post("/books")
async def create_book(request: Request):
    """Add a new book to the catalog"""
    global next_book_id
    body = await request.json()

    if "title" not in body or "author" not in body:
        raise HTTPException(status_code=400, detail="Title and author required")

    new_book = {
        "id": next_book_id,
        "title": body["title"],
        "author": body["author"],
        "isbn": body.get("isbn", f"AUTO-{next_book_id}"),
        "available": True,
        "publishedYear": body.get("publishedYear", datetime.utcnow().year)
    }

    books_db[next_book_id] = new_book
    next_book_id += 1

    return new_book

@router.get("/books/{bookId}")
async def get_book(bookId: int):
    """Get a specific book by ID"""
    if bookId not in books_db:
        raise HTTPException(status_code=404, detail=f"Book {bookId} not found")

    return books_db[bookId]
EOF
```

Step 5: Deploy Custom Handler

Before deploying the handler, remove the OpenAPI spec to avoid route conflicts:

```
# Remove library-api spec (keep sample-crud)
kubectl get configmap df-adaptive-api-sample-specs -n svraft \
  -o jsonpath='{.data.sample-crud\.openapi\.yaml}' > /tmp/sample-crud.openapi.yaml

kubectl create configmap df-adaptive-api-sample-specs \
  --from-file=sample-crud.openapi.yaml=/tmp/sample-crud.openapi.yaml \
  -n svraft --dry-run=client -o yaml | kubectl apply -f -

# Deploy custom handler
kubectl create configmap df-adaptive-api-handlers \
```

```
  --from-file=library-api_handler.py=/tmp/library-api_handler.py \
  -n svraft --dry-run=client -o yaml | kubectl apply -f -

# Restart deployment
kubectl rollout restart deployment df-adaptive-api -n svraft

# Wait for pod to be ready
kubectl wait --for=condition=ready pod -l app=df-adaptive-api -n svraft --timeout=60s
```

Step 6: Test Custom Endpoints

Test the real business logic:

```
# List books (returns real data from in-memory storage)
curl -u username:password https://localhost/api/adaptive/library-api/books

# Add a new book (actually creates it)
curl -u username:password -X POST https://localhost/api/adaptive/library-api/books \
  -H "Content-Type: application/json" \
  -d '{"title":"Docker Deep Dive","author":"Mike Container","isbn":"978-5566778899"}'

# Verify it was added
curl -u username:password https://localhost/api/adaptive/library-api/books

# Get the new book by ID
curl -u username:password https://localhost/api/adaptive/library-api/books/2
```

The handler now maintains real state with business logic validation.

Advanced Topics

Multiple Handlers

You can deploy multiple handlers in the same ConfigMap:

```
kubectl create configmap df-adaptive-api-handlers \
  --from-file=library-api_handler.py=/tmp/library-api_handler.py \
  --from-file=weather_handler.py=/tmp/weather_handler.py \
  --from-file=products_handler.py=/tmp/products_handler.py \
  -n svraft --dry-run=client -o yaml | kubectl apply -f -
```

Each handler will be mounted at its own path.

Custom Endpoints Not in Spec

Handlers can add endpoints not defined in the OpenAPI spec:

```
@router.get("/stats")
```

```
async def get_stats():
    """Custom endpoint - not in OpenAPI spec"""
    return {"total_books": len(books_db)}
```

These will automatically appear in the Swagger UI.

Database Integration

For production use, replace in-memory storage with database access:

```
import psycopg2
from os import getenv

DB_HOST = getenv("DB_HOST", "df-backend-postgresql")
DB_NAME = getenv("DB_NAME", "datafabric")

@router.get("/books")
async def list_books():
    conn = psycopg2.connect(host=DB_HOST, database=DB_NAME, ...)
    cursor = conn.cursor()
    cursor.execute("SELECT * FROM books")
    # ... process results
```

Additional Resources

- Application code: https://github.com/raft-tech/rdp/tree/main/sdl-adaptive-api
- Helm chart: `~/rdp/charts/sdl-adaptive-api`
- Complete example: See `EXAMPLE.md` in application repository

Scalar

Scalar renders each of the OpenAPI specifications with an interactive API reference, and enables you to invoke the services directly from the browser.

This is helpful for testing out endpoints, or quickly seeing what's available.

Accessing Scalar

Scalar is available from the main nav bar in the SDL UI under the **API Docs** dropdown.

Each service has its own OpenAPI specification displayed through Scalar.

Authorization

Before you can invoke any of the endpoints, you must configure your credentials. Click the **"Authentication"** section in the left sidebar to configure one of the supported methods:

- `OAuth2` - OAuth2 single-sign on.

- `Basic` - Basic authorization with username/password credentials.
- `Bearer` - Bearer authorization with a JSON Web Token (JWT).

Only one method is required.

If you supply **Basic** credentials, SDL will use those credentials to retrieve a JWT from Keycloak and use that JWT for all downstream authorizations.

If you supply a **JWT**, it will be passed down as is.

Trying an Endpoint

Once you have configured authentication, you can invoke endpoints by:

1. Select an endpoint from the left sidebar to view its details.
2. Review the **Parameters** section for available parameters and request body (if applicable).
3. Review the **Responses** section for documented response types.
4. Click **"Send"** to invoke the endpoint.
5. The response section will display the status code, headers, and body that came back.

Identification

`/api/id`

The `/api/id` endpoint provides some general identity information of the running instance.

```
GET /api/id
```


```
{
  "classification": "unclassified",
  "name": "SOF Data Layer",
  "version": "1.17.65"
}
```


Health

`/api/health`

The `/api/health` endpoint provides a live health check of the cluster, executing a new health check every time it is invoked.

```
GET /api/health
```

```
{
  "status": "UP",
  "components": {
    "gateway": {
      "status": "UP",
      "components": {
        "auth": {
          "status": "UP"
        },
        "hello": {
          "status": "UP"
        },
        "openapi": {
          "status": "UP"
        }
      }
    },
    "kafka": {
      "status": "UP",
      "components": {
        "connection": {
          "status": "UP"
        }
      }
    },
    "minio": {
      "status": "UP",
      "components": {
        "connection": {
          "status": "UP"
        }
      }
    },
    "v1audit": {
      "status": "UP",
      "components": {
        "eventcount": {
          "status": "UP"
        },
        "openapi": {
          "status": "UP"
        }
      }
    },
    "v1courier": {
      "status": "UP",
      "components": {
        "listinboxes": {
          "status": "UP"
        },
```

```
        "openapi": {
          "status": "UP"
        }
      }
    },
    "v1kafka": {
      "status": "UP",
      "components": {
        "config": {
          "status": "UP"
        },
        "openapi": {
          "status": "UP"
        }
      }
    },
    "v1lakehouse": {
      "status": "UP",
      "components": {
        "openapi": {
          "status": "UP"
        },
        "schemas": {
          "status": "UP"
        }
      }
    },
    "v1registry": {
      "status": "UP",
      "components": {
        "openapi": {
          "status": "UP"
        },
        "schemas": {
          "status": "UP"
        },
        "version": {
          "status": "UP"
        }
      }
    },
    "v1s3": {
      "status": "UP",
      "components": {
        "countbuckets": {
          "status": "UP"
        },
        "openapi": {
          "status": "UP"
        }
      }
```

```
    },
    "v2catalog": {
      "status": "UP",
      "components": {
        "datasources": {
          "status": "UP"
        },
        "openapi": {
          "status": "UP"
        }
      }
    }
  },
  "groups": [
    "api",
    "liveness",
    "readiness"
  ]
}
```

You can also get the health from just a group of components. Below is an example of the health of just Kafka.

```
GET /api/health/kafka
```

```
{
  "status": "UP",
  "components": {
    "connection": {
      "status": "UP"
    }
  },
  "groups": [
    "api",
    "liveness",
    "readiness"
  ]
}
```

Test

`/api/test/hello`

The `/api/test/hello` endpoint is an unauthenticated endpoint that lets you test basic connectivity.

```
GET /api/test/hello
```

```
{
  "msg": "Hello SDL client!!"
}
```

/api/test/auth

The `/api/test/auth` endpoint can verify your auth credentials.

```
GET /api/test/auth \
  -H 'Authorization: Basic YWRtaW46VmpadlZSMXV0ZW15WFhmZ1JkaUJuMG9G'
```

If successful, your authenticated principal is returned.

```
{
  "principal": "af24a6cb-08d4-4f54-864d-177700b43316"
}
```

If unsuccessful, a `401 Unauthorized` is returned.

```
HTTP/1.1 401 Unauthorized
```

Data Pipelines

Introduction

SDL Data Pipelines provide capabilities to design, configure, deploy, and manage complex data processing jobs without being restricted to underlying tools like Kubernetes.

The user documentation contains help sections for authoring Transformers and deploying them to **SDL**; the developer documentation provides more information on the underlying capabilities and concepts, major architectural components, and design decisions.

High-Level Overview

Raft built the Data Pipelines system to address several pain points. Design will go into more detail, but at a glance:

1. Deploying and managing ETL jobs is difficult to do; many ETL platforms are not optimized for Kubernetes and require additional management.
2. In many enterprise systems, the question of "what data pipelines are running?" is non-trivial to answer. Even more difficult is the question of "what steps comprise a data pipeline?"
3. Many ETL systems are sticky; it's difficult to maintain a system that uses more than one tool (ex. Flink, Spark, NiFi, Airflow, etc.).

There are three main concepts that are used:

1. **Dataset**: A logical grouping of data in **SDL**. For example, a Kafka topic, a bucket or key prefix on S3, or a table in PostgreSQL.
2. **Transformer**: Any process that acts upon a Dataset (or another Transformer), producing zero or more Datasets as output.
3. **Pipeline**: An undirected, possibly cyclic graph that describes a series of Transformers and their connections to Datasets.

Additionally, there are two other terms that are important:

1. **Transformer Template** (also called a **Template**): A record of a **Transformer** that is registered with the Pipeline Engine so that it can be used in Pipelines.
2. **Pipeline Template**: A pre-defined scaffold of a Pipeline with options for what Transformers can be used at each step, simplifying the process of creating commonly-needed Pipelines.

Native Transformation Path

Transformers can also convert between arbitrary formats without forcing conformity to a data model:

- **Format Bridges**: Convert between coalition partner formats

- **Legacy Adapters**: Translate protocols for systems that cannot be modified
- **High-Fidelity Passthrough**: Preserve source format when downstream systems require it

Path 2 enables interoperability scenarios that forced conformity would break.

Example: Format Bridge Pipeline

```
AXS --> AXS-to-AFATDS Transformer --> System B (Original AXS transformed to AFATDS for
Multinational Operational Environments)
```

.1. Access Control

By default, every Transformer and Pipeline has access controls applied to it. As with other components in **SDL**, access control decisions are made by OPA according to a policy that is customizable for a particular use case, rather than being baked into the code directly. The policy controlling the Data Pipeline Engine is located here.

Because all access control decisions are done via OPA, the associated policy is what controls the semantics described below. Any discussion of access controls in this document uses the default policy above as reference. A team wanting to extend or modify the access control paradigm for Pipelines and Transformers should generally start with the builtin policy and modify it as needed, to maintain existing functionality to the extent possible.

Overview

Each Pipeline and Transformer is annotated with three fields related to access control:

1. `created_by`: This is filled in by the engine when a Pipeline or Transformer is created, according to the identity associated with the JWT used to authenticate with the API. By default, the entity referred to by `created_by` has both read and write permissions to the resource.
2. `access_controls`: This is an array of complex objects whose structure is defined below. Essentially, this allows any entity with write access to share the resource with another user or group, as defined in Keycloak.
3. `security_markings`: This is a free-text field intended to contain security markings for the resource (for example, classification level). In order to maintain generality across different use cases, the pipeline engine does not perform any validation on this field, allowing entities to write anything into it. Additionally, the default OPA policy does not perform any filtering by this field. This is intended to be modified by an individual use case to allow mandatory access controls beyond what exist as attributes and groups for an entity in Keycloak. For example, an OPA policy could be written to take advantage of a classification parser to perform ACCM filtering on a pipeline or transformer automatically.

The combination of these three fields is intended to support various access control scenarios, including:

- Basic access control, which restricts visibility of pipelines and transformers to the user that

created them.

- Sharing, where a user grants another user or group read or read+write permissions to a Pipeline or Transformer.
- Creation on-behalf-of, where a service creates a Pipeline or Transformer but makes it visible (potentially read-only) to a user or group.
- Mandatory access control, where the security markings applied to a Pipeline or Transformer restrict its visibility beyond the attributes of a user or their membership in a particular group.

Data Model

Entity URNs

Access control information is tied to users and/or groups as defined in Keycloak (or whatever OIDC provider is being used). The way this information is encoded in the Pipeline Engine is via a URN namespaced to the specific type. Three formats are allowed:

- `urn:rdp:keycloak-user:<user>`: The most common. This should map to the `preferred_username` of a user as referenced in Keycloak.
- `urn:rdp:keycloak-group:<group>`: A group as referenced in Keycloak. Note that it is assumed that all users and groups are in the same realm.
- `urn:rdp:keycloak-svc-acct`: Currently defined but unsupported. The intent would be to allow service accounts to create Pipelines or Transformers on behalf of users.

While it is technically possible to create Keycloak Groups with special characters in their name (like spaces), it is not recommended if they will be used with the Data Pipelines.

`access_controls`

By default, a pipeline or transformer is created with no `access_controls`; only the user who created it is able to view and modify it. Additionally, when creating or updating a Pipeline or Transformer, a user can add a set of additional permissions, like the format below:

```
[
  {
    "entity": "urn:rdp:keycloak-user:user-a",
    "read": true,
    "write": true
  },
  {
    "entity": "urn:rdp:keycloak-group:Group-A",
    "read": true,
    "write": false
  }
]
```

The format should be self-documenting. The result of these access controls would be that the given entity (Pipeline or Transformer) would be:

- Readable by both `urn:rdp:keycloak-user:user-a` and `urn:rdp:keycloak-group:Group-A` (i.e. GET and LIST requests will return the entity)
- Writable by `urn:rdp:keycloak-user:user-a`. Note that this would allow `user-a` to delete the Transformer.

Even if a Pipeline or Transformer grants write permissions to another user or group, the original `created_by` remains, even if it is further updated.

.2. Monitoring

For more information on the design decisions behind Pipeline monitoring, see Design Documents - Monitoring

Overview

Monitoring Data Pipelines is intended to answer the following questions:

1. For a given Pipeline, what is its top-level status? Is it running, completed, or in a degraded state?
2. For a given Pipeline, what are the individual statuses of each Transformer in it? Are they running, completed, or in a degraded state?
3. Why did a Pipeline's status transition from one state to another?
4. When did a Pipeline's status transition?

This document will outline the ways that status is reported for a Pipeline, intended for developers working on **SDL** Data Pipelines.

Where Status is Reported

The source of truth for the pipeline CR comes from the operator code.

Pipelines are instantiated via the `Pipeline` Kubernetes CR, which tracks **both** the spec for the Pipeline (i.e. how each resident Transformer should be instantiated) as well as the status of the Pipeline (via the Status subresource). There are three top-level fields denoting status:

- `currentStatus`: A top-level indication of the current status of the Pipeline. Options are `Inactive`, `Active`, `Finished`, or `Invalid`.
- `conditions`: An array of Kubernetes Conditions referring to the object. Note that Kubernetes Conditions serve a very specific role and may not be what you expect.
- `stages`: Individual statuses for each Transformer in this Pipeline.

See Pipeline Status for more information

A Transformer status has the following structure:

- `status`: The status of this individual Transformer.
- `message`: A human-readable message referring to the status of this Transformer.
- `started`: When this Transformer started (if at all)
- `stopped`: When this Transformer stopped (if at all)

See Transformer Status for more information.

Pipeline Status

A Pipeline's `currentStatus` may be one of a few values:

- `Active`: The Pipeline is currently active without errors. Note that in the case of periodic tasks (i.e. cron jobs), this is higher-level than simply "are all stages running?"
- `Finished`: All stages in the Pipeline have finished.
- `Degraded`: One or more Pipeline stages have failed, though all were instantiated successfully..
- `Invalid`: One or more Pipeline stages failed to be instantiated, which caused a rollback.

Transformer Status

A Transformer's `status` may be one of a few values:

- `Unknown`: The controller failed to determine the current state of the stage.
- `FailedToInstantiate`: The stage failed to be instantiated.
- `Active`: The stage is currently running.
- `Complete`: All work for this stage has completed.
- `Degraded`: One or more operations in this stage have failed. Note that this doesn't mean ALL have failed.

.3. Writing Your First Transformer

Follow these steps to develop and deploy your first **SDL** transformer.

1. Set up your environment

Make sure your development environment is fully configured - this includes installing all necessary tools, dependencies, and starting your local cluster.

Refer to Tools and Environment for a step-by-step guide.

2. Learn about pipelines and transformers

Understand the core concepts behind transformers and their purpose in the pipeline engine.

Pages in the `user` docs site have more detailed information on:

- What a transformer is

- Required input/output contracts
- How to write a valid `transformer.json` file

3. Write the Transformer Code

The sdl-transformer repository contains the majority of SDL's built-in transformations and is the best place to look for examples. Connectors handle I/O (Kafka, stdin/stdout, TAK servers, PostGIS, Foundry, Lattice, etc.) while transformers handle format conversion.

All configuration is done via environment variables — see the sdl-transformer README for the full list of supported formats, connectors, and configuration options.

If sdl-transformer does not fit your use case (e.x. needing to write the transformer in a different programming language), you can write a standalone transformer. There are examples of this in the df-pipeline-transformers repository.

4. Write a Makefile and build your Docker image

If you added your transformation sdl-transformer, use the `make kind` command to update the image in the cluster.

If you created your own transformer, you should create a Makefile to automate building your Docker image and loading it into your local cluster.

You can find examples in existing **SDL** pipeline repositories for reference.

Your Makefile should include targets similar to the following:

```
docker:
    docker build -t ghcr.io/raft-tech/<image-name>:dev .

kind:
    kind load docker-image ghcr.io/raft-tech/<image-name>:dev --name <cluster-
namespace>
```

Replace `image-name`, and `cluster-namespace` with the appropriate information.

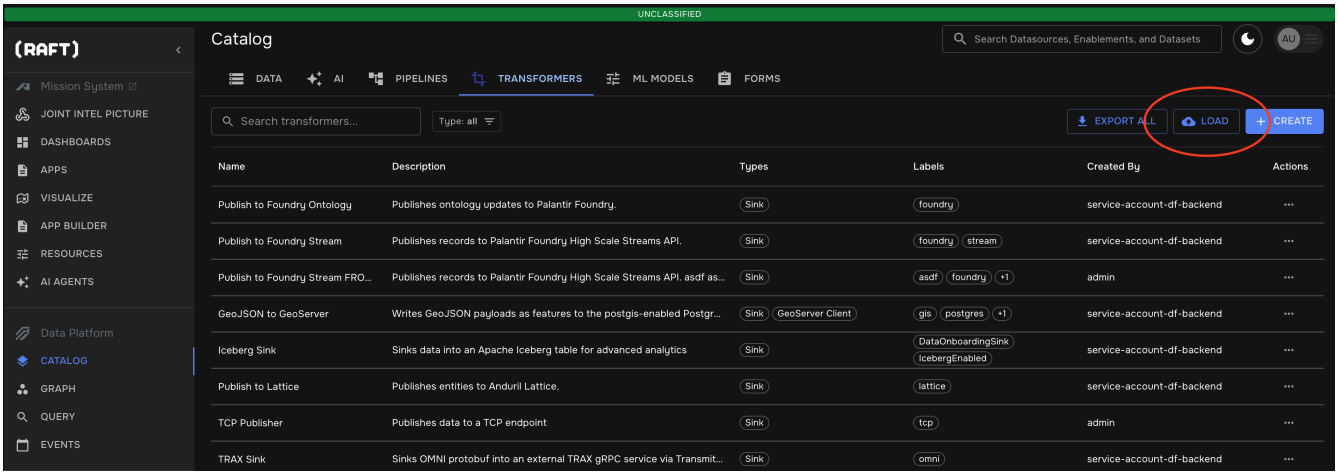
You can then run:

```
make docker   # Build the Docker image
make kind     # Load the image into your cluster
```

5. Register your transformer

Navigate to the "Transformers" tab on the "Catalog" page to add your transformer in the system. This step applies the `transformer.json` configuration to the cluster.

Make sure the image referenced in your `transformer.json` matches the image you built and loaded into the cluster in step 4.



If you are updating an existing transformer, you can click on that transformer to edit its configuration on the same page.

6. Run your pipeline

With everything deployed:

- Apply any required input/output datasets in the UI.

Your pipeline should now be ready to process data through your transformer.

.4. Design Documents

.4.1. System

Terms

- **Data Tool** - Any software component, usually open-source, used for processing, storing, or recording data. Examples may include Kafka, Flink, Spark, Beam, NiFi, Arroyo, Airflow, or others.
- **Dataset** - A set of connected data objects, intended to be used/processed as a unit. A dataset may be bounded or unbounded.
- **Pipeline** - A series of operations performed on a dataset. A pipeline may be cyclic or acyclic and may connect to other data pipelines.
- **Stage** - A single operation contained in a data pipeline.

Rationale

The Data Pipeline Engine (DPE) was conceived from a set of pain points encountered by Raft over years of working with different organizations. Those pain points will be covered later, but to start with, a number of functional and non-functional requirements were established to guide development. They are:

1. The solution should be deployed closely within **SDL**, utilizing as much of the current stack as possible. This should include things like:
    - Existing infrastructure like Kafka, MinIO, etc.
    - Existing **SDL** core concepts like Datasets and Datasources.
    - Kubernetes primitives, potentially including Kubernetes Operator-like capacities
2. The solution should provide a pathway to integration with the **SDL** Data Catalog for 'closed-loop' data pipelines that contribute new datasets to the catalog.

Multiple data tools may be involved in projects

This may not be true for small, very focused, or greenfield projects, where technology selection is limited to a small set of engineers. However, as new engineers onboard to a project, new user requirements come up that weren't accounted for in the original design, or technological best practices evolve, this ceases to be the case. It is assumed that nearly all data-heavy, successful projects evolve to the point where multiple data tools are used for processing.

The proliferation of data tools in recent years is with good reason. At sufficient scales of data or system complexity (many times at lower thresholds than you may think) minor design decisions between different data tools become apparent-Kafka Streams may be preferable to Airflow, or Flink over Beam, etc. Data engineers should then understand to operate in these conditions.

In this light, in designing the technical stack for a new system, architects have three options for visualizing, managing, and deploying data pipelines:

1. Mandate that only a single data tool is used. It should be apparent that this option has signficant drawbacks, including being tethered to a single data tool/community and the ability of that data tool to encompass all current and future use cases.
2. Accept (implicitly or explicitly) that multiple data tools will be used, but don't plan for a way to deal with that consequence. Projects architected with this option are typically characterized by disorganized or fragemented deployment processes, delays in implementing conceptually simple changes that affect multiple tools, and a complete inability to visualize or manage these complex data flows without manual methods (ex. hand-drawn architecture diagrams).
3. Explicitly accept this eventuality, but plan for the consequences, including how to deploy, visualize, monitor, and update data pipelines encoded in multiple tools.

Pain Points

The following pain points inform much of the core functionality of **SDL** Data Pipelines.

Visualizing data pipelines is hard

In terms of pipeline complexity, pre-MVCR 1 (2023) CBC2 wasn't hugely complicated. The team was managing essentially three data pipelines, each with between 2 and 5 stages. All data were streamed in via Kafka and a subset were persisted to Postgres for retrieval via a REST API. Being Kafka-heavy, most applications were written as Kubernetes `Deployment`s, some utilizing Kafka Streams.

Even with only a single data tool being used (Kafka/KStreams), visualizing the pipelines was

extremely difficult. Most of the time, deciphering what a particular pipeline was doing required examining each set of `Deployment` objects to figure out the interconnections (Kafka Topics). This worked well enough for engineers, but anyone non-technical (or even engineers not well-versed in the system) was in the dark. The team ended up keeping an architecture diagram to track individual data flows, complete with Kafka Topics and boxes representing pipeline stages, to show internal and external stakeholders the system.

This wasn't the best solution, for several reasons:

1. Architecture diagrams, as they usually live in dedicated graphical tools outside the codebase, tend to stay slightly (or majorly) out-of-date. Updates to the data pipelines had to be intentionally reflected in the diagram.
2. Knowledge of **how** to create the architecture diagram was still maintained by engineers. Non-technical or outside parties may have a snapshot of the data architecture **at a point in time**, but that had limited usefulness for anything besides record-keeping.
3. Manually creating and updating diagrams can be time-intensive and requires broad system knowledge to get right. Though understanding the system broadly is probably a good thing, spending engineering time on architecture diagrams should be minimized to a reasonable extent.

When a project utilizes a single data tools, there is sometimes an open-source or proprietary visualization tool, which can help ease this pain point somewhat. However, in many cases that tool doesn't exist, and in the event of another data tool being added, the benefits are significantly lessened.

Lesson Learned

Given that data pipelines are often heterogeneous with respect to data tools, they should be visualized **outside** the tools themselves. In other words, visualization of a data pipeline **should not** rely on features inside the data tools themselves.

This is a different direction from tools like Apache NiFi, which ship with built-in UIs. However, we see the benefits of those built-in UIs quickly deteriorate as pipelines are built outside of those tools.

Data pipelines are hard to deploy

This problem gets exponentially more difficult as more data tools are used in a given project, but still exists in the case of a single tool. On CBC2, pipelines were deployed as Helm charts, which has benefits (easy to templatize deployments between environments) and drawbacks (somewhat difficult to thread pipelines together, lots of modifications if stages added/removed).

Many popular data tools like Airflow, NiFi, and Arroyo bake deployment into their tools, either via UIs or their own deployment languages (like Python for Airflow).

.4.2. Monitoring

Overview

One crucial aspect of deploying a data pipeline is knowing what's happening with it. At a minimum,

we need to know whether it:

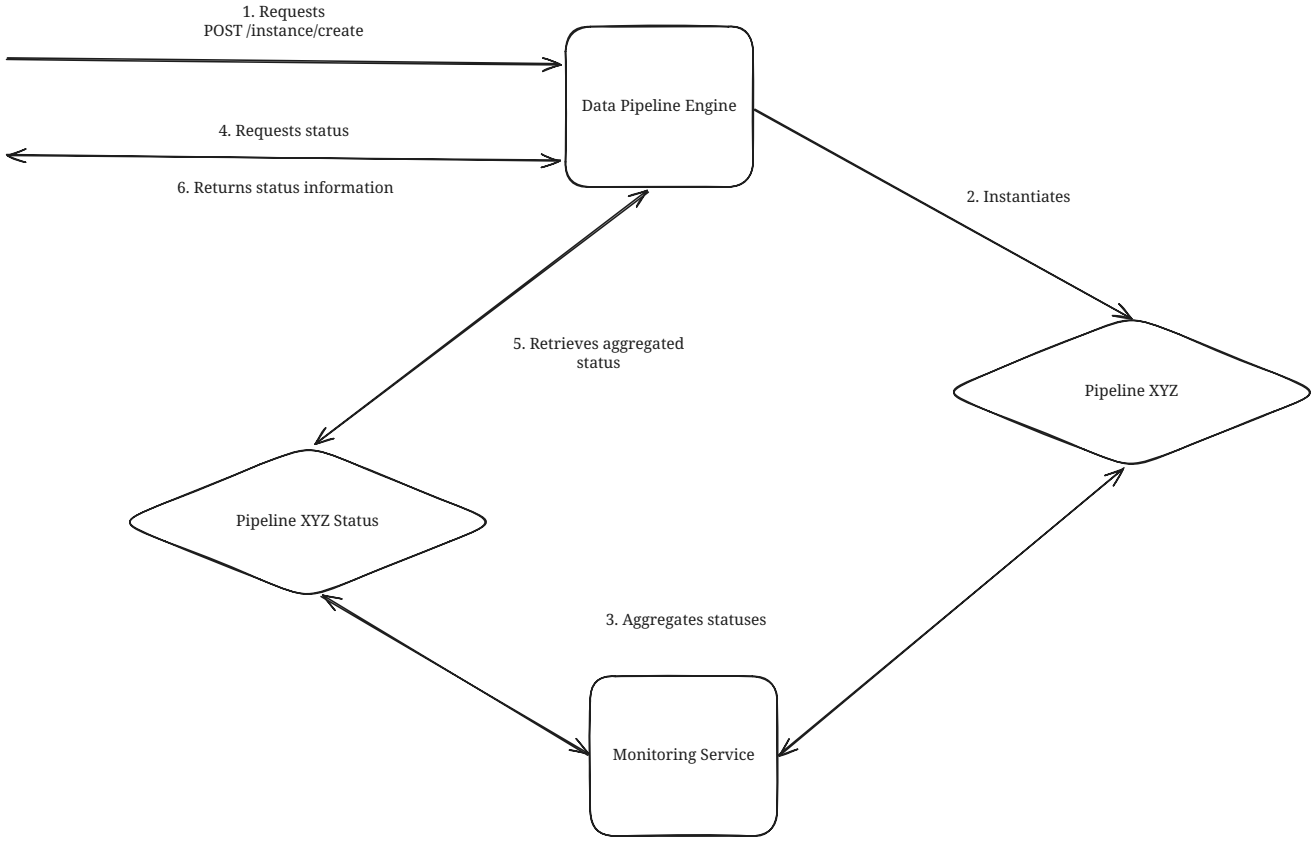
1. Is still running without error
2. Has completed (in the case of a discrete job)
3. Is in a degraded state

There are plenty of systems that do similar things - Argo Workflows is a really great tool - so we can take inspiration there.

Requirements

1. The system should report Pipeline-level status:
    - **Inactive** - not started
    - **Active** - currently running
    - **Degraded** - one or more Transformers in the Pipeline are failing
    - **Completed** - all Transformers in the Pipeline have finished
2. The system should report detailed statuses for each Transform in a Pipeline:
    - **Active** - currently running
    - **Degraded** - the Transformers has failed (and possibly restarted)
    - **Completed** - the Transformer has finished
3. The system should report these things asynchronously, i.e the pipeline API shouldn't need to aggregate Pipeline- and Transformer-level status every time a request comes in.
4. The types of statuses should be abstracted **above** a given instantiation type. For example, we can't expect that every Transformer is instantiated as a Kubernetes `Job`.
5. A Pipeline may be instantiated in multiple underlying ways, but status information should be reported in a single place. In other words, the monitoring system should provide an abstraction over multiple instantiation types.

Abstract Solution



This diagram shows the logical flow between the Data Pipeline Engine and the hypothetical monitoring service. Keeping the creation of the `Pipeline XYZ Status` separate from the REST request in Step #4 allows the flow to be asynchronous from the user's perspective - status information is always generally available without scraping the pipeline objects.

Since monitoring affects the life cycle of a Pipeline, it's useful to break it down into parts:

1. Pipeline submission/validation
2. Pipeline instantiation
3. Pipeline status updates
4. Pipeline removal

As the alternatives will go through, points #2 and #3 can either be served by separate services or together, which has consequences for scalability and reliability.

Alternatives

There were a few different alternatives explored for how this functionality could be implemented. As a litmus test, the following scenario should perform somewhat well:

- 100 users
- 10 pipelines/user
- 5 transformers/pipeline

1. Monitor directly in the Pipeline Engine

This is the simplest option, as it doesn't require any additional components to be managed. The Pipeline Engine simply retrieves status information at request-time by query each Transformer in a Pipeline, then aggregates it to form a top-level object.

Pros:

- Simplistic deployment; no additional components.
- Full integration with other Pipeline Engine primitives.
- Can use caching to improve performance somewhat.

Cons:

- Pipeline status aggregation is synchronous with API requests, even when cached
- Scaling is sub-optimal; in test case above, Pipeline Engine needs to make `10 * 5 = 50` API calls to retrieve the current pipeline statuses for a **single user**.

2. Monitor via a separate service, writing to the pipeline DB

In this alternative, an `sdl-pipeline-monitor` service is created that handles status updates on Pipeline objects. Importantly, validation and instantiation of a Pipeline is still done in the Pipeline Engine; the monitor is only responsible for handling status changes. Writing results to the pipeline database means the Pipeline Engine only needs to retrieve information from a single source.

Pros:

- Single access point for data is easier to reason through.
- Asynchronous status updates are much more scalable.

Cons:

- Concepts like redundancy, fault tolerance, etc. need to be implemented from scratch.
- Monitor starts to look like a Kubernetes Operator without the benefits of one.
- Sharing data structures/table schemas between the engine and monitor will be clunky.

3. Monitor via a Kubernetes Operator

This alternative is the largest change to the current Pipeline Engine. Technically a Kubernetes Operator could simply watch underlying resources like Jobs and Services, but you lose many benefits of using an operator (self-healing, lifecycle management). Because of this, using a Kubernetes Operator would entail moving Pipeline instantiation out of the Pipeline Engine, a significant change from the current state. Instead of creating Jobs and Services, the Pipeline Engine would instead create a Pipeline CR containing those specs; the operator would then handle creating and managing them.

Pros:

- Kubernetes Operators built with `kubebuilder` have lots of functionality out of the box -

redundancy, horizontal scaling, fault tolerance, etc.

- Asynchronous instantiation and monitoring makes the API more responsive.
- `sdl-operator` already exists, so no net new microservices to be added.
- Existing parts of **SDL** already integrate with Kubernetes API, so this would be easy to integrate them with in the future.

Cons:

- Moving instantiation out of the engine makes the system somewhat more complex.
- Operator reconciliation logic is complex.
- Although Kubernetes CRDs are well-integrated with the API/client libraries, still some friction keeping them in sync between multiple services.
- Ties the system somewhat tightly to Kubernetes, although non-Kubernetes instantiation targets (Flink, Airflow, etc.) can still be targeted in the operator logic.
- There have been issues installing CRDs in multi-tenant environments.

Final Implementation

The team decided on alternative #3 for monitoring. As described, a Kubernetes Operator pattern is used to instantiate, reconcile, and monitor Pipelines. Pipeline- and Transformer-level status information is reflected in the Status subresource of the CR, meaning given the litmus test above, the Pipeline Engine needs to simply list Pipeline CRs to retrieve their statuses. Although the implementation makes the system more complex, it increases the scalability and reliability of the system and also improves visibility (listing the CRs will show what Pipelines are currently running).

.4.3. Simulation

Overview

SDL supports the ability to simulate the execution of individual Transformers on predefined sets of inputs. This allows users to iteratively build and test their Transformers before deploying a Pipeline. This is particularly useful in the case of Dynamic Transformers, which allow users to write custom code snippets.

This document is intended for Transformer authors who want their Transformers to support simulation. It outlines the contract that the Data Pipeline Engine expects for Transformer Templates that advertise the ability to be simulated, as well as the functional flow that occurs when a simulation request is received.

Limitations

Currently, the two `conn_type` variantes that are supported for simulation are `INTERNAL_KAFKA`, `INTERNAL_POSTGRES`, and `INTERNAL_ICEBERG`. This means that simulation is limited to Transformer Templates that **only** define inputs and outputs of those types. When registering a Transformer Template, the Data Pipeline Engine will validate that these constraints are met and will return an error if the Transformer Template cannot be used for simulation.

Enabling Simulation for a Transformer Template

Transformer Templates that want to use simulation must include the `simulation-enabled` label in their definition:

```
{
    "uid": "1afc0a55-ca7e-40db-9efc-971f85f48b42",
    "labels": [
        {"name": "simulation-enabled"}
    ]
    // Rest of definition
}
```

Simulation API - Kubernetes-based Transformers

This is the contract that the Data Pipeline Engine expects each Transformer Template to adhere to if it defines itself as having the `simulation-enabled` label. Any Transformer author that wants to enable simulation for their Transformer Template must adhere to this API for simulation to work correctly.

Note that this contract applies only to Transformer Templates that are specified as containerized images to be run in Kubernetes; other instantiation types will require separate simulation APIs.

Assumptions/Prerequisites

1. Simulation cannot occur with Transformer Templates that have any `input` connections of type `TRANSFORMER`, because the method to supply inputs to those connections is unknown by the Data Pipeline Engine.
2. A Transformer Template should support running **either** as a Transformer directly or in simulation mode without rebuilding or changing the image.

Detecting Simulation

Transformer Templates should read the value of the environment variable named `TRANSFORMER_SIMULATION_MODE` to determine whether or not to start in Simulation (FaaS) or Transformer mode:

- If the environment variable **exists and has the value of `true`**, then the Transformer should start up in Simulation mode.
- In any other case, the Transformer should start normally.

Startup

If running in Simulation mode, a Transformer should not run an event loop or listener, nor should it expect that any configuration values to connect to data sources are present. For example, the configuration contract for `INTERNAL_KAFKA` connections will not be fulfilled in Simulation mode. However, any user-provided configuration values (as specified in the Transformer Template's `configuration` block) **will** be filled in by the Data Pipeline Engine.

Instead, the Transformer should launch an HTTP server on port `8111` that contains handlers for two routes:

1. `/health`: This route should return `200 OK` once the Transformer is finished startup and is ready to accept the simulation request. The `GET` HTTP verb should be handled.
2. `/simulate`: This route should handle the simulation request coming from the Data Pipeline Engine via an HTTP `POST` request, with the JSON-encoded body containing the simulation request itself.

Handling `/simulate`

The `/simulate` route should be handled in a very specific way in order to be as efficient as possible. Transformers should anticipate that **only a single request to `/simulate`** will be made, and that after returning an HTTP response, the Transformer should shut down. Note that repeated simulation requests from users for the same Transformer Template will result in additional replicas being instantiated, **not** that the same Transformer will be used to handle more than one request. While this may impact performance slightly, it ensures that repeated simulation requests are stateless.

Request Body

The request body for `/simulate` is very specific, and is always encoded as `application/json`:

```
{
    "inputs": {
        // Each item corresponds to the name of an input connection in the Transformer
Template
        "CONNECTION_1": {
            // Note that conn_type is elided here, as it isn't important
            // Assume for this example that CONNECTION_1 is of type INTERNAL_KAFKA.
            "msgs": [
                // Each object here should be fed into the Transformer code as though
it were coming from Kafka.
                {"key": "key1", "headers": {}, "value": "Hello, world!"},
                {"key": "key2", "headers": {"__from__": "value1"}, "value": "Hello,
world!"},
                {"key": "key3", "headers": {}, "value": "Hello, world!"},
            ]
        },
        "CONNECTION_2": {
            // Assume for this example that CONNECTION_1 is of type INTERNAL_ICEBERG
or INTERNAL_POSTGRES.
            "table": {
                // This is a JSON representation of a table that should be created in
the Transformer as input.
                "cols": ["col1", "col2", "col3"],
                "rows": [
                    [1, "one", false],
                    [2, "two", true],
                    [3, "three", false]
                ]
```

```
            }
        }
    }
}
```

Upon receipt of this request body, the Transformer should route these inputs into its core transformation logic as though they were coming directly from the connected data sources.

Response

Transformers should attempt to run the simulation request as similiarly to normal operation as they can; ideally the code paths for running simulated requests and normal operation are the same.

Note that normal logging to stdout or stderr **will not** be captured by the Data Pipeline Engine when returning results to the client. Any logs the Transformer wants to return must be encoded in the response body, as outlined below.

Once a result has been obtained, the Transformer should return a response to the Data Pipeline Engine containing one of these HTTP status codes:

`200 OK`

Simulation succeeded. The Transformer should return the simulation results in a JSON response body matching this format:

```
{
    // Additional arbitrary values or objects may be returned by the Transformer in
this block. Keys beginning
    // with a double-underscore (__) will NOT be returned to the client, but any
others are.
    "metadata": {
        "simulation_time_ms": 26, // Will be returned to the client.
        "__trace_info": { // Will not be returned to the client.
            "id": 273465,
        }
    },
    // `logs` contains zero or more log messages, as output by the Transformer.
    "logs": [
        {"msg": "Hello, world!"},
        {"msg": "Second log"},
        {"msg": "Third log"}
    ],
    // `outputs` should correspond to the list of output connections defined by the
Transformer Template.
    // Note that the data model returned in each output (table or msgs) is validated
by the Data Pipeline Engine
    // against the conn_type of the output; a 500 will be returned if the returned
data do not match the expected
    // conn_type.
    "outputs": {
```

```
        "ICEBERG_TABLE_OUTPUT_CONN": {
            "table": {
                "cols": ["output1", "output2"],
                "rows": [
                    [2, "asdf"],
                    [3, "asdf"],
                ]
            }
        },
        "KAFKA_OUTPUT_CONN": {
            "msgs": [
                {"key": "key1", "headers": {}, "value": "Hello, world!"},
                {"key": "key2", "headers": {"__from__": "value1"}, "value": "Hello,
world!"},
                {"key": "key3", "headers": {}, "value": "Hello, world!"},
            ]
        },
    }
}
```

400 Bad Request

The Transformer failed to run the simulation request due to issues with the data. - For example, the input Kafka messages were raw JSON when the Transformer expected raw XML.

The Transformer should return the error in a JSON response body matching this format:

```
{
    // Additional arbitrary values or objects may be returned by the Transformer in
this block. Keys beginning
    // with a double-underscore (__) will NOT be returned to the client, but any
others are.
    "metadata": {
        "simulation_time_ms": 26, // Will be returned to the client.
        "__trace_info": { // Will not be returned to the client.
            "id": 273465,
        }
    },
    // `logs` contains zero or more log messages, as output by the Transformer.
    "logs": [
        {"msg": "Hello, world!"},
        {"msg": "Second log"},
        {"msg": "Third log"}
    ],
    "errors": [
        {"msg": "Connection XYZ must contain XML data"}
    ]
}
```

500 Internal Server Error

The Transformer failed to run the simulation request for some internal reason. Note that this will be returned to the client as a server-side error, rather than an error on the client's part.

The Transformer should return the error in a JSON response body matching this format:

```
{
    // Additional arbitrary values or objects may be returned by the Transformer in
this block. Keys beginning
    // with a double-underscore (__) will NOT be returned to the client, but any
others are.
    "metadata": {
        "simulation_time_ms": 26, // Will be returned to the client.
        "__trace_info": { // Will not be returned to the client.
            "id": 273465,
        }
    },
    // `logs` contains zero or more log messages, as output by the Transformer.
    "logs": [
        {"msg": "Hello, world!"},
        {"msg": "Second log"},
        {"msg": "Third log"}
    ],
    "errors": [
        {"msg": "Connection XYZ must contain XML data"}
    ]
}
```

Shutdown

As mentioned above, once a request to `/simulate` has been handled and responded to, the Transformer is obligated to shut down as quickly as possible, to ensure that simulation requests are as efficient as possible.

Functional Flow

When the Data Pipeline Engine receives a simulation request from a client, it requires the following information to be present:

1. The UID of the Transformer Template to execute the simulation request against. This tells the Data Pipeline Engine how to instantiate the app that will receive the simulation request.
2. Configuration for each `input` connection on the Transformer Template, which will be passed to the app at runtime. This will usually take the form of discrete data, as opposed to a connection to a Dataset.
3. General Transformer configuration (environment variables, file configurations, etc.)-effectively the same as what configuration is passed to a Transformer when it is used in a Pipeline.

Based on these pieces of information, the Data Pipeline Engine performs the following steps:

1. It retrieves the Transformer Template with the matching UID from the persistence layer.

2. It validates the configuration and input blocks from the simulation request against the Transformer Template.
3. If successful, it instantiates the given Transformer Template as a run-once task.
4. Once the Transformer is active, it forwards the simulation request to it and waits for a response.
5. Once it has received a response, it shuts the Transformer down (if it has not shut itself down already).
6. It returns the simulation response, with additional metadata, to the client.

Chapter 1. Query & Analytics

1.1. Federated SQL Engine

SDL leverages a distributed SQL query engine to federate queries across the data lake, relational databases, streaming topics, and other data sources.

Trino is required to be deployed with HTTPS enabled as well as JWT as the authentication type. == Connecting to Trino via API 1. From the SDL UI, navigate to the SDL OpenAPI spec and select the Lakehouse link. 2. Once on the Scalar API reference page, click on Authentication 3. You have an option of providing a username and password or JWT for authentication. Authentication will be handled by Keycloak and authorization will be handled by OPA policies. 4. Execute any of the APIs listed available for Lakehouse. == Connecting to Trino via CLI . Requirements to connect via CLI is that you should be an admin and should have updated the /etc/hosts for trino.localhost or any DNS used other than localhost to correctly map the subdomain to the correct local or VM IP address. . SSH onto Trino Coordinator. . Example connecting from CLI

+

```
trino --server  trino.localhost:443 --user=admin --insecure --access-token=<token>
--catalog tpch
```

1. CLI connection template

   ```
   trino --server  trino.<DNS>:443 --user=<username> --insecure --access-token=<token>
   --catalog <catalog name>
   ```

Connecting to Trino via JDBC

Example using Python sqlalchemy. User will need to update DNS and catalog.

1. Pre-requirements: user must have JWT and OPA policy access.
2. Launch python 3 in a terminal.
3. Execute the following.
4. Connection example

   ```
   import sqlalchemy as db
   from sqlalchemy.sql.expression import select, text
   from trino.auth import JWTAuthentication

   engine = db.create_engine('trino://admin@trino.localhost:443/tpch',connect_args={
       "auth": JWTAuthentication("<token>"),
        "http_scheme":"https",
           "verify": False})
   connection = engine.connect()
   ```

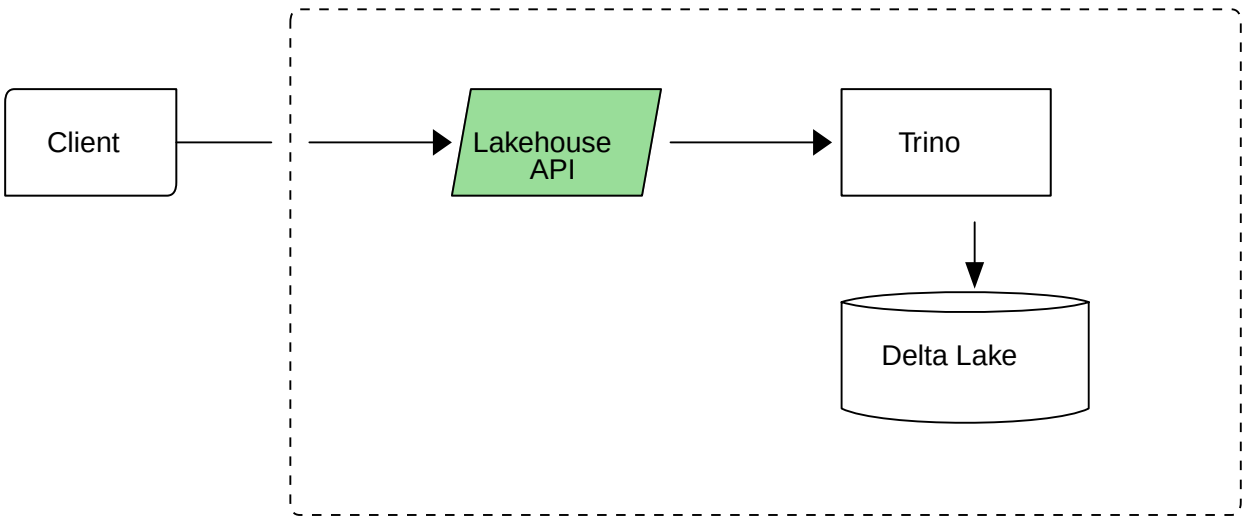
```
rows = connection.execute(text('select * from sf1.region')).fetchall()
print(rows)
```

5. Connection template

```
import sqlalchemy as db
from sqlalchemy.sql.expression import select, text
from trino.auth import JWTAuthentication

engine = db.create_engine('trino://<user>@trino.<domain>:443/<catalog
name>',connect_args={
    "auth": JWTAuthentication("<token>"),
     "http_scheme":"https",
        "verify": False})
connection = engine.connect()
rows = connection.execute(text('<query>')).fetchall()
print(rows)
```

1.2. Lakehouse

SDL provides a set of APIs exposed from the Open API Spec that expose Lakehouse data to users.



Purpose

The Lakehouse APIs provide users the ability to interact with the SDL Lakehouse.

The Lakehouse is structured with a medallion architecture and the layers are exposed through various schemas.

The SDL uses Trino as a distributed query engine which has a direct connection to the Lakehouse. Trino is useful in this architecture for several reasons: . Trino has out-of-the-box support for

interacting with the Lakehouse. . Trino allows us to implement Open Policy Agent for tighter controls on data access. . Trino Gateway supports a load balancer, proxy server, and configurable routing gateway for multiple Trino clusters. . Trino supports large workloads and queries by streaming results.

Endpoints

Operation	Endpoint	Description
GET	api/v1/lakehouse/schemas	Exposes the available schemas.
GET	api/v1/lakehouse/schemas/{schema}	Exposes all of the tables which belong to the particular {schema}.
GET	api/v1/lakehouse/schemas/{schema}/{table}	Exposes all of the column data belonging to the {schema}/{table}.
POST	api/v1/lakehouse/schemas/{schema}	Allows the user to provide custom queries to the Lakehouse.

Enablement

To utilize SDL Lakehouse APIs, users must have an account with the program Keycloak or SDL's Keycloak. The Open API Spec expects a username and password or JSON Web Token (JWT) to authenticate with the Lakehouse APIs. Users must belong to a group upon onboarding that has an associated Open Policy Agent (OPA) policy for data access. If you require access, need access to a group, or need the group's policy updated, reach out to our help desk.

1.3. Real-Time Analytics

Setup (Pre-demo)

1. Update the df-raft-trino-coordinator configMap to enable group-based access control:

```
apiVersion: v1
data:
  access-control.properties: |
    fine-grained-access-control-enabled=true
    access-control.name=OPA
```

Demo (Part 1) – Viewing Pinot Data with Proper Access Control

1. Create an enablement for mock UDL.
2. Enable the track data set once it is available.
3. Next, configure a local terminal to connect to SDL:

```
# Set environment variables for password and client secret
DF_ADMIN_PASSWORD="admin-user-pw"
```

```
DF_BACKEND_CLIENT_SECRET=$(kubectl get secret -n data-fabric keycloak-realm-init \
                          -o jsonpath='\{.data.keycloakAdminClientSecret}' | base64 -d
)
BASE_URL="localhost"
# Get access token from Keycloak and save to variable
AUTH_TOKEN=$(curl --request POST \
  --url http://$BASE_URL/auth/realms/data-fabric/protocol/openid-connect/token \
  --header 'Content-Type: application/x-www-form-urlencoded' \
  --data username=admin \
  --data password=$DF_ADMIN_PASSWORD \
  --data client_secret=$DF_BACKEND_CLIENT_SECRET \
  --data client_id=df-backend \
  --data scope=openid \
  --data grant_type=password | jq -r '.access_token')

echo "Access token: $AUTH_TOKEN"
```

1. Fetch the information necessary to create a Pinot table. It's best to get the Kafka topic value from the webpage:

```
# The name of the Pinot schema, NOT the schema in the catalog!
PINOT_SCHEMA=udl
# The name of the Pinot table
NEW_TABLE_NAME=udltrack
RETENTION_DAYS=7
# This command will only fetch the correct kafka topic if there is only a single mock
UDL enabled (and nothing else). It's best to get copy this value from the frontend
manually.
KAFKA_TOPIC=$(kubectl get datasets -A -o
jsonpath='{.items[0].metadata.labels.datafabric\.goraft\.tech/dataset}')
# Get kafka password from secret
KAFKA_PASSWORD=$(kubectl -n data-fabric get secrets/df-kafka-user-internal
--template={{.data.password}} | base64 -d)
```

1. POST the schema to SDL. This schema should match the shape of the data the Pinot table is supposed to fetch from Kafka.

```
# Create Pinot schema
curl -X POST http://$BASE_URL/api/internal/pinot/schemas \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $AUTH_TOKEN" \
  -d '{
    "schemaName": "udl",
    "dimensionFieldSpecs": [
      {"name": "origin", "dataType": "STRING"},
      {"name": "dataMode", "dataType": "STRING"},
      {"name": "asset", "dataType": "INT"},
      {"name": "objType", "dataType": "STRING"},
```

```
      {"name": "id", "dataType": "STRING"},
      {"name": "createdBy", "dataType": "STRING"},
      {"name": "classificationMarking", "dataType": "STRING"},
      {"name": "objIdent", "dataType": "STRING"},
      {"name": "origNetwork", "dataType": "STRING"},
      {"name": "trkId", "dataType": "STRING"},
      {
        "name": "__security__",
        "dataType": "JSON"
      }
    ],
    "metricFieldSpecs": [
      {"name": "alt", "dataType": "DOUBLE"},
      {"name": "lon", "dataType": "DOUBLE"},
      {"name": "lat", "dataType": "DOUBLE"}
    ],
    "dateTimeFieldSpecs": [
      {
        "name": "time",
        "dataType": "LONG",
        "notNull": false,
        "format": "1:MILLISECONDS:EPOCH",
        "granularity": "1:MILLISECONDS"
      }
    ]
  }'
```

1. POST the table configuration to SDL. The schema referenced should be the one that was saved on the prior step.

```
curl -X POST http://$BASE_URL/api/internal/pinot/tables \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer $AUTH_TOKEN" \
  -d "{
  \"tableName\": \"$NEW_TABLE_NAME\",
  \"tableType\": \"REALTIME\",
  \"segmentsConfig\": {
    \"timeColumnName\": \"time\",
    \"timeType\": \"MILLISECONDS\",
    \"retentionTimeUnit\": \"DAYS\",
    \"retentionTimeValue\": $RETENTION_DAYS,
    \"replication\": 1,
    \"replicasPerPartition\": 1,
    \"schemaName\": \"$PINOT_SCHEMA\"
  },
  \"tableIndexConfig\": {
    \"loadMode\": \"MMAP\",
    \"streamConfigs\": {
      \"streamType\": \"kafka\",
      \"stream.kafka.consumer.type\": \"lowLevel\",
```

```
      \"stream.kafka.topic.name\": \"$KAFKA_TOPIC\",
      \"stream.kafka.decoder.class.name\":
\"org.apache.pinot.plugin.stream.kafka.KafkaJSONMessageDecoder\",
      \"stream.kafka.consumer.factory.class.name\":
\"org.apache.pinot.plugin.stream.kafka20.KafkaConsumerFactory\",
      \"stream.kafka.broker.list\": \"df-kafka-bootstrap:9092\",
      \"key.serializer\":
\"org.apache.kafka.common.serialization.StringDeserializer\",
      \"value.serializer\":
\"org.apache.kafka.common.serialization.StringDeserializer\",
      \"stream.kafka.consumer.prop.auto.offset.reset\": \"smallest\",
      \"security.protocol\": \"SASL_PLAINTEXT\",
      \"sasl.jaas.config\": \"org.apache.kafka.common.security.scram.ScramLoginModule
required username=\\\"internalkafkauser\\\" password=\\\"$KAFKA_PASSWORD\\\";\",
      \"sasl.mechanism\": \"SCRAM-SHA-512\",
      \"stream.kafka.decoder.prop.format\": \"JSON\",
      \"stream.kafka.decoder.prop.schema.registry.rest.url\": \"http://df-schema-
registry:8081\"
    }
  },
  \"ingestionConfig\": {
    \"transformConfigs\": [
      {
        \"columnName\": \"time\",
        \"transformFunction\": \"now()\"
      }
    ]
  },
  \"tenants\": {
    \"broker\": \"DefaultTenant\",
    \"server\": \"DefaultTenant\"
  },
  \"metadata\": {
    \"customConfigs\": {}
  }
}"
```

1. With `df-backend` versions older than `1.15.165`, you need to update the catalog data set storages manually. This will trigger the authorization service to create permissions to the data set in Pinot based on the data set enablement's groups:

```
# Get information about enabled data sets.
RESPONSE=$(curl
"http://localhost/api/v2/catalog/datasources/all/enablements/all/datasets" \
        -H 'Content-Type: application/json' \
        -H "Authorization: Bearer $AUTH_TOKEN")

# Parse the id of the storage for the enablement. NOTE: This assumes there's ONE
enablement only!
STORAGE_ID=$(echo $RESPONSE | jq -r '.[] | select(.name == "track") | .storage[] |
```

```
select(.name == "default") | .id')
echo $STORAGE_ID

# Parse the "self" link that makes up /datasources/{datasource-
id}/enablements/{enablement-id}/datasets/{dataset-id}
PREFIX_URL=$(echo $RESPONSE | jq -r '.[0].links[] | select(.rel == "self") | .href')
echo $PREFIX_URL

STORAGE_URL="${PREFIX_URL}/storages/${STORAGE_ID}"
echo $STORAGE_URL

curl -vvv "http://{sdl-host}/api/v2/catalog/$STORAGE_URL" -H 'Content-Type:
application/json' -H "Authorization: Bearer $AUTH_TOKEN" | jq
```

1. Connect to Trino and try to view data from the Pinot catalog. There should be a `default` schema and a `udltrack` table available. Query the data and confirm the results are visible to the user.
2. To test that the group-based access control, sign out of SDL and sign back in as a user that is NOT in any of the groups the enablement was created for.
   a. Repeat step 8. This time, no schema nor table suggestions should show up in the dropdown.
   b. Try to run the query `SELECT * FROM default.udltrack` and confirm that a Trino error is returned to the user.

Demo (Part 2) – Creating a Dashboard with Pinot Data

1. Switch back to the Admin user (or whatever user can view the enabled data set).
2. Create a data set from a SQL query:
   a. Run the `SELECT * FROM default.udltrack LIMIT 100` query.
   b. On the "Save" button, hit the dropdown, and save the results as a data set named "test."
   c. Hit "Save and explore" to get redirected to the chart editing page.
3. Press "View all charts."
   a. On the left side of the screen, click "Map."
   b. Select the `deck.gl Scatterplot` and press the "Select" button.
4. Update the "Longitude & Latitude" fields to point to `lon` and `lat`, respectively.
5. On the "Map" dropdown, play around with the `https://map.localhost/geoserver/wms` URL until you get the `nasa:blue_marble` WMS Layer value.
6. Increase the point size from `1000` to `7000`.
7. Change the point color to bright red.
8. Click "Update Chart" and wait for the map to render.
9. Click "Save" on the top-right to save the chart to and existing dashboard, to save it to a new dashboard.

Chapter 2. Virtual Knowledge Graph

2.1. RDF & OBDA

Chapter 3. Streaming & Storage

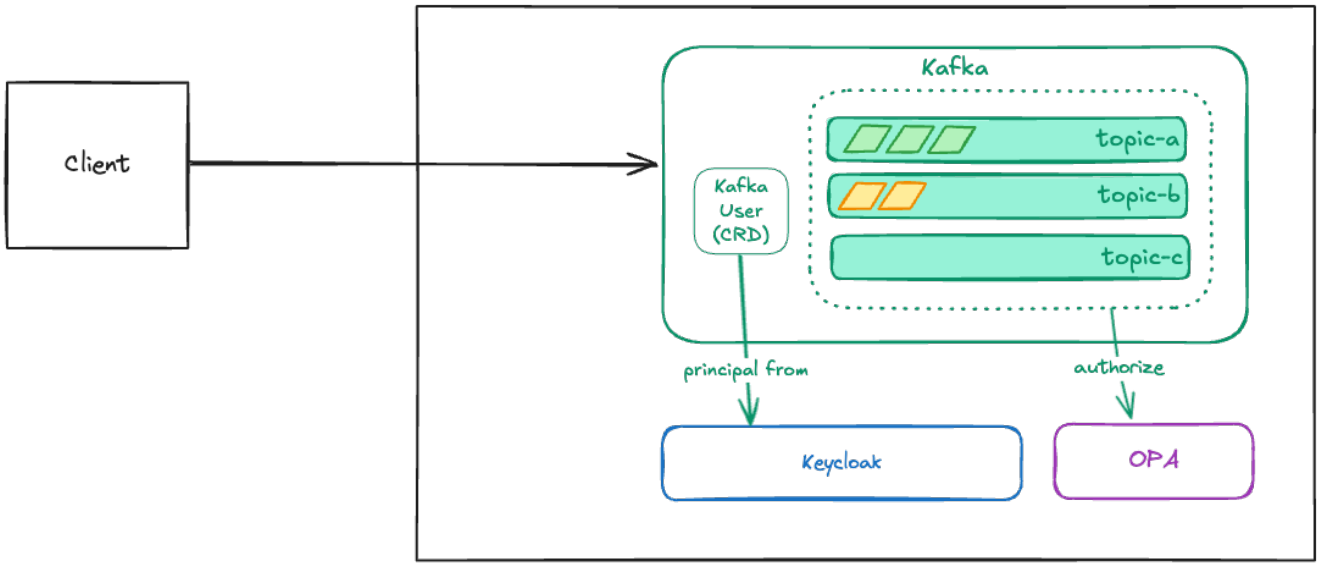
3.1. Event Streaming

> To explore the full API, download the OpenAPI or view it in the Scalar docs.

SDL runs and internally manages an event streaming broker for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications. The event streaming backbone handles data ingestion and processing pipelines.

Users are managed by the Strimzi Kubernetes Operator, which creates a `KafkUser` CRD for each user that is granted access to Kafka (by an administrator). The `KafkaUser` ID matches the user's principal ID in Keycloak. This enables Kafka to identify the user by their principal ID in Keycloak (instead of a separate access key and secret key), which is needed to authorize the user for access to specific objects.

For authorization decisions, Kafka defers to Open Policy Agent (OPA) on a per-topic basis. This is where security policies are applied, such as classification access controls.



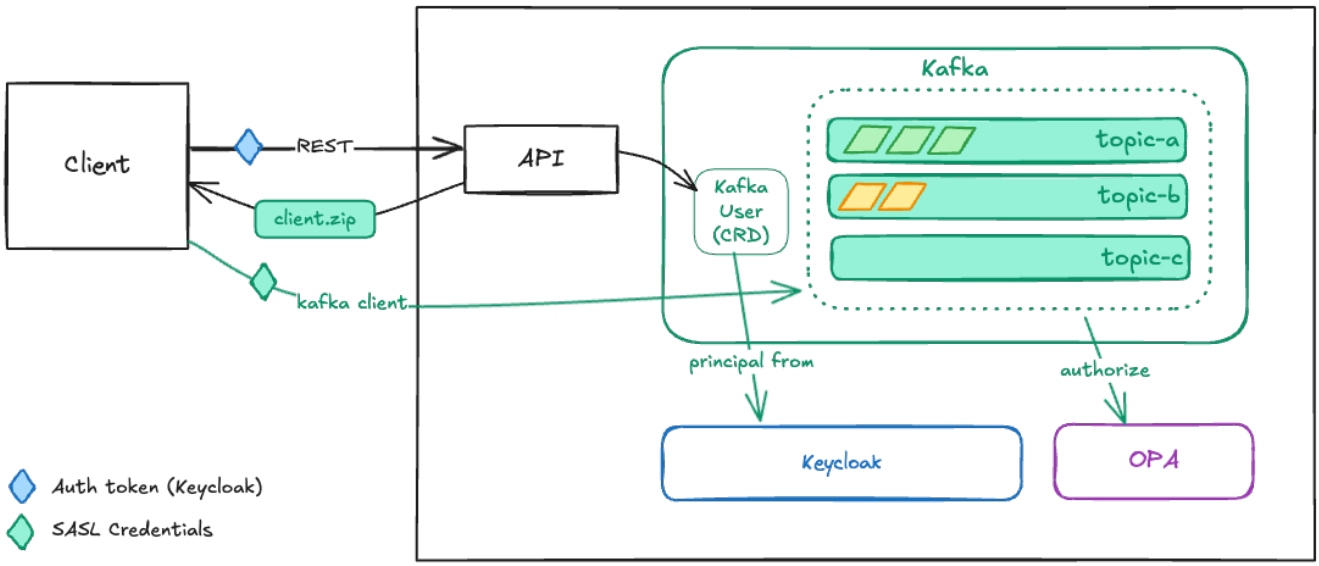
See Clients for getting connected.

3.1.1. Clients

Kafka is available to external clients over `SASL_SSL` using `SCRAM-SHA-512`.

Credentials

To connect to Kafka, each client will need to download their own connection credentials from the `/api/v1/kafka/client` endpoint.



Get as `.zip` File

If you include an `Accept` header of `application/octet-stream`, you will be given a binary stream to download as a `.zip` file.

Request

```
curl -X 'GET' \
  https://localhost/api/v1/kafka/client \
  -H 'accept: application/octet-stream' \                  ①
  -H 'Authorization: Basic YWRtaW46eUNITHBDaG12dDRuTVUwNWpaZTZBbGl0' \  ②
  --output df-kafka.zip ③
```

① Downloads a `.zip` file.

② Can use `Basic` (username/password) or `Bearer` token.

③ Filename of downloaded config zip.

If you get a `401 Unauthorized` response, you will need an administrator to grant you access to Kafka.

This will provide you with a `df-kafka.zip` file containing:

- `df-kafka.zip` - Client certificate needed for the connection.
- `kcat.conf` - A pre-configured configuration file for testing your connection with `kcat` (a lightweight Kafka CLI client).

Your client credentials are in the `kcat.conf` as `sasl.username` and `sasl.password`.

`kcat.conf`

```
# Usage: kcat -b kafka-bootstrap.localhost:443 -C -t test-topic -F kcat.conf
ssl.ca.location=df-kafka.crt
security.protocol=SASL_SSL
sasl.mechanism=SCRAM-SHA-512
```

```
sasl.username=9dfb067a-fd75-4b50-a0ea-8654487babd6
sasl.password=evqp2h0mt3l9h1lkpph0
```

Get as JSON

If you include an `Accept` header of `application/json`, you will be given a more programmatically friendly response.

Request

```
curl -X 'GET' \
  https://localhost/api/v1/kafka/client \
  -H 'accept: application/json' \                    ①
  -H 'Authorization: Basic YWRtaW46eUNITHBDaG12dDRuTVUwNWpaZTZBbGl0'   ②
```

① Ask for JSON.

② Can use `Basic` (username/password) or `Bearer` token.

If you get a `401 Unauthorized` response, you will need an administrator to grant you access to Kafka.

This will provide you with a JSON response containing the same details as the `.zip` (above).

```
{
  "bootstrapServers": "kafka-bootstrap.localhost:443",
  "certificate": "-----BEGIN CERTIFICATE-----\n<certification-body>\n-----END
CERTIFICATE-----\n",
  "sasl": {
    "mechanism": "SCRAM-SHA-512",
    "username": "8a81001a-28c0-4257-a2de-7f247e42c10e",
    "password": "o5t0v8pz2etego0apqav"
  }
}
```

Connection Test

To test your client credentials, you can run `kcat` with the `kcat.conf` configuration file provided in the `df-kafka.zip` download.

In the `kcat.conf` file you will find an example usage in the comment at the top of the file. Copy that line and run it in your terminal in the same directory containing the `kcat.conf` file.

```
kcat -b kafka-bootstrap.localhost:443 -C -t test-topic -F kcat.conf
```

If successful, you should see output similar to:

```
% Reading configuration from file kcat.conf
% Reached end of topic test-topic-u [0] at offset 0
```

The offset will vary depending on the current size of the `test-topic` topic.

3.1.2. Administrators

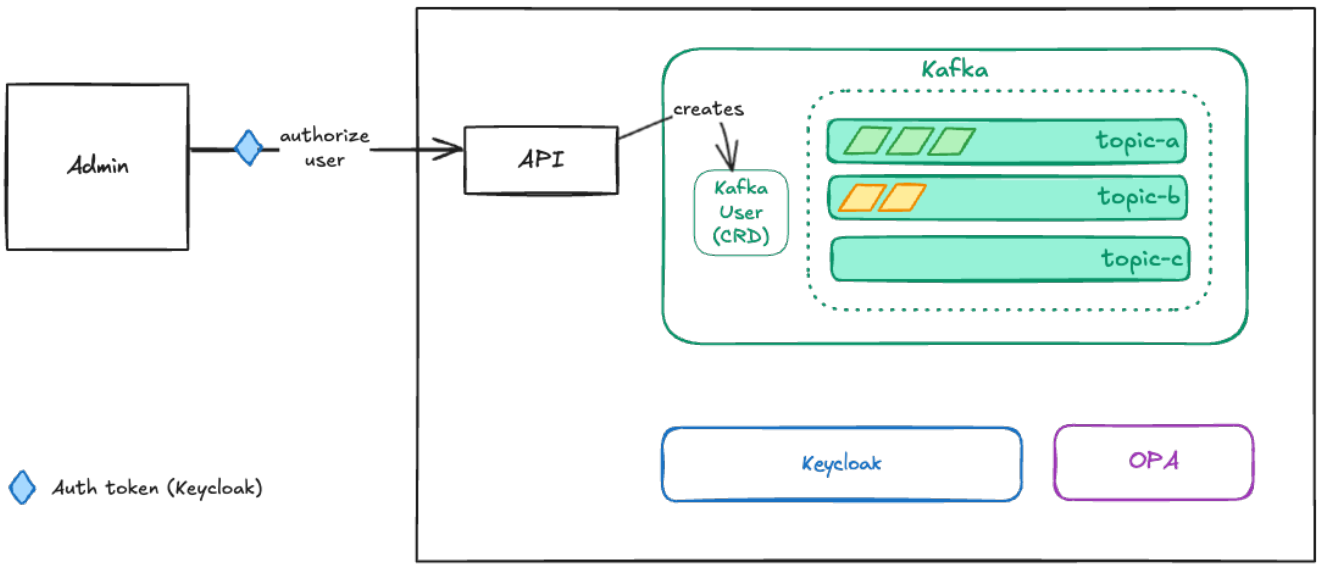
Client Access

Before a client can access Kafka, their user principal must be granted access by an administrator.

Grant User Access

To grant a user access to Kafka, a user with admin privileges issues a `PUT` request to `/api/v1/kafka/users/{user}` where `{user}` is the user's principal ID (in Keycloak).

A user can get their principal from the auth test API.



This creates a `KafkaUser` CRD, along with an associated `Secret` containing the user's credentials.

Users can download their credentials, along with everything else they need (SSL certificate, host URL, etc.), from the client credentials API endpoint.

Anyone accessing Kafka *must* have a `KafkaUser` CRD instance.

Request

```
curl -X 'PUT' \
  'http://localhost/api/v1/kafka/users/21feb230-3ac4-471c-b90d-6d145a8b3f6e' \ ①
  -H 'accept: application/json' \
  -H 'Authorization: Basic YWRtaW46eUNITHBDaG12dDRuTVUwNWpaZTZBbGl0' ②
```

① The user principal to grant access to Kafka.

② Credentials for an admin user (can use `Basic` username/password or `Bearer` token).

If successful, you will get a response that echos back a confirmation of the user's principal that was granted.

Response

```
{
  "name": "21feb230-3ac4-471c-b90d-6d145a8b3f6e"
}
```

List Authorized Users

Admins can get a list of currently authorized users with a `GET` to `/api/v1/kafka/users`.

Request

```
curl -X 'GET' \
  'http://localhost/api/v1/kafka/users' \
  -H 'accept: application/json' \
  -H 'Authorization: Basic YWRtaW46eUNITHBDaG12dDRuTVUwNWpaZTZBbGl0'
```

Response

```
[
  {
    "name": "21feb230-3ac4-471c-b90d-6d145a8b3f6e"
  },
  {
    "name": "972f1aff-9c94-4e73-92be-f45a9156c83e"
  },
  {
    "name": "2204a4ec-9137-4d3d-a61a-b24409a9cd20"
  }
]
```

Revoke User Access

To revoke a user's access to Kafka, an admin issues a `DELETE` request to `{url-base/users/{user}` where `{user}` is the user's principal ID (in Keycloak).

A user can get their principal from the auth test API.

Request

```
curl -X 'DELETE' \
  'http://localhost/api/v1/kafka/users/21feb230-3ac4-471c-b90d-6d145a8b3f6e' \ ①
  -H 'accept: application/json' \
```

```
  -H 'Authorization: Basic YWRtaW46eUNITHBDaG12dDRuTVUwNWpaZTZBbGl0' ②
```

① The user principal to revoke access to Kafka.

② Credentials for an admin user (can use `Basic` username/password or `Bearer` token).

If successful, you will get a response that echos back a confirmation of the user's principal that was revoked.

Response

```
{
  "name": "21feb230-3ac4-471c-b90d-6d145a8b3f6e"
}
```

3.1.3. Access Controls

Kafka access controls are divided into two pieces: **Authentication** and **Authorization**.

Authentication

Client authentication with Kafka is managed by the Strimzi Operator, which looks for a `KafkaUser` custom resource definition (CRD) for the user attempting to authenticate with Kafka.

`KafkaUser`

```
apiVersion: kafka.strimzi.io/v1beta2
kind: KafkaUser
metadata:
  labels:
    strimzi.io/cluster: df
  name: 346bb0f4-c8f6-46de-a3c5-6b06c7e4ce81                    ①
  namespace: data-fabric
spec:
  authentication:
    password:
      valueFrom:
        secretKeyRef:
          key: password
          name: df-kafka-client-346bb0f4-c8f6-46de-a3c5-6b06c7e4ce81    ②
    type: scram-sha-512
```

① The user ID, which matches the user's principal ID in Keycloak.

② Name of the `Secret` containing the user's auto-generated password for Kafka.

Matching the user's principal is key to performing fine access controls, in which the user's attributes and group memberships are looked up in Keycloak using their principal.

Managing Users

All management of the `KafkaUser` and his associated `Secret` are managed by the Kafka Administrator, which handles creating them when granting access to a user.

Reflexively, the admin API also handles removing the `KafkaUser` and his `Secret` when the user's access is revoked.

Accessing Kafka

To access Kafka as a client, see Clients.

Authorization

Kafka is configured to defer all authorization decisions to OPA.

Per-Topic Authorization

The OPA policies for Kafka control access at the **topic level**. That is, topics can be controlled individually but their individual messages within the *same* topic cannot be differentiated.

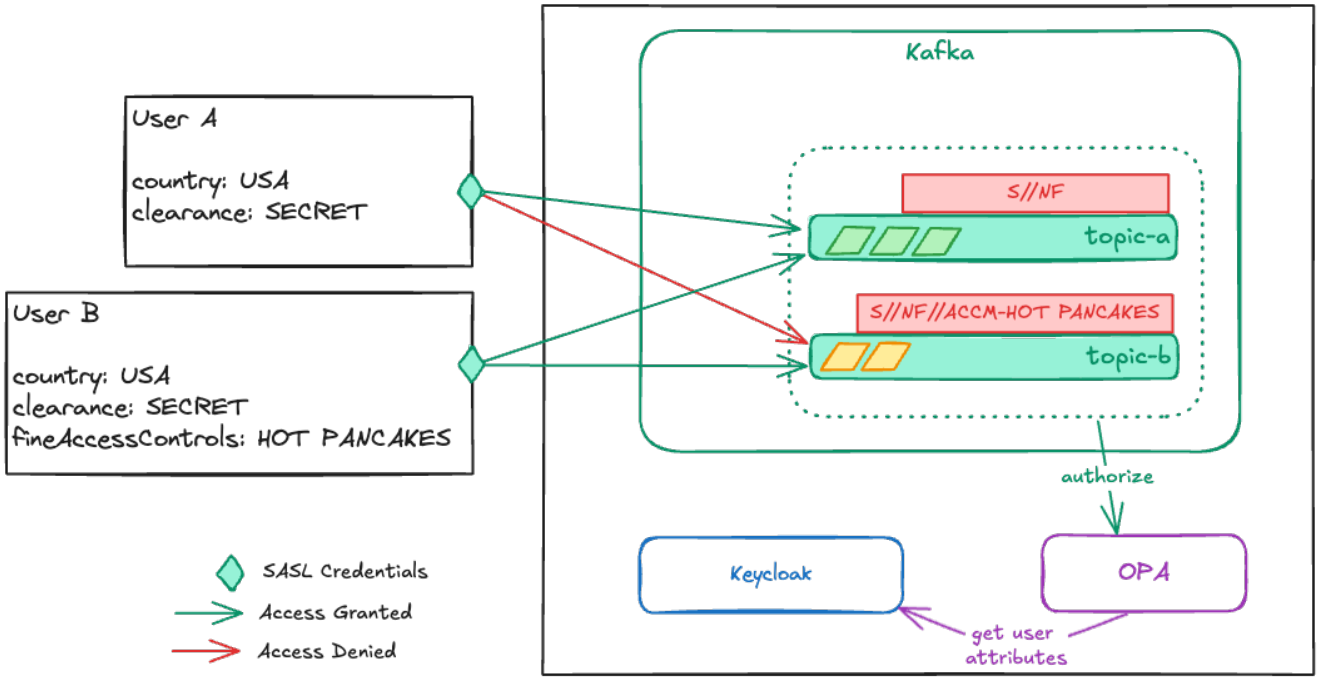
If a user has access to a particular topic, they have access to *all* messages on that topic.

Topic authorization can change dynamically, either by updating policies or by modifying the user's attributes in Keycloak.

If a user has an active session with a topic, and the policy for that topic changes such that the user should no longer have access, the user's session will be terminated by Kafka.

Classification Controls

Each topic can be (optionally) marked with a classification marking using IC Classification and Control Markings.



Set Classification Marking

To set a classification marking for a topic, a user issues a `PUT` request to the `/topics/{topic}/classification` endpoint.

Request

```
curl -X 'PUT' \
  'http://localhost/api/v1/kafka/topics/test-
topic/classification?classification=S%2F%2FNF' \  ①
  -H 'accept: application/json' \
  -H 'Authorization: Basic YWRtaW46eUNITHBDaG12dDRuTVUwNWpaZTZBbGl0'  ②
```

① Provide the marking in the `classification` URL arg (be sure to url-escape the value). Here we set it to `S//NF`.

② Can use `Basic` (username/password) or `Bearer` token.

To successfully set (or change) a topic's classification,

1. The invoking user must be cleared for the marking they are providing (cannot mark beyond their own clearance).
2. If the topic has an existing marking (changing marking), the user must also be cleared for the existing marking.

Get Classification Marking

To see the current classification marking for a topic, a user issues a `GET` request to the `/topics/{topic}/classification` endpoint.

Request

```
curl -X 'GET' \
  'http://localhost/api/v1/kafka/topics/test-topic/classification' \
  -H 'accept: application/json' \
  -H 'Authorization: Basic YWRtaW46eUNITHBDaG12dDRuTVUwNWpaZTZBbGl0'
```

This will return the normalized form of the marking, which includes a `"raw"` attribute containing the originally provided marking.

Response

```
{
  "raw": "S//NF",
  "components": {
    "classification": "S",
    "disseminationControls": [
      "NF"
    ],
    "ownerProducer": [
      "USA"
    ],
    "releasableTo": []
  }
}
```

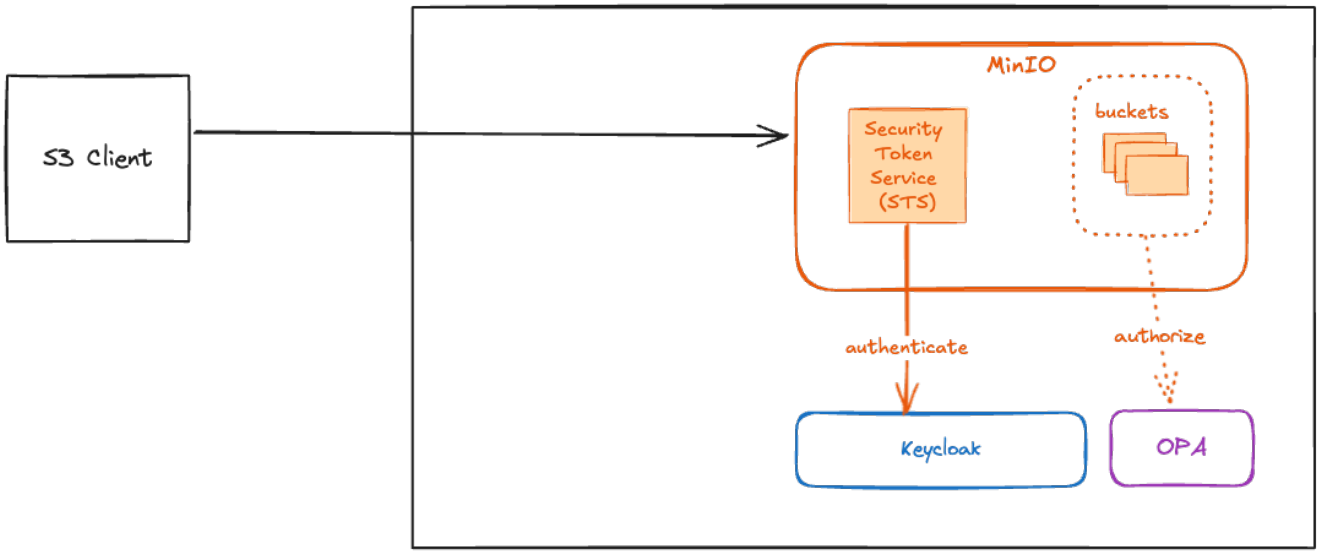
To explore the full API, download the OpenAPI or view it in the Scalar docs.

3.2. Object Storage

SDL runs and internally manages an S3-compatible object storage cluster for blob storage.

It uses a Security Token Service (STS) for user JWT token authentication. This enables the object storage layer to identify the user by their principal ID in the identity provider (instead of a separate access key and secret key), which is needed to authorize the user for access to specific objects.

For authorization decisions, the object storage layer defers to the policy engine on a per-object basis. This is where security policies are applied, such as classification access controls.



See Clients for getting connected.

3.2.1. Clients

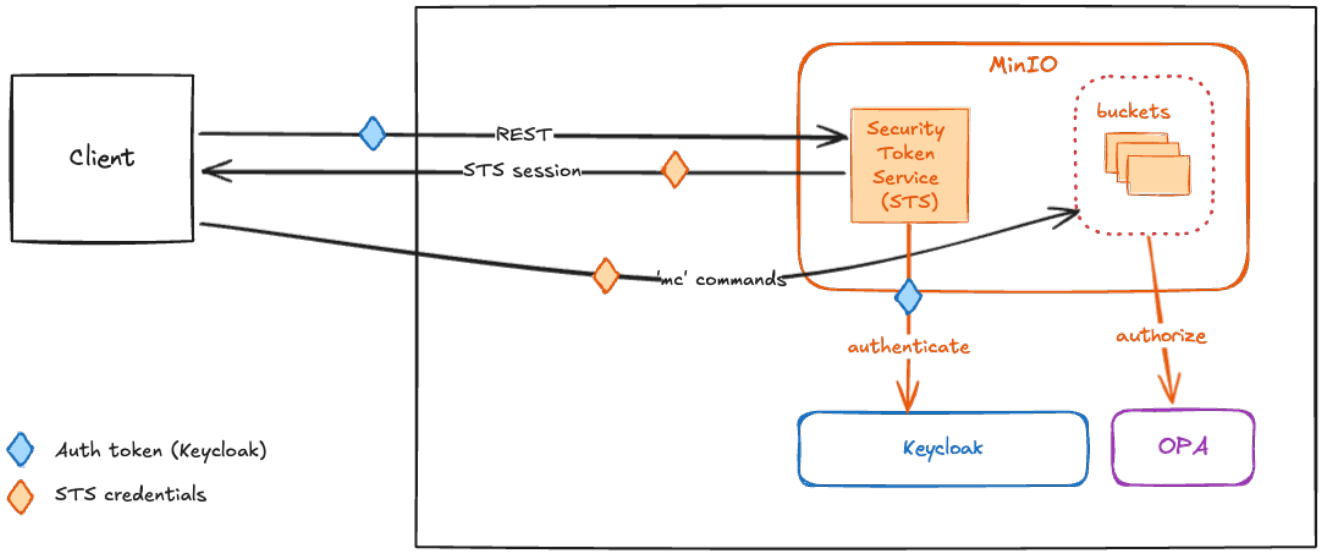
S3 is available to external clients over `s3`, as well as a few supported operations over `http` REST (for convenience).

The `mc` client is recommended, as that gives you the most efficient experience. Also see the example go client that uses a dedicated MinIO client library.

The REST client should be used sparingly, as that creates a new STS session for every invocation.

`mc` Client

To use MinIO's `mc` client, you will need to perform a token exchange with MinIO's Security Token Service (STS). This creates a temporary STS session with your user credentials.



The basic flow is, with your SDL auth token:

1. Get an STS session.
2. Use the STS session access key, secret and token with your `s3` client of choice.

The procedures listed below use the following to assist in extracting elements from service response payloads:

- `jq`
- `yq`

If you don't have these, you can still run through the steps but you will need to extract the parts from the responses yourself.

Auth Token

If you do not have one yet, get an auth token for yourself from SDL with an audience for the `df-minio` keycloak client.

```
HOST=localhost
USER=your_username
PASS=your_password
AUTH_TOKEN=$(curl -sk -X POST \
  "https://$HOST/api/v1/auth/token" \
  -H 'accept: application/json' \
  -H 'Content-Type: application/x-www-form-urlencoded' \
  -d "grant_type=password&username=$USER&password=$PASS&client_id=df-
minio&client_secret=" \
  | jq -r '.access_token')
```

Open an STS Session

Using your `AUTH_TOKEN`, get an STS session credentials.

```
HOST=localhost
STS_CREDENTIALS=$(curl -sk -X POST \
  "https://s3.$HOST?Action=AssumeRoleWithWebIdentity&Version=2011-06-
15&WebIdentityToken=$AUTH_TOKEN" \
  | yq --input-format xml
'.AssumeRoleWithWebIdentityResponse.AssumeRoleWithWebIdentityResult.Credentials')
```

From the `STS_CREDENTIALS`, parse out the session access key, secret and token.

```
STS_ACCESS_KEY=$(echo -n $STS_CREDENTIALS | yq '.AccessKeyId')
STS_SECRET_KEY=$(echo -n $STS_CREDENTIALS | yq '.SecretAccessKey')
STS_SESSION_TOKEN=$(echo -n $STS_CREDENTIALS | yq '.SessionToken')
```

STS sessions are short-lived, and will expire after 15 minutes.

Configure `mc`

Export a `MC_HOST_{alias}` environment variable to configure the `mc` client, using your STS access key, secret and session token.

```
export
MC_HOST_myalias=https://$STS_ACCESS_KEY:$STS_SECRET_KEY:$STS_SESSION_TOKEN@s3.localhos
t
```

If the certificate your SDL instance is running with cannot be verified, you will need to tell `mc` to ignore TLS errors.

```
export MC_INSECURE=true
```

Then use `mc` as you normally would.

```
mc ls myalias
```

Why STS?

STS session tokens are used to tie the session to a user in Keycloak. This is needed to ensure proper access controls (including classification markings) are in place for the user.

Without an STS token, clients will be restricted from accessing any object with a classification marking.

For more info, see the STS docs.

`go` Client

The `go` client for MinIO handles the STS token exchange for you, and will also renew the token if/when it expires.

To set up the client, you just pass it the URL to MinIO and your SDL auth token.

The example `go` code below is a function you can drop into your `go` app for creating a `minio.Client` instance.

```
import (
    "fmt"
    "github.com/minio/minio-go/v7"
    "github.com/minio/minio-go/v7/pkg/credentials"
    "github.com/rs/zerolog/log"
    "net/url"
)
```

```
// newMinioStsClient creates a new minio client with "STSWebIdentity" credentials.
// The `stsEndpoint` should be the URL to minio's REST API (ex: http://s3.localhost).
// The `authToken` should be the invoking user's base64 encoded JWT (note that this
needs an audience with the `df-minio` client).
func newMinioStsClient(stsEndpoint, authToken string) (*minio.Client, error) {
    log.Debug().Msgf("STS endpoint: %s", stsEndpoint)

    var getWebTokenExpiry = func() (*credentials.WebIdentityToken, error) {
        return &credentials.WebIdentityToken{
            Token: authToken,
        }, nil
    }

    stsUrl, err := url.Parse(stsEndpoint)
    if err != nil {
        return nil, fmt.Errorf("Failed to parse STS Endpoint '%s':  %w", stsEndpoint,
err)
    }

    sts, err := credentials.NewSTSWebIdentity(stsEndpoint, getWebTokenExpiry)
    if err != nil {
        return nil, fmt.Errorf("Could not get STS credentials from '%s':  %w",
stsEndpoint, err)
    }

    opts := &minio.Options{
        Creds:        sts,
        BucketLookup: minio.BucketLookupAuto,
    }

    return minio.New(stsUrl.Host, opts)
}
```

Using the function above, create a minio client for a user's session with their auth token.

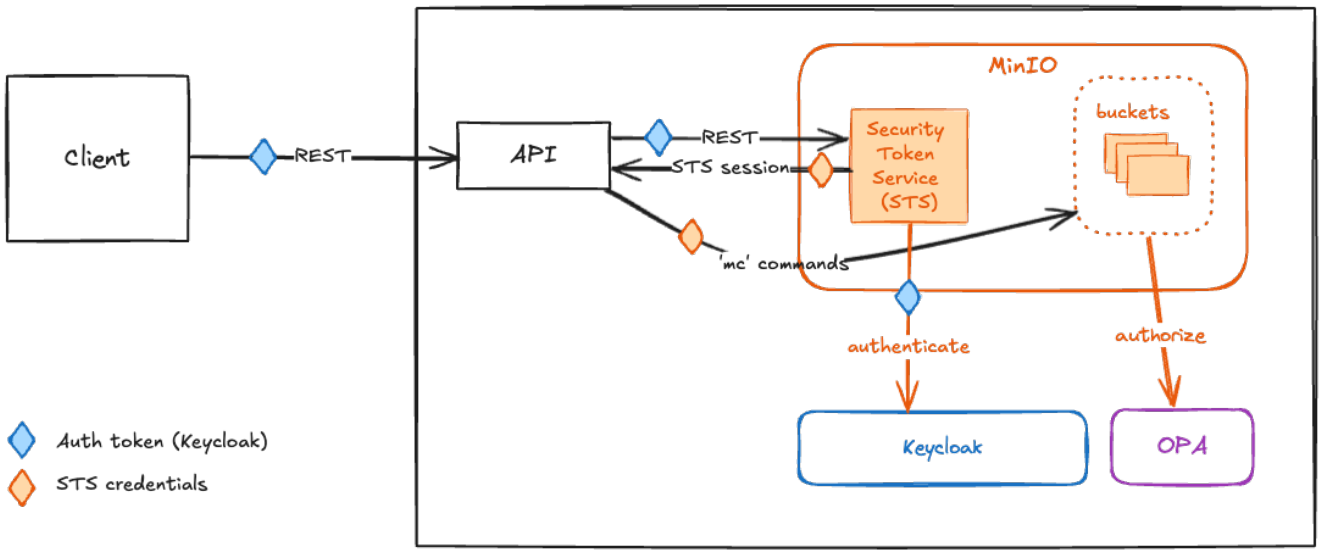
```
mc, _ := newMinioStsClient("http://s3.localhost", token)
```

And retain it for the life of your user auth token session. The client will renew the STS session when needed.

```
buckets, _ := mc.ListBuckets(context.Background())
```

REST Client

The REST API is rooted at `/api/v1/s3` and provides basic bucket and object operations suitable for simple workloads.



Below are just a few basic examples using the REST API.

> To explore the full API, download the OpenAPI or view it in the Scalar docs.

If you need deeper `s3` protocol integration, see the `mc` or `go` client examples.

List Available Buckets

Request

```
curl -X 'GET' \
  'http://localhost/api/v1/s3/buckets' \
  -H 'accept: application/json' \
  -H 'Authorization: Basic YWRtaW46eUNITHBDaG12dDRuTVUwNWpaZTZBbGl0' ①
```

① Can use `Basic` (username/password) or `Bearer` token.

Response

```
[
  {
    "name": "df-backups",
    "creationDate": "2024-11-04T18:41:50.255Z"
  },
  {
    "name": "df-notebooks",
    "creationDate": "2024-11-04T18:41:51.321Z"
  },
  {
    "name": "df-schemas",
    "creationDate": "2024-11-04T18:41:56.739Z"
  },
  {
    "name": "inbox-public",
    "creationDate": "2024-11-04T18:42:00.742Z"
```

```
  }
]
```

Actual bucket listing will vary.

Upload a File

Request

```
curl -X 'PUT' \
  'http://localhost/api/v1/s3/buckets/inbox-public/objects/content?objectName=my-
file.txt&classification=UNCLASSIFIED' \ ①
  -H 'accept: application/json' \ ②
  -H 'Authorization: Basic YWRtaW46eUNITHBDaG12dDRuTVUwNWpaZTZBbGl0' \ ③
  -H 'Content-Type: application/json' \
  -d 'This is my file.' ④
```

① Uploading file `my-file.txt` to bucket `inbox-public` with marking `UNCLASSIFIED`.

② The response will be a JSON payload describing the uploaded object.

③ Can use `Basic` (username/password) or `Bearer` token.

④ Raw content of file.

Response

```
{
  "name": "my-file.txt",
  "etag": "32ea80dfecd60916d9091b304dd4efb0-1",
  "sizeBytes": 16
}
```

List Objects in Bucket

Request

```
curl -X 'GET' \
  'http://localhost/api/v1/s3/buckets/inbox-public/objects' \ ①
  -H 'accept: application/json' \
  -H 'Authorization: Basic YWRtaW46eUNITHBDaG12dDRuTVUwNWpaZTZBbGl0' ②
```

① List contents of bucket `inbox-public`.

② Can use `Basic` (username/password) or `Bearer` token.

Response

```
[
  {
    "name": "my-file.txt",
    "tags": {},
```

```
    "etag": "32ea80dfecd60916d9091b304dd4efb0-1",
    "classification": {
      "raw": "U",
      "components": {
        "classification": "U",
        "disseminationControls": [],
        "ownerProducer": [
          "USA"
        ],
        "releasableTo": []
      }
    },
    "lastModified": "2024-11-04T22:05:18.561Z",
    "sizeBytes": 16
  }
]
```

Download a File

Request

```
curl -X 'GET' \
  'http://localhost/api/v1/s3/buckets/inbox-public/objects/content?objectName=my-
file.txt' \ ①
  -H 'accept: */*' \
  -H 'Authorization: Basic YWRtaW46eUNITHBDaG12dDRuTVUwNWpaZTZBbGl0' ②
```

① Downloading file `my-file.txt` from bucket `inbox-public`.

② Can use `Basic` (username/password) or `Bearer` token.

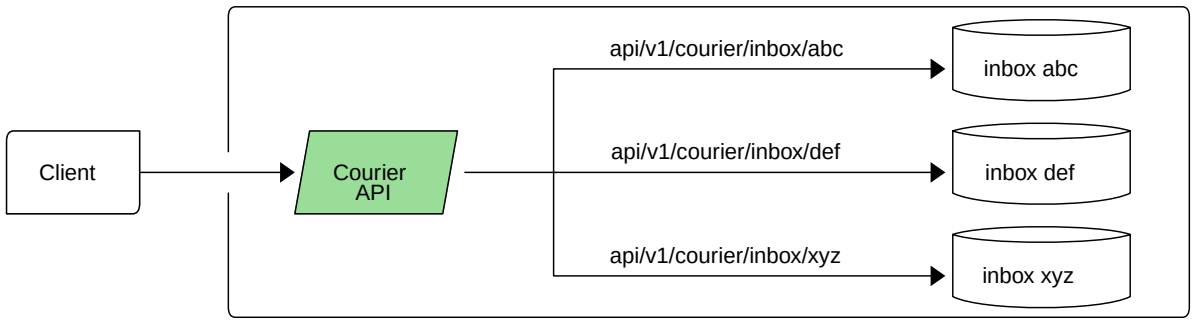
The response will be an octet stream of the binary content of the file.

> To explore the full API, download the OpenAPI or view it in the Scalar docs.

3.2.2. Courier

The Courier API is rooted at `/api/v1/courier` and provides a general "inbox"-like landing zone for sending data to SDL.

There are various "inboxes", organized by system or data type, and each can have unique security access controls (if necessary).



The inboxes listed above are (obviously) for example only.

> To explore the full API, download the OpenAPI or view it in the Scalar docs.

Purpose

The inboxes are a simple means for dropping data off to SDL. This can be useful for ad-hoc data ingestion, or simply "parking" data that has no other ICD mechanism.

Each inbox is "write-only", though you can read the current contents of an inbox.

Once data is submitted to an inbox, it may be ingested and/or moved by SDL (depending on the data).

3.2.3. Access Controls

S3 access controls are divided into two pieces: **Authentication** and **Authorization**.

Authentication

Client authentication for S3 is managed by MinIO's Security Token Service (STS), which accepts a user's authentication token and exchanges it for a session token with MinIO. The session token from STS retains the user's principal ID, and combines it with all the necessary information MinIO requires to authenticate and authorize the user for all actions they perform within the session.

See the `mc` client for an overview of how this works.

Authorization

MinIO is configured to defer all authorization decisions to OPA.

Per-Object Authorization

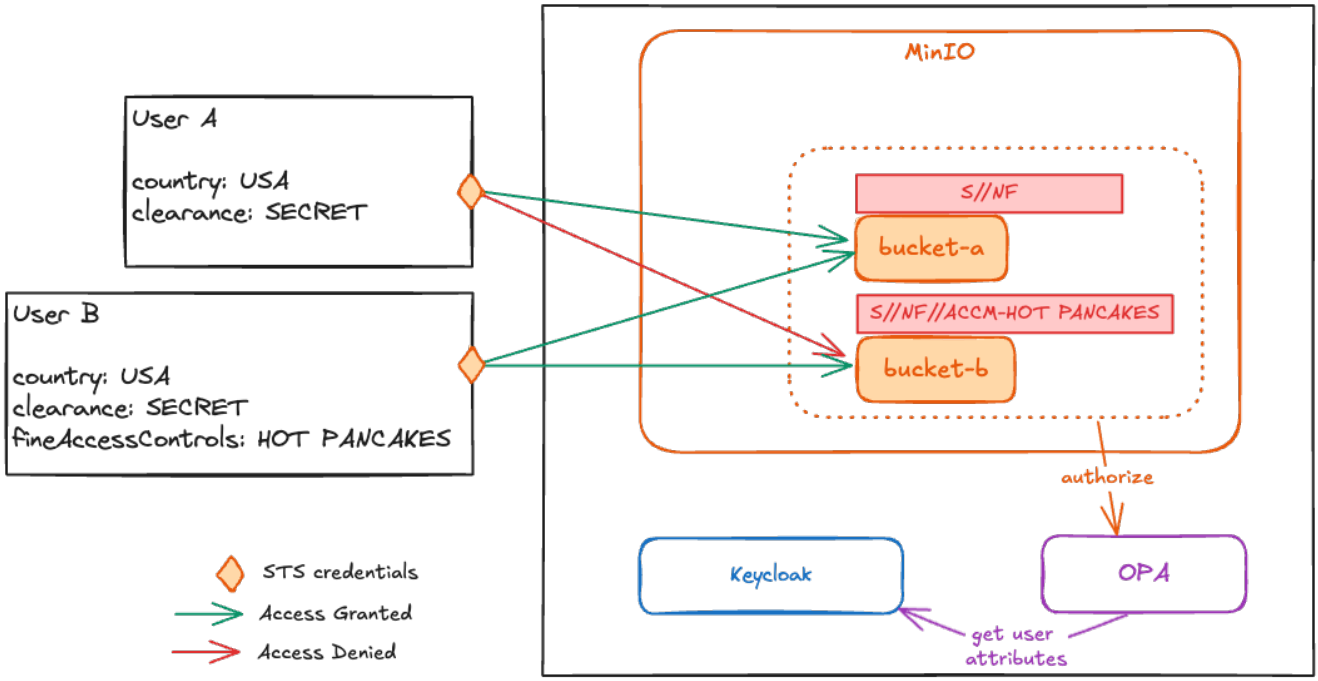
The OPA policies for S3 control access at the **object level**. That is, objects can be controlled independently even within the same bucket.

Object authorization can change dynamically, either by updating policies or by modifying the user's attributes in Keycloak.

> If a user has an active session, changes to their attributes or the governing policy take effect on their next object operation. In other words, OPA is consulted for every operation the user performs.

Classification Controls

Each object can be (optionally) marked with a classification marking using IC Classification and Control Markings.



Set Classification Marking

You can include the classification marking when uploading (or modifying) the content of an object. To do so, the user issues a `PUT` request to the `{object-endpoint}`.

Upload an Object

```
curl -X 'PUT' \
  'http://localhost/api/v1/s3/buckets/inbox-public/objects/content?objectName=my-
file.txt&classification=S%2F%2FNF' \ ①
  -H 'accept: application/json' \
  -H 'Authorization: Basic YWRtaW46eUNITHBDaG12dDRuTVUwNWpaZTZBbGl0'  ②
  -H 'Content-Type: application/json' \
  -d 'my file content'  ③
```

① Provide the marking in the `classification` URL arg (be sure to url-escape the value). Here we set it to `S//NF`.

② Can use `Basic` (username/password) or `Bearer` token.

③ The file content.

You can also set or change the classification marking on an existing object with a `PUT` request to the `/buckets/{bucket}/objects/classification` endpoint.

Set Classification

```
curl -X 'PUT' \
  'http://localhost/api/v1/s3/buckets/inbox-
public/objects/classification?objectName=my-
file.txt&classification=S%2F%2FREL%20TO%20USA%2C%20FVEY' \ ①
  -H 'accept: application/json' \
  -H 'Authorization: Basic YWRtaW46eUNITHBDaG12dDRuTVUwNWpaZTZBbGl0'
```

① Provide the *new* marking in the `classification` URL arg (be sure to url-escape the value). Here we change it to `S//REL TO USA, FVEY`.

To successfully set (or change) an object's classification,

1. The invoking user must be cleared for the marking they are providing (cannot mark beyond their own clearance).
2. If the object has an existing marking (changing marking), the user must also be cleared for the existing marking.

Get Classification Marking

To see the current classification marking for an object, a user issues a `GET` request to the `/buckets/{bucket}/objects/classification` endpoint.

Request

```
curl -X 'GET' \
  'http://localhost/api/v1/s3/buckets/inbox-
public/objects/classification?objectName=my-file.txt' \
  -H 'accept: application/json' \
  -H 'Authorization: Basic YWRtaW46eUNITHBDaG12dDRuTVUwNWpaZTZBbGl0'
```

This will return the normalized form of the marking, which includes a `"raw"` attribute containing the originally provided marking.

Response

```
{
  "raw": "S//REL TO USA, FVEY",
  "components": {
    "classification": "S",
    "disseminationControls": [
      "REL"
    ],
    "ownerProducer": [
      "USA"
    ],
    "releasableTo": [
      "USA",
```

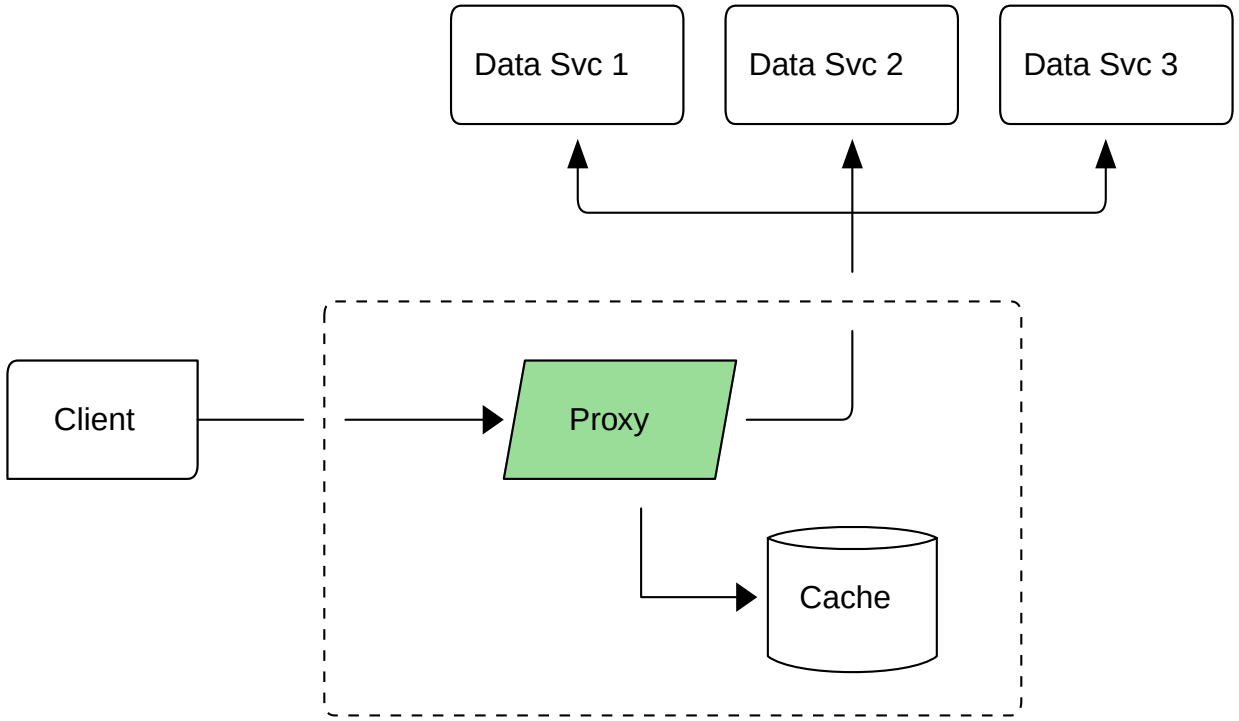
```
      "FVEY"
    ]
  }
}
```

To explore the full API, download the OpenAPI or view it in the Scalar docs.

Chapter 4. Platform Services

4.1. Proxy

SDL provides a cache-enabled REST proxy to various external services. The list of currently proxied services is documented on the root API landing.



Purpose

The proxy serves a few primary purposes for SDL clients, including:

- Central location for all data outreach (simplifying network egress controls and management).
- Ability to cache response data for reducing latency and operating in DDIL environments.
- Instrumented for observability to monitor all data request activity across all data consumer applications (clients of the proxy).

Enablement

Each proxied service must first be enabled from the Catalog. During enablement, the SDL user (or system account) provides the credentials necessary to authenticate with the target service.

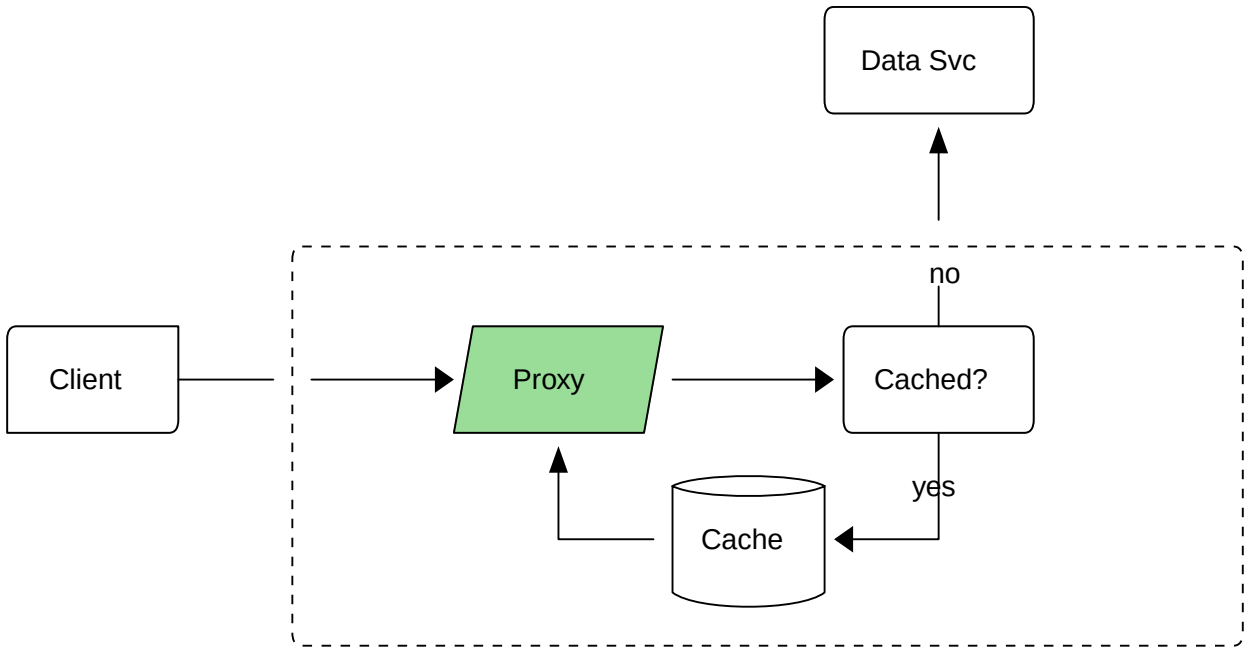
The credentials are then stored securely in a Kubernetes `Secret`, and used by SDL to invoke the target service on behalf of the user (or system account).

If a client attempts to invoke a proxied service *without first enabling it*, SDL will respond with a `407 Proxy Authentication Required` error code and a message indicating the service has not been

enabled for that client.

Caching

One of the benefits to going through the SDL proxy is leveraging the built-in cache. For any given request, the response from its upstream service is cached locally. Upon the next request for the same data, the proxy responds with the previously cached response. Once the response expires from the cache, the next request will again attempt to fetch from the upstream service (and cache the new response).



Cached Responses

If a response came from the cache, the proxy will add a `Cache-Ttl` header to the response. The value will be the time left until the response expires from the cache.

```
Cache-Ttl: 55m47s
```

The absence of `Cache-Ttl` in the response indicates the response was fetched from the upstream service.

Controlling the Cache Behavior

There may be scenarios where you want to invalidate or skip over the cache for a particular request (testing a new upstream service, forcing a cache update, etc.).

To do this, the SDL proxy supports an optional `Cache-Mode` request header which can have one of the following values.

- `Default` - Default cache behavior (same as omitting the `Cache-Mode` header all together).
- `None` - Ignore the cache completely (strictly pass-through to the upstream service). *Useful for testing if the upstream is currently available without impacting the cache.*
- `Only` - Only return the cached response. If no cached response exists, returns `404`. *Useful for avoiding any external outreach attempts.*
- `Invalidate` - Ignore any existing cached response and fetch from upstream service. Update cache with new response. *Useful for purging the cache of old data.*

4.2. Catalog

The Catalog service is the central repository for managing datasets and data sources within the SOF Data Layer. It provides APIs and interfaces for discovering, organizing, and accessing data assets across the platform.

Key Features

- **Data Source Management**: Register and manage connections to various data sources
- **Dataset Organization**: Organize datasets with metadata, labels, and hierarchical structures
- **Schema Management**: Integration with Schema Registry for schema versioning and validation
- **Dataset Filtering**: Create filtered, real-time views of datasets using powerful query expressions
- **Access Control**: Fine-grained permissions for dataset access and management

Components

API Documentation

- Dataset Schema API - Retrieve and understand dataset structures and field information
- Dataset Filtering API - Create filtered datasets with KSQL-based stream processing

User Interface

- Catalog UI Components - Web interface for browsing and managing catalog entries

Getting Started

To start using the Catalog service:

1. **Browse Available Data**: Use the Catalog UI to discover available datasets and data sources
2. **Access Dataset Schemas**: Query dataset schemas to understand data structure
3. **Create Filtered Views**: Use the Dataset Filtering API to create customized data streams
4. **Monitor Data Flow**: Track ingestion status and data pipeline health

Related Topics

- Kafka Integration
- Data Pipelines
- S3 Storage

4.2.1. UI

The Catalog directory serves as the core component for interacting with the Catalog API. It supports querying, updating, and managing both datasets and data sources. This directory provides the structure for different levels of data exploration, filtering, and management.

Page: Catalog

Description: This page provides an overview of the available data sources and datasets, with basic filtering and pagination.

ResultsCard: Displays individual catalog items.

CenteredCard: Provides a "View All" option to navigate to dedicated pages for data sources or datasets.

Page: Datasets

Description: A dedicated page for exploring datasets, offering advanced filtering, sorting, and pagination options.

Components:

- **DatasetsTable**: Displays dataset details in a tabular format.
- **MultiDatasourceInfo**: Shows detailed information about datasets.
- **DatasetContextFilter**: Provides filtering options for datasets.
- **Pagination**: Manages pagination for datasets.

Page: Datasource

Description: Dedicated to managing data sources, including sorting, filtering, and adding new sources.

Components:

- **DataSourceInfo**: Displays data source details.
- **DataSourceTable**: Renders data sources in a grid or list view.
- **AddDataSourceModal**: Allows users to add new data sources.
- **DatasourceContextFilter**: Provides filtering options for data sources.

Catalog Dynamic Directories

These dynamic route pages facilitate advanced interactions with the catalog, enabling users to search, filter, and view details about data sources and datasets. They use Next.js's dynamic routing capabilities to handle user-specific queries and parameters.

- **Dynamic Routing**: Uses Next.js's dynamic routing to handle [Query], [datasource], and [dataset] paths.
- **State Management**: Uses React's useState, useReducer, and context (UserContext) for managing data loading, filtering, and user-specific group access.
- **Pagination & Filtering**: Pagination and filtering are handled on dataset and data source pages, allowing users to customize the display and sort results based on their preferences.

Pages: Catalog [datasource] and Catalog [dataset]

Description: These pages provide detailed information about individual data sources or datasets, including an overview and related datasets. The pages are dynamic, adapting to the specific catalog entry that the user selects.

- **SourceHeader**: Displays header information about the data source or dataset.
- **Tabs**: Allows navigation between "Overview" and "Datasets" sections.
- **AccordionSection**: Provides expandable details about specific sections, such as stewardship or requirements.
- **EnablementsSelection**: Dropdown for selecting enablements related to the data source or dataset.
- **DatasetInfo**: Renders detailed information about datasets.

Page: Dynamic Dataset Pages

These pages allow users to interact with specific datasets within a given data source, providing detailed information, management options, and enablement features. The pages are organized into four sub-sections: Overview, Bucket, Topics, and Enablements.

- **Dynamic Routing**: Uses Next.js dynamic routing to handle routes like [datasource]/dataset/[dataset].
- **State Management**: Uses useReducer for managing page-specific state, with initial states and reducers defined for each sub-page.
- **Pagination & Filtering**: Provides pagination for topics and enablements, allowing users to control rows per page.
- **Enablement Toggling**: Allows users to enable or disable dataset-related enablements, with real-time updates and success/error notifications.

Page: [Dataset] Overview

Description: Provides a detailed overview of the selected dataset, including general information, enablement details, and storage options.

- **SourceHeader**: Displays the dataset's header information.
- **PageNavigation**: Navigation tabs for switching between Overview, Bucket, Topics, and Enablements.
- **DatasetBanner**: Displays additional information about the dataset.
- **AccordionSection**: Contains expandable sections for more information about enablements.
- **SetStorage**: Manages the dataset's storage settings.

Page: [Dataset] Bucket

Description: Displays the storage details for the dataset, specifically S3 buckets and Delta storage options.

- **SourceHeader**: Displays dataset header information.
- **PageNavigation**: Navigation tabs for switching between sections.
- **DatasetBanner**: Displays additional information about the dataset.
- **SetCloudStorage**: Manages cloud storage settings for the dataset.

Page: [Dataset] Topics

Description: Displays and manages Kafka topics related to the dataset.

- **SourceHeader**: Displays dataset header information.
- **PageNavigation**: Navigation tabs for switching between sections.
- **DatasetBanner**: Displays additional information about the dataset.
- **SetTopics**: Manages the topics related to the dataset.

Page: [Dataset] Enablements

Description: Provides a comprehensive view of all enablements for the selected dataset, including management options to toggle enablement statuses. Users can enable or disable specific enablements and receive feedback on success or failure.

- **SourceHeader**: Displays dataset header information.
- **PageNavigation**: Navigation tabs for switching between sections.
- **DatasetBanner**: Displays additional information about the dataset.
- **SetEnablements**: Manages enablement statuses for the dataset.

4.2.2. Dataset Schemas

The Dataset Schema API enables developers to retrieve and understand the structure of datasets in the catalog. This API integrates with the Schema Registry to provide both raw schema definitions and normalized field information for filtering and querying.

Overview

The Schema API provides two modes of operation:

- **Raw Schema Mode**: Returns the actual schema definition as stored in the Schema Registry
- **Normalized Mode**: Returns a standardized field list optimized for building filter expressions

This dual approach allows developers to:

- Access original schema definitions for data processing
- Build dynamic filter interfaces using normalized field information
- Support multiple schema formats (Avro, Protobuf, JSON Schema)
- Validate data structures before processing

API Endpoints

Get Dataset Schema

Retrieves the schema for a specific dataset.

```
GET /datasources/{datasourceId}/enablements/{enablementId}/datasets/{datasetId}/schema
```

Query Parameters

Parameter	Type	Description
raw	boolean	When `true`, returns the raw schema content. When `false` or omitted, returns normalized field information. Default: `false`

Response Types

Schema Format	Raw Mode Content-Type	Normalized Mode Content-Type
Avro	`application/json`	`application/json`
Protobuf	`application/x-protobuf-schema`	`application/json`
JSON Schema	`application/json`	`application/json`

Raw Schema Responses

When using `?raw=true`, the API returns the actual schema definition from the Schema Registry.

Avro Schema Example

Avro schemas are returned as JSON with full type information and documentation.

Request:

```
GET
/datasources/{datasourceId}/enablements/{enablementId}/datasets/{datasetId}/schema?raw
=true
```

Response Headers:

```
Content-Type: application/json
```

Response Body:

```
{
  "type": "record",
  "name": "CustomerRecord",
  "namespace": "io.raft.datafabric.customer",
  "doc": "A record representing customer data",
  "fields": [
    {
      "name": "customerId",
      "type": "long",
      "doc": "Unique customer identifier"
    },
    {
      "name": "firstName",
      "type": "string",
      "doc": "Customer's first name"
    },
    {
      "name": "lastName",
      "type": "string",
      "doc": "Customer's last name"
    },
    {
      "name": "email",
      "type": ["null", "string"],
      "default": null,
      "doc": "Customer's email address"
    },
    {
      "name": "age",
      "type": "int",
      "doc": "Customer's age"
    },
    {
      "name": "accountBalance",
      "type": {
        "type": "bytes",
        "logicalType": "decimal",
        "precision": 10,
```

```
        "scale": 2
      },
      "doc": "Customer's account balance"
    },
    {
      "name": "registrationDate",
      "type": {
        "type": "long",
        "logicalType": "timestamp-millis"
      },
      "doc": "Date when customer registered"
    },
    {
      "name": "preferences",
      "type": {
        "type": "map",
        "values": "string"
      },
      "doc": "Customer preferences"
    },
    {
      "name": "tags",
      "type": {
        "type": "array",
        "items": "string"
      },
      "doc": "Tags associated with the customer"
    },
    {
      "name": "status",
      "type": {
        "type": "enum",
        "name": "CustomerStatus",
        "symbols": ["ACTIVE", "INACTIVE", "SUSPENDED", "DELETED"]
      },
      "doc": "Customer account status"
    },
    {
      "name": "address",
      "type": ["null", {
        "type": "record",
        "name": "Address",
        "fields": [
          {"name": "street", "type": "string"},
          {"name": "city", "type": "string"},
          {"name": "state", "type": "string"},
          {"name": "postalCode", "type": "string"},
          {"name": "country", "type": "string"}
        ]
      }],
      "default": null,
```

```
      "doc": "Customer's address"
    }
  ]
}
```

Protobuf Schema Example

Protobuf schemas are returned in their text format with the appropriate content type.

Request:

```
GET
/datasources/{datasourceId}/enablements/{enablementId}/datasets/{datasetId}/schema?raw
=true
```

Response Headers:

```
Content-Type: application/x-protobuf-schema
```

Response Body:

```
syntax = "proto3";

package io.raft.datafabric.customer;

option java_package = "io.raft.datafabric.customer.proto";
option java_outer_classname = "CustomerProto";

message CustomerRecord {
  // Unique customer identifier
  int64 customer_id = 1;

  // Customer's name
  string first_name = 2;
  string last_name = 3;

  // Optional email
  optional string email = 4;

  // Customer's age
  int32 age = 5;

  // Account balance with decimal precision
  double account_balance = 6;

  // Registration timestamp (Unix epoch milliseconds)
  int64 registration_date = 7;
```

```
  // Customer preferences as key-value pairs
  map<string, string> preferences = 8;

  // Tags associated with the customer
  repeated string tags = 9;

  // Customer status
  enum CustomerStatus {
    CUSTOMER_STATUS_UNSPECIFIED = 0;
    CUSTOMER_STATUS_ACTIVE = 1;
    CUSTOMER_STATUS_INACTIVE = 2;
    CUSTOMER_STATUS_SUSPENDED = 3;
    CUSTOMER_STATUS_DELETED = 4;
  }
  CustomerStatus status = 10;

  // Nested address message
  message Address {
    string street = 1;
    string city = 2;
    string state = 3;
    string postal_code = 4;
    string country = 5;
  }

  // Optional address
  optional Address address = 11;

  // Contact preference using oneof
  oneof contact_method {
    string phone = 12;
    string mobile = 13;
    string work_phone = 14;
  }

  // Nested repeated messages for order history
  message Order {
    string order_id = 1;
    int64 order_date = 2;
    double total_amount = 3;

    enum OrderStatus {
      ORDER_STATUS_UNSPECIFIED = 0;
      ORDER_STATUS_PENDING = 1;
      ORDER_STATUS_PROCESSING = 2;
      ORDER_STATUS_SHIPPED = 3;
      ORDER_STATUS_DELIVERED = 4;
      ORDER_STATUS_CANCELLED = 5;
    }
    OrderStatus status = 4;
  }
```

```
  repeated Order orders = 15;
}
```

JSON Schema Example

JSON schemas follow the JSON Schema specification and include validation rules.

Request:

```
GET
/datasources/{datasourceId}/enablements/{enablementId}/datasets/{datasetId}/schema?raw
=true
```

Response Headers:

```
Content-Type: application/json
```

Response Body:

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "$id": "https://raft.io/schemas/customer-record.json",
  "title": "Customer Record",
  "description": "A record representing customer data",
  "type": "object",
  "required": ["customerId", "firstName", "lastName", "age", "status"],
  "properties": {
    "customerId": {
      "type": "integer",
      "description": "Unique customer identifier",
      "minimum": 1
    },
    "firstName": {
      "type": "string",
      "description": "Customer's first name",
      "minLength": 1,
      "maxLength": 100
    },
    "lastName": {
      "type": "string",
      "description": "Customer's last name",
      "minLength": 1,
      "maxLength": 100
    },
    "email": {
      "type": ["string", "null"],
      "description": "Customer's email address",
```

```
      "format": "email",
      "default": null
    },
    "age": {
      "type": "integer",
      "description": "Customer's age",
      "minimum": 0,
      "maximum": 150
    },
    "accountBalance": {
      "type": "number",
      "description": "Customer's account balance",
      "multipleOf": 0.01
    },
    "registrationDate": {
      "type": "string",
      "description": "Date when customer registered",
      "format": "date-time"
    },
    "preferences": {
      "type": "object",
      "description": "Customer preferences",
      "additionalProperties": {
        "type": "string"
      }
    },
    "tags": {
      "type": "array",
      "description": "Tags associated with the customer",
      "items": {
        "type": "string"
      },
      "uniqueItems": true
    },
    "status": {
      "type": "string",
      "description": "Customer account status",
      "enum": ["ACTIVE", "INACTIVE", "SUSPENDED", "DELETED"]
    },
    "address": {
      "type": ["object", "null"],
      "description": "Customer's address",
      "default": null,
      "properties": {
        "street": {"type": "string"},
        "city": {"type": "string"},
        "state": {"type": "string"},
        "postalCode": {"type": "string", "pattern": "^[0-9]{5}(-[0-9]{4})?$"},
        "country": {"type": "string"}
      },
      "required": ["street", "city", "country"]
```

```
    }
  },
  "additionalProperties": false
}
```

Normalized Schema Response

When `raw` is `false` or omitted, the API returns a normalized `SchemaInfo` object containing field information optimized for building filter expressions.

Request:

```
GET /datasources/{datasourceId}/enablements/{enablementId}/datasets/{datasetId}/schema
```

Response:

```
{
  "schemaId": "123",
  "schemaType": "AVRO",
  "fields": [
    {
      "path": "customerId",
      "name": "customerId",
      "type": "LONG",
      "description": "Unique customer identifier"
    },
    {
      "path": "firstName",
      "name": "firstName",
      "type": "STRING",
      "description": "Customer's first name"
    },
    {
      "path": "lastName",
      "name": "lastName",
      "type": "STRING",
      "description": "Customer's last name"
    },
    {
      "path": "email",
      "name": "email",
      "type": "STRING",
      "nullable": true,
      "description": "Customer's email address"
    },
    {
      "path": "age",
      "name": "age",
      "type": "INTEGER",
```

```
      "description": "Customer's age"
    },
    {
      "path": "status",
      "name": "status",
      "type": "STRING",
      "enumValues": ["ACTIVE", "INACTIVE", "SUSPENDED", "DELETED"],
      "description": "Customer account status"
    },
    {
      "path": "address.street",
      "name": "street",
      "type": "STRING",
      "nullable": true,
      "description": "Street address"
    },
    {
      "path": "address.city",
      "name": "city",
      "type": "STRING",
      "nullable": true,
      "description": "City"
    },
    {
      "path": "tags",
      "name": "tags",
      "type": "ARRAY",
      "description": "Tags associated with the customer"
    },
    {
      "path": "tags[]",
      "name": "tags[]",
      "type": "STRING",
      "description": "Array element"
    }
  ]
}
```

Field Type Mapping

The normalized response maps schema-specific types to a common set of field types:

Normalized Type	Avro Types	Protobuf Types	JSON Schema Types
STRING	string, enum	string, enum	string
INTEGER	int, long	int32, int64, sint32, sint64, fixed32, fixed64	integer
DOUBLE	float, double, decimal	float, double	number
BOOLEAN	boolean	bool	boolean

Normalized Type	Avro Types	Protobuf Types	JSON Schema Types
ARRAY	array	repeated fields (shown as field[] in API)	array
MAP	map	map	object (with additionalProperties)
RECORD	record	message	object

Nested Field Access

Nested fields are flattened using dot notation in the `path` property:

- `address.street` - Access the street field within the address object
- `orders[].orderId` - Access orderId within an array of orders (the `orders[]` entry itself would have type ARRAY)
- `metadata.tags[]` - Access an array of tags within metadata

Array fields are represented with two entries: 1. The array field itself (e.g., `tags` with type `ARRAY`) 2. The array element type (e.g., `tags[]` with the element's type like `STRING`, `OBJECT`, etc.)

When filtering, you can use specific array indices (e.g., `tags[0]`, `tags[5]`) even though the schema only defines the general pattern `tags[]`. The filtering API automatically validates these indexed paths against the array element definition.

Error Responses

The API returns appropriate HTTP status codes and error messages for various failure scenarios.

Dataset Not Found (404)

```
{
  "timestamp": "2025-07-27T13:30:45.123Z",
  "status": 404,
  "error": "Not Found",
  "message": "Dataset not found: d456",
  "path": "/datasources/ds123/enablements/e123/datasets/d456/schema"
}
```

Schema Not Found (404)

```
{
  "timestamp": "2025-07-27T13:30:45.123Z",
  "status": 404,
  "error": "Not Found",
  "message": "Schema not found in registry: schema-123",
  "path": "/datasources/ds123/enablements/e123/datasets/d456/schema"
}
```

Dataset Has No Schema (404)

```
{
  "timestamp": "2025-07-27T13:30:45.123Z",
  "status": 404,
  "error": "Not Found",
  "message": "Dataset has no Kafka storage with schema",
  "path": "/datasources/ds123/enablements/e123/datasets/d456/schema"
}
```

Schema Registry Unavailable (503)

```
{
  "timestamp": "2025-07-27T13:30:45.123Z",
  "status": 503,
  "error": "Service Unavailable",
  "message": "Schema Registry is currently unavailable",
  "path": "/datasources/ds123/enablements/e123/datasets/d456/schema"
}
```

Invalid Schema Format (500)

```
{
  "timestamp": "2025-07-27T13:30:45.123Z",
  "status": 500,
  "error": "Internal Server Error",
  "message": "Failed to parse schema: Invalid Avro schema format",
  "path": "/datasources/ds123/enablements/e123/datasets/d456/schema"
}
```

Usage Examples

Example 1: Building a Dynamic Filter UI

When building a user interface for dataset filtering, use the normalized schema response:

```
// Fetch normalized schema
const response = await fetch('/datasources/ds1/enablements/e1/datasets/d1/schema');
const schemaInfo = await response.json();

// Build filter options from fields
// Note: Array element fields (those with [] in the path) are typically
// not used directly in filters - use the parent array field instead
const filterableFields = schemaInfo.fields
  .filter(field => !field.path.includes('[]'))  // Exclude array elements
  .map(field => ({
    label: field.description || field.name,
```

```
    value: field.path,
    type: field.type,
    enumValues: field.enumValues
  }));

// For array fields, you can allow users to specify indices
function getArrayFieldsWithIndices(schemaInfo) {
  const arrayFields = schemaInfo.fields
    .filter(field => field.type === 'ARRAY');

  const arrayElementFields = schemaInfo.fields
    .filter(field => field.path.includes('[]'));

  // For each array field, find its element fields
  return arrayFields.map(arrayField => {
    const elementFields = arrayElementFields
      .filter(ef => ef.path.startsWith(arrayField.path + '[]'))
      .map(ef => ({
        ...ef,
        // Allow user to specify index
        pathTemplate: ef.path.replace('[]', '[${index}]')
      }));

    return {
      arrayField,
      elementFields
    };
  });
}

// Create appropriate input based on field type
function createFilterInput(field) {
  switch(field.type) {
    case 'STRING':
      return field.enumValues
        ? createDropdown(field.enumValues)
        : createTextInput();
    case 'INTEGER':
    case 'DOUBLE':
      return createNumberInput();
    case 'BOOLEAN':
      return createCheckbox();
    default:
      return createTextInput();
  }
}
```

Example 2: Validating Data Before Publishing

Use the raw schema to validate data before publishing to Kafka:

```
// Fetch raw Avro schema
HttpResponse<String> response = httpClient.send(
    HttpRequest.newBuilder()

.uri(URI.create("/datasources/ds1/enablements/e1/datasets/d1/schema?raw=true"))
        .build(),
    HttpResponse.BodyHandlers.ofString()
);

// Parse and use for validation
Schema schema = new Schema.Parser().parse(response.body());
GenericDatumReader<GenericRecord> reader = new GenericDatumReader<>(schema);

// Validate data against schema
try {
    GenericRecord record = reader.read(null, decoder);
    // Data is valid
} catch (AvroTypeException e) {
    // Data does not match schema
}
```

Example 3: Schema Evolution Monitoring

Monitor schema changes by comparing raw schemas:

```
# Get current schema
curl -s "/datasources/ds1/enablements/e1/datasets/d1/schema?raw=true" \
  | jq . > current_schema.json

# Compare with previous version
diff previous_schema.json current_schema.json

# Check for breaking changes
if grep -q '"type".*"null"' current_schema.json; then
  echo "Warning: New nullable fields detected"
fi
```

Best Practices

1. **Cache Schema Information**: Schemas change infrequently. Cache normalized schema responses to reduce API calls.
2. **Handle Schema Evolution**: Always handle nullable fields and new fields gracefully in your applications.
3. **Use Appropriate Mode**:
   - Use normalized mode (`raw=false`) for building UIs and filter expressions
   - Use raw mode (`raw=true`) for data validation and processing

4. **Error Handling**: Implement retry logic for 503 errors, as Schema Registry may be temporarily unavailable.
5. **Field Path Navigation**: Use the dot notation paths from normalized responses to access nested fields in your data.

Schema Registry Integration

The Dataset Schema API integrates with Confluent Schema Registry to provide:

- Centralized schema storage and versioning
- Schema evolution and compatibility checking
- Multiple format support (Avro, Protobuf, JSON Schema)

Datasets must have Kafka storage configured with a schema ID to use this API. The schema ID links the dataset to its schema definition in the registry.

Related Topics

- Catalog Overview
- Dataset Filtering API
- Kafka Integration
- Confluent Schema Registry Documentation

4.2.3. Dataset Filtering

The Dataset Filtering API enables developers to create filtered datasets from existing Kafka-based datasets. This powerful feature allows for real-time stream processing using KSQL queries generated from intuitive filter expressions.

Overview

Dataset filtering provides:

- Schema-aware validation of filter expressions
- Automatic KSQL query generation for stream processing
- Support for complex filter combinations
- Dry-run mode for testing filters before deployment
- Comprehensive operator support for various data types

Prerequisites

Before using the filtering API, ensure:

- The source dataset has Kafka storage configured
- The dataset has a valid schema registered in the Schema Registry

- You have the necessary permissions to create filtered datasets

API Endpoints

Create Filtered Dataset

Creates a new filtered dataset from an existing dataset.

```
POST
/datasources/{datasourceId}/enablements/{enablementId}/datasets/{datasetId}/filters
```

Request Body

```
{
  "name": "Filtered Dataset Name",
  "acronym": "FDS",
  "description": "Description of the filtered dataset",
  "labels": ["tag1", "tag2"],
  "autoStart": true,
  "filter": {
    // Filter expression (see Filter Expressions section)
  }
}
```

Response

```
{
  "datasourceId": "550e8400-e29b-41d4-a716-446655440001",
  "enablementId": "550e8400-e29b-41d4-a716-446655440002",
  "datasetId": "550e8400-e29b-41d4-a716-446655440003",
  "ingestionStarted": true,
  "ksqlQuery": "SELECT * FROM d WHERE d.status = 'ACTIVE'"
}
```

Validation (Dry Run)

Test filter expressions without creating a dataset by adding the `dryRun` parameter.

```
POST
/datasources/{datasourceId}/enablements/{enablementId}/datasets/{datasetId}/filters?dr
yRun=true
```

Response

```
{
  "valid": true,
```

```
  "ksqlQuery": "SELECT * FROM d WHERE d.status = 'ACTIVE'",
  "warnings": [],
  "errors": []
}
```

Filter Expressions

Filter expressions define the criteria for selecting records from the source dataset. There are two types:

Filter Predicate

A basic condition that evaluates a single field.

```
{
  "field": "status",
  "operator": "EQUALS",
  "value": "ACTIVE"
}
```

Compound Filter

Combines multiple filters using logical operators.

```
{
  "operator": "AND",
  "filters": [
    {
      "field": "status",
      "operator": "EQUALS",
      "value": "ACTIVE"
    },
    {
      "field": "priority",
      "operator": "GREATER_THAN",
      "value": 5
    }
  ]
}
```

Filter Operators

Equality Operators

EQUALS

Tests if a field equals a specific value.

Supported Types	All field types (STRING, INTEGER, DOUBLE, BOOLEAN)
Example Filter	`{ "field": "status", "operator": "EQUALS", "value": "ACTIVE" }`
Generated KSQL	`SELECT * FROM d WHERE d.status = 'ACTIVE'`

NOT_EQUALS

Tests if a field does not equal a specific value.

Supported Types	All field types
Example Filter	`{ "field": "status", "operator": "NOT_EQUALS", "value": "INACTIVE" }`
Generated KSQL	`SELECT * FROM d WHERE d.status != 'INACTIVE'`

Comparison Operators

GREATER_THAN

Tests if a field is greater than a value.

Supported Types	INTEGER, DOUBLE, STRING (lexicographic)
Example Filter	`{ "field": "priority", "operator": "GREATER_THAN", "value": 5 }`
Generated KSQL	`SELECT * FROM d WHERE d.priority > 5`

LESS_THAN

Tests if a field is less than a value.

Supported Types	INTEGER, DOUBLE, STRING (lexicographic)
Example Filter	`{ "field": "temperature", "operator": "LESS_THAN", "value": 32.0 }`
Generated KSQL	`SELECT * FROM d WHERE d.temperature < 32.0`

GREATER_THAN_OR_EQUALS

Tests if a field is greater than or equal to a value.

Supported Types	INTEGER, DOUBLE, STRING (lexicographic)
Example Filter	`{ "field": "score", "operator": "GREATER_THAN_OR_EQUALS", "value": 80 }`
Generated KSQL	`SELECT * FROM d WHERE d.score >= 80`

LESS_THAN_OR_EQUALS

Tests if a field is less than or equal to a value.

Supported Types	INTEGER, DOUBLE, STRING (lexicographic)
Example Filter	`{ "field": "age", "operator": "LESS_THAN_OR_EQUALS", "value": 65 }`
Generated KSQL	`SELECT * FROM d WHERE d.age <= 65`

Range Operators

BETWEEN

Tests if a field value falls within a range (inclusive).

Supported Types	INTEGER, DOUBLE, STRING (lexicographic)
Example Filter	`{ "field": "temperature", "operator": "BETWEEN", "lowerBound": 20, "upperBound": 30 }`
Generated KSQL	`SELECT * FROM d WHERE d.temperature BETWEEN 20 AND 30`

NOT_BETWEEN

Tests if a field value falls outside a range.

Supported Types	INTEGER, DOUBLE, STRING (lexicographic)
Example Filter	`{ "field": "age", "operator": "NOT_BETWEEN", "lowerBound": 18, "upperBound": 65 }`
Generated KSQL	`SELECT * FROM d WHERE d.age NOT BETWEEN 18 AND 65`

Set Operators

IN

Tests if a field matches any value in a list.

Supported Types	All field types
Example Filter	`{ "field": "category", "operator": "IN", "values": ["electronics", "computers", "phones"] }`
Generated KSQL	`SELECT * FROM d WHERE d.category IN ('electronics', 'computers', 'phones')`

NOT_IN

Tests if a field does not match any value in a list.

Supported Types	All field types
Example Filter	`{ "field": "status", "operator": "NOT_IN", "values": ["CANCELLED", "FAILED", "EXPIRED"] }`
Generated KSQL	`SELECT * FROM d WHERE d.status NOT IN ('CANCELLED', 'FAILED', 'EXPIRED')`

Null Operators

IS_NULL

Tests if a field is null.

Supported Types	All field types
Example Filter	`{ "field": "deletedAt", "operator": "IS_NULL" }`
Generated KSQL	`SELECT * FROM d WHERE d.deletedAt IS NULL`

IS_NOT_NULL

Tests if a field is not null.

Supported Types	All field types
Example Filter	`{ "field": "email", "operator": "IS_NOT_NULL" }`
Generated KSQL	`SELECT * FROM d WHERE d.email IS NOT NULL`

String Pattern Operators

CONTAINS

Tests if a string field contains a substring.

Supported Types	STRING only
Example Filter	`{ "field": "description", "operator": "CONTAINS", "value": "urgent" }`
Generated KSQL	`SELECT * FROM d WHERE d.description LIKE '%urgent%'`

STARTS_WITH

Tests if a string field starts with a prefix.

Supported Types	STRING only
Example Filter	`{ "field": "name", "operator": "STARTS_WITH", "value": "John" }`
Generated KSQL	`SELECT * FROM d WHERE d.name LIKE 'John%'`

ENDS_WITH

Tests if a string field ends with a suffix.

Supported Types	STRING only
Example Filter	`{ "field": "email", "operator": "ENDS_WITH", "value": "@company.com" }`
Generated KSQL	`SELECT * FROM d WHERE d.email LIKE '%@company.com'`

NOT_CONTAINS

Tests if a string field does not contain a substring.

This operator is validated but not currently supported in KSQL generation.

MATCHES_REGEX

Tests if a string field matches a regular expression.

This operator is validated but not currently supported in KSQL generation.

Geographic Operators

Geographic operators enable location-based filtering. While these operators are validated by the API, they are not currently supported in KSQL generation and will produce placeholder comments in the generated query.

WITHIN_RADIUS

Tests if coordinates are within a specified radius of a center point.

Required Fields	**geoFields (latitude/longitude mapping), center, radius**
Example Filter	`{ "operator": "WITHIN_RADIUS", "geoFields": { "latitudeField": "lat", "longitudeField": "lon", "altitudeField": "alt" }, "center": { "latitude": 40.7128, "longitude": -74.0060 }, "radius": 10000 }`
Generated KSQL	`TBD`

WITHIN_BOUNDS

Tests if coordinates are within a geographic bounding box.

Required Fields	geoFields (latitude/longitude mapping), bounds

Example Filter	<pre>{
	  "operator": "WITHIN_BOUNDS",
	  "geoFields": {
	  "latitudeField": "lat",
	  "longitudeField": "lon"
	  },
	  "bounds": {
	  "northLatitude": 40.917577,
	  "southLatitude": 40.477399,
	  "eastLongitude": -73.700272,
	  "westLongitude": -74.259090
	  }
	}</pre>

Generated KSQL	TBD

WITHIN_MGRS_GRID

Tests if an MGRS (Military Grid Reference System) field matches a grid reference.

Required Fields	field, mgrsGrid

Example Filter	<pre>{
	  "field": "mgrsLocation",
	  "operator": "WITHIN_MGRS_GRID",
	  "mgrsGrid": "18TWL"
	}</pre>

Generated KSQL	TBD

Compound Filters

Compound filters combine multiple filter expressions using logical operators.

AND Operator

All conditions must be true.

```
{
  "operator": "AND",
  "filters": [
    {
      "field": "status",
      "operator": "EQUALS",
      "value": "ACTIVE"
```

```
    },
    {
      "field": "priority",
      "operator": "GREATER_THAN",
      "value": 5
    }
  ]
}
```

Generated KSQL:

```
SELECT * FROM d WHERE (d.status = 'ACTIVE') AND (d.priority > 5)
```

OR Operator

At least one condition must be true.

```
{
  "operator": "OR",
  "filters": [
    {
      "field": "status",
      "operator": "EQUALS",
      "value": "FAILED"
    },
    {
      "field": "retryCount",
      "operator": "GREATER_THAN",
      "value": 3
    }
  ]
}
```

Generated KSQL:

```
SELECT * FROM d WHERE (d.status = 'FAILED') OR (d.retryCount > 3)
```

NOT Operator

Negates a filter condition. Must contain exactly one filter.

```
{
  "operator": "NOT",
  "filters": [
    {
      "field": "environment",
```

```
      "operator": "EQUALS",
      "value": "production"
    }
  ]
}
```

Generated KSQL:

```
SELECT * FROM d WHERE NOT (d.environment = 'production')
```

Field Type Compatibility

The following table shows which operators are compatible with each field type:

Operator	STRING	INTEGER	DOUBLE	BOOLEAN
EQUALS	✔	✔	✔	✔
NOT_EQUALS	✔	✔	✔	✔
GREATER_THAN	✔	✔	✔	✘
LESS_THAN	✔	✔	✔	✘
GREATER_THAN_OR_EQUALS	✔	✔	✔	✘
LESS_THAN_OR_EQUALS	✔	✔	✔	✘
BETWEEN	✔	✔	✔	✘
NOT_BETWEEN	✔	✔	✔	✘
IN	✔	✔	✔	✔
NOT_IN	✔	✔	✔	✔
IS_NULL	✔	✔	✔	✔
IS_NOT_NULL	✔	✔	✔	✔
CONTAINS	✔	✘	✘	✘
STARTS_WITH	✔	✘	✘	✘
ENDS_WITH	✔	✘	✘	✘
NOT_CONTAINS	✔	✘	✘	✘
MATCHES_REGEX	✔	✘	✘	✘

Nested Field Access

Access nested fields in complex JSON structures using dot notation:

```
{
```

```
  "field": "user.profile.age",
  "operator": "GREATER_THAN",
  "value": 18
}
```

Generated KSQL:

```
SELECT * FROM d WHERE d.user.profile.age > 18
```

Array Field Access

Arrays can be filtered in two ways:

Filtering the Array Field Directly

When filtering on the array field itself, use operators like `CONTAINS` to check if any element matches:

```
{
  "field": "tags",
  "operator": "CONTAINS",
  "value": "urgent"
}
```

Generated KSQL:

```
SELECT * FROM d WHERE d.tags LIKE '%urgent%'
```

Filtering Specific Array Indices

When a schema defines array fields using the `[]` notation (e.g., `measurements[].value`), you can filter on specific array indices without the schema explicitly declaring each index:

```
{
  "field": "measurements[0].value",
  "operator": "GREATER_THAN",
  "value": 100.0
}
```

Generated KSQL:

```
SELECT * FROM d WHERE d.measurements[0].value > 100.0
```

This works for any valid array index: * `items[0].name` - First item * `items[5].price` - Sixth item * `locations[2].coordinates.latitude` - Nested field in third location

The filter validation automatically recognizes array index patterns and validates them against the schema's array element definitions.

Example: Comparing Array Filtering Approaches

Given a schema with an array of sensor readings:

```
// Schema fields:
// - sensors (type: ARRAY)
// - sensors[].id (type: STRING)
// - sensors[].temperature (type: DOUBLE)
// - sensors[].status (type: STRING)
```

Option 1: Filter any sensor (uses array field directly)

```
{
  "field": "sensors",
  "operator": "CONTAINS",
  "value": "ALARM"
}
```

Option 2: Filter specific sensor (uses array index)

```
{
  "field": "sensors[0].status",
  "operator": "EQUALS",
  "value": "ALARM"
}
```

Option 3: Complex filter on multiple specific sensors

```
{
  "operator": "OR",
  "filters": [
    {
      "field": "sensors[0].temperature",
      "operator": "GREATER_THAN",
      "value": 100
    },
    {
      "field": "sensors[1].temperature",
      "operator": "GREATER_THAN",
      "value": 100
    }
  ]
}
```

Error Handling

The API provides detailed validation feedback:

```
{
  "valid": false,
  "errors": [
    "Field 'nonExistentField' does not exist in schema",
    "Operator 'CONTAINS' is not compatible with field type 'INTEGER'"
  ],
  "warnings": [
    "OR filters may impact performance on large datasets"
  ]
}
```

Common validation errors include:

- Field does not exist in schema
- Operator not compatible with field type
- Missing required parameters
- Dataset has no Kafka storage
- Schema not found or unavailable

Complete Examples

Active Premium Users Filter

Filter for active premium users in specific regions with recent activity:

```
{
  "name": "Active Premium Users - US West",
  "filter": {
    "operator": "AND",
    "filters": [
      {
        "field": "status",
        "operator": "EQUALS",
        "value": "ACTIVE"
      },
      {
        "field": "subscription.type",
        "operator": "EQUALS",
        "value": "PREMIUM"
      },
      {
        "field": "region",
        "operator": "IN",
        "values": ["us-west-1", "us-west-2"]
```

```
      },
      {
        "field": "lastActivityDate",
        "operator": "GREATER_THAN",
        "value": "2024-01-01"
      },
      {
        "field": "deletedAt",
        "operator": "IS_NULL"
      }
    ]
  }
}
```

Error Event Detection

Filter for events indicating errors or requiring attention:

```
{
  "name": "Error Events",
  "filter": {
    "operator": "OR",
    "filters": [
      {
        "field": "level",
        "operator": "IN",
        "values": ["ERROR", "CRITICAL", "FATAL"]
      },
      {
        "operator": "AND",
        "filters": [
          {
            "field": "httpStatus",
            "operator": "BETWEEN",
            "lowerBound": 500,
            "upperBound": 599
          },
          {
            "field": "retryCount",
            "operator": "GREATER_THAN",
            "value": 0
          }
        ]
      },
      {
        "field": "message",
        "operator": "CONTAINS",
        "value": "Exception"
      }
    ]
```

```
  }
}
```

High-Value Customer Segmentation

Filter for high-value customers for targeted marketing:

```
{
  "name": "High Value Customers",
  "filter": {
    "operator": "AND",
    "filters": [
      {
        "operator": "OR",
        "filters": [
          {
            "field": "totalPurchases",
            "operator": "GREATER_THAN",
            "value": 10000
          },
          {
            "field": "customerTier",
            "operator": "IN",
            "values": ["GOLD", "PLATINUM"]
          }
        ]
      },
      {
        "field": "accountAge",
        "operator": "GREATER_THAN",
        "value": 365
      },
      {
        "field": "email",
        "operator": "ENDS_WITH",
        "value": "@corporate.com"
      },
      {
        "operator": "NOT",
        "filters": [
          {
            "field": "optedOut",
            "operator": "EQUALS",
            "value": true
          }
        ]
      }
    ]
  }
```

```
}
```

Sensor Data Monitoring

Filter for anomalous readings from specific sensors in an array:

```
{
  "name": "Anomalous Sensor Readings",
  "filter": {
    "operator": "OR",
    "filters": [
      {
        "operator": "AND",
        "filters": [
          {
            "field": "sensors[0].temperature",
            "operator": "GREATER_THAN",
            "value": 85.0
          },
          {
            "field": "sensors[0].status",
            "operator": "EQUALS",
            "value": "ACTIVE"
          }
        ]
      },
      {
        "field": "sensors[1].pressure",
        "operator": "NOT_BETWEEN",
        "lowerBound": 25.0,
        "upperBound": 35.0
      },
      {
        "field": "alerts[0].severity",
        "operator": "IN",
        "values": ["HIGH", "CRITICAL"]
      }
    ]
  }
}
```

This example demonstrates: - Filtering on specific array indices (`sensors[0]`, `sensors[1]`) - Accessing nested fields within array elements (`sensors[0].temperature`) - Combining array index filters with other operators

Array Filtering Comparison

Here's how different array filtering approaches work for various use cases:

Use Case 1: Find documents with a specific tag anywhere in the array

```
{
  "name": "Documents with Urgent Tag",
  "filter": {
    "field": "tags",
    "operator": "CONTAINS",
    "value": "urgent"
  }
}
```

Use Case 2: Check if the first author is from a specific department

```
{
  "name": "Primary Author from Engineering",
  "filter": {
    "field": "authors[0].department",
    "operator": "EQUALS",
    "value": "Engineering"
  }
}
```

Use Case 3: Complex filtering - any high-priority alert OR first sensor overheating

```
{
  "name": "Critical Conditions",
  "filter": {
    "operator": "OR",
    "filters": [
      {
        "field": "alerts",
        "operator": "CONTAINS",
        "value": "high-priority"
      },
      {
        "field": "sensors[0].temperature",
        "operator": "GREATER_THAN",
        "value": 100
      }
    ]
  }
}
```

Best Practices

1. **Test with Dry Run**: Always validate filters using `dryRun=true` before creating filtered datasets
2. **Consider Performance**: Complex OR filters and nested conditions may impact stream

processing performance

3. **Use Appropriate Types**: Ensure filter values match the field types in your schema
4. **Leverage Nested Access**: Use dot notation for filtering on nested JSON structures

Related Topics

- Catalog Overview
- Kafka Integration
- Data Pipelines

4.3. Database as a Service

This guide will walk you through the process of requesting a new managed PostgreSQL database instance.

Prerequisites

- Valid SDL account credentials

Steps

1. Log into SDL web client
2. Navigate to the Forms page Click on the Data Collection menu item to the left.
3. Locate and click on the "Managed Database Request" form
4. Fill out the request form with the requested information.
5. Review your inputs and submit the form

What Happens Next?

After submission:

1. The Raft team will review your request
2. The Raft team will schedule a quick TEM to go over your requirements.
3. Upon approval, you will receive connection details and credentials
4. Standard provisioning time is approximately 1 week.

4.4. Operational Monitoring

Introduction

Purpose

This documentation serves as a comprehensive guide to using Grafana for application activity monitoring, including health checks and general usage telemetry, specifically utilizing Prometheus

for metrics and Loki for log aggregation. It covers the setup process, key functionalities, and advanced features to enable users to monitor application and infrastructure health, performance, and usage.

Target Audience

This guide is designed for users of the SDL Application who need to monitor and analyze system health, performance, and usage metrics. It is aimed at both technical and non-technical users, including developers, operators, and data analysts, who want to gain insights into the application's activity and performance using Grafana dashboards.

Scope

This guide outlines how to set up Grafana, integrate Prometheus for metrics, and Loki for log monitoring, create dashboards, configure alerts, and troubleshoot common issues. The document also highlights use cases for monitoring application performance and provides best practices for scalable and efficient dashboard design.

Overview

Capability Description

Grafana is a powerful tool for visualizing and monitoring data from multiple sources, like Prometheus and Loki. It allows users to create custom dashboards for real-time and historical analysis, helping track system performance and detect issues. Key features include multi-source integration, dynamic dashboards, and alerting to ensure system reliability and quick issue resolution.

Key Features

- **Metrics Monitoring with Prometheus**: Collect and visualize CPU usage, memory consumption, request rates, and other telemetry data.
- **Log Aggregation with Loki**: Centralized logging, making it easy to correlate logs with metrics for comprehensive monitoring.
- **Alerting**: Trigger alerts when specific thresholds are met, such as high error rates or degraded system performance.
- **Multi-Source Integration**: Support for multiple data sources allows you to combine metrics (from Prometheus) with logs (from Loki) in a single dashboard.
- **Custom Dashboards**: Build flexible and intuitive dashboards for both real-time and historical data analysis.

Setup

The following steps guide you through the initial setup process to start using Grafana for monitoring and visualization within Raft's SDL:

Step 1: Accessing Grafana

There are two ways to navigate to Grafana: * When you first log in, you will see the Grafana tile on the Home page.

- You can also locate Grafana under the Data Insights tab on the left.

Ensure your login credentials are set up with Keycloak for authentication.

Step 2: Configuring Authentication

Log in using your Keycloak credentials, which manage access to dashboards and data sources.

Step 3: Grafana Buttons

- **General / Home**: Links to Grafana documentation, tutorials, community, and public Slack.
- **Search dashboards by name**
- **Your starred dashboards**
- **Dashboards**: Browse, Playlists, Snapshots, Library Panels, + New dashboard, + New folder, + Import
- **Explore**
- **Alerting**: Alert rules, Contact points, Notification policies, Silences, Groups, Admins, + New alert rule
- **Configurations**: Service accounts, API Keys, Preferences, Plugins, Teams, Users, Data sources
- **User profile**: Sign out, Notification history, Preferences

The **Help** option is also available under General / Home.

Step 4: Adding Data Sources

After logging in, navigate to **Configurations** > **Data sources** to add a new data source.

Grafana supports multiple data sources. SDL includes:

- Prometheus for DF system metrics
- Loki for DF log aggregation

Configure the connection with the following details:

- Server address (URL or IP of the data source)
- Authentication details (e.g., API key, credentials, or OAUTH)
- Query parameters as required for the specific data source

```
# Example server address configuration
server_address: "http://localhost:9090"
```

```
authentication: "OAUTH"
```

Step 5: Verifying Data Source Connections

Once configured, use the “Test” button to ensure that Grafana can successfully connect to the data source.

If the test fails, verify your authentication credentials and data source server address.

Step 6: Initial Dashboard Setup

Grafana allows you to create a dashboard immediately after setting up your data source.

1. Navigate to **Dashboards** > **New Dashboard**.
2. Add a new panel and select your configured data source.
3. Customize the panel by selecting the query, time range, and visualization type.
4. Save your dashboard to begin real-time monitoring.

Usage

Getting Started

Grafana’s flexibility makes it easy to create dashboards by connecting to pre-built integrations like Prometheus for metrics (CPU, memory, request rates) and Loki for logs, providing a comprehensive view of application activity.

Beyond these built-in integrations, Grafana supports a wide range of custom data sources, allowing connections to databases, APIs, and systems like MySQL, PostgreSQL, or custom APIs. With the right data source plugin, you can monitor real-time metrics, visualize trends, and ensure your infrastructure’s health. Additionally, you can extend Grafana in the SDL App by integrating external sources like additional databases, logs, or third-party APIs. This enables centralized monitoring of both SDL and external systems, offering a broader perspective on system metrics.

Grafana’s robust alerting system lets you set thresholds for key metrics and receive notifications via Mattermost, email, or webhooks, ensuring you’re alerted in real-time to any issues. You can also use annotations to mark critical events like deployments or incidents directly on your graphs, making it easier to track performance changes. With data transformations, you can filter, aggregate, and modify data within Grafana itself, creating more meaningful visualizations. Finally, multi-source dashboards enable you to combine data from multiple sources into a single, unified view, providing a holistic perspective of your infrastructure and application health.

Common Use Cases

Use Case 1: Comprehensive Monitoring of Dataset Query Volume, Performance, and Load Balancing

Users of the SDL Application need to retrieve datasets for analysis or reporting. Monitoring both query volume and performance, while ensuring the query load is evenly distributed across

datasets, is crucial for efficient data retrieval and system optimization. This helps prevent individual datasets from being overloaded while maintaining fast response times.

Example: A panel shows the number of dataset queries executed per hour across different datasets, highlighting any dataset with disproportionately high load. Another panel tracks the average time taken to execute queries across datasets, providing insights into system performance under varying load.

Step 1 - Create a New Dashboard

- Navigate to Dashboards > + New Dashboard.
- Click Add New Panel to begin creating the first panel.

Step 2 - Add Dataset Query Volume Panel

- **Data Source**: Select Prometheus as the data source.
- **Metric Query**: In the query builder, use a metric like `query_requests_total{dataset=~".*"}` to track the total number of dataset queries made to the SDL system across all datasets.
- **Example query**: `query_requests_total{dataset="all-datasets"}`

Panel Configuration:

- Set Panel Type to Graph to visualize query volume trends over time.
- In the Legend, name it "Dataset Query Volume."
- **Filters**: Apply filters to focus on specific datasets if needed:
  - **Example filter**: `query_requests_total{dataset="customer-data"}` to focus on a specific dataset.
- **Save the Panel**: Click Apply to save the panel.

Step 3 - Add Dataset Query Performance Panel

- Add a New Panel by clicking the "+" icon.
- **Metric Query**: Use a query such as `query_execution_duration_seconds{dataset=~".*"}` to track the time it takes to execute queries across datasets.
- **Example query**: `query_execution_duration_seconds{dataset="all-datasets"}`

Panel Configuration:

- Set Panel Type to Graph to visualize query execution time over a set period.
- Name the panel "Query Execution Time" in the Legend field.
- **Save the Panel**: Click Apply to save the panel.

Step 4 - Add Query Load Balancing Panel

- Add a New Panel by clicking + Add New Panel.
- **Metric Query**: Use a query like `query_requests_total{dataset=~".*"}` to track how evenly

queries are distributed across multiple datasets.

- **Example query**: `query_requests_total{dataset=~".*"}` with each dataset being filtered individually to see query volume per dataset.

Panel Configuration:

- Set Panel Type to Bar Gauge or Table to clearly visualize how the query load is spread across datasets.
- Name the panel "Query Load Distribution."
- **Save the Panel**: Click Apply.

Use Case 2: Monitoring Data Ingestion Pipeline Performance

Users of the SDL Application often rely on data ingestion pipelines to bring in new datasets for analysis. Monitoring the performance of these pipelines—specifically the rate at which data is ingested and the time it takes to process data—ensures that ingestion tasks are completed efficiently and without delay.

Example: One panel tracks the data ingestion rate, showing how many datasets or records are ingested per minute, while another tracks the time taken for ingestion tasks to complete. This helps users identify potential bottlenecks or performance degradation in the data ingestion process.

Step 1 - Create a New Dashboard * Navigate to Dashboards > + New Dashboard. * Click Add New Panel.

Step 2 – Add Data Ingestion Rate Panel

- **Data Source**: Select Prometheus.
- **Metric Query**: Use a query like `ingestion_requests_total` to track how many datasets or records are ingested over time.
- **Example query**: `ingestion_requests_total{pipeline=~".*"}` to track ingestion across all pipelines.

Panel Configuration:

- Set Panel Type to Graph to visualize the ingestion rate.
- Name the panel "Data Ingestion Rate."
- **Save the Panel**: Click Apply.

Step 3 – Add Data Ingestion Time Panel

- **Metric Query**: Use a query like `ingestion_duration_seconds` to track the time taken to complete data ingestion tasks.
- **Example query**: `ingestion_duration_seconds{pipeline=~".*"}` to track duration across all pipelines.

Panel Configuration:

- Set Panel Type to Graph to visualize the ingestion time for each task.
- Name the panel "Data Ingestion Time" in the Legend.
- **Save the Panel**: Click Apply.

Step 4 – Set Alerts for Ingestion Failures

- Create an alert to trigger when ingestion time exceeds a set threshold (e.g., >5 minutes).
- Configure the Notification Channels to send alerts via email, Webhook, or Mattermost when ingestion tasks are slow or failing.

Use Case 3: Tracking Kafka Topic Growth Trends

Users of the SDL Application may need to monitor Kafka topic growth to maintain system performance and manage storage effectively. By tracking the log size growth of topics over time, users can identify topics that consume large amounts of storage and take necessary actions, such as managing retention policies or optimizing resource allocation. Monitoring topic growth is crucial to ensuring that Kafka clusters run efficiently without unnecessary resource overuse.

Example: In this use case, users can set up a dashboard to monitor the log size for each Kafka topic over time. Panels display the current log size and highlight any trends or significant increases in storage consumption for specific topics. This helps users quickly identify which topics are growing rapidly and take action, such as increasing resource allocation or adjusting retention settings to prevent excessive storage consumption.

Step 1 - Create a New Dashboard

- Navigate to Dashboards > + New Dashboard.
- Click Add New Panel.

Step 2 - Add Kafka Log Size Panel

- **Data Source**: Select Prometheus.
- **Metric Query**: Use a query like `sum(kafka_log_log_size{topic!~"(_)strimzi."})` to track the total log size for each Kafka topic.

Step 3 - Add a Time-Series Panel for Log Growth Over Time

- **Data Source**: Select Prometheus.
- **Metric Query**: Use a query like `sum(rate(kafka_log_log_size{topic!~"(_)strimzi."}[$__rate_interval]))` to track the growth of Kafka log size over time.

Step 4 - Set Alerts for Excessive Kafka Topic Growth

- Create an alert to trigger when Kafka topic log size exceeds a set threshold (e.g., >10 GB).
- Configure the Notification Channels to send alerts via email, Webhook, or Mattermost when certain Kafka topics experience rapid growth that may affect resource allocation.

Advanced Features

Feature 1: Dynamic Dashboards with Variables

Grafana allows template variables to create dynamic dashboards. This means you can create a single dashboard that adjusts based on user input, such as selecting different servers or data sources.

Steps:

- Navigate to the Variables section in the dashboard settings.
- Add a new variable (e.g., server).
- Use the variable in your queries to dynamically change the data displayed based on user selection.

Feature 2: Advanced Alerting

Grafana's alerting system can be set up to trigger notifications when a monitored metric crosses a specific threshold.

Steps:

- In the panel settings, click on Alert > Create Alert.
- Define the condition for the alert (e.g., CPU usage > 80%).
- Set up the notification channel (e.g., Webhook).
- Save the alert and monitor the dashboard for real-time updates.

Feature 3: Grafana Annotations

Annotations allow users to mark specific events on graphs for easier correlation between system events and metric changes.

Steps:

- Open the dashboard settings and navigate to the "Annotations" tab.
- Set up an annotation query based on log data or events.
- Visualize annotations as markers on the graph, correlating them with data trends.

Feature 4: Dashboard Playlist

Playlist mode in Grafana allows you to set up a rotating view of multiple dashboards, useful for operations centers and monitoring rooms.

Steps:

- Create a set of dashboards that you want to monitor.
- Open the playlist feature from the Grafana toolbar.
- Configure the rotation interval and start the playlist mode.

Best Practices

General Best Practices

- Keep dashboards simple by focusing on key metrics and avoiding an overload of panels on a single dashboard.
- Leverage Grafana's template variables to create flexible and reusable dashboards.

Performance Optimization

- Ensure queries are optimized to avoid overloading data sources, especially when working with large time-series datasets.
- Use caching mechanisms in Grafana for frequently used queries to improve dashboard performance.

Security Considerations

- Use role-based access control (RBAC) to manage permissions effectively. Ensure only authorized users can access or modify sensitive data.
- Ensure proper integration with Keycloak for secure authentication using OAUTH mechanisms.

Troubleshooting

Common Issues

- **Dashboard Not Loading**: Check the data source connection settings and verify that the data source is online.
- **Query Errors**: Ensure that the queries used in panels are correct and compatible with the data source.

Error Messages

- **Data Source Error**: Indicates an issue with the connection to your data source. Check server settings and authentication credentials.

Diagnostic Steps

- Check the status of the data source (Prometheus, Loki, etc.) from the Data Sources tab.
- Test the queries in the query editor to ensure they return data.

FAQs

- **How do I add a new data source in Grafana?** Go to Settings > Data Sources and select your data source type. Configure the connection with the necessary credentials and server details.
- **Can I set alerts for specific metrics?** Yes, alerts can be configured in each panel. Set the condition, and Grafana will notify you when the metric crosses the defined threshold.
- **How can I resolve "Permission Denied" when accessing a dashboard?** Ensure that you have

been assigned the correct user role in Keycloak.

- **Why can't I connect to my data source?** This issue may arise due to incorrect server credentials or configuration settings. Double-check the server URL, authentication details, and that the data source is running. If the problem persists, test the connection using Grafana's "Test" button under **Settings › Data Sources.**
- **Can I export a dashboard for use in other Grafana instances?** Yes, Grafana allows you to export dashboards in JSON format. Navigate to the dashboard you want to export, click on the "Share" button, and choose "Export." This JSON file can be imported into other Grafana instances.
- **What should I do if Grafana is running slow?** If Grafana's performance is slow, check if your queries are optimized. Large data sets or inefficient queries can slow down the dashboard.

Reference Materials

- Grafana Official Site
- Get Started with Grafana and Prometheus
- Loki Overview - Getting Started

Glossary

- **Dashboard**: A collection of panels that visualize data from one or more data sources.
- **Panel**: A single visualization in a Grafana dashboard (e.g., a graph, gauge, or table).
- **Data Source**: A system that provides data for visualization in Grafana (e.g., Prometheus, Loki, Elasticsearch).

Delivered Dashboards

API Gateway Dashboard

Description: This dashboard monitors the health and performance of the API Gateway, providing insights into key metrics like uptime, operation rates, response times, and outcomes for various API routes. It allows users to observe the performance of both fabric services and support services, helping them track and troubleshoot issues in real-time.

Data Sources: Prometheus

Key Panels:

Uptime (Top Panel): Displays the API Gateway's uptime, sourced in seconds (`process_uptime_seconds`), but converted and shown in days for readability. The panel uses color thresholds where uptime over 1 hour is shown in green. No critical thresholds are defined, but the panel visually indicates uptime status.

Proxy Services (Panel Group):

Operation rate: A time-series graph visualizing the rate of the operations (requests per second) for

API Gateway proxy routes. The query tracks the rate of requests using the `spring_cloud_gateway_requests_seconds_count` metric, filtered by the namespace, job, and route ID pattern (i.e., routes starting with proxy-). If no routes match this pattern, or if no data is present for the selected time range, the panel will display "No Data."

Response Times: A time-series graph tracking the maximum response times for API requests through proxy routes. The graph shows the slowest response times for each HTTP method and route that matches the proxy- route pattern.

Outcomes: A time-series panel showing the outcomes of requests through proxy routes, such as SUCCESSFUL or CLIENT_ERROR, aggregated by HTTP method and route ID for proxy routes.

Fabric Services (Panel Group):

Operation Rate: Tracks the operation rate (requests per second) for API routes such as v1-pipelines, categorized by HTTP method (e.g., DELETE, GET, POST).

Response Times: Shows the maximum response times for API requests, visualized as a time-series graph. The panel displays the slowest response times for each HTTP method and route.

Outcomes: Displays the results of API requests, such as CLIENT_ERROR and SUCCESSFUL, aggregated by HTTP method and route.

Support Services (Panel Group):

Operation Rate: Monitors the operation rate for support services such as /api/test/auth, /api/v1/auth/token, and /api/test/hello.

Response Times: Visualizes the response times for support services, providing detailed latency data for these API routes.

Outcomes: Displays the results of API requests for support services, showing success and unknown outcomes.

CPU Usage (Instance-Level Panel): A gauge displaying the CPU usage of the API Gateway, using the `process_cpu_usage` metric. The panel uses color-coded thresholds to indicate the CPU usage levels: * Green: Normal usage, below 20% CPU. * Yellow: Warning state, between 20% and 40% CPU usage. * Red: Critical state, above 40% CPU usage.

Additional Features:

Annotations & Alerts: Supports annotations for tracking key events. Alerts can be set up for critical metrics like operation rate or response times.

Template Variables: Utilizes variables like namespace and gateway_job to easily switch between different namespaces and jobs without modifying queries.

Time Selection & Auto-Refresh: Time range selector and auto-refresh features available.

Alerting Dashboard

Description: This dashboard monitors the status and performance of the alerting system within the SDL application. It provides insights into key metrics like alerts received, alerts published, filtering, and errors encountered in the alerting pipeline. Additionally, it monitors the health and uptime of the alerting services and provides data on the REST API for alert operations.

Data Sources: Prometheus

Key Panels:

Uptime (Top Panel): Displays the uptime of the alerting service, sourced from the `process_uptime_seconds` metric, converted to hours for readability. The panel uses color thresholds, with uptime shown in green if greater than 1 hour.

Alerts Section:

Alerts Published (Top Panel): Displays the total number of alerts published after being processed by the alerting system. Sourced from the `alerts_published_total` metric, this stat panel shows "No Data" when no alerts are published. The panel is color-coded blue when data is available.

Alerts Filtered (Top Panel): Tracks the total number of alerts filtered out by the system, using the `alerts_filtered_total` metric. This stat panel shows "No Data" if no filters are applied or no alerts pass through the filters, with an orange color when data is present.

Alert Errors (Top Panel): Monitors errors related to alerts, sourced from the `alerts_errors_total` metric. This stat panel turns red when errors are detected in the alerting process, with "No Data" shown if no errors occur.

Alerts Received Section:

Alerts Received (by source): This time-series panel shows alerts received, categorized by their source. Sourced from the `alerts_received_total` metric, it groups alerts by the source label. When no data is available, the panel displays "No Data."

Alerts Received (by type): This time-series graph displays the received alerts grouped by their type, showing how many of each type were processed. It is sourced from the same metric (`alerts_received_total`) but filtered by the type label.

Alerts Published Section:

Alerts Published (by sink): This time-series graph tracks the number of alerts published, grouped by the sink (destination) to which the alerts were sent, sourced from `alerts_published_total`. If no alerts are published, "No Data" is shown.

Alerts Published (by type): Similar to the above panel, this one tracks the alerts published, categorized by their type, using the `alerts_published_total` metric.

Alert Filtering Section:

Alerts Filtered (by filter): This panel shows the total number of alerts filtered, grouped by the filters applied to the alerts. The data is sourced from the `alerts_filtered_total` metric and

categorized by the filter label.

Alerts Filtered (by type): Tracks the alerts filtered, grouped by the type of alert being filtered. It uses the same `alerts_filtered_total` metric but categorized by type.

Alert Errors Section:

Filter Errors: This panel shows errors encountered during the filtering of alerts, grouped by the filters in question. The data is sourced from `alerts_errors_total`, specifically focusing on filter-related errors.

Publication Errors: Tracks errors encountered when publishing alerts to sinks, using the `alerts_errors_total` metric. The panel groups errors by the sinks and shows "No Data" if no errors occur.

REST API Section:

REST Operation Rate: This time-series panel monitors the rate of REST API operations related to alerts. It tracks the request rate using the `http_server_requests_seconds_count` metric, filtered for URIs matching the /api/.* pattern.

REST Response Times: Tracks the response times for the REST API operations related to alerting, using the `http_server_requests_seconds_max` metric. It visualizes the slowest response times for each URI that matches the /api/.* pattern.

REST Outcomes: This panel tracks the outcomes of REST API operations related to alerting, such as successful requests or errors, grouped by their status. It uses the `http_server_requests_seconds_count` metric.

Instances Section:

CPU Usage (Instance-Level Panel): A gauge panel displaying the CPU usage of the alerting service, using the `process_cpu_usage` metric. The panel uses thresholds to indicate different CPU levels: * Green: Normal usage (below 20%) * Yellow: Warning state (20%–40%) * Red: Critical usage (above 40%)

Additional Features:

Annotations & Alerts: Supports annotations for key event tracking. Users can configure alerts based on metrics like alert errors, published alerts, or CPU usage.

Template Variables: Uses variables like namespace and alert_api_job to easily switch between different jobs and namespaces for tailored views.

Time Selection & Auto-Refresh: Time range selector and auto-refresh features available.

Home Dashboard

Description: This dashboard serves as the central hub for monitoring the overall health of the SDL application. It provides users with quick access to error logs, API dashboards, Kafka dashboards, and alert dashboards. By aggregating key dashboards in one place, it allows users to seamlessly

navigate and investigate different areas of the system's performance. It also features a quick view of error logs from SDL, helping users quickly spot issues across the environment.

Data Sources: * Prometheus (for dashboard lists) * Loki (for error logs)

Key Panels:

Error Logs FD (Top Panel): A time-series panel tracking the occurrence of errors in SDL using logs from Loki. The query counts log entries that contain the term "error" over time (`count_over_time`). This provides a quick overview of any error spikes in the environment.

API Dashboards (Panel Group):

- A dashboard list panel that provides access to API-related dashboards within SDL. Key dashboards include:
- API Gateway: Focused on monitoring the health and performance of API Gateway services.

Kafka Dashboards (Panel Group):

- A dashboard list panel focused on monitoring Kafka clusters. Available dashboards include:
- Kafka Cluster - Brokers Overview: Monitors the health and performance of Kafka brokers.
- Kafka Cluster - Connections: Tracks connection metrics for Kafka.
- Kafka Cluster - Consumer Lag: Measures the lag between Kafka consumers and producers.
- Kafka Cluster - Topics: Displays metrics about Kafka topics.
- Kafka Cluster - Topics Comparison: Compares various metrics between Kafka topics.
- Kafka Cluster - Topics Overview: An overall view of all Kafka topics within the cluster.

Alert Dashboards (Panel Group):

- A dashboard list providing access to alerting-related dashboards in SDL, including:
- Alerting: The primary dashboard for monitoring alert performance in the system.

Additional Features:

Annotations & Alerts: Supports annotations for tracking key events, though no alerts are pre-configured. Users can add alerts based on logs or API performance metrics.

Navigation: The Home Dashboard provides direct links to specific dashboards within the SDL environment, including Error Logs, API Gateway, Kafka (Brokers Overview, Connections, Consumer Lag, Topics), and Alerting Dashboards. This centralized access allows users to quickly switch between dashboards to monitor different parts of the system and troubleshoot issues more efficiently.

Time Selection & Auto-Refresh: Time range selector and auto-refresh features available.

Kafka Cluster – Brokers Overview Dashboard

Description: This dashboard provides an overview of key metrics related to Kafka brokers,

including the number of online brokers, controller status, partition health, and traffic rates. It allows users to monitor the health and performance of Kafka clusters in real-time and quickly identify any issues related to replication, partition availability, and broker performance. The dashboard is essential for administrators who need to maintain the stability of Kafka clusters and ensure smooth message processing.

Data Sources: Prometheus (Metrics for Kafka brokers, partitions, and controllers)

Key Panels:

Top Panel

Brokers Online: Displays the current number of Kafka brokers that are online. The panel uses Prometheus metrics (`kafka_controller_kafkacontroller_activebrokercount`) and is updated every 5 seconds. Thresholds are color-coded: * Green: 2 or more brokers online * Yellow: 1 broker online * Red: 0 brokers online

Active Controllers: Shows the number of active Kafka controllers. This panel uses `kafka_controller_kafkacontroller_activecontrollercount` to track the active controllers in the Kafka cluster.

Unclean Leader Election Rate: Monitors the rate of unclean leader elections in Kafka, which can indicate instability or failure in the Kafka cluster. The panel uses `kafka_controller_controllerstats_uncleanleaderelectionenablerateandtimems` to calculate the metric. Thresholds are color-coded for alerting purposes.

Online Partitions: Displays the number of online partitions in the Kafka cluster. The `kafka_server_replicamanager_partitioncount` metric is used to determine the number of partitions that are online and functioning correctly.

Under Replicated Partitions: Tracks the number of under-replicated partitions using `kafka_server_replicamanager_underreplicatedpartitions`. If the number of under-replicated partitions rises, it indicates that some replicas are not in sync, which can lead to data loss if a broker fails.

Offline Partitions Count: Displays the number of offline partitions in the Kafka cluster. Using the `kafka_controller_kafkacontroller_offlinepartitionscount` metric, it highlights any partitions that are currently unavailable.

Traffic and Performance Panels:

Messages In / Second: Tracks the number of incoming messages per second using `kafka_server_brokertopicmetrics_messagesin_total`. This panel shows the overall message rate being processed by the Kafka brokers.

Bytes In / Second & Bytes Out / Second: These panels track the incoming and outgoing data throughput in bytes per second. The `kafka_server_brokertopicmetrics_bytesin_total` and `kafka_server_brokertopicmetrics_bytesout_total` metrics are used to monitor the rate of data flowing into and out of Kafka brokers.

Messages In / Second / Broker & Bytes In / Second / Broker: These time-series panels provide a

breakdown of message and byte throughput per individual broker. Metrics are displayed for each broker (df-kafka-0, df-kafka-1, df-kafka-2) to help identify imbalances in the load distribution.

Partitions / Broker: Shows the number of partitions per broker using the `kafka_server_replicamanager_partitioncount` metric. This panel helps in understanding how partitioned data is distributed across brokers.

Partition Leaders / Broker: Displays the number of partition leaders per broker, which is crucial for balancing load and ensuring the efficient handling of partitions.

Under Replicated Partitions / Broker: Tracks the number of under-replicated partitions for each broker. The panel provides a broker-wise view of replication health, helping to identify which brokers are struggling to keep their replicas in sync.

Kafka Log Size by Broker: Displays the total log size of each Kafka broker using the `kafka_log_log_size` metric. This helps in monitoring storage usage on each broker and ensuring enough disk space is available.

JVM Panel:

JVM Metrics (Threads): Shows JVM metrics like the number of threads per broker. This provides insight into the resource utilization of the JVM running on Kafka brokers.

Additional Features:

Annotations & Alerts: Supports annotations to track key events. Alerts can be configured for broker count, partition status, replication health, and message throughput.

Threshold-Based Coloring: Color-coded thresholds for critical metrics like broker count, CPU usage, and partition replication to highlight potential issues.

Time Selection & Auto-Refresh: Time range selector and auto-refresh features available.

10.5 Kafka Cluster – Connections Dashboard

Description: This dashboard provides an overview of Kafka cluster connections, focusing on connection counts per listener and the versions of client software connected to the brokers. It helps users to monitor Kafka's connection health, identifying potential bottlenecks or anomalies in listener activity and client distribution across software versions.

Data Source: Prometheus

Key Panels:

Connection Count / Listener (Time-Series Panel): This panel shows the count of active connections per listener (CONTROLPLANE-9090, PLAIN-9092, REPLICATION-9091) over time. The data is sourced from the `kafka_server_socket_server_metrics_connections_software` metric, grouping by listener. The panel visualizes the last and maximum values for each listener, allowing for real-time monitoring of connection patterns.

Client Versions (Bar Gauge Panel): This panel displays the active client versions connected to the

Kafka brokers, visualizing how many connections are made by each Kafka client version. The data is sourced from the same metric, grouped by `clientSoftwareName` and `clientSoftwareVersion`. It allows the user to track the distribution of client software across the cluster, which helps identify the most common client versions in use.

Additional Features:

Annotations & Alerts: Supports annotations to track key events. Alerts can be configured based on connection metrics, listener activity, and client versions.

Template Variables: Allows users to select specific listeners and client versions to filter and focus the view.

Time Selection & Auto-Refresh: Time range selector and auto-refresh features available.

Kafka Cluster – Consumer Lag Dashboard

Description: This dashboard provides an in-depth view of consumer lag within a Kafka cluster. It helps users monitor lag at both the topic and partition levels, track current offsets, and evaluate the lag per consumer group. The dashboard is designed to help pinpoint issues that may affect data consumption and processing performance.

Data Source: Prometheus

Key Panels:

Consumer Lag By Topic (Time-Series Panel): This panel shows the consumer lag for different Kafka topics over time. The data is sourced from the `kafka_consumergroup_lag` metric, grouped by topic and consumer group. It allows users to identify which topics are experiencing the most lag, providing insights into where delays in message consumption are occurring.

Consumer Lag By Partition (Time-Series Panel): This panel displays the consumer lag for different Kafka partitions. The data is sourced from the same `kafka_consumergroup_lag` metric, but it breaks down the lag at the partition level. This enables users to monitor lag on a more granular scale, helping to identify specific partitions that may be causing issues.

Consumer Lag Table (Table Panel): This panel provides a detailed table view of consumer lag, showing the topic, partition, current offset, and lag by offset. The table is essential for users who need to track exact lag metrics and see how much delay each partition is experiencing.

Consumer Lag By Group (Bar Gauge Panel): This panel visualizes consumer lag grouped by consumer group. It provides a comparative view of how different consumer groups are performing in terms of lag, helping to quickly identify groups that may be lagging.

Additional Features:

Annotations & Alerts: Supports annotations for tracking key events. Users can configure custom alerts based on consumer lag metrics such as lag by partition, topic, or consumer group.

Detailed Consumer Lag Monitoring: Provides in-depth monitoring of consumer lag at the topic, partition, and group levels, helping to quickly identify issues.

Time Selection & Auto-Refresh: Time range selector and auto-refresh features available.

Kafka Cluster – Topics Dashboard

Description: This dashboard provides insights into the consumption metrics of Kafka topics, including topic size, partition count, message count, and I/O data rates. It is designed to help users monitor topic performance, assess data flow, and identify potential issues such as under-replicated partitions.

Data Sources: Prometheus

Key Panels:

Topic Size (Stat Panel): Displays the total size of the selected Kafka topic in megabytes (MB). The value updates based on the selected time range (default 24 hours). Helps to monitor data growth for each topic.

Partition Count (Stat Panel): Shows the total number of partitions for the selected topic. Kafka partitions provide concurrency, and tracking their count can help assess the distribution of data.

Current Message Count (Stat Panel): This panel displays the current count of messages in the topic, providing a snapshot of how much data is currently present.

Total Messages Produced (Stat Panel): Displays the total number of messages produced to the topic over time. It is calculated based on the Kafka log end offset metric grouped by partition, providing visibility into the throughput of the topic.

Messages In / Sec (Stat Panel): Displays the rate of messages being produced to the selected topic in real-time (messages per second). It helps assess the load being handled by the topic.

Bytes In / Sec (Stat Panel): Shows the rate at which data is being ingested into the topic, measured in bytes per second. This metric helps to track the inbound data flow to the topic.

Bytes Out / Sec (Stat Panel): Displays the rate of outbound data from the topic in bytes per second. This panel tracks how much data is being consumed from the topic, though in this case, it currently shows "N/A," indicating no outbound data at the moment.

Messages In / Second (Time-Series Panel): A time-series graph showing the rate of messages being ingested into the topic over the selected time frame (e.g., last 24 hours). This graph provides a visual indication of message ingestion trends and spikes.

Bytes In / Second (Time-Series Panel): A time-series graph displaying the rate of inbound data (bytes per second) over time, helping users observe changes in data flow.

Bytes Out / Second (Time-Series Panel): A time-series panel is meant to display the rate of outbound data. Currently, it shows no data, indicating that the topic has no data being read from it during the selected timeframe.

Additional Features:

Annotations & Alerts: Annotations are supported to track key events. Users can configure custom

alerts for critical metrics like message rates, data throughput, and partition replication status.

Template Variables: Provides a dropdown to select specific Kafka topics, allowing for easy comparison of metrics across topics.

Time Selection & Auto-Refresh: Time range selector and auto-refresh features available.

Kafka Cluster – Topics Comparison Dashboard

Description: This dashboard offers a comparative overview of key metrics across different Kafka topics, such as message rates and byte throughput. It enables users to analyze message production and consumption patterns, helping to identify anomalies or underutilized topics over the selected time range.

Data Source: Prometheus

Key Panels:

Messages In / Sec (Stat Panel): Displays the rate of messages being produced to the selected topic in real-time (messages per second). It allows users to assess the load being handled by the topic and monitor message throughput over time.

Bytes In / Sec (Stat Panel): Shows the rate at which data is being ingested into the topic, measured in bytes per second. This panel provides insights into the inbound data flow to the topic.

Bytes Out / Sec (Stat Panel): Displays the rate of data being consumed from the topic in bytes per second. Currently, it shows "N/A," indicating no outbound data during the selected time period.

Messages In / Second (Time-Series Panel): A time-series graph that tracks the rate of messages being produced to the topic over time. It provides a detailed look at message throughput, helping users visualize spikes or drops in activity.

Bytes In / Second (Time-Series Panel): A time-series graph displaying the rate of inbound data to the topic over the selected timeframe. This graph helps users understand how the data volume changes over time.

Bytes Out / Second (Time-Series Panel): A time-series graph intended to track the rate of outbound data from the topic. However, it currently shows no data, indicating that the selected topic has not had any data consumed during the observed period.

Additional Features:

Annotations & Alerts: Supports annotations to track key events related to Kafka topics or infrastructure changes. These annotations help correlate Kafka metrics with external factors that might impact performance. Although no alerts are pre-configured, users can add alerts to monitor thresholds such as message throughput or byte consumption rates.

Topic Selection Dropdown: This dashboard includes a dynamic topic selection dropdown that allows users to select and compare multiple Kafka topics, such as _consumer_offsets, df.meilisearch.datasources, and others. The selected topic determines the data shown in the time-series and stat panels, enabling users to toggle between different topics to analyze and monitor

performance.

Time Selection & Auto-Refresh: Time range selector and auto-refresh features available.

Dynamic Data Updates: All panels, including "Messages In / Sec," "Bytes In / Sec," and "Bytes Out / Sec," update dynamically based on the selected topic and time range. This enables continuous monitoring of Kafka performance metrics without needing manual refreshes.

Kafka Cluster – Topics Overview Dashboard

Description: This dashboard provides an overview of Kafka topics, offering insights into message rates, data flow, disk storage, and partition replication. It helps track Kafka topic activity and ensures fault tolerance by monitoring under-replicated partitions.

Data Source: Prometheus

Key Panels:

Topics (Stat Panel): Displays the total number of Kafka topics in the cluster. Useful for keeping track of the overall topic count.

_strimzi Topics (Stat Panel): Shows the number of Strimzi-managed topics, which can help monitor and manage topics created by the Strimzi Kafka operator.

All Topics (Stat Panel): Indicates the total number of topics across all categories, providing a complete view of topics being tracked.

Disk Storage (Stat Panel): Displays the total disk space used by Kafka topics, excluding Strimzi-managed topics, helping monitor overall storage consumption.

_strimzi Disk Storage (Stat Panel): Shows the disk usage for Strimzi-managed topics, which is useful for understanding the storage footprint of topics managed by the Strimzi operator.

All Disk Storage (Stat Panel): Displays the total disk space occupied by all Kafka topics, both system and user-managed, providing an overview of storage usage in the cluster.

Messages In / Second / Topic (Time-Series Panel): A time-series graph tracking the rate of messages produced per topic over time. This helps visualize throughput per topic, enabling users to detect spikes or anomalies in message production rates.

Bytes In / Second / Topic (Time-Series Panel): Monitors the rate of inbound data for each Kafka topic over time. This helps users understand the data flow for each topic, aiding in capacity planning and identifying bottlenecks.

Bytes Out / Second / Topic (Time-Series Panel): Tracks the outbound data rate per topic, indicating data consumption activity. This helps ensure topics are being consumed as expected and that consumers are working effectively.

Local Storage – Log Size by Topic (Time-Series Panel): A graph showing log size by topic over time. This panel helps identify topics that are consuming significant storage, allowing users to manage disk usage and plan for storage expansion.

Under-Replicated Partitions by Topic (Time-Series Panel): Displays the number of under-replicated partitions per topic, which is crucial for Kafka's fault tolerance. It helps monitor if any partitions are not meeting the replication factor, highlighting potential data redundancy issues.

Additional Features:

Annotations & Alerts: Supports annotations for tracking key events. Alerts can be set for metrics like under-replicated partitions, message rates, and log size.

Topic Filtering: The dashboard provides a dropdown to select specific Kafka topics for focused monitoring, allowing users to filter data by topics like df.meilisearch.datasets.

Time Selection & Auto-Refresh: Time range selector and auto-refresh features available.

Detailed Legends for Time-Series Panels: Each time-series panel comes with detailed legends that show real-time metrics such as "last" and "max" values for topics. This makes it easy to compare the relative activity of different topics and identify anomalies.

Appendices

Common Data Source Queries

Here are some additional example queries to get the most out of the SDL's integration with Prometheus and Loki:

Total CPU Usage for a Service:

```
sum(rate(process_cpu_seconds_total{job="your-service"}[5m]))
```

Memory Consumption by Namespace:

```
sum(container_memory_usage_bytes{namespace="data-fabric"}) by (pod)
```

Error Rate for a Specific API Endpoint:

```
rate(http_requests_total{status_code=~"5..", job="api-gateway", route="/v1/pipelines"}[5m])
```

Importing Dashboards in Grafana

You can easily import dashboards from JSON files or from Grafana.com to replicate pre-built dashboards.

How to Import a Dashboard:

1. Navigate to the New button at the top right in the Dashboard page.
2. Select Import from the dropdown menu.

You will have three options:

- **Upload JSON File**: Upload the JSON file of a dashboard you have saved.
- **Import via grafana.com**: Provide the URL or ID of the dashboard from Grafana's public repository.

- **Import via Panel JSON**: Paste a JSON configuration for a specific panel to add it to your dashboard.

Customizing Grafana with Variables

SDL supports the use of template variables for dynamic dashboards. Below are examples of how to configure template variables for more flexible monitoring.

Example: Variable for Kafka Topics: * Navigate to Dashboard Settings > Variables. * Add a new variable: - **Name**: topic - **Query**: `label_values(kafka_topic)`

Use this variable in your queries:

Example: `sum(rate(kafka_log_log_size{topic="$topic"}[5m]))`

Example: Variable for Data Sources: * Add a new variable for switching data sources dynamically: - **Name**: datasource - **Type**: Datasource - **Query**: Prometheus or Loki

Use `$datasource` in any query to switch between data sources without editing the panel.

Alerting: Setting Up a Webhook Alert in SDL

In this section, we will guide you through the steps of setting up a webhook alert in Grafana within the SDL environment. Webhook alerts are useful for sending notifications to external systems like chat applications, issue trackers, or custom monitoring services when an alert condition is triggered.

Prerequisites:

- **Access to Grafana**: Ensure you have access to the Grafana instance within SDL.
- **Webhook URL**: You will need the URL of the webhook endpoint where you want to send alerts.
- **Authentication (if required)**: If the webhook requires authentication, have the necessary credentials (e.g., API key or token) ready.

Step 1: Create a New Notification Channel for Webhooks:

Login to Grafana: Using your Keycloak credentials, log in to Grafana in the SDL environment.

Navigate to Notification Channels: 1. From the Grafana homepage, click the □□ (gear icon) on the left sidebar. 2. Under the Alerting section, select Notification Channels.

Create a New Notification Channel:

1. Click the + New Channel button at the top right of the Notification Channels page.
2. Configure the Webhook Notification Channel:
    - **In the Name field**, enter a name for the webhook notification channel (e.g., "SDL Webhook").
    - **In the Type dropdown**, select Webhook.
    - **Webhook URL**: In the URL field, enter the webhook endpoint where the alert notifications

should be sent (e.g., `https://your-webhook-service.com/alert`).

- If your webhook requires authentication, add any necessary headers by clicking **Add Custom HTTP Header** and entering the key (e.g., Authorization) and the value (e.g., `Bearer your-token-here`).

Test the Webhook (optional but recommended):

1. Click **Test** to send a test notification to the webhook URL.
2. Verify that the external service receives the notification and processes it as expected.

Save the Notification Channel:

1. Scroll down and click **Save** to finish setting up the webhook notification channel.

Step 2: Configure Alerts for a Panel:

Open the Dashboard: Navigate to the dashboard you want to monitor. For example, if you want to monitor CPU usage, open the CPU Usage dashboard.

Select a Panel for Alerting:

1. In the dashboard, identify the panel where you want to set up the alert (e.g., a graph tracking CPU usage).
2. Click the panel title, then select **Edit** from the dropdown menu.

Create a New Alert:

1. In the panel editor, navigate to the **Alert** tab.
2. Click **Create Alert**.

Define the Alert Condition:

1. Evaluation of Time Interval: Set how often the alert should evaluate data. For example, check every minute.
2. Alert Condition: Define the condition for triggering the alert (e.g., WHEN avg() OF query (A, 5m) IS ABOVE 80). This will trigger an alert when the average CPU usage over a 5-minute window exceeds 80%.
3. Adjust the time range and query if necessary.

Define Alert Behavior:

1. Alert States: Grafana alerts have three states: OK, Pending, and Alerting. Configure these states based on how long the condition should persist before triggering the alert.
    - **Example**: If CPU usage remains above 80% for 2 minutes, trigger the alert.

Add Notification Channel:

1. Under the Notifications section, click **Add Notification Channel**.

2. Select the webhook notification channel you created earlier from the dropdown list (e.g., "SDL Webhook").
3. Optionally, you can add more notification channels (e.g., email, Mattermost) if you'd like to receive alerts through multiple methods.

Customizing the Alert Message (optional):

1. You can customize the message that is sent to the webhook. Click **Edit Message** and enter your custom text, including variables like `${metric}` to include dynamic values.

Step 3: Testing and Saving the Alert:

Test the Alert:

1. Once the alert is set up, scroll down to the bottom of the panel editor and click **Test Rule** to evaluate the alert condition.
2. This will run the alert condition and trigger a notification if the condition is met. You should verify that the webhook received the alert.

Save the Panel:

1. Once satisfied with the alert setup, click **Apply** to save the alert configuration.
2. Save the dashboard by clicking **Save Dashboard** to ensure the changes persist.

Step 4: Monitoring Alerts:

Monitoring Active Alerts:

1. Navigate to the **Alerting** section in the Grafana menu to see the status of your alerts.
2. Here, you can monitor all active alerts, check which conditions are being evaluated, and view their current state (OK, Pending, or Alerting).

Viewing Alert History:

1. Click on **Alert History** in the same section to see past alert notifications and check if alerts were delivered as expected. This section provides insight into when and why alerts were triggered, helping you understand system behavior and assess whether alert conditions are configured appropriately.

Modifying Alerts:

1. To make adjustments to an existing alert, return to the relevant panel and open the **Alert** tab in the panel settings.
2. Adjust the conditions, notification channels, or alert thresholds as needed. Ensure you save changes to both the panel and the dashboard.

Disabling Alerts:

1. If you need to temporarily disable an alert without deleting it, navigate to the **Alert** tab and

toggle the enable/disable switch. This allows you to retain the alert configuration while preventing notifications from being sent.

Custom Query Examples for Advanced Users

This section provides a series of custom Prometheus and Loki queries that advanced users can leverage to enhance their Grafana dashboards in the SDL environment.

Prometheus Queries:

Track Memory Usage Across All Pods in a Namespace:

```
`sum(container_memory_usage_bytes{namespace="data-fabric"}) by (pod)`
```

Monitor API Error Rates by Endpoint:

```
`rate(http_requests_total{status_code=~"5..", job="api-gateway",
route="/v1/endpoint"}[5m])`
```

CPU Usage by Instance:

```
`avg(rate(container_cpu_usage_seconds_total{image!="", namespace="data-fabric"}[1m]))
by (instance)`
```

Loki Queries:

Search for Specific Log Entries Containing Errors:

```
`{namespace="data-fabric"} |= "ERROR"`
```

Count of Errors Over a Time Range:

```
`sum(count_over_time({namespace="data-fabric"} |= "ERROR" [5m]))`
```

These queries can be further customized to fit specific monitoring requirements by adjusting filters, labels, and aggregation functions.

4.5. Data Science Notebooks

Introduction

Purpose: A comprehensive guide to understanding and utilizing JupyterHub's capabilities within Raft's SDL and the SDL.

Target Audience: Data Stewards, Data Analysts

Scope: The JupyterHub documentation for a data platform provides users with detailed guidance on using JupyterHub for working within SDL.

Overview

Capability Description: JupyterHub is a powerful, scalable, multi-user platform that allows users to create and run Jupyter notebooks on a shared SDL server. JupyterHub allows users to store, access, and stream their data through their browser, which is a powerful tool for data-driven projects.

Key Features:

- **Interactive Data Analysis**: Provides access to Jupyter Notebooks for interactive data exploration, visualization, and machine learning. This is especially useful in air-gapped environments, where SDL's JupyterHub helps solve the problem of finding, accessing, and analyzing data.
- **Libraries**: JupyterHub allows Python libraries to be installed on-the-fly if there is internet connectivity. However, in air-gapped environment, SDL can be connected to CDAO's Python library repository, where most (if not all) common Python libraries are copied. This allows on-the-fly library installation even without internet access.
- **Multi-user Environment**: Supports multiple users with individual environments, enabling collaboration while maintaining data privacy.
- **Data Storage and Streaming**: Users can store, access, and stream data through their Jupyter environment, making it ideal for large-scale data processing tasks when running on SDL's scalable platform.

Setup

Access JupyterHub

From the left side panel, select **Data Insights → Notebooks**.

Log in with Keycloak if necessary. This will start a JupyterHub session. You should see something like this:

Usage

The following example notebooks are available to help users get started with interacting with SDL data sources:

- `kafka_consumer.ipynb` (Kafka Consumer example)
- `kafka_producer.ipynb` (Kafka Producer example)
- `object_store.ipynb` (MinIO example)
- `trino.ipynb` (Trino SQL query example)

Run Example Notebooks

Kafka Producer and Consumer Interaction

The `kafka_producer.ipynb` and `kafka_consumer.ipynb` notebooks are complementary, demonstrating a typical Kafka message flow within SDL. The producer sends messages to a Kafka topic, and the consumer listens to that topic and processes the messages in real-time.

Why Use It? This notebook is helpful when you need to simulate a real-time data pipeline. It allows users to verify that messages are being sent and received properly, ensuring the data flow through Kafka is working as expected.

Example Use Case Imagine you're working on a system that streams real-time data, like sensor readings or log files: 1. Send your data (sensor readings, logs, etc.) to a Kafka topic using the producer notebook. 2. Consume the data in real-time with the consumer notebook, ensuring that the data is flowing correctly and can be processed further.

Steps to Run the Producer and Consumer Notebooks:

1. Start the Kafka Consumer:
    - First, run the `kafka_consumer.ipynb` notebook to start listening for messages on the Kafka topic.
        - The consumer is configured to subscribe to the `test-topic`. **It uses the `python-consumer` group ID to manage offset tracking and allow parallel consumption with other consumers if necessary.
            - Run the first cell to set up the configuration, environment variables, and logging.
            - Run subsequent cells to start the Kafka consumer. **This consumer will remain active and continuously listen for new messages on `test-topic`.** You should see log outputs indicating the consumer is connected and ready to receive messages.

Example:

```
Listening for messages on 'test-topic'...
```

1. Produce Messages with the Kafka Producer:
    - Next, open and run the `kafka_producer.ipynb` notebook. This notebook generates and sends messages to the same Kafka topic (`test-topic`), which the consumer is already listening to.
        - Run the first cell to set up the environment and configuration.
        - Generate messages: In the subsequent cells, a loop generates 10 messages by default, which are sent to the Kafka topic. Each message is sent to `test-topic`, and a confirmation log is produced for each message. You can customize the number of messages by changing the `number_of_messages_to_generate` variable.
2. Switch Back to the Kafka Consumer:
    - Once the producer starts sending messages, switch back to the `kafka_consumer.ipynb` notebook to observe the incoming messages.

        - The consumer will print each message it receives from the Kafka topic.
        - You'll see logs similar to the following as the messages are consumed:
3. Logging and Error Handling:
    - Both notebooks implement logging, which helps you monitor their operation. If there are any errors in producing or consuming messages, they will be captured and displayed.
        - The Kafka consumer logs errors if there are issues with message fetching, connection problems, or configuration mismatches.
        - The Kafka producer logs errors if it fails to send messages or connect to the Kafka server.

Congratulations! You just produced and consumed messages with Kafka running on SDL!

MinIO and Kafka Avro Producer Example

This example notebook demonstrates how to interact with MinIO and produce Kafka messages using Avro schemas. It shows the integration between the MinIO object storage service and Kafka for producing messages with schema validation.

Why Use It? This notebook is useful when you need to store data in MinIO and send it as structured messages to Kafka. It ensures that the data conforms to a specific format, which is important for systems that rely on consistent and reliable data, like real-time analytics or event-driven processing.

Example Use Case Suppose you're a data engineer working on a project that processes sensor data: 1. Store sensor data in MinIO as JSON files. 2. Use the notebook to generate an Avro schema from this data. 3. Send the data to Kafka, where other services can pick it up for further analysis or processing.

Steps to Use the Notebook:

Below is the function of each cell in order.

1. Connecting to MinIO:
    - The first cell establishes a connection to MinIO using environment variables for the access key and secret key. Users can manage MinIO buckets and objects once the connection is established.
2. Listing MinIO Buckets:
    - This cell lists all available MinIO buckets in the environment. Users can check if their buckets are accessible.
3. Uploading an Object to MinIO:
    - In this cell, the notebook uploads a file (e.g., `example.json`) to a specific bucket (`inbox-public`) in MinIO. This is useful for storing data that will be used later or shared with other services.
4. Listing Objects in MinIO:
    - After uploading the file, this cell lists all objects in the specified MinIO bucket, confirming

the file was uploaded successfully.

5. Generating Avro Schema:
    - These cells define the logic for generating an Avro schema based on the data being sent to Kafka. The notebook dynamically creates a schema that ensures all messages conform to a standardized format.
6. Validating Missing Fields:
    - In this cell, the notebook checks if any fields in the data are missing from the generated schema, ensuring that all fields are accounted for.
7. Configuring Kafka Producer:
    - The Kafka producer is configured in this cell with the necessary security settings and the topic to which messages will be sent.
8. Producing Messages to Kafka:
    - This cell produces messages to Kafka, using the Avro schema generated earlier. The messages are sent to the Kafka topic specified in the configuration.
9. Configuring Kafka Settings:
    - In this cell, environment variables like `KAFKA_USERNAME` and `KAFKA_PASSWORD` are set up to ensure the connection to Kafka is properly authenticated.
10. Producing More Kafka Messages:
    - Additional messages are produced to Kafka in this cell. Users can modify the fields to produce different data.
11. Listing Objects in MinIO:
    - This cell lists objects in the `inbox-public` bucket again to ensure everything remains accessible after Kafka message production.
12. Retrieving and Reading an Object from MinIO:
    - Finally, this cell retrieves and reads the contents of the file (`example.json`) from MinIO, decoding the JSON data to verify that it was stored correctly.

Object Store Example

This `object_store.ipynb` example notebook demonstrates how to interact with MinIO for uploading files and listing objects in a MinIO bucket. It helps users manage their data in an S3-compatible object storage system within the SDL.

Why Use It? This notebook is useful when you need to store and manage files in MinIO. It allows users to easily upload files (e.g., datasets, configuration files) to a shared storage system, verify the contents of MinIO buckets, and ensure data is safely stored for future use or processing.

Example Use Case

Imagine you are working on a project where you need to store documentation, datasets, or configuration files:

1. Upload your file (e.g., `getting-started.md`) to a MinIO bucket (`inbox-public`) for storage and easy

retrieval later.

2. List objects in the MinIO bucket to verify the upload and check the other available files.
3. Use the uploaded file in future data analysis tasks, configuration processes, or any other project-related activities.

Steps to Use the Notebook:

1. Connecting to MinIO:
    - The first cell establishes a connection to MinIO using the Minio client. The access key and secret key are retrieved from environment variables to securely connect to MinIO. Once connected, users can interact with the MinIO buckets to upload, retrieve, and manage files.
2. Listing MinIO Buckets:
    - This cell lists all available buckets in MinIO, allowing users to check that the connection is successful and see which storage buckets are accessible for file uploads and management.
3. Uploading an Object to MinIO:
    - In this cell, the notebook uploads a file (`getting-started.md`) to the `inbox-public` bucket. This is useful for storing data files, configuration files, or any other data that needs to be shared or processed later.
4. Listing Objects in MinIO:
    - After uploading the file, this cell lists all objects in the `inbox-public` bucket. This confirms that the file has been successfully uploaded and gives users an overview of all the files stored in the bucket.

Trino Example

This example notebook demonstrates how to connect to a Trino instance using SQLAlchemy and JWT authentication. It shows users how to retrieve an authentication token from SDL, establish a secure connection to Trino, and run SQL queries to interact with Trino's catalogs.

Why Use It? This notebook is useful for users who need to run SQL queries on Trino while using secure JWT authentication. It provides an easy way to query data from various catalogs in Trino, allowing users to explore datasets stored within the SDL using a Python interface.

Example Use Case

Imagine you're working on a project where you need to query and analyze data from multiple sources within Trino:

1. Authenticate using credentials to retrieve a secure token from SDL.
2. Connect to Trino with the obtained token, allowing you to securely run queries against various data catalogs.
3. Run SQL queries to explore datasets, pull relevant data, and perform analysis or data processing directly from your notebook.

Steps to Use the Notebook:

1. Setting Up Authentication:
    - The first cells disable warnings and set up environment variables for the username (`DF_USER`) and password (`DF_PASS`). These credentials are used to authenticate with the SDL API and retrieve a JWT token that will be used to connect to Trino.
2. Getting the Token:
    - These cells send a request to the SDL API to obtain a JWT authentication token. The token is returned in JSON format, and the notebook extracts the token for use in the connection to Trino.
3. Connecting to Trino:
    - The connection to Trino is established using SQLAlchemy. The JWT token obtained earlier is passed in the connection, and the notebook uses HTTPS with the option to disable SSL verification in local environments.
4. Running Queries on Trino: This cell runs a sample SQL query (`SHOW CATALOGS`) to display the available catalogs in Trino. The results are loaded into a Pandas DataFrame for easy viewing, and users can modify the query to explore other datasets or perform further analysis.

Best Practices

Efficient Resource Usage

- Limit Notebook Resources: Use small datasets and run lightweight operations to test functionality before executing large-scale computations. This helps prevent overloading the system and ensures efficient use of CPU and memory. For example, if working with large datasets from Delta Lake or Kafka streams, apply filters early to avoid loading unnecessary data.
- Shut Down Idle Kernels: Notebooks left idle or running indefinitely consume system resources. Always shut down kernels when you're done with a notebook session by using the **Kernel → Shutdown** option.
- Run Long Jobs in Off-Peak Hours: If your work involves long-running processes or resource-intensive tasks, consider running them during off-peak hours to avoid affecting other users.

Troubleshooting

1. **Notebook is Running Slowly or Crashing** Cause: Your notebook may be using too much memory or CPU, especially when working with large datasets or running complex computations. Solution: Try using smaller data subsets or breaking your computations into smaller steps. Clear any unnecessary variables or data that are no longer needed using `%reset` or `del` commands to free up memory. If the notebook is still unresponsive, restart the kernel from the Kernel menu.
2. **Kernel is Not Responding** Cause: The kernel may have become overwhelmed due to a heavy computation or large data load. Solution: Restart the kernel by going to **Kernel → Restart Kernel**. After restarting, re-run the necessary cells to continue your work. If the problem persists, consider optimizing your code or using smaller data batches.
3. **Seek Help from Administrators** If the issue seems related to the underlying environment or

services (like Kafka or MinIO), it might be necessary to contact your administrator for assistance. Make sure to provide any relevant error messages and details about what you were trying to do.

FAQs

1. How do I save my work in JupyterHub? JupyterHub automatically saves your notebook every few minutes, but you can also manually save by clicking **File → Save and Checkpoint**. It's a good habit to save your progress regularly, especially when running long computations.
2. How do I shut down a notebook that's no longer in use? You can shut down a notebook by going to the **Kernel** menu at the top of the notebook and selecting **Shutdown**. Alternatively, you can shut down kernels from the JupyterHub dashboard by clicking the **Running** tab and manually stopping the notebook.
3. What should I do if my notebook is running slowly?

If your notebook is running slowly, try the following:

- Shut down other notebooks that you're not using.
- Clear unnecessary variables or data from memory by using `%reset` or `del` commands.
- Restart the kernel and re-run only the cells you need.
    1. What should I do if my notebook kernel becomes unresponsive? If the kernel is unresponsive, you can restart it by going to **Kernel → Restart Kernel**. This will reset the notebook without closing it, allowing you to re-run the cells. If that doesn't work, try closing the notebook and re-opening it from the JupyterHub dashboard.
    2. How do I install new Python libraries in JupyterHub? In many cases, Python libraries are pre-installed. If you need to install a new library, you can try running `!pip install <library-name>` in a notebook cell. However, if you're in a restricted environment, you may need to request that the library be added by an administrator.
    3. What should I do if I encounter a connection error when accessing Kafka or MinIO? Verify your connection details, including the host, access keys, and any authentication tokens required. Ensure the correct endpoint and bucket names are being used for MinIO, or the correct topic for Kafka. If you still have issues, reach out to the administrator.
    4. How do I know which libraries and tools are available in JupyterHub? You can check the available Python libraries by running `!pip list` in a notebook cell. For system tools or external services (like MinIO or Trino), refer to the documentation provided with your JupyterHub environment, or ask your administrator for a list of pre-configured services.

Reference Materials

- JupyterHub Documentation: https://jupyterhub.readthedocs.io/en/stable/
- MinIO Documentation: https://docs.min.io/
- Kafka Documentation: https://kafka.apache.org/documentation/
- Confluent Kafka Python Client: https://docs.confluent.io/platform/current/clients/python.html
- Trino Documentation: https://trino.io/docs/current/

- SQLAlchemy Documentation: https://docs.sqlalchemy.org/en/14/
- Python Requests Library: https://docs.python-requests.org/en/latest/
- Pandas Documentation: https://pandas.pydata.org/docs/

Glossary

- **Avro**: A data serialization format commonly used with Kafka to encode data into a compact, binary format. Avro schemas ensure that data is structured consistently across producers and consumers.
- **Catalog (Trino)**: A namespace in Trino that contains multiple databases. Users can query catalogs in Trino to access datasets stored in the SDL.
- **Consumer (Kafka)**: A process or service that reads data from a Kafka topic. Consumers receive messages sent by producers and process or analyze the data in real-time.
- **JWT (JSON Web Token)**: A token used for securely transmitting information between a client and a server. In this context, JWT tokens are used to authenticate connections to Trino.
- **Kafka**: A distributed streaming platform used for building real-time data pipelines. It allows users to produce (send) and consume (receive) messages to and from topics.
- **Kernel**: The computational engine in Jupyter notebooks that runs your code. You can restart or shut down kernels when they become unresponsive or after completing a task.
- **MinIO**: A distributed object storage system used to store large datasets and files. It is similar to Amazon S3 and allows users to upload, list, and retrieve files via notebooks.
- **Notebook**: An interactive document in JupyterHub that contains code, text, and visualizations. Notebooks are the primary tool for running code, exploring data, and documenting your analysis.
- **Producer (Kafka)**: A process or service that writes (produces) data to a Kafka topic. Producers generate messages that are sent to Kafka topics, where they can be consumed by other services or applications.
- **Pandas**: A Python library used for data manipulation and analysis. It is often used in Jupyter notebooks to load, filter, and analyze data in a structured format (e.g., DataFrames).
- **SQLAlchemy**: A Python SQL toolkit and Object Relational Mapper (ORM) used to connect to databases and run SQL queries. It's used in the Trino notebook to query data from the SDL.
- **Token (Authentication Token)**: A digital key used to authenticate users and authorize access to services, such as Trino or Kafka. In this context, authentication tokens are retrieved from SDL and passed to secure the connection.
- **Topic (Kafka)**: A feed or category in Kafka where messages are published. Producers send data to topics, and consumers read data from these topics for real-time processing or analysis.
- **Trino**: An open-source SQL query engine designed for running interactive queries across distributed data sources. It allows users to query datasets across multiple catalogs in the SDL.

Appendices

Appendix A: Useful JupyterHub Shortcuts

Here are some useful keyboard shortcuts for JupyterHub to make your workflow more efficient:

- Run a cell: `Shift + Enter`
- Insert a new cell below: `B`
- Delete a cell: `D + D` (press `D` twice)
- Save the notebook: `Ctrl + S` or `Cmd + S` (Mac)
- Open command palette: `P`

Appendix B: Code Snippet – Restarting the Kernel

If your notebook becomes unresponsive or you need to free up resources, you can restart the kernel with the following steps: - Go to the top menu and select **Kernel** → **Restart Kernel**.

Appendix C: Example Python Libraries Used

Here are some Python libraries commonly used in the example notebooks and their main purpose:

- **Pandas**: Used for data manipulation and analysis in the notebooks. Pandas allows users to work with tabular data efficiently.
  - Import statement: `import pandas as pd`
- **SQLAlchemy**: Used for database connections and running SQL queries against Trino.
  - Import statement: `from sqlalchemy.engine import create_engine`
- **MinIO Python Client**: Used for interacting with the MinIO object storage service.
  - Import statement: `from minio import Minio`
- **Confluent Kafka**: Used for producing and consuming Kafka messages.
  - Import statement: `from confluent_kafka import Producer, Consumer`

Appendix D: Sample Queries for Trino

Here are some sample SQL queries that can be run in the `trino.ipynb` notebook to query data catalogs: - Show all catalogs:

```
SHOW CATALOGS;
```

- Show all schemas in a catalog:

```
SHOW SCHEMAS FROM <catalog_name>;
```

- List all tables in a schema:

```
SHOW TABLES FROM <catalog_name>.<schema_name>;
```

- Run a simple query:

```
SELECT * FROM <catalog_name>.<schema_name>.<table_name> LIMIT 10;
```

4.6. Geospatial Services

This guide walks you through the process of managing layers in GeoServer, including uploading layers, setting classification metadata, and configuring access rules.

Uploading and Managing Layers

Through the Web Interface

1. Log in to the GeoServer Web Interface
2. Navigate to `Data > Layers`
3. Click `Add a new layer`
4. Select your data source (nasa:blue_marble)
5. Configure the layer settings:
    - Name and title
    - Coordinate reference system
    - Bounding box
    - Coverage parameters and band details

Through the REST API

```
# Upload a new layer (example for Blue Marble)
curl -v -XPUT -H 'Content-type: text/plain' \
     -H "Authorization: Bearer $AUTH_TOKEN" \
     -d $GEOTIFF_PATH \

map.$DF_HOST/geoserver/rest/workspaces/$WORKSPACE/coveragestores/$STORE/external.geoti
ff?configure=first\&coverageName=$LAYER_NAME

# List all layers
curl -H "Authorization: Bearer $AUTH_TOKEN" \
    http://map.$DF_HOST/geoserver/rest/layers

# Get layer details
curl -H "Authorization: Bearer $AUTH_TOKEN" \
    http://map.$DF_HOST/geoserver/rest/layers/$LAYER_NAME
```

Setting Classification Metadata

Users that have been given access to the layer administration page should be able to set a classification keyword on a layer. Users must take precaution to avoid setting an invalid classification, or setting multiple classifications on a layer. Failure to do so will put the layer in an invalid state that will require intervention from the **SDL** admin to correct. For more information, see the "Troubleshooting" section below.

Through the Web Interface

1. Navigate to `Layers > [the layer to classify]`
2. Under `Keywords`, create a new keyword for the classication:
    - Type the layer classification into the `New Keyword` text box
    - In the dropdown, select "English"
    - Type "Classification" into the `Vocabulary` text box
3. Once all information has been inserted, click `Add Keyword` and confirm the classification appears under the `Current Keywords` field
4. Click `Save` at the bottom of the page.

Through the REST API

TBD

Troubleshooting

When a layer's classification is misconfigured, the layer becomes invisible to end users. Once a layer is in this state, a user will have to submit a ticket to **SDL** support for to recover the layer. The ticket should be titled "Suspected Layer Classification Misconfiguration" and should include the following information:

1. The name of the layer
2. The desired classification level of the layer
3. An email used to alert the user that the request has been fulfilled

Configuring Layer Access Rules

Through GeoFence UI

1. Navigate to `Security > GeoFence Data Rules`
2. Click `Add New Rule`
3. Configure the rule:
    - Select Priority
    - Select the Role/User
    - Select services and requests to allow/deny

- Select the workspace and layer
- Define access type (ALLOW, DENY, LIMIT)
- Save the rule

Through the REST API

```
# Create a new access rule (Example for ALLOW all)
curl -XPOST -H "Content-type: application/json" \
  -H "Authorization: Bearer $AUTH_TOKEN" \
  -d '{
      "Rule": {
      "priority":0,
      "userName":"*",
      "roleName":"*",
      "service":"*",
      "request":"*",
      "workspace":"*",
      "layer":"*",
      "access":"ALLOW"
    }
  }' \
  http://map.$DF_HOST/geoserver/rest/geofence/rules

# List all rules
curl -H "Authorization: Bearer $AUTH_TOKEN" \
  http://map.$DF_HOST/geoserver/rest/geofence/rules

# Update an existing rule. Unspecified fields remain unchanged
curl -XPOST -H "Content-type: application/json" \
  -H "Authorization: Bearer $AUTH_TOKEN" \
  -d '{
    "rule": {
      "roleName": "$ROLE_NAME"
      "layer": "$LAYER_NAME",
      "access": "DENY",
    }
  }' \
  http://map.$DF_HOST/geoserver/rest/geofence/rules/id/$RULE_ID
```

4.6.1. Configuration Guide

This guide covers essential configuration tasks for GeoServer in the SDL platform, including GeoWebCache setup, data stores, and security settings.

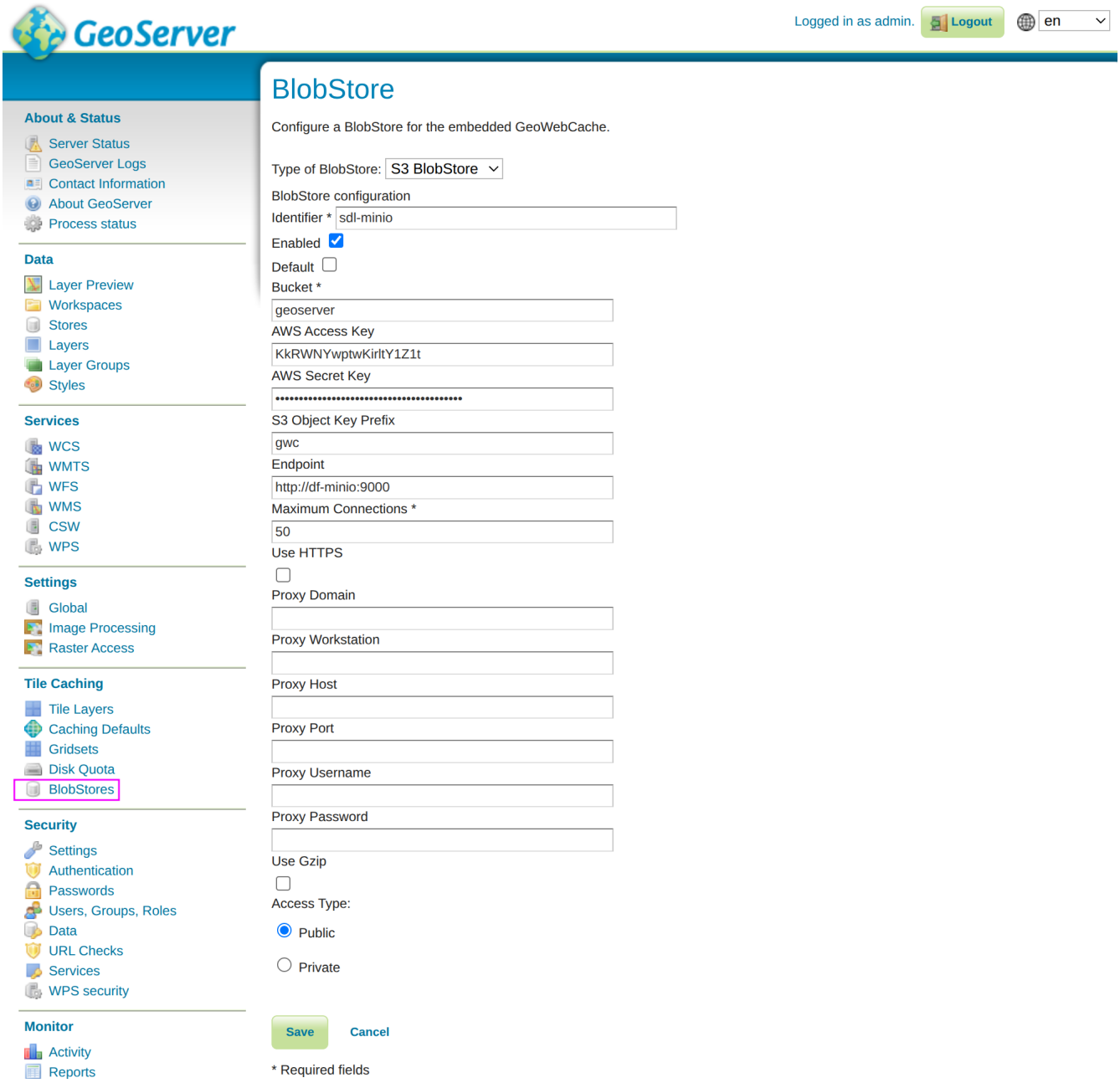
GeoWebCache S3 BlobStore Configuration

GeoWebCache in SDL is configured to use MinIO (S3-compatible storage) for distributed tile caching. This configuration enables multiple GeoServer instances to share the same tile cache and

provides better scalability.

Configuration Steps

1. Navigate to **Tile Caching** > **BlobStores** in the GeoServer admin interface
2. Click **Add new BlobStore**
3. Select **S3 BlobStore** from the Type dropdown
4. Fill in the configuration fields as shown below:



Configuration Field Reference

Field	Description
Identifier	Name for the BlobStore
Enabled	Self-explanatory

Field	Description
Default	Make this the default BlobStore for new layers
Bucket	S3 bucket name for storing tiles
AWS Access Key	MinIO access key (from df-minio secret, get from k9s or see commands below)
AWS Secret Key	MinIO secret key (from df-minio secret, get from k9s or see commands below)
S3 Object Key Prefix	Minio directory name within `geoserver` bucket
Endpoint	MinIO service URL, e.g. http://df-minio:9000
Maximum Connections	Connection pool size
Use HTTPS	Enable HTTPS for S3 connections, off if e.g. http://df-minio:9000
Use Gzip	Enable gzip compression for transfers
Access Type	Public or Private bucket access

Retrieving MinIO Credentials

The MinIO credentials are stored in the `df-minio` Kubernetes secret:

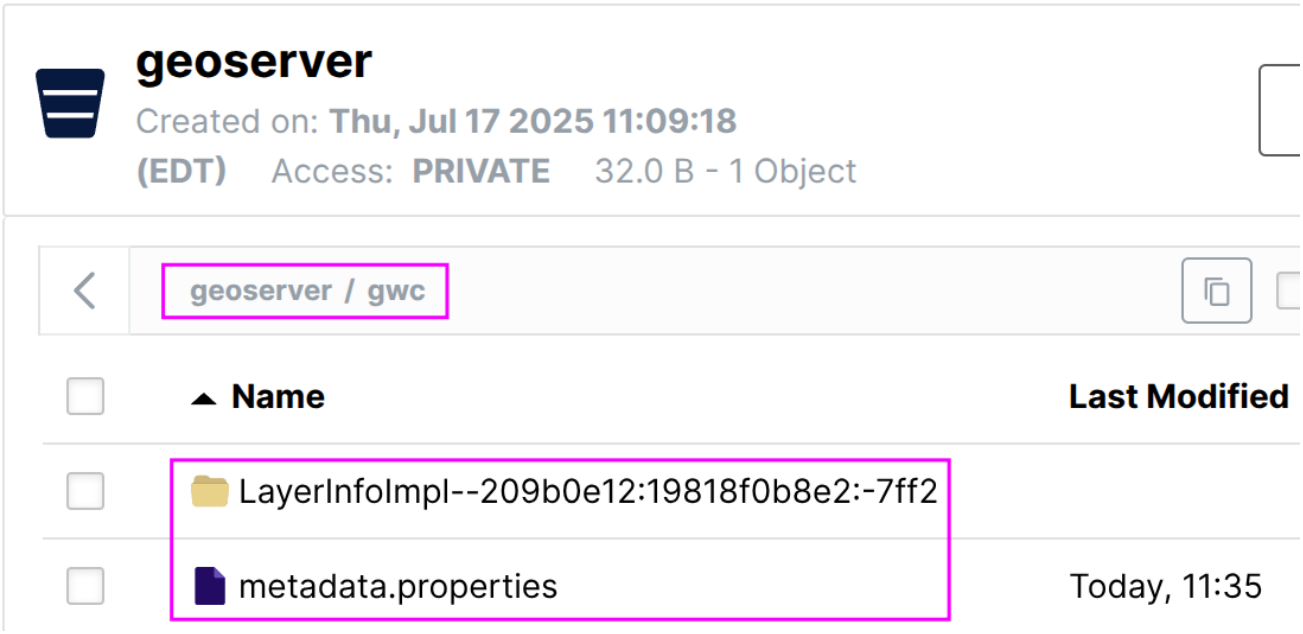
```
# Get MinIO access key
kubectl get secret -n data-fabric df-minio -o jsonpath='{.data.rootUser}' | base64 -d

# Get MinIO secret key
kubectl get secret -n data-fabric df-minio -o jsonpath='{.data.rootPassword}' | base64
-d
```

Verifying Configuration

After saving the BlobStore configuration:

1. Navigate to **Tile Caching** > **Tile Layers**
2. Select a layer to cache
3. In the layer configuration, set **BlobStore** to `sdi-minio`
4. Save the layer configuration
5. Test by going to map.yourURL/geoserver/gwc/demo, clicking `Seed this layer`, submitting a seed job, and checking if tiles appear in MinIO.


GeoServer
About & Status
Server Status
GeoServer Logs
Contact Information
About GeoServer
Process status
Data
Layer Preview
Workspaces
Stores
Layers
Layer Groups
Styles
Services
WCS
WMTS
WFS
WMS
CSW
WPS
Edit Layer
Edit layer data and publishing
nasa:blue_marble
Configure the resource and publishing information for the current layer
Data
Publishing
Dimensions
Tile Caching
Tile cache configuration
Create a cached layer for this layer
Enable tile caching for this layer
Enable In Memory Caching for this Layer.
BlobStore
sdl-minio
Metatiling factors
4
tiles wide by
4
tiles high
Gutter size in pixels
GeoWebCache
List
this Layer tasks
(there are no tasks for other Layers)
Kill
all
Tasks for Layer 'nasa:blue_marble'.
Submit
List of currently executing tasks:
Id
Layer
Status
Type
Estimated # of tiles
Tiles completed
Time elapsed
Time remaining
Tasks
3
nasa:blue_marble
RUNNING
SEED
1,398,109
1,472
5 seconds
1 hour 19 m
(Task 1 of 1)
Kill Task
Refresh list



PostGIS Datastore Configuration

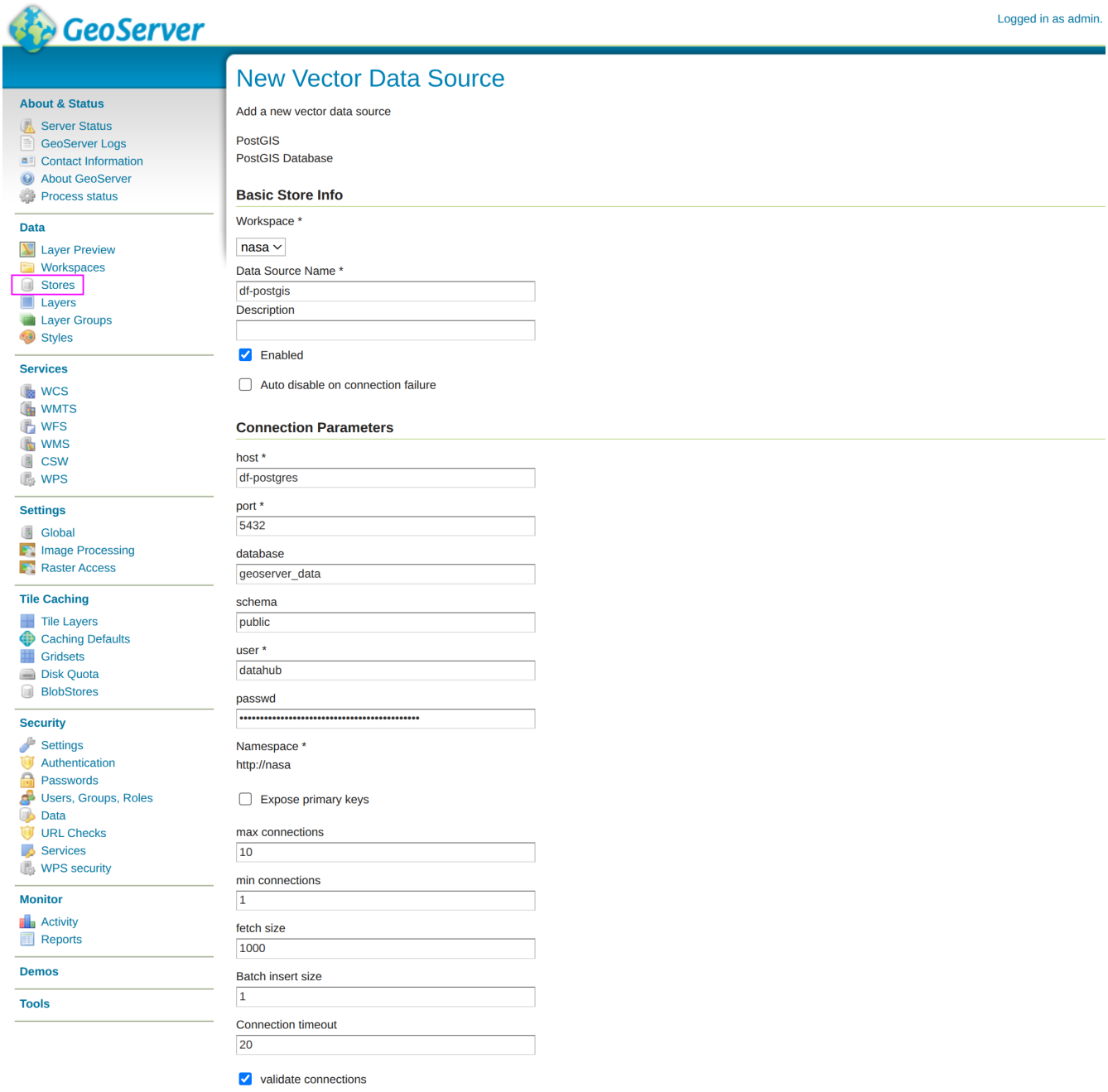
Creating a PostGIS Datastore

1. Navigate to **Data** > **Stores** in GeoServer
2. Click **Add new Store**
3. Select **PostGIS** under Vector Data Sources
4. Configure the connection:

Parameter	Description	Value
Workspace	Workspace for this datastore	`nasa`
Data Source Name	Unique name for this datastore	`df-postgis`
Description	Optional description	`PostGIS spatial database`
host	Database server hostname	`df-postgres`
port	Database server port	`5432`
database	Database name	`geoserver_data`
schema	Database schema	`public`
user	Database username	`datahub`
passwd	Database password	Retrieved from `postgres-secrets (see command below)`
Validate connections	Test connections before use	✔ (checked)

Total connections across all GeoServer instances should not exceed PostgreSQL's `max_connections` setting.

```
# Get Postgres Datahub Password
kubectl get secret -n data-fabric postgres-secrets -o
jsonpath='{.data.postgresDatahubSecret}' | base64 -d
```



Security Configuration

Keycloak Integration Settings

The OIDC plugin configuration for Keycloak integration:

Setting	Value
Client ID	`df-geoserver`
Client Secret	From `keycloak-realm-init` secret

Setting	Value
Discovery URL	`http://df-keycloak/auth/realms/data-fabric/.well-known/openid-configuration`
Roles Claim	`resource_access.df-geoserver.roles`
Send Client Secret	`IN_PARAMS`
Logout URI	`/geoserver/web/logout`

Performance Tuning

JVM Settings

Optimal JVM settings for production:

```
env:
  - name: JAVA_OPTS
    value: >-
      -Xms4g -Xmx8g
      -XX:+UseG1GC
      -XX:MaxGCPauseMillis=200
      -XX:ParallelGCThreads=4
      -XX:ConcGCThreads=2
      -Dfile.encoding=UTF-8
      -Djava.awt.headless=true
      -Djavax.servlet.request.encoding=UTF-8
      -Djavax.servlet.response.encoding=UTF-8
      -Duser.timezone=UTC
      -Dorg.geotools.coverage.jaiext.enabled=true
```

Resource Limits

Configure Kubernetes resource limits appropriately:

```
resources:
  requests:
    cpu: "2"
    memory: "4Gi"
  limits:
    cpu: "4"
    memory: "8Gi"
```

Next Steps

- Infrastructure Guide - Deep dive into deployment architecture
- Layer Management - Managing layers and access control
- API Reference - REST API documentation

4.6.2. Infrastructure & Development

This guide provides comprehensive information for developers and operators working with GeoServer in the SDL platform, covering infrastructure architecture, configuration management, backup strategies, and development workflows.

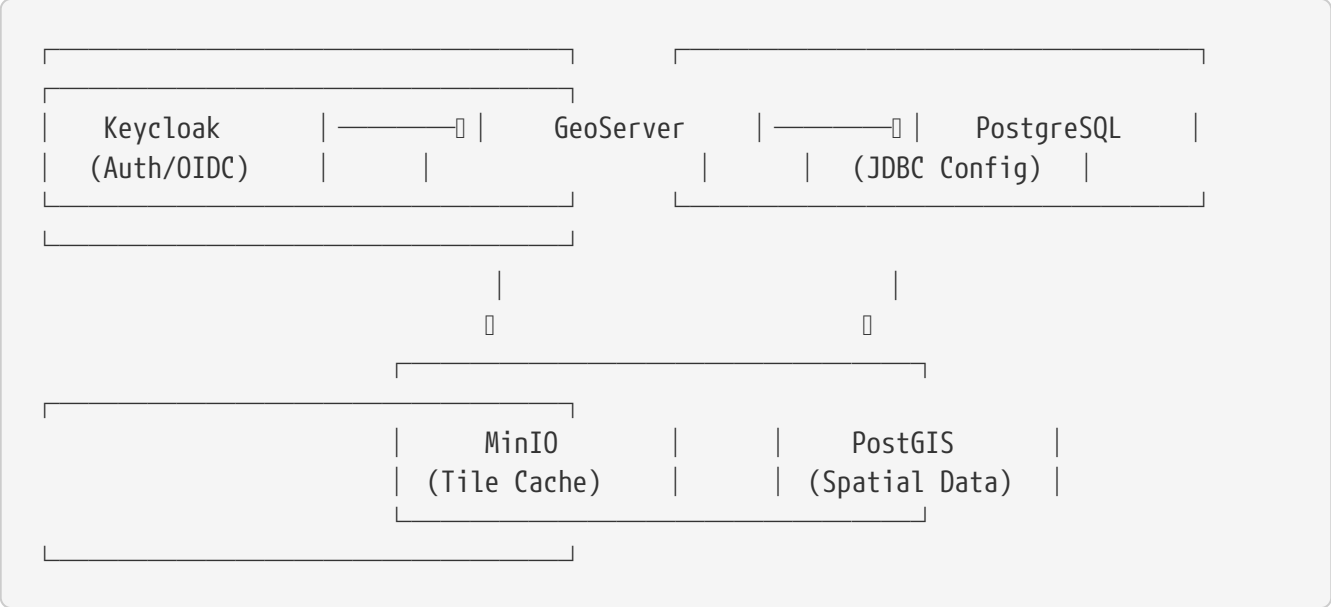
Overview

GeoServer in SDL is deployed as a containerized service with extensive customizations for security, persistence, and integration with other platform components. The deployment leverages:

- **JDBC Configuration Storage** - PostgreSQL-based configuration management
- **Automated Security Backups** - Continuous backup of critical security files
- **Persistent Volume Management** - Selective persistence for non-database data
- **PostGIS Integration** - Native spatial database support
- **MinIO S3 Storage** - Distributed tile caching
- **Keycloak OIDC** - Enterprise authentication and authorization

Architecture

Component Overview



Key Components

1. **GeoServer Container** (`df-geoserver`)
   - Based on Kartoza GeoServer image with custom extensions
   - ABAC (Attribute-Based Access Control) support
   - OIDC/Keycloak integration
   - JDBC Configuration module
2. **Security Backup Service**

- Runs as separate deployment
- Backs up security files every 10 seconds
- Stores in PostgreSQL tables

3. **Init Containers**
   - Database state checker
   - Configuration injector
   - Security restoration

Configuration Management

JDBC Configuration

Most GeoServer configurations are stored in PostgreSQL using the JDBC Config community module. This provides:

- Centralized configuration storage
- Easy backup and restore
- Support for multiple GeoServer instances
- Configuration versioning

Database Schema

```
-- Main configuration database
Database: geoserver_configs

-- Key tables created by JDBC Config:
- object           -- Main configuration objects
- property         -- Object properties
- property_type    -- Property type definitions
- default_object   -- Default configurations
```

Dynamic Initialization

The system automatically detects database state on startup:

```
# Check if JDBC config tables exist
TABLE_EXISTS=$(psql ... -c "SELECT EXISTS (SELECT FROM information_schema.tables WHERE
table_name = 'object');")

if [ "$TABLE_EXISTS" = "t" ]; then
    # Tables exist - check for data
    CONFIG_COUNT=$(psql ... -c "SELECT COUNT(*) FROM object;")

    if [ "$CONFIG_COUNT" -gt "0" ]; then
        # Existing config - don't reinitialize
```

```
        initdb=false
        import=false
    else
        # Empty tables - import from filesystem
        initdb=false
        import=true
    fi
else
    # No tables - full initialization
    initdb=true
    import=true
fi
```

Security Backup System

Critical security files are continuously backed up to prevent data loss during pod restarts or failures.

Backed Up Directories

```
/opt/geoserver/data_dir/security/
├──── usergroup/default/     # User and group definitions
├──── role/default/          # Default role mappings
└──── role/keycloak_service/ # Keycloak role mappings
```

Backup Tables

```
-- Security file backups
CREATE TABLE geoserver_security_backup (
    id SERIAL PRIMARY KEY,
    file_path VARCHAR(255) NOT NULL UNIQUE,
    file_content TEXT NOT NULL,      -- Base64 encoded
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Keystore backup
CREATE TABLE geoserver_keystore (
    id SERIAL PRIMARY KEY,
    jceks BYTEA,                     -- Binary keystore data
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Restoration Process

During startup, the bootstrap script:

1. Checks for backup existence in PostgreSQL
2. Restores files if backups exist
3. Creates marker file (`.security_restored`)
4. Prevents backing up vanilla configurations on first startup

Persistent Volumes

For data that cannot be stored in JDBC Config, selective PVC mounting is used:

```
persistence:
  enabled: true
  size: 1Gi
  directories:
    - gwc                # GeoWebCache configuration
    - gwc-layers         # Layer-specific cache settings
    - user_projections   # Custom CRS definitions
    - styles            # SLD/CSS style files
    - logs              # Application logs
    - monitoring        # Performance data
```

Development Workflows

Local Development Setup

Prerequisites

1. Docker and Docker Compose
2. Access to SDL container registry
3. PostgreSQL client tools
4. kubectl and Helm

Running Locally

```
# Clone repositories
git clone https://github.com/your-org/df-geoserver.git
git clone https://github.com/your-org/sdl.git

# Build custom GeoServer image
cd df-geoserver
docker build -t df-geoserver:local .

# Deploy with Helm (from SDL repo)
cd ../sdl
helm install geoserver kubernetes/helm/charts/geoserver \
  --set image.geoserver.tag=local \
  --set global.namespace=default
```

Adding Extensions

The Kartoza GeoServer image provides multiple ways to add extensions:

Stable Extensions

Enable stable extensions using the `STABLE_EXTENSIONS` environment variable:

```
env:
  - name: STABLE_EXTENSIONS
    value: "charts-plugin,db2-plugin,mongodb-plugin"
```

Use the plugin name without the full filename. For example, use `wps-plugin` instead of `geoserver-2.27.1-wps-plugin.zip`.

Community Extensions

Enable community extensions that are still in development:

```
env:
  - name: COMMUNITY_EXTENSIONS
    value: "gwc-sqlite-plugin,ogr-datastore-plugin,sec-keycloak-plugin"
  - name: FORCE_DOWNLOAD_COMMUNITY_EXTENSIONS
    value: "true"  # Force download of latest versions
```

Default Extensions

By default, these extensions are activated: - `vectortiles-plugin` - Vector tiles support - `wps-plugin` - Web Processing Service - `libjpeg-turbo-plugin` - Optimized JPEG encoding - `control-flow-plugin` - Request flow control - `pyramid-plugin` - Image pyramid support - `gdal-plugin` - GDAL raster formats - `monitor-plugin` - Request monitoring - `inspire-plugin` - INSPIRE compliance - `csw-plugin` - Catalog Service for Web

To customize which default extensions are active:

```
env:
  - name: ACTIVE_EXTENSIONS
    value: "control-flow-plugin,csw-plugin,wps-plugin"
    # This will ONLY activate these three, skipping other defaults
```

Custom JAR Extensions

For custom extensions not available in the repositories:

1. **Build-time approach** - Add to Dockerfile:

   ```
   # In df-geoserver Dockerfile
   ```

```
FROM kartoza/geoserver:${GS_VERSION}

# Add custom extension JARs
COPY --chown=geoserveruser:geoserverusers \
  ./extensions/custom-extension-*.jar \
  /usr/local/tomcat/webapps/geoserver/WEB-INF/lib/

# For {sdl-product-acronym} ABAC extension
COPY --chown=geoserveruser:geoserverusers \
  ./df-geoserver-abac/target/df-geoserver-abac.jar \
  /tmp/df-geoserver-abac.jar
```

2. **Runtime approach** - Configure in bootstrap.sh:

```
# Enable custom extensions conditionally
if [[ "$GEOSERVER_ABAC" =~ ^[Tt][Rr][Uu][Ee]$ ]]; then
  echo "Support for GeoServer ABAC has been enabled."
  cp /tmp/df-geoserver-abac.jar \
     /usr/local/tomcat/webapps/geoserver/WEB-INF/lib/df-geoserver-abac.jar

  # Set proper ownership
  chown geoserveruser:geoserverusers \
     /usr/local/tomcat/webapps/geoserver/WEB-INF/lib/df-geoserver-abac.jar
fi
```

3. **Volume mount approach** - For development:

```
volumes:
  - name: custom-extensions
    hostPath:
      path: /path/to/extensions
      type: Directory

volumeMounts:
  - name: custom-extensions
    mountPath: /opt/geoserver/data_dir/extensions

env:
  - name: GEOSERVER_DATA_DIR_EXTENSIONS
    value: "/opt/geoserver/data_dir/extensions"
```

Extension Configuration Example

Complete example combining multiple extension types:

```
# In values.yaml or deployment.yaml
env:
  # Stable extensions
```

```
  - name: STABLE_EXTENSIONS
    value: "monitor-plugin,importer-plugin,web-resource-plugin"

  # Community extensions
  - name: COMMUNITY_EXTENSIONS
    value: "sec-keycloak-plugin,jdbcconfig-plugin"

  # Force latest community versions
  - name: FORCE_DOWNLOAD_COMMUNITY_EXTENSIONS
    value: "true"

  # Custom extensions
  - name: GEOSERVER_ABAC
    value: "true"

  # Override default active extensions
  - name: ACTIVE_EXTENSIONS
    value: "control-flow-plugin,monitor-plugin,wps-plugin,csw-plugin"
```

Creating Custom Datastores

PostGIS Datastore Example

```
<dataStore>
  <name>df-postgis</name>
  <type>PostGIS</type>
  <enabled>true</enabled>
  <connectionParameters>
    <host>df-postgres</host>
    <port>5432</port>
    <database>spatial_data</database>
    <user>datahub</user>
    <passwd>${POSTGRES_PASSWORD}</passwd>
    <dbtype>postgis</dbtype>
    <schema>public</schema>
    <Loose bbox="">true</Loose>
    <Estimated extends="">true</Estimated>
    <validate connections="">true</validate>
    <Connection timeout="">20</Connection>
    <min connections="">1</min>
    <max connections="">10</max>
  </connectionParameters>
</dataStore>
```

Implementing Custom Security

ABAC Extension

The custom `df-geoserver-abac` extension provides attribute-based access control:

```
// Example ABAC rule evaluation
public boolean evaluate(Authentication auth, Resource resource) {
    // Get user attributes from JWT
    Map<String, Object> attributes = extractAttributes(auth);

    // Check against resource requirements
    return checkAccess(attributes, resource.getRequiredAttributes());
}
```

Keycloak Integration

Token validation and role extraction:

```
# OIDC configuration
oidc:
  clientId: df-geoserver
  clientSecret: ${GEOSERVER_CLIENT_SECRET}
  discoveryUrl: http://df-keycloak/auth/realms/data-fabric/.well-known/openid-
configuration
  rolesClaim: resource_access.df-geoserver.roles
```

Integration Examples

Accessing GeoServer from Applications

Python Example

```
import requests
from requests.auth import HTTPBasicAuth

class GeoServerClient:
    def __init__(self, base_url, token):
        self.base_url = base_url
        self.headers = {'Authorization': f'Bearer {token}'}

    def get_layers(self):
        response = requests.get(
            f"{self.base_url}/rest/layers",
            headers=self.headers
        )
        return response.json()

    def get_wms_image(self, layers, bbox, width=800, height=600):
        params = {
            'service': 'WMS',
            'version': '1.1.0',
            'request': 'GetMap',
            'layers': layers,
```

```
            'bbox': bbox,
            'width': width,
            'height': height,
            'srs': 'EPSG:4326',
            'format': 'image/png'
        }
        response = requests.get(
            f"{self.base_url}/wms",
            params=params,
            headers=self.headers
        )
        return response.content
```

JavaScript/OpenLayers Example

```
// Initialize map with GeoServer WMS layer
const wmsLayer = new ol.layer.Tile({
  source: new ol.source.TileWMS({
    url: 'https://map.example.com/geoserver/wms',
    params: {
      'LAYERS': 'nasa:blue_marble',
      'TILED': true
    },
    serverType: 'geoserver',
    crossOrigin: 'anonymous'
  })
});

// Add authentication header
wmsLayer.getSource().setTileLoadFunction((tile, src) => {
  const xhr = new XMLHttpRequest();
  xhr.open('GET', src);
  xhr.setRequestHeader('Authorization', `Bearer ${authToken}`);
  xhr.responseType = 'blob';
  xhr.onload = () => {
    tile.getImage().src = URL.createObjectURL(xhr.response);
  };
  xhr.send();
});
```

Federation

Multi-Instance Deployment

For high availability, deploy multiple GeoServer instances sharing the same JDBC config:

```
# In values.yaml
replicaCount: 3
```

```
# Ensure all instances use same database
env:
  - name: POSTGRES_HOST
    value: df-postgres-primary  # Use primary for writes
```

Federation with Other SDL Nodes

Configure cascading WMS/WFS to federate layers from other nodes:

```
<remoteStore>
  <name>remote-sdl-node</name>
  <type>WMS</type>
  <capabilitiesURL>
    https://remote.sdl.example.com/geoserver/wms?request=GetCapabilities
  </capabilitiesURL>
  <username>federation-user</username>
  <password>${FEDERATION_PASSWORD}</password>
</remoteStore>
```

References

- GeoServer Documentation
- Kartoza Docker GeoServer
- JDBC Config Module
- GeoWebCache Documentation
- Keycloak Integration Guide
- MinIO S3 Storage Guide

4.7. Logging

SDL Developers are required to adhere to a standardized logging format in support of ATO / STIG compliance. Given that SDL applications are language agnostic, the following document outlines a JSON logging format with example implementations in a variety of languages found within the system. Where possible, usage of existing logging libraries or creation of library wrappers around these tools is encouraged.

Format and Content Requirements

All log events must be JSON and minimally contain the following fields:

Required Fields

- `level`
  - Severity level of the emitted event
  - Actual values are library dependent, but should generally be FATAL, ERROR, WARN, INFO,

DEBUG, or TRACE

- `time`
  - ISO 8601 UTC timestamp minimally with millisecond precision
  - Format: `YYYY-MM-DDTHH:mm:ss.sssZ`
- `appId`
  - A unique application identifier
- `appVersion`
  - The application version
- `class`
  - The fully qualified class name, including the package/module
- `method`
  - The calling function or method
- `line`
  - Line number of the log event occurrence
- `message`
  - The message for the event, subject to additional requirements depending on context
- `entity`
  - The entity, username, or user ID associated with the event
  - For applications or code where this information is not applicable, use a value of `system`

Optional Fields

Additional fields can be specified on a per-application basis. For example:

- `traceId`
  - If set, used to correlate a chain of events across different services
  - For example, a traceId may be generated by a user interacting with an API endpoint. This traceId is then passed to all follow-on calls across downstream services so that a single lineage can be constructed
  - Trace IDs may be particularly useful for correlating logs within Pipelines
- `spanId`
  - If set, used to identify a specific operation within a trace

Log Event Emit Requirements

- Log events marked with an appropriate severity level and runtime configuration ensure that only events up to the desired level are reported
- Within the application, a log event must be generated for any data access or modification by a user

    - For example, an API service must log the user and action taken on a given endpoint
- When communicating with an external service, the service name and IP address must be logged
- Sensitive information such as tokens or passwords must not be logged. If a message is emitted with this information, the content must be masked

Language-Specific Logging Libraries

The following libraries are easily configured to adhere to the above format and content restrictions:

- **Java**
    - Slf4j with Logback
- **Go**
    - Zerolog
- **Python**
    - Python JSON Logger

Sample Log Events

Standard Application Event

```
{
  "level": "DEBUG",
  "time": "2025-03-15T12:15:40.555Z",
  "appId": "json-processor",
  "appVersion": "v1.0.0",
  "class": "com.example.processing.DataHandler",
  "method": "transformJsonData",
  "line": 99,
  "message": "Attempting to transform json data '{\"key\":\"val\"}'",
  "entity": "system",
  "traceId": "a64af53a-acb4-4db3-b328-c9e504b0e1d1",
  "spanId": "16e2dcfa-2be6-49ab-8a21-a8841a77b020"
}
```

Standard User Event

```
{
  "level": "INFO",
  "time": "2024-01-15T14:30:25.123Z",
  "appId": "user-service",
  "appVersion": "v1.2.3",
  "class": "com.example.userservice.UserController",
  "method": "getUserById",
  "line": 45,
  "message": "Successfully retrieved user profile",
```

```
  "entity": "userAbc"
}
```

Event With Masked Sensitive Information

```
{
  "level": "INFO",
  "time": "2024-01-15T15:35:12.653Z",
  "appId": "user-service",
  "appVersion": "v1.2.3",
  "class": "com.example.userservice.UserController",
  "method": "authenticateUser",
  "line": 150,
  "message": "Authenticating user with token: ***",
  "entity": "userAbc"
}
```

Chapter 5. Federation

Federation enables multiple SDL instances to communicate and share data with each other. Federation provides deliberate, controlled data exchange between echelons for any data type.

Key Capabilities

Controlled Data Exchange

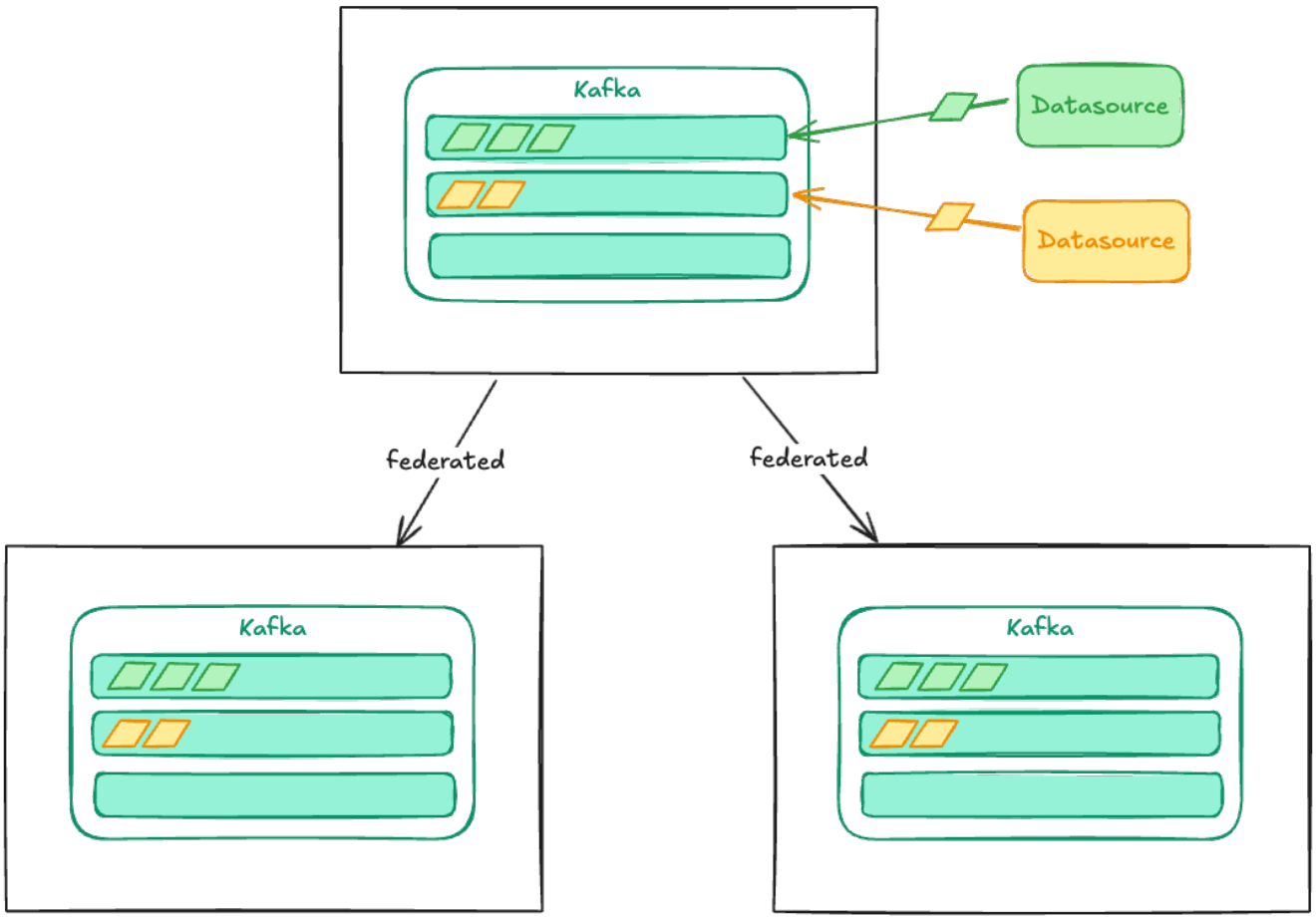
Federation supports intentional data movement between SDL nodes for arbitrary data sets — streaming topics, files, and other data. Operators configure which data flows between which nodes.

Hierarchical Echelon Support

Federation topology mirrors operational command structures. Data flows can be configured to match hierarchical echelon architectures (e.g., Platoon → Company → Battalion), with bidirectional or unidirectional edges depending on the use case.

Classification Boundary Awareness

Federation respects classification boundaries. Data marked at a specific classification level can only federate to nodes authorized at that level or above, enforced by the policy engine.



See Management for configuring federation.

5.1. Management

Federation is managed from **Admin** → **Federation**, available to users with the `federation-admin` role.

Add Node

Adding a node makes that node available for federation, but does not federate with it yet (that happens when it is enabled).

A quick check of the node's availability and identification information (name, version, classification) is performed, so it can be included in the table listing.

It's OK if the node is unavailable at the time it is added. Its identification information will be checked again when it is enabled.

1. Click the **Add Node** button in the upper right.
2. Input the URL of the other node to be federated with.

The URL should not contain any trailing forward slash `"/"` character at the end of the URL.

3. Click **Confirm**.
4. The new node is added to the table. If the node is available, it's `Name`, `Version` and `Classification` details will be set.

Enable Node

Enabling a node initiates federation with the remote node.

You will be prompted for the client credentials the local node is to use when communicating with the remote node. These credentials are stored as long as federation with the node is enabled.

1. In the **Actions** column, click the network icon. Its tooltip should display the text **"Enable"**, if it says **"Disable"** then it is already enabled.
2. This will present a modal asking for OAuth Credentials for the target node. This should be a client that has privileges to pull data from the remote node. Typically, this can be the `df-federation` client from the remote node's Keycloak.
3. Click **Enable**.

The client credentials you use to enable the federated node must have access to any/all data you wish to have federated access to. For example, catalog data can be restricted to certain user groups. In order for group-protected data to be federated, the client must also be part of those groups.

At this point the local node will begin crawling the remote node's catalog in the background. It is synchronizing the local catalog with the datasources and datasets that are available from the

remote node.

These will appear as "Federated" enablements in the local catalog.

Disable Node

Disabling a node stops all federation activity with the remote node, and removes all remnants of the federated enablements from the local catalog.

This is a destructive operation to the local catalog. Any data that has been pulled from the remote node is deleted when its federated enablements are removed.

To **disable** a node, click the same network icon you used to enable it.

To **delete** a node, click the trash can icon.

Deleting a node performs both disable and delete in one operation.

Delete Node

Deleting a node removes it from the federation list, and forgets the node entirely.

If the node happens to be enabled, it is first disabled and then deleted.

Chapter 6. Security

This section covers security-related details of SDL.

Ingress Networking & Authentication

SDL exposes port `443` for the majority of network traffic, with the exception being Postgres, which listens on port `5432`. All inbound network traffic is handled by the NGINX Ingress kubernetes proxy.

SDL Data Access Traffic:

- `HTTPS/443`
  - API traffic is forwarded by NGINX Ingress to the SDL internal API Gateway, where the request is authenticated first against the internal Keycloak before being forwarded on to its destination service.
    - All internal `REST` communications uses `Bearer` tokens service-to-service.
    - `Basic` username/password credentials are automatically converted to `Bearer` tokens at the API Gateway.
  - Static resource requests, like those for loading web clients for various tools in SDL, are routed by NGINX Ingress directly to the target service.
- Trino `JDBC/ODBC` clients use `TCP/443` under the hood, and are routed from NGINX Ingress directly to Trino. Trino traffic is secured by Keycloak issued tokens.
- Kafka uses a custom binary `TCP` based protocol, and is routed by NGINX Ingress directly to Kafka. Kafka traffic is secured by `SASL/OAUTHBEARER`, which is SASL protocol with Keycloak issued tokens.
- `s3` traffic also uses `HTTPS/443` under the hood, and are routed from NGINX Ingress directly to MinIO. SDL MinIO supports Keycloak based authentication via MinIO's `Secure Token Service (STS)`, which authenticates users against their keycloak credentials, and issues temporary tokens similar to those used in `OAUTH/OIDC` flows.
- Postgres clients use `JDBC/5432`, and that traffic is routed from NGINX Ingress directly to Postgres. Postgres uses internal `Basic` authentication. Keycloak authentication support via Postgres Pluggable Authentication Modules (PAM) is planned for a future PI.

TLS terminates at NGINX Ingress.

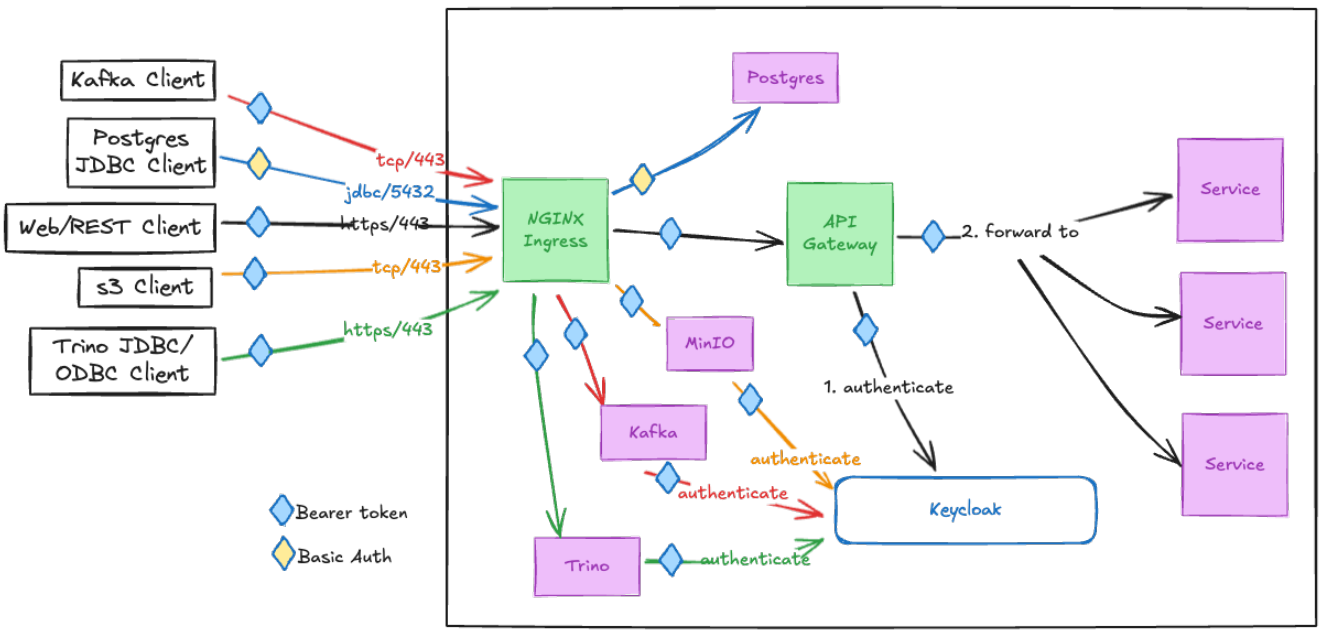


Figure 3. Ingress Overview

Ingress Routes

Protocol/Port	Subdomain	Path(s)	Destination Service	Authentication
https/443		/api/**	df-api-gateway / http/80	Bearer Token, Basic Auth
https/443		/auth/**	df-keycloak / http/8080	Bearer Token
https/443		/backend/**	df-backend / http/8180	Bearer Token
https/443		/, /api/auth, /api/frontend	df-frontend / http/3000	Bearer Token
tcp/443	kafka-bootstrap		df-kafka-external-bootstrap / tcp/9095	SASL/OAUTHBEARER
tcp/443	kafka-broker-0		df-kafka-external-0 / tcp/9095	SASL/OAUTHBEARER
tcp/443	kafka-broker-1		df-kafka-external-1 / tcp/9095	SASL/OAUTHBEARER
tcp/443	kafka-broker-2		df-kafka-external-2 / tcp/9095	SASL/OAUTHBEARER
https/443	s3		df-minio / http/9000	MinIO STS token
https/443	trino		df-raft-trino / http/8080	Bearer Token
jdbc/5432	postgres.dbaas		df-postgres-external / jdbc/5432	Basic Auth

Protocol/Port	Subdomain	Path(s)	Destination Service	Authentication
https/443	grafana		grafana / http/3000	Bearer Token
https/443	hub		df-jupyterhub / http/80	Bearer Token
https/443	minio		df-minio-console / http/9001	MinIO STS Token
https/443	map		geoserver / http/80	Basic Auth

APIs

All the available APIs are listed on the main API landing page rooted at https://DF_HOST/api.

Endpoint	Description
/api/id	General identification of a cluster.
/api/health	Get health status of a cluster.
/api/test	Test auth credentials and basic connectivity.
Platform Services	
/api/proxy	Proxy to external services, with payload caching.
/api/v1/auth	Auth utilities (get a token, etc).
/api/v1/courier	Drop data into SDL.
/api/v1/kafka	Connect to Kafka.
/api/v1/s3	S3 object storage.
Data Services	
/api/v2/catalog	Data catalog.
/api/v1/lakehouse	Data lake APIs.

Data Access Authorization

SDL primary data access authorization strategy is a hybrid of the 3 following techniques:

1. Role Based Access Control (RBAC)
2. Attribute Based Access Control (ABAC)
3. Policy Based Access Control (PBAC)

Role Based Access Control (RBAC)

Role Based Access Control is a course-grained authorization strategy that relies on checking the requestors membership in specific roles/groups in order to grant access to resources.

SDL tends to use RBAC to authorize access to different datasets, or perform admin functions across

various services. This authorization is performed at the service layer.

Attribute Based Access Control (ABAC)

Attribute Based Access Control is a fine-grained authorization strategy that is based on attributes of the user, resource, action, and environment. ABAC enforces policies dynamically by considering contextual data such as:

- User attributes (e.g., role, department, clearance level).
- Resource attributes (e.g., classification, owner, category).
- Action attributes (e.g., read, write, delete).
- Environmental attributes (e.g., time of access, location).

Policy Based Access Control (PBAC)

Policy Based Access Control is a fine-grained authorization strategy that uses a collections of predefined policies as rules-based decision making in order to grant access to resources.

Policies can be role-based (RBAC), attribute-based (ABAC), or other criteria.

SDL provides end (admin) user configurable authorization policies powered by Open Policy Agent (OPA), an Open Source, Rego based policy engine. This allows data access and authorization business rules to be implemented as configuration and easily modifiable due to evolving requirements, rather than hardcoded into custom applications.

SDL's core implementation of classification, dissemination, and ACCM data access controls are implemented as OPA policies, based on the standardized `Information Security Model (ISM)` adopted across the program. This classification/dissemination control metadata structure is in every SDL data payload, or tagged to s3 objects.

Access control logic is split across multiple files by domain and function. Some files contain plumbing to connect to Keycloak to reference the user's metadata, such as Roles, Groups, and Attributes.

OPA listens to REST calls on `8181`, and serves the following types of requests:

1. authorization checks
2. policy management
3. telemetry

None of the above functions are directly exposed outside of SDL.

OPA in SDL

The diagram below shows the connectivity of OPA to various major services in SDL. OPA references SDL Keycloak as needed to grab the roles, groups, and attributes of the user that OPA is performing an access decision on. The Keycloak attributes contain the user's clearance, nationality, compartments and program reasons, etc.

SDL Keycloak in turn federates with the larger program Keycloak/Identity Provider (IDP) solution to get the federated users, attributes, and any program wide roles and groups.

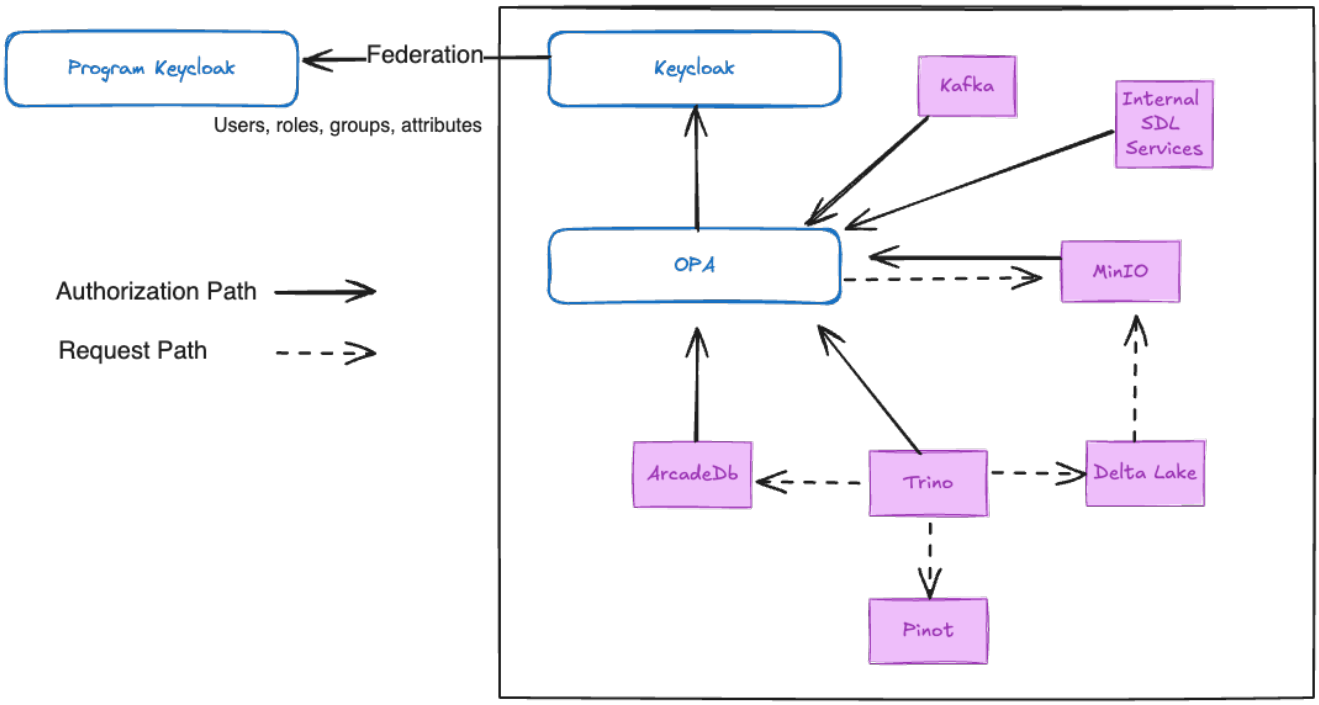


Figure 4. Authorization Overview

There are two patterns of OPA authorization in SDL:

Resource based authorization

Resource based authorization requests are when authorization checks to individual resources are made, one at a time, to OPA, and authorized if the requesting user has sufficient clearance and Need to Know (NTK) to the requested resource. These are used in the following ways:

1. Kafka
    a. Classification/dissemination based authorization checks for kafka topics, leveraging the program ISM tags to filter available topics for a user..
2. MinIO (s3)
    a. Classification/dissemination based authorization checks for each object in any requested bucket, leveraging the program ISM tags on the objects.

Predicate based authorization

Predicate based authorization requests are when the shape of a user's data access is exported from OPA, and converted into query predicates. Those query predicates are transparently applied to the original user data request to limit the data returned or modified to just what they have access to.

This technique is extremely useful when data access controlling query surfaces over large datasets, where lazily checking permission to every single object would be prohibitively expensive. Each datastore or query surface that uses this technique will have applied the program adopted ISM data access controls to every data element in their system.

The following query surfaces/datastores use predicate based authorization:

1. Delta Lake (via Trino)
2. Pinot (via Trino)
3. Trino

SBOM

Below is the list of 3rd party images used.

```
{registry}/dockerhub/apache/flink-kubernetes-operator:1.8.0
{registry}/dockerhub/bitnami/kafka:3.5.1
{registry}/dockerhub/bitnami/os-shell:12
{registry}/dockerhub/bitnami/schema-registry:7.8.0
{registry}/dockerhub/bluenviron/mediamtx:1.9.0
{registry}/dockerhub/busybox:1.36
{registry}/dockerhub/curlimages/curl:8.12.1
{registry}/dockerhub/datarhei/restreamer:2.11.0
{registry}/dockerhub/dpage/pgadmin4:2024-05-28-1
{registry}/dockerhub/getmeili/meilisearch:v1.4.0
{registry}/dockerhub/grafana/grafana:9.2.4
{registry}/dockerhub/grafana/loki:2.4.2
{registry}/dockerhub/jimmidyson/configmap-reload:v0.5.0
{registry}/dockerhub/jupyterhub/k8s-image-awaiter:3.0.0
{registry}/dockerhub/jupyterhub/k8s-network-tools:3.0.0
{registry}/dockerhub/jupyterhub/k8s-secret-sync:3.2.1
{registry}/dockerhub/kindest/node:v1.30.0
{registry}/dockerhub/nginx:1.27.2
{registry}/dockerhub/postgis/postgis:16-3.4
{registry}/dockerhub/postgres:14
{registry}/dockerhub/provectuslabs/kafka-ui:v0.7.1
{registry}/dockerhub/swaggerapi/swagger-ui:v5.18.2
{registry}/dockerhub/liquibase/liquibase:4.25
{registry}/dockerhub/redis:7.0.11-alpine
{registry}/dockerhub/shadowtraffic/shadowtraffic:0.6.3
{registry}/dockerhub/wiremock/wiremock:3.11.0
{registry}/dockerhub/yuzutech/kroki:0.25.0
{registry}/dockerhub/yuzutech/kroki-bpmn:0.25.0
{registry}/dockerhub/yuzutech/kroki-excalidraw:0.25.0
{registry}/dockerhub/yuzutech/kroki-mermaid:0.25.0
{registry}/gcr/kubebuilder/kube-rbac-proxy:v0.8.0
{registry}/ghcr/raft-tech/df-geoserver:1.14.1
{registry}/ghcr/raft-tech/df-postgis:1.15.6
{registry}/ghcr/stakater/reloader:v1.0.121
{registry}/ironbank/bitnami/zookeeper:3.9.3
{registry}/ironbank/afrl-dcgs/stream/ingress-nginx-controller:v1.9.4
{registry}/ironbank/opensource/apache-pinot:1.2.0
{registry}/ironbank/opensource/grafana/promtail:v2.9.4
{registry}/ironbank/opensource/jupyterhub/configurable-http-proxy:4.6.1
```

```
{registry}/ironbank/opensource/jupyterhub/k8s-hub:4.0.0
{registry}/ironbank/opensource/kubernetes/kube-state-metrics:v2.8.0
{registry}/ironbank/opensource/minio/minio:RELEASE.2024-06-04T19-20-08Z
{registry}/ironbank/opensource/minio/mc:RELEASE.2024-11-17T19-35-25Z
{registry}/ironbank/opensource/openpolicyagent/opa:0.61.0
{registry}/ironbank/opensource/redis/redis7:7.2.4
{registry}/k8s/ingress-nginx/controller:v1.9.4
{registry}/k8s/ingress-nginx/kube-webhook-certgen:v1.1.1
{registry}/k8s/kube-scheduler:v1.19.15
{registry}/k8s/pause:3.5
{registry}/ironbank/jetstack/cert-manager-cainjector:v1.9.1
{registry}/ironbank/jetstack/cert-manager-controller:v1.9.1
{registry}/ironbank/jetstack/cert-manager-ctl:v1.9.1
{registry}/ironbank/jetstack/cert-manager-webhook:v1.9.1
{registry}/quay/kiwigrid/k8s-sidecar:1.19.2
{registry}/quay/prometheus-operator/prometheus-config-reloader:v0.60.1
{registry}/quay/prometheus-operator/prometheus-operator:v0.60.1
{registry}/quay/prometheus/alertmanager:v0.24.0
{registry}/quay/prometheus/node-exporter:v1.3.1
{registry}/quay/prometheus/prometheus:v2.39.1
{registry}/quay/strimzi/kafka:0.44.0-kafka-3.8.0
{registry}/quay/strimzi/operator:0.44.0
```

Chapter 7. Admin

7.1. Configuration Management

This document outlines the configuration management strategy for SDL. The primary approach is decentralized configuration management. This ensures consistency and simplifies updates across all deployments. This approach emphasizes distributed control, autonomy, and flexibility while maintaining consistency and governance.

Whenever possible, use GitOps principles, managing our configurations as code in a Git repository.

Key Principles

1. **Autonomy**: Each instance of SDL must have control over their own services and configurations
2. **Consistency**: Shared standards and best practices across all instances of SDL
3. **Transparency**: Configurations are visible and auditable across the organization
4. **Flexibility**: Each instance of SDL can adapt configurations to a specific need
5. **Governance**: Providing central oversight to ensures security and compliance across instances of SDL

Implementation Details

- SDL uses Helm for packaging and deploying applications with default and minimal configuration overrides
- Short-lived environment should leverage the Hacks Scripts and [Override Strcture] to deploy
- Lond lived environments should leverage ArgoCD and IaC for their deployment
- Each instance of SDL requires a Git repository for its unique configurations leveraging Customize for environment-specific customizations to overlay changes that are instance-specific
- Changes made at the root helm repo for SDL propagate to every instance, unless that instance has an override specific to it
- SDL can leverage HashiCorp Vault with a federated setup for distributed secrets management
- SDL can leverage Terraform to manage our infrastructure as code as well

Disaster Recovery

The ability to recreate an instance of SDL is built into the system. SDL uses Velero for cross-cluster backup and restore See df-backup-restore for more details

Challenges and Mitigations

Challenge	Mitigation
Configuration drift	Regular compliance checks and automated remediation
Increased complexity	Comprehensive documentation and training programs
Performance impact of federated services	Careful performance testing and optimization
Security risks from decentralized control	Strict policy enforcement and regular security audits

Edge Deployments and Policy Management

Edge deployments in our Kubernetes-based data platform introduce unique challenges and opportunities for configuration management. This section outlines how we handle policy changes at the edge and the process of suggesting these changes back to the central deployment.

Edge Deployment Architecture

Our edge deployments consist of:

- Lightweight Kubernetes clusters (e.g., KinD)
- Local data processing and storage capabilities
- Reduced set of services compared to central deployment
- Local policy enforcement mechanisms

Policy Management at the Edge

1. **Local Policy Enforcement:**
    - Use Open Policy Agent (OPA) for policy enforcement at edge locations
    - Deploy a subset of central policies to edge clusters
2. **Policy Customization:**
    - Allow limited policy customization for edge-specific requirements
    - Use Kustomize overlays for edge-specific policy adjustments
3. **Policy Synchronization:**
    - Implement a pull-based model for edge clusters to fetch policy updates
    - Use GitOps principles for policy distribution

Suggesting Policy Changes from Edge to Central

We implement a feedback loop that allows edge deployments to suggest policy changes back to the central deployment:

1. **Change Proposal Process:**
    - Edge cluster administrators can propose policy changes through a Git-based workflow

    - Use pull requests to submit change proposals
2. **Automated Validation:**
    - Implement CI/CD pipelines to validate proposed policy changes
    - Use policy simulation tools to assess the impact of changes
3. **Review and Approval:**
    - Central platform team reviews proposed changes
    - Collaborate with edge administrators to refine proposals
4. **Integration and Testing:**
    - Merge approved changes into a staging environment
    - Conduct thorough testing, including simulation of edge scenarios
5. **Gradual Rollout:**
    - Implement a phased rollout of policy changes
    - Use feature flags or policy versioning for controlled deployment

Example: Edge-to-Central Policy Suggestion Workflow

1. Edge administrator identifies a need for a policy adjustment
2. Administrator creates a branch in the policy Git repository
3. Policy changes are made and committed to the branch
4. A pull request is opened with the proposed changes
5. CI/CD pipeline runs automated tests and policy simulations
6. The central platform team reviews the proposal
7. Iterative feedback and adjustments are made
8. Approved changes are merged into the staging environment
9. Comprehensive testing is performed
10. Changes are gradually rolled out to production, including edge deployments

Challenges and Considerations

- **Connectivity:** Ensure the policy suggestion process works with intermittent connectivity
- **Conflict Resolution:** Develop strategies for resolving conflicting policy suggestions from multiple edge deployments
- **Performance Impact:** Assess the performance impact of policy changes on edge deployments with limited resources
- **Compliance:** Ensure all policy changes meet regulatory and compliance requirements across different edge locations

7.2. DBaaS: Managed Postgres

Overview

This guide walks through the process of creating a new managed PostgreSQL database instance for a user, including user creation and initial setup.

Prerequisites

- Administrative access to the target cluster, specifically command line and k9s

Steps

1. Connect to Kubernetes Cluster

Obtain the hostname/IP of the target Kubernetes cluster or appropriate jump server, and switch local kubectl context as needed.

2. Launch K9s CLI Tool

1. Run the following command.

```
k9s
```

1. Navigate to the appropriate namespace containing the PostgreSQL cluster as needed.

3. Open a second terminal (recommended)

Follow steps 1 and 2 again. You will need to reference randomly generated credentials, and copy/paste is easier.

4. Get the external postgres cluster superuser credentials

1. In k9s, switch to secrets
2. Scroll/search for `df-pg-external-superuser`, and press `x` to decode.

5. Connect to PostgreSQL Cluster

1. In the second terminal - with k9s, go to `services`
2. Shell into `df-pg-external-rw`
3. Run the following command, replace the placeholder value with the username from the secret.

```
psql -h df-pg-external-rw -U postgres -d postgres
```

1. When prompted, enter the password from the secret.

6. Generate random initial password for new user

Still in the psql cli session, run the following command.

```
SELECT
string_agg(substr('ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789',
(random() * 61)::int + 1, 1), '') AS secure_password
FROM generate_series(1, 24);
```

7. Create New User Role

1. Run the following command. Replace `<username>` with the designated username (e.g., `vendor_a`), and the password with the previously generated string.

```
CREATE ROLE <username> WITH
    LOGIN
    PASSWORD '<generated_secure_password>'
    NOSUPERUSER
    NOCREATEDB
    INHERIT;
```

8. Create New Database

Replace `<database_name>` with the requested database name, and username with the previously created user

```
CREATE DATABASE <database_name>
    OWNER <username>
    ENCODING 'UTF8';
```

9. Send Welcome Email

Send two separate emails to the requestor:

Email 1: Welcome and Access Details

```
Subject: Your Raft Managed PostgreSQL Database is Ready

Hello [Requestor Name],

Your new managed PostgreSQL database has been created with the following details:

Host: postgres.dbaas.<SDL deployed FQDN>
Port: 5432
Database: <database_name>
Username: <username>
```

```
You will receive a separate encrypted email containing your initial database password.

Please confirm that you can successfully connect to your database at your earliest
convenience.

If you experience any issues, please contact our support team.

Best regards,
<your name>
```

Email 2: Encrypted Password

Send a separate encrypted email containing only the initial password.

Security Notes

- Always use strong, randomly generated passwords
- Send password information only via encrypted channels
- Encourage requestors to change their password upon first login

Next Steps

- Monitor for email confirmation of successful access

7.3. GeoServer Administration

This guide provides information for **SDL** Administrators on managing GeoServer authentication, user roles, and system configuration.

User Administration

Making a User a GeoServer Administrator

This is only applicable if you already have an admin account in Keycloak. Otherwise ask an SDL admin to make you a GeoServer administrator.

1. Log in to Keycloak with existing admin credentials
2. Create a new user/update an existing user
3. Select the `Role Mappings` tab
4. Add the `GEOSERVER_ADMIN` role

Authentication Configuration

Verifying Authentication Filter Chain

1. Navigate to `Security > Authentication`

2. Current filter chain configuration is displayed under `Filter Chains`
3. For OIDC, the filter chain (in web and rest) should be:
    - OIDC
    - Anonymous

Modifying Filter Chain

1. Under `Filter Chains`, select the chain to modify
2. Use the arrows to change filter order or add/remove filters
3. Click `Close` then `Save` to apply changes

OIDC Configuration

UI Location

1. Navigate to `Security > Authentication`
2. Look for `oidc` authentication filter

For the OIDC filter to work, the following environment variable had to be set:

```
- name: DISABLE_SECURITY_FILTER
  value: "true"
```

Setting this variable will disable the `antiClickJackingOption` security filter from the Tomcat configuration.

Important configuration files are located at:

- Web config: `/build_data/web.xml`
- Auth filter config: `/opt/geoserver/data_dir/security/config.xml`
- OIDC config: `/opt/geoserver/data_dir/security/filter/oidc/config.xml`
- Keycloak config: `/opt/geoserver/data_dir/security/role/keycloak_service/config.xml`

Troubleshooting

Layer Classification Misconfiguration

Layer classification misconfigation occurs when a user saves an invalid classfication string as a layer's classication keyword, or when a user sets multiple classifications on a layer. If a user misconfgures a layer's classification, the **SDL** user will need to refer to a user's support ticket for the *intended* layer classification and correct their issue via the following steps:

1. In the `Layers` view, identify the misconfigured error — the misconfigured layer's name should be in the ticket

2. Enter the layer's configuration page and navigate to the `Keywords` section and delete all classification keywords present on the layer
3. Create the correct classification:
    - Type the layer classification into the `New Keyword` text box
    - In the dropdown, select "English"
    - Type "Classification" into the `Vocabulary` text box
4. Click `Add Keyword`
5. Click `Save` at the bottom of the page.

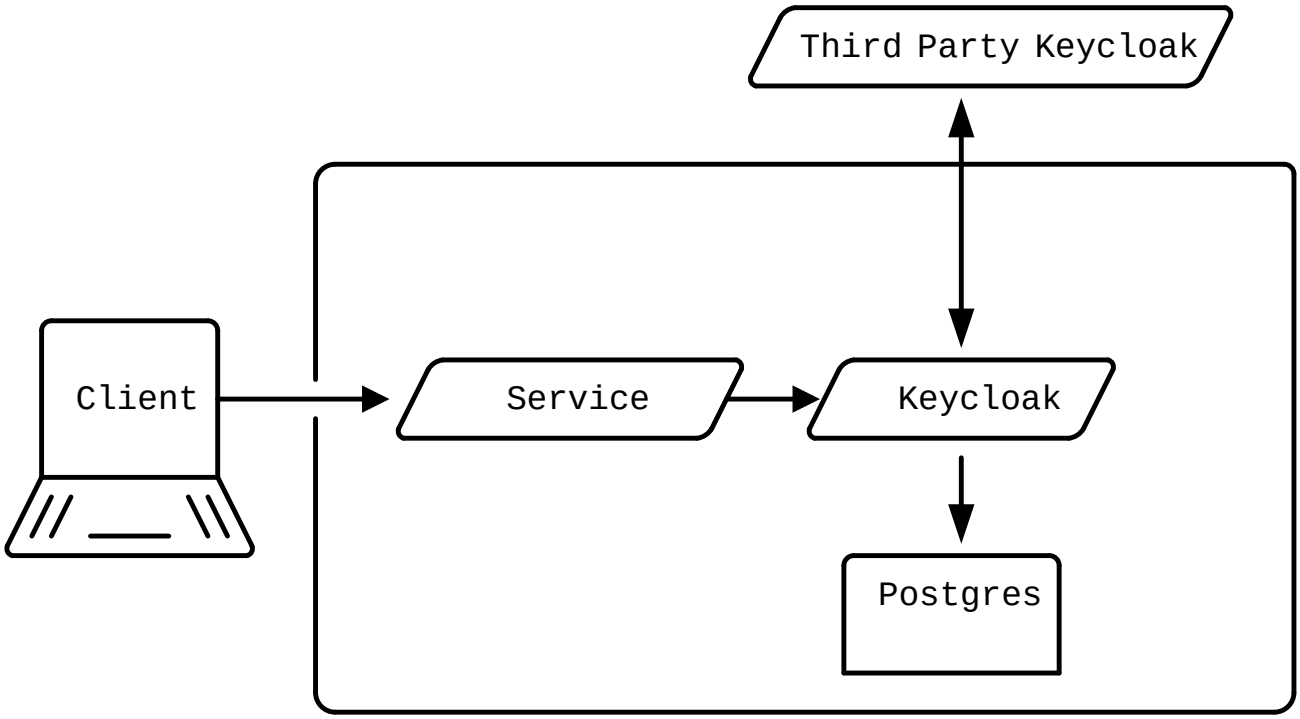
7.4. Keycloak

SDL leverages Keycloak for Identity and Access Management.

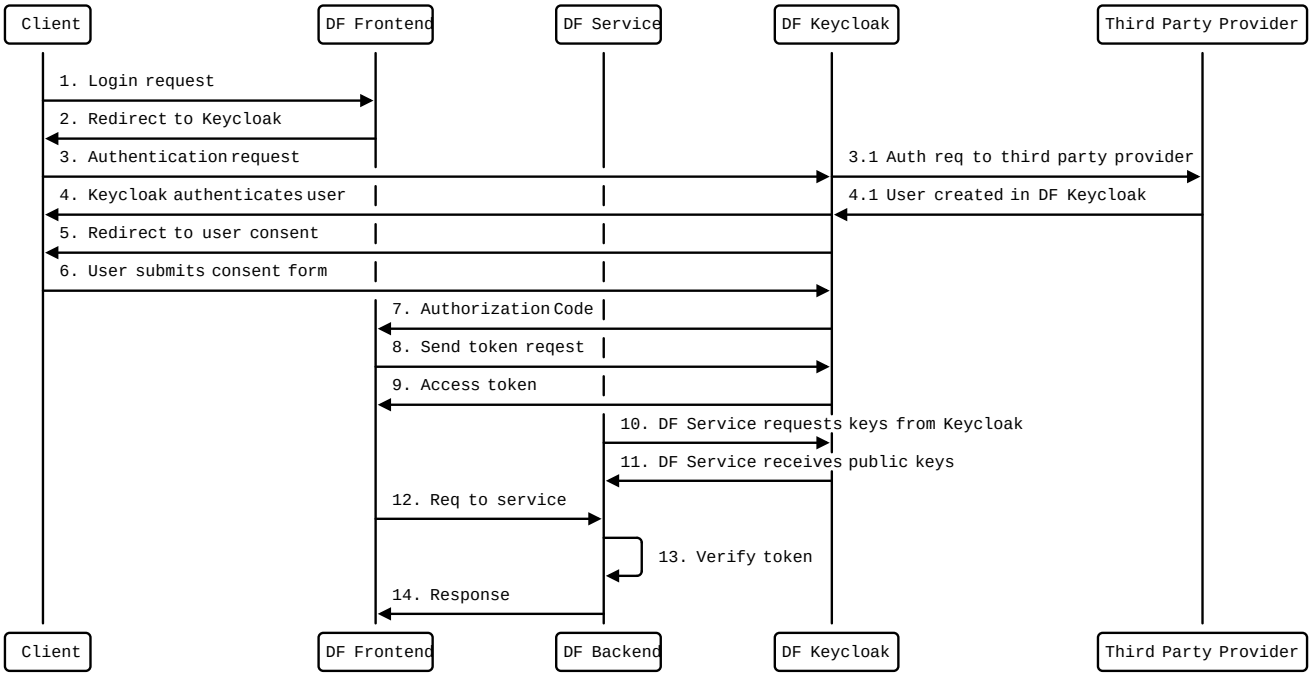
Keycloak is a lightweight feature rich Identity and Access Management tool that abstracts the need to implement any authentication mechanisms into our own applications. Additionally, applications will not have direct access to user credentials.

SDL uses OAuth 2.0 and OpenID Connect which Keycloak offers out of the box.

SDL Keycloak Diagram



SDL Keycloak Sequence Diagram



1. DF User attempts to login to SDL.
2. User is redirected to Keycloak login page.
3. User fills out login page username and password and submits the form. 3.1 If the user has an account from a third party, the user will fill out the login page from third party provider and submit the form.
4. Keycloak returns an authorization code to the user. 4.1 If the user did not exist in DF Keycloak, the user will be created from the third party provider.
5. Once the user is authenticated, they are redirected to the consent form.
6. User must consent to accepting access to application.
7. Keycloak issues an authorization token.
8. Application uses authorization token to fetch access token, refresh token, and id token from Keycloak.
9. Access token, ID Token, and refresh token response.
10. If the DF Service has not already requested the keys from Keycloak, the request will be made and keys will be cached.
11. Keycloak sends the Service the public keys.
12. A request is made to a DF Service.
13. The DF Service validates the access token with the Keycloak public keys. Verifies the user is able to access other resources given they have the appropriate roles.
14. DF Service returns the reponse.

7.4.1. Attribute Mappers

This page describes the required attribute mappers that need to be configured on identity providers in Keycloak. These mappers are responsible for mapping token claims from external

identity providers to user attributes in SDL.

Configuring Attribute Mappers

To configure these mappers for an identity provider:

1. Navigate to the identity provider's configuration in Keycloak
2. Select the "Mappers" tab
3. Click "Add mapper"
4. On the mapper configuration page:
    - Set the mapper type to "Attribute Importer"
    - Give the mapper a meaningful name
    - Specify the name of the claim in the token
    - Specify the name of the user attribute the claim should be mapped to

Example 1: Mapping a "classification" claim

Mapper name	`classification_attribute_mapper`
Mapper type	`Attribute Importer`
Claim	`classification`
User attribute name	`classification`

Example 2: Mapping a "fineAccessControls" claim

Mapper name	`fine_access_controls_attribute_mapper`
Mapper type	`Attribute Importer`
Claim	`fineAccessControls`
User attribute name	`fineAccessControls`

Example 3: Mapping a "country" claim

Mapper name	`country_attribute_mapper`
Mapper type	`Attribute Importer`
Claim	`country`
User attribute name	`country`

Example 4: Mapping a "citizenship" claim to "us_citizen"

Mapper name	`us_citizen_attribute_mapper`
Mapper type	`Attribute Importer`
Claim	`citizenship`
User attribute name	`us_citizen`

7.4.2. Users & Groups

SDL User and Group Management

SDL Authentication Utilizes Keycloak

Accessing the Keycloak instance for your cluster

1. Note the SDL/Data-Fabric URL used for accessing the WebGUI
2. Append "/auth" to default URL, for example: `https://{{ default_URL_for_accessing_SDL }}/auth`
3. Open a browser and navigate to the URL that includes the `/auth` path
4. Click on ***Administration Console***
5. Login with provided credentials

SDL Users and Groups

For all SDL configurations, be sure to verify that Keycloak realm ***data-fabric***` is selected from the upper-left drop-down. By default, the ***Master*** realm will be selected. > Note: Not completing this step of selecting the ***data-fabric*** realm will cause any subsequent configurations to be moot and invisible to SDL.

Creating Groups

Select ***Groups*** from the

1. In the *Groups* pane, click ***Create group***
2. Input the desired name for the new group
3. Click ***Create***

Creating Users

Select ***Users*** from the

1. In the *User list* pane, click ***Add user***
2. If the desire is to have the new user reset their password on their first login, then select ***Update Password*** from the field *Required user actions*
3. Complete *Username* field with desired username for the new user account
4. Complete any additional fields as necessary for new user account

5. Add new user account to respective user groups by clicking ***Join Groups***
6. Click ***Create***

Adding Existing Users to a Group

Select ***Users*** from the

1. In the *User list* pane, click on the ***{{ username }}*** of user who will be added to a group
2. In the *{{ username }}* pane, click on the ***Groups*** tab, near the top
3. Click ***Join Group***
4. In the *Join groups for user {{ username }}* pane, select the checkboxes for each group that user should be included in. > Note: Multiple groups can be selected in this view.
5. Click ***Join***

There are many additional functions and features that can be configured and managed from Keycloak, however, these procedures only focus on the prediscussed topics for the current effort. For more information, please contact RAFT or consult Keycloak documentation.

7.4.3. Token Exchange

SDL supports token exchange through the API gateway, allowing for secure token exchange with configured OpenID Connect (OIDC) identity providers.

Configuration

Token exchange is enabled when the API gateway is configured with the `OIDC_TOKEN_EXCHANGE_IDP_LIST` environment variable. This variable should contain the URLs of all configured OIDC identity providers in Keycloak.

If the value "default" is present in the `OIDC_TOKEN_EXCHANGE_IDP_LIST`, all identity providers will be ignored and no token exchange will be performed. This effectively disables the token exchange functionality.

How It Works

The API gateway acts as a token exchange broker, allowing clients to exchange tokens from trusted identity providers for SDL tokens. This enables:

- Single sign-on (SSO) across different identity domains
- Secure token exchange between trusted systems
- Integration with external identity providers

Token Caching

By default, the API gateway caches exchanged tokens in Redis to improve performance and reduce the load on Keycloak. The caching behavior works as follows:

- Tokens are cached using the external token's JWT ID (jti) as the lookup key in Redis
- The cache expiration time is set to match the external token's expiration time — this ensures that cached tokens are automatically invalidated when the original token expires

Token caching can be disabled by setting the `ENABLE_REDIS_CACHING` environment variable to `false` in the API gateway configuration.

Configuring Token Exchange for an Identity Provider

To enable token exchange for an existing identity provider, follow these steps:

1. Navigate to the identity provider's configuration in Keycloak
2. Select the "Permissions" tab
3. Enable permissions for the identity provider
4. From the permissions list, select the "token exchange" scope
5. On the token exchange configuration page, associate an appropriate policy with this permission

The policy you associate with the token exchange permission will determine which clients are allowed to exchange tokens with this identity provider. Choose a policy that aligns with your security requirements.

When configuring the token exchange permission, only modify the fields specified in these steps. Leave all other fields with their default values as populated by Keycloak.

Creating a Token Exchange Policy

If you need to create a new policy for token exchange:

1. When configuring the token exchange permission, click on the "policies" text box
2. Select the "create new policy" option
3. Choose "Client" as the policy type
4. Give the policy a meaningful name (e.g., "gateway-token-exchange")
5. In the clients list, select the client(s) that are configured for token exchange

When creating a new policy, only modify the fields specified in these steps. Leave all other fields with their default values as populated by Keycloak. For example, the "Logic" field should remain set to "positive".

Supported Identity Providers

The token exchange functionality supports any OIDC-compliant identity provider that is:

1. Configured in Keycloak

2. Listed in the `OIDC_TOKEN_EXCHANGE_IDP_LIST` environment variable
3. Properly configured for token exchange

Security Considerations

- Only trusted identity providers should be added to the exchange list
- Token exchange requests are validated for proper scope and audience
- All exchanged tokens must meet SDL's security requirements

Validating Token Exchange

You can validate that token exchange is working correctly by following these steps:

First, obtain a token from your trusted identity provider:

```
curl -k -X POST "$TRUSTED_PROVIDER_URL/auth/realms/$REALM/protocol/openid-
connect/token" \
 -H "Content-Type: application/x-www-form-urlencoded" \
     -d "grant_type=password" \
     -d "client_id=$IDP_CLIENT_ID" \
     -d "client_secret=$IDP_CLIENT_SECRET" \
     -d "username=$USERNAME_ON_IDP" \
     -d "password=$PASSWORD" \
     -d "scope=openid" | jq -r '.access_token'
```

Next, save the returned token to an environment variable:

```
export PROVIDER_TOKEN=<token_from_previous_step>
```

Finally, make a request to a protected endpoint that is *behind* the gateway:

```
curl -k -v -X GET https://map.$RDP_HOST/geoserver/rest/workspaces/nasa/coveragestores
\
    -H "Authorization: Bearer $PROVIDER_TOKEN"
```

If token exchange is working correctly, the API gateway will:

1. Receive the request with the provider's token
2. Exchange it for a SDL token
3. Forward the request to the backend service with the new token

The `-k` flag is used to skip SSL certificate validation. In production environments, you should use proper SSL certificates and remove this flag.

Testing Token Exchange Directly with Keycloak

To isolate and test the token exchange configuration in Keycloak (without involving the API gateway), you can use the following command:

```
curl -X POST \
        $RDP_HOST/auth/realms/data-fabric/protocol/openid-connect/token \
        -H "Content-Type: application/x-www-form-urlencoded" \
        -d "grant_type=urn:ietf:params:oauth:grant-type:token-exchange" \
        -d "subject_token=$IDP_TOKEN" \
        -d "subject_token_type=urn:ietf:params:oauth:token-type:jwt" \
        -d "client_id=$TOKEN_EXCHANGE_ENABLED_CLIENT" \
        -d "requested_token_type=urn:ietf:params:oauth:token-type:access_token" \
        -d "scope=openid"
```

This command tests the token exchange functionality directly at the Keycloak level, which can be useful for:

- Verifying that the identity provider is properly configured for token exchange
- Testing token exchange permissions and policies
- Troubleshooting token exchange issues without the API gateway

7.5. OPA

- Policy Management

7.5.1. Policy Management

A comprehensive Open Policy Agent (OPA) management interface for creating, editing, testing, and managing authorization policies written in the Rego language.

Overview

This policy management system provides a complete solution for managing OPA policies with a modern, user-friendly interface. It enables policy authors to create, edit, test, and visualize authorization policies with real-time feedback, making policy development faster and more accessible.

Features

Policy Management

- **Create New Policies**: Interactive policy creation with templates and guidance
- **Edit Existing Policies**: Full-featured Monaco editor with Rego syntax highlighting
- **Policy Versioning**: Track changes and maintain policy history
- **Real-time Sync**: Automatic synchronization with OPA decision engine

- **Import/Export**: Support for policy import and export functionality

Policy Testing

- **Interactive Testing**: Test policies with custom JSON input
- **Real-time Evaluation**: See policy results instantly
- **Input Generation**: Auto-generate test inputs from policy structure
- **Error Handling**: Comprehensive error reporting and validation
- **Performance Monitoring**: Track policy evaluation performance

Policy Visualization

- **Statement Cards**: Visual representation of policy rules and statements
- **Diff View**: Compare policy changes with side-by-side visualization
- **Syntax Highlighting**: Color-coded Rego syntax for better readability
- **Line Indicators**: Highlight specific lines when editing or debugging
- **Change Tracking**: Visual indicators for added, modified, and removed policies

Security & Access Control

- **Built-in Policies**: Read-only system policies for core functionality
- **User-defined Policies**: Full CRUD operations for custom policies
- **Access Control**: Role-based access to policy management features
- **Audit Trail**: Track all policy changes and user actions

Technical Architecture

Frontend Stack

- React 18 with TypeScript
- Next.js for server-side rendering and API routes
- Material-UI (MUI) for consistent UI components
- Monaco Editor for advanced code editing
- TanStack Query for efficient data fetching and caching
- Highlight.js for syntax highlighting

API Integration

- OPA REST API: Direct integration with Open Policy Agent
- Policy Storage: Persistent storage for policy definitions
- Real-time Updates: WebSocket connections for live policy status
- Batch Operations: Support for bulk policy operations

Language Support

- Rego Language: Full support for OPA's policy language
- JSON Input/Output: Standard format for policy testing
- Custom Parsers: Advanced Rego parsing for visualization

File Structure

```
pages/policies/
 ├──── index.tsx              # Main policy management page
 └──── README.md              # This documentation

components/Service/OPA/
 ├──── policies.d.ts          # TypeScript type definitions
 ├──── PolicyList.tsx         # Policy navigation and management
 ├──── PolicyToolbar.tsx      # Policy action buttons and controls
 ├──── EditorPanel.tsx        # Monaco editor integration
 ├──── TestingPanel.tsx       # Policy testing interface
 ├──── StatementCard.tsx      # Visual policy statement representation
 ├──── StatementCards.tsx     # Collection of statement cards
 ├──── StatementPanel.tsx     # Statement management panel
 ├──── Welcome.tsx            # Welcome screen and onboarding
 ├──── Utils.ts               # Policy parsing and utility functions
 ├──── CreateNewPolicyDialog.tsx # New policy creation dialog
 ├──── PolicyVersionControl.tsx  # Version control interface
 └──── Editor/                # Advanced editor components
     ├──── Utils.ts           # Editor utilities
     └──── [other editor files]
```

Usage Guide

Getting Started

1. **Navigate to Policies**: Access the policy management interface at `/policies`
2. **View Existing Policies**: Browse the policy list in the left sidebar
3. **Create New Policy**: Click the "Create New" button to start a new policy
4. **Edit Policy**: Select a policy from the list and click the edit icon

Creating a New Policy

1. Click **"Create New"** button in the policy list
2. Enter your policy name and description
3. Write your Rego policy in the editor
4. Use the **Testing Panel** to validate your policy
5. **Save** your policy to make it active

Testing Policies

1. Select a policy from the list
2. Open the **Testing Panel**
3. Enter JSON input data or click **"Generate Input from Policy"**
4. Click **"Evaluate Policy for Input"** to see results
5. Review the output and any warnings or errors

Understanding Policy Statements

- **Statement Cards**: Each policy is broken down into visual statement cards
- **Rule Types**: Different colors indicate rules, variables, and functions
- **Change Indicators**: Modified policies show visual change indicators
- **Line Highlighting**: Click on statement cards to highlight corresponding editor lines

Policy Development Best Practices

Writing Effective Policies

1. **Keep it Simple**: Write clear, readable policies
2. **Use Comments**: Document complex logic with comments
3. **Test Thoroughly**: Always test policies with various inputs
4. **Version Control**: Save policies frequently and track changes
5. **Follow Naming Conventions**: Use descriptive policy and rule names

Common Patterns

```
# Allow access based on user role
allow {
    input.user.role == "admin"
}

# Conditional access with multiple criteria
allow {
    input.user.role == "user"
    input.resource.type == "document"
    input.action == "read"
}

# Time-based access control
allow {
    input.user.role == "employee"
    time.now_ns() < input.resource.expires_at
}
```

Error Handling

- **Syntax Errors**: The editor highlights syntax issues in real-time
- **Policy Conflicts**: Visual indicators show conflicting rules
- **Performance Issues**: Monitor evaluation times for complex policies
- **Load Failures**: Red indicators show policies that failed to load into OPA

API Integration

Policy CRUD Operations

```
// Get all policies
GET /api/frontend/df-services/opa/get_policies

// Get active policies from OPA
GET /api/frontend/df-services/opa/get_policies_active

// Evaluate policy
POST /api/frontend/df-services/opa/evaluate/{policyPath}

// Save policy
PUT /api/frontend/df-services/opa/policies/{policyId}

// Delete policy
DELETE /api/frontend/df-services/opa/policies/{policyId}

// Chatbot integration
POST /api/v1/prediction/{chatflowId}
```

Policy Types

- **USER_DEFINED**: Custom policies created by users
- **BUILTIN**: System policies (read-only)

Configuration

Theme Configuration

The system supports both light and dark themes with automatic switching based on system preferences.

Troubleshooting

Common Issues

1. **Policy Not Loading**
    a. Check for syntax errors in the Rego code
    b. Verify policy format matches OPA requirements

    c. Review server logs for loading errors
2. **Testing Failures**
    a. Ensure input JSON is valid
    b. Check policy rules for logic errors
    c. Verify policy is loaded in OPA
3. **Performance Issues**
    a. Review complex policies for optimization opportunities
    b. Monitor evaluation times in testing panel
    c. Consider breaking down complex policies

Support

For additional help:

- Check the OPA documentation for Rego language reference
- Review policy examples in the built-in policies

Contributing

When contributing to the policy management system:

1. Follow TypeScript best practices
2. Add comprehensive tests for new features
3. Update documentation for any API changes
4. Ensure backward compatibility with existing policies
5. Test with both built-in and user-defined policies

Security Considerations

- All policies are validated before activation
- Built-in policies are protected from modification
- User actions are logged for audit purposes
- Policy evaluation is sandboxed within OPA
- Regular security updates are applied to dependencies

Chapter 8. SDL Onboarding

8.1. Tools and Environment

1. Environment Setup

Due to the architecture of **SDL**, it is typically possible to run **SDL** on a developer-issued laptop (e.g., Apple Silicon MacBook Pro). By utilizing Apple Rosetta, you can run amd64 processes/containers with near-native performance.

If you prefer to work on an amd64 Linux environment, on-prem VMs are available for this purpose. The steps below are for development on a Mac machine, but the command-line tools should be installed wherever you plan to deploy **SDL** or build code. Editors like VSCode allow for development through SSH, which can be useful when working on a remote VM.

The **SDL** team doesn't enforce any specific environment setup. However, using the same set of tools as the team can make debugging easier, especially during the first few weeks. Below is the typical setup used by the **SDL** development team:

All Developers

Optional Tools

- Terminal app:
  - Built-in Terminal.app (basic but functional)
  - iterm2
  - alacritty

Required Tools for macOS

- **homebrew** (an unofficial package manager for macOS)
- **A container management tool**, depending on your OS:
  - Raft is moving to **Podman** over Docker Desktop for macOS. There may be some initial hiccups as **SDL** development moves onto it.
  - For VM development, using Docker is fine as it doesn't require a team license.
  - Podman and Podman Desktop are the recommended tools.
  - Other options like **colima** may work, but YMMV.
  - **Rancher Desktop** is not recommended due to sensitivities around using Rancher for DoD applications.

Required Tools (All Platforms)

- **kubectl** - Kubernetes command-line tool
- **helm** - Kubernetes package manager

- **kind** - Kubernetes in Docker (for local development)
- **k9s** - Optional tool for Kubernetes cluster management
- **jq** - JSON processor
- **yq** - YAML processor
- **make** - Build automation tool
- **aws-cli** - AWS command-line tool (optional)
- **dfdev** - Simplified **SDL** deployment/management (optional)

Backend Development

- Common programming languages used include:
  - Go, Java, Python, and Rust (roughly in that order of precedence).
- For JVM languages:
  - **IntelliJ IDEA** (community edition is typically fine).
- Other editors that are used:
  - **VSCode**
  - **Zed**
  - **Vim/NeoVim**
  - **IntelliJ Ultimate** (paid version)

Frontend Development

- A mix of JavaScript and TypeScript is used.
- **VSCode** is primarily used for frontend development.

2. Deploy a Minimal SDL

Create a GitHub PAT and clone repo

If you haven't already, provide a GitHub username to be added to the `raft-tech` organization.

See the GitHub docs for more information on how to create a GitHub PAT.

Although fine-grained tokens are recommended, given that we use PATs to auth with ghcr.io, we need to create classic tokens instead.

The primary repository to start with is **rdp**, which contains deployment files and helper scripts to get everything set up.

Environment and Deployment

Set up the following environment variables:

```
export RDP_HOME=/path/to/your/rdp
export GHP_USERNAME=your_github_username
export GHP_SECRET=your_github_personal_access_token
```

The scripts provided in `/scripts` can be used to simplify local development.

To create a Kind cluster to work with:

```
cd $RDP_HOME
./scripts/rdp_create_kind.sh
```

Once the cluster is created, deploy SDL using:

```
./scripts/rdp_deploy.sh -e dev-minimal
```

Development Cluster Routing

Typically, production SDL instances have a public DNS record associated with them. For development, we typically don't set this up. However, there are two ways to handle this:

1. If using the instructions above, your cluster will be available on `localhost`.
2. If you prefer a non-public DNS record, deploy a cluster with a non-public DNS record and update your `/etc/hosts` file to point to it:

```
./scripts/rdp_deploy.sh -e dev-minimal -u sdl.local
```

Then, in your `/etc/hosts` file, add the following entry:

```
127.0.0.1 sdl.local
```

Note: You'll need to create identical records in your `/etc/hosts` file for every subdomain **SDL** routes on. You can see these subdomains by examining the Ingress Kubernetes resources:

```
kubectl get ingresses -n data-fabric
```

Chapter 9. Usage

9.1. S3 Events

How to consume a message from a Kafka Topic and be notified when a new file is placed into a MinIO bucket?

- All buckets in SDL have event notifications attached to them and will publish any new events to the Kafka topic: “minio-events”

Internal Consume

Set Up

1. Log into SDL JupyterHub
    - Navigate to JupyterHub
    - Home/Analyze/Jupyterhub
    - Click on examples folder
    - Double click `minio-events-consumer.ipynb`
    - This is a template with the topic name of “minio-events”
        1. Open SDL Kafka in a separate tab
            1. Click on Topics
            2. Search for “minio-events” Topic and select
            3. Click Messages
            4. Sort by Newest First (dropdown on right side of window)
        2. Open SDL Object Store in a separate tab
        3. Optional, you can open the SDL Topics page to verify messages were published from the minio bucket to the Kafka producer topic “minio-events”.

Test

1. In SDL JupyterHub
    1. `minio-events-consumer.ipynb` is designed to consume the “minio-events” topic
    2. Run all 3 code blocks and start the consumer
2. Go to SDL Object Store
    1. Upload a file into the bucket of choice
3. Go to SDL Kafka
    1. Refresh message page of the “minio-events” topic and you will see the event has been published to the “minio-events” topic
4. Go to SDL JupyterHub

1. You will see the message output of the consumer for the “minio-events” topic

External Consume

1. Navigate to SDL Topics
    - Home/Explore/Topics
    - Click KAFKA CA EXPORT
    - This will download a zip file with configurations and sample code for running an external Kafka Consumer
    - Use the example template `minio-events-consumer.ipynb` as a starting point

Options for filtering messages on the topic you are consuming

Offset

Offset

This is a unique identifier assigned to each message within a partition in Kafka. It is essentially a sequence number that allows consumers to keep track of their position within a partition.

Filtering on Offset

This means consuming messages starting from a specific offset or within a range of offsets. For example, you might consume messages starting from offset 10 up to offset 20 in a partition. This allows you to skip earlier messages and start consuming from a particular point of interest.

Partitions

Partition

Kafka topics are divided into partitions for parallel processing and scalability. Each partition is an ordered, immutable sequence of messages.

Filtering on Partitions

This involves specifying which partitions you want to consume messages from. For example, if a topic has 10 partitions, you might decide to consume messages only from partitions 2, 4, and 6.

Who Decides Partitions

Partitions are typically defined by the producer when the topic is created or configured. The producer decides how to distribute messages across partitions, which can be based on the message key (using a hash function) or a round-robin mechanism if no key is provided.

Key Serde and Value Serde

Serde

Stands for Serializer/Deserializer. It defines how keys and values are serialized (converted to bytes) when producing messages and deserialized (converted back to their original form) when consuming messages.

Filtering on Key Serde and Value Serde

This means filtering messages based on their serialized forms. For example, if you are only interested in messages where the key or value matches a certain pattern or format, you can apply filtering logic during deserialization to select only those messages.

Key Serde

Handles the serialization and deserialization of the message key.

Value Serde

Handles the serialization and deserialization of the message value.

MinIO Event Notification Types

The following event notification types are available on each bucket:

PUT

This event is triggered when an object is created or updated in the bucket. In the context of MinIO, this corresponds to the `putObject` operation, where new objects are uploaded or existing objects are overwritten.

GET

This event is triggered when an object is read from the bucket. In MinIO, this corresponds to the `getObject` operation, where objects are retrieved or accessed from the storage.

DELETE

This event is triggered when an object is removed from the bucket. This corresponds to the `removeObject` operation, where objects are deleted from the storage.

REPLICA

This event is related to the replication process. It is triggered when an object is replicated from one bucket to another, typically in a multi-region or disaster recovery setup. This ensures data redundancy and availability across different geographical locations.

ILM (Intelligent Lifecycle Management)

This event is triggered by actions related to lifecycle management policies. ILM policies automate the transition of objects between different storage classes (e.g., from standard to archival storage) or the expiration of objects after a certain period.

SCANNER

This event is triggered by background scanner processes. In MinIO, scanners typically run to verify the integrity of the stored objects, check for data consistency, or perform data healing in erasure-coded setups.

9.2. Video Streams

Videos in SDL

Working with videos

SDL currently ships with 2 video handling services, Restreamer and Oryx.

Restreamer is the primary means to consume an external Stream. For RTSP streams specifically, follow the steps below.

Fresh stream setup:

1. Navigate to the Restreamer UI
2. If this is the first time logging in, the credentials are already in the form. Otherwise use keycloak admin creds
3. If this is the first time using restreamer in a deployment, then the new stream wizard is already open and guiding you through.
4. Click `Network Source` option if not already selected, and continue.
5. Video setup- enter the rtsp url, and continue. Basic auth credentials are unlikely
6. Video setup profile- default selection is probably adequate, if there are even multiple options
7. Audio setup- use default
8. Metadata, leave blank for now. Can eventually add SDL stuff if we stick with restreamer long term.
9. License- select none
10. You may get an error that "the video could not be loaded". Half the time a simple refresh fixes it.
11. If refresh doesn't fix:
    a. go into the stream settings (click the pencil icon towards the upper right of the video player)
    b. edit video settings (pencil icon). Select `Disconnect and continue` if prompted.
    c. Click the big probe button towards the bottom of the form
    d. Change the codec from `Passthrough` to `h.264 (libx264)`
    e. select the `Audio` button at the bottom of the form to advance
    f. click `Finish` to advance
    g. click `Save`
    h. click `Connect` to restart the stream

If you already have one or more streams created, and you wish to add 1 more,

1. click the camera icon in the upper right of the webpage
2. click the plus button to the upper right of the channels modal that pops up
3. Add a name for the new channel
4. Follow the steps above starting at #4

Grabbing the stream from an external application

For this initial setup, we are using HLS (http live streaming)

For a stream you wish to get a url for: . select the stream in restreamer (such that you are seeing the video player) . click the hls button to the bot right of the video player to copy the url. . Adjust the hostname as needed to make it externally accessible

Displaying the stream in df-frontend

1. Navigate to the media store (no menu item exists, manually update the url)
    1. Click the Livestreams tab
    2. In the middle of the empty video player, click on the `Select a stream to add` dropdown
    3. click add UUID
    4. navigate to restreamer. Get the UUID from the url
    5. Enter the UUID and stream name in the df-frontend modal
    6. Wait the better part of a min (known issue)

Setup stream recording

This currently uses df-oryx. Soon to be refactored out.

1. Navigate to the video service
2. Log in with keycloak admin creds.

Oryx has issues clearing expired auth data from the browser. You may need to manually clear tokens/cookies if trying to log back in after an expired session.

3. Click the `Record` tab
4. Click the `Enable Record` expansion item if not already expanded
5. Click the `Start Record` button
6. Click the `Streaming` tab
7. Expand the `RTMP: OBS or VMix` if not already selected
8. Copy the `Streamkey` under step 3
9. Switch back to restreamer
10. Navigate to the stream you want to record
11. On the `Publications` widget, click the plus sign
12. Select `RTMP`
13. For address, enter `df-video:1935/live/`
14. Paste in the stream key you copied earlier
15. Click save
16. Click the enable toggle next to the stream name you just created in the publication widget

17. Navigate back to oryx
18. To confirm the stream is coming through, click the `Simple Player` hyperlink on the same line as step 4.1 on the "RTMP: OBS or vMix" section
19. Back in oryx, click the Record tab
20. Expand the `Record Tasks` expansion item
21. Wait a minute. You should see a new task appear
22. Navigate to MinIO, `streams` bucket
23. You should see fragmented video files appear under a UUID folder. A single large mp4 will be created from all those after the stream ends

9.3. Query Parquet File

How To: Upload a parquet file to Minio and query from Trino

To accomplish this following these instructions, you will be required to have Trino CLI app installed.

- Upload the target parquet (`.parquet`) file to the corresponding s3 bucket via UI, CLI or Inbox.
- Login to via trino cli:
  - An example command for logging in using Trino CLI:

```
https://trino.{ base_url_of_data-fabric } --user=admin --access-token={ access_token }
--catalog df-hive
```

If your local instance of SDL does not have proper certs issued, you can still successfully accomplish this by adding the `--insecure` flag to the previous command

To find the correct token, you can use the bash script packaged with SDL repo, located here: `data-fabric/hacks/df_token.sh`

- Create a table via Trino CLI:
  - Here is an example table one might use:

```
CREATE TABLE orders
(
    o_orderkey       BIGINT ,
    o_custkey        BIGINT ,
    o_orderstatus    CHAR(1) ,
    o_totalprice     DOUBLE PRECISION ,
    o_orderdate      DATE ,
    o_orderpriority  CHAR(15) ,
```

```
    o_clerk          CHAR(15) ,
    o_shippriority   INTEGER ,
    o_comment        VARCHAR(79)
)
WITH ( format = 'PARQUET', external_location = 's3a://inbox-public/' )
;
```

- Verify the tables were properly rendered:

```
describe default.orders;
```

- View the data with a query:
    - Here is an example query one might use:

```
select * from default.orders limit 5;
```

9.4. Data Examples

9.4.1. OpenSky

Endpoints

The OpenSky ingester pulls data from four endpoints.

`/states/all`

- Two bounding boxes are configured to extract data from this endpoint.
- Data extraction begins 48 hours ago.
- The endpoint is accessed every 20 minutes to comply with the limit of 400 credits per day.
- Each access retrieves data for the previous 10 seconds.

`/flights/all`

- Data extraction begins 48 hours ago.
- The endpoint is accessed every 60 minutes, retrieving 60 minutes of data.
- A distinct list of airports is extracted from each record using the estDepartureAirport and estArrivalAirport fields.

`/flights/arrivals`

- Data extraction begins 48 hours ago.
- The endpoint is accessed every 60 minutes, retrieving 60 minutes of data.
- Uses the distinct list of airports retrieved from the /flights/all endpoint.

`/flights/departures`

- Data extraction begins 48 hours ago.
- The endpoint is accessed every 60 minutes, retrieving 60 minutes of data.
- Uses the distinct list of airports retrieved from the /flights/all endpoint.

Examples

Request

```
GET /states/all with a bounding box for China
```

The states property is a two-dimensional array. Each row represents a state vector and contains the following fields:

Field Name	Data Type	Description
`icao24`	`string`	Unique ICAO 24-bit address of the transponder in hex string representation.
`callsign`	`string`	Callsign of the vehicle (8 chars). Can be null if no callsign has been received.
`origin_country`	`string`	Country name inferred from the ICAO 24-bit address.
`time_position`	`int`	Unix timestamp (seconds) for the last position update. Can be null if no position report was received by OpenSky within the past 15s.
`last_contact`	`int`	Unix timestamp (seconds) for the last update in general. This field is updated for any new, valid message received from the transponder.
`longitude `	`float`	WGS-84 longitude in decimal degrees. Can be null.
`latitude`	`float`	WGS-84 latitude in decimal degrees. Can be null.
`baro_altitude`	`float`	Barometric altitude in meters. Can be null.
`on_ground`	`boolean`	Boolean value which indicates if the position was retrieved from a surface position report.

Field Name	Data Type	Description
velocity	float	Velocity over ground in m/s. Can be null.
true_track	float	True track in decimal degrees clockwise from north (north=0°). Can be null.
vertical_rate	float	Vertical rate in m/s. A positive value indicates that the airplane is climbing, a negative value indicates that it descends. Can be null.
sensors	int[]	IDs of the receivers which contributed to this state vector. Is null if no filtering for sensor was used in the request.
geo_altitude	float	Geometric altitude in meters. Can be null.
squawk	string	The transponder code aka Squawk. Can be null.
spi	boolean	Whether flight status indicates special purpose indicator.
position_source	int	Origin of this state's position. • 0 = ADS-B • 1 = ASTERIX • 2 = MLAT • 3 = FLARM

Request

```
GET /flights/all
GET /flights/arrival
GET /flights/departure
```

The response is a JSON array of flights where each flight is an object with the following properties:

Field Name	Data Type	Description
Icao2	string	Unique ICAO 24-bit address of the transponder in hex string representation

Field Name	Data Type	Description
`firstSee`	`integer`	Estimated time of departure for the flight as Unix time (seconds since epoch)
`estDepartureAirport`	`string` or `null`	ICAO code of the estimared departure airport. Can be null if the airport could not be identified<
`lastSeen`	`string` or `null`	Estimated time of arrival for the flight as Unix time (seconds since epoch)
`estArrivalAirport`	`string` or `null`	ICAO code of the estimated arrival airport. Can be null if the airport could not be identified.
`callsign`	`string` or `null`	Callsign of the vehicle (up to 8 characters). Can be null if no callsign has been received.
`estDepartureAirportHorizDistance`	`integer` or `null`	Horizontal distance of the last received airborne position to the estimated departure airport in meters.
`estDepartureAirportVertDistance`	`integer` or `null`	Vertical distance of the last received airborne position to the estimated departure airport in meters.
`estArrivalAirportHorizDistance`	`integer` or `null`	Horizontal distance of the last received airborne position to the estimated arrival airport in meters.
`estArrivalAirportVertDistance`	`integer` or `null`	Vertical distance of the last received airborne position to the estimated arrival airport in meters.
`departureAirportCandidatesCount`	`integer` or `null`	Number of other possible departure airports. These are airports in short distance to the estimated departure airport.
`arrivalAirportCandidates`	`integer` or `null`	Number of other possible arrival airports. These are airports in short distance to the estimated arrival airport.