Ops Catalog

None

None

None

Table of contents

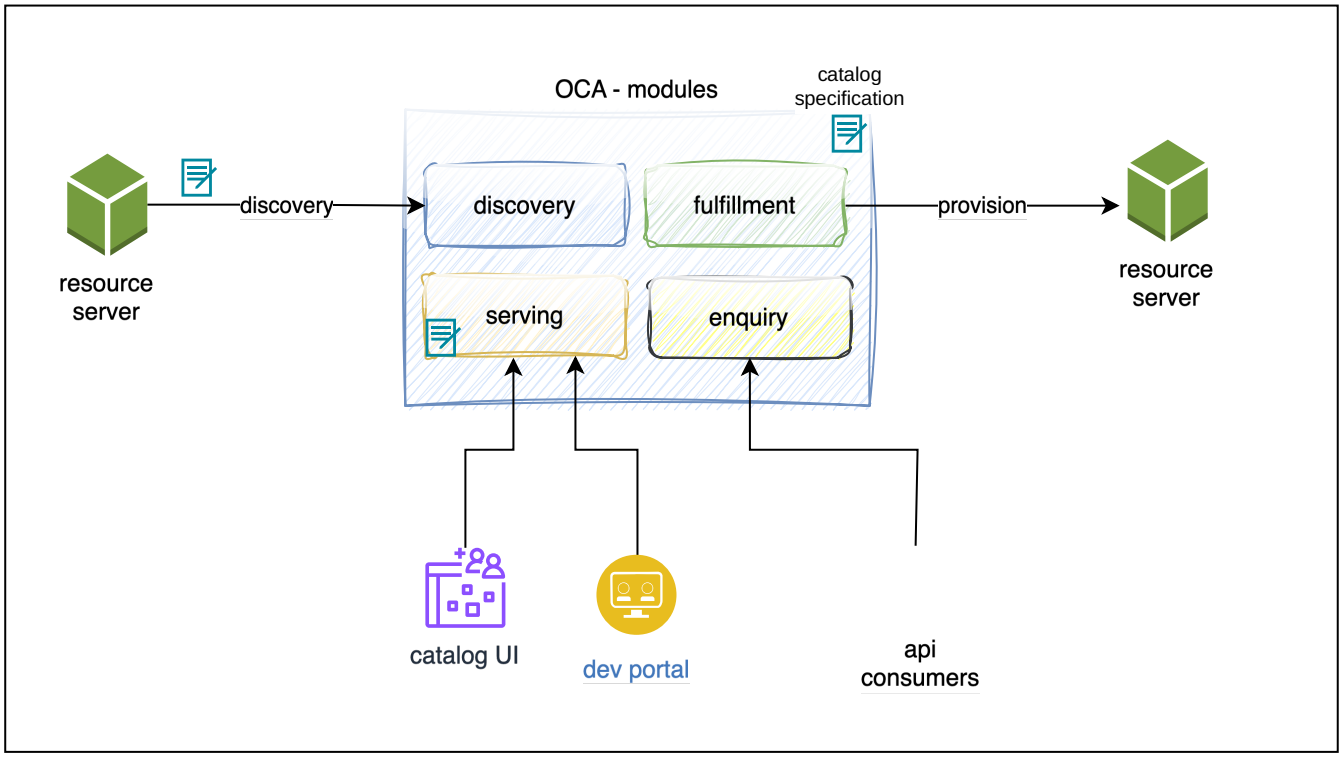
1. What is Ops Catalog 3
1.1 Key Principles 3
1.2 Other Benefits 4
1.3 Dive Deeper 4
2. Problem Statement 5
3. Why Ops Catalog 6
4. Use Cases 9
5. What is it? 12
6. Ops Catalog Examples 15
6.1 Running Ops Catalog Api 15
6.2 Recipes 15
6.3 Cleanup Activity 17
7. Ecosystem 18
8. Specification 19
8.1 Structure 19
8.2 Types 28
8.3 Catalog 35
8.4 Discovery 41
8.5 Fulfillment 50
8.6 Scorecard 51
9. Examples 52
9.1 My Account 52
9.2 Deployer 55
10. Integrations 56
10.1 Backstage 56
11. FAQs 60

1. What is Ops Catalog

Ops Catalog is a modern, collaborative solution for managing and utilizing software engineering metadata. It empowers DevOps teams to build internal SaaS platforms by making system information readily available for automation and consumer applications.

An asset catalog that increases developer happiness by putting Developers and DevOps front and centre.

- Specification based collaborative catalog solution that empowers DevOps teams to automate workflows, enhance visibility, and drive efficiency.
- Build your modern internal SaaS platform with an API-first, open-standard catalog designed for maximum flexibility and automation.
- Discover, manage, and provision software assets with ease. Empower your DevOps teams with federated data management, powerful APIs, and built-in gamification.



1.1 Key Principles

Federated Data Mgmt

Everyone authors what they own. This distributed ownership model ensures data accuracy and accountability, as those closest to the systems are responsible for their information.

API First API

Access asset information via API or subscribe to events. An API-first approach enables seamless integration with existing tools and automated workflows, making the catalog a dynamic part of your infrastructure.

Open Standard

Utilizes open standards and attributes for modern workflows. This promotes interoperability and allows for a flexible, extensible system that can adapt to evolving needs.

1.2 Other Benefits

Quick Bootstrap

Discovery modules available to take stock of your assets. Quickly populate your catalog by automatically discovering existing resources across your environments.

Fulfillment

Fulfillment plugins help with provisioning assets. Integrate with automation tools to enable self-service provisioning directly from the catalog.

Friendly Competition & Gamification

Scorecards and gamification elements can show where teams or services are in terms of documentation completeness, adherence to standards, or other desirable metrics, encouraging improvement.

1.3 Dive Deeper

Ready to learn more?

Get Started with Ops Catalog

Or explore specific areas:

- Why Ops Catalog? - Understand the value proposition.
- Explore Use Cases - See how it can be applied.
- Read the Specification - Delve into the technical details.

2. Problem Statement

2.0.1 Problems

A handful of problems related to asset catalog are shared here to highlight how operations efficiency has been impacted in general.

Owner Identification is hard

In a modern api/data architecture, there is a proliferation of assets across teams. Identifying Asset Owners is a challenging task. Often times we have a problem with an application, a system or an appliance and it takes a long time to get to the owner and to discuss the issue and seek help.

Obscure and Inaccessible

Ease of use is a priority and complexity is not a necessity. Most CMDB were written before microservices and cloud native applications became popular. Not only are new attributes missing, existing ones are hard to get access, retrieve and integrate with.

No support for self-service and Innovation

Self-service platforms are being built by various platform and feature teams alike. Modern application workflows demand new attributes which are accessible to streamline tooling.

Lack of an open approach

No single consistent approach exists for system data collection for App Team, Infra Team and Ops Team. Operational configuration often resides as tribal knowledge among developers, making it invisible to operations. It is hard to give a consistently good experience to your internal and external customers when data is managed in silos.

Lack of context

Enterprise-wide initiatives are great ideas and are meant to solve specific problems. However, they do not integrate with team specific data and processes and the data available in them stays high level. Making decisions using the outdated data or context can be inefficient and this also hides the inherent risk to the business.

2.0.2 Self Assessment

You can try to answer the following questions and see where you sit on this.

How do you find all the database schemas, message queues and assets a team has?

Are resources like repository, schema, topics provisioned through ticketing by special admins?

Do you calculate impact prior to software release and exercise different levels of testing with various integration points?

Do you know what configuration a given microservice is running, who the owner is and how quickly the owner is contacted when problems occur?

If you did not confidently answer the above questions' answers to yourself, there may be a good chance Ops Catalog could help with your DevOps workflows.

3. Why Ops Catalog

Just as you wouldn't navigate a book without an index, you can't steer a sizable business effectively without an Ops Catalog.

The Challenge: Scattered Knowledge and Manual Drudgery

In many organizations, information about infrastructure, APIs, services, and ownership resides in disparate silos—spreadsheets, wikis, even tribal knowledge. Developers waste countless hours searching for what they need, contacting other teams for basic information, and manually provisioning resources. These friction points create frustration, delays, and potential misconfigurations.

Conceptual Illustration: The Old Way (Scattered Knowledge)

```
graph LR
    A[Developers] --> B{Looking for Info}
    B --> C[Spreadsheets]
    B --> D[Wikis]
    B --> E[Tribal Knowledge]
    B --> F[Other Teams]
    C --> G((Pain Point: Wasted Time & Errors))
    D --> G
    E --> G
    F --> G
```

The Ops Catalog Solution: A Central Source of Truth

An Ops Catalog system establishes a centralized, API-driven repository of information about all the components that make up an organization's technology stack. Key features drive value:

- Automated Discovery : Specialized modules continually discover and update information about infrastructure, APIs, microservices, databases—all the building blocks developers rely on.
- Standardized Specification : Data adheres to a clear format, making it easily understandable and usable, both by humans and machines.
- API-Driven API: The API unlocks automation, letting developers integrate Ops Catalog data and workflows into their existing tools and CI/CD pipelines.
- Automated Provisioning : Actions are taken when new resource requests are added to the catalog which promotes a scalable approach to providing self-service capability without manual toil.

Conceptual Illustration: The Ops Catalog Way (Centralized Hub)

```
graph LR
    subgraph Ops Catalog
        direction LR
        H[Central Repository & API]
    end
    A[Developers] --> H
    I[Automation Tools] --> H
    J[CI/CD Pipelines] --> H
    K[Consumer Apps] --> H
    H --> L((Benefit: Efficiency & Single Source of Truth))
    H --> M((Benefit: Consistency & Automation))
```

3.0.1 Process

From Searching to Finding

Developers no longer spend hours hunting for obscure details. The Ops Catalog delivers accurate data in seconds.

Self-Service Empowerment

With ready access to resource descriptions and potential automation for provisioning, developers can independently experiment, test, and deploy new solutions.

Knowledge Sharing and Collaboration

The Ops Catalog encourages teams to document their components and ownership. This fosters transparency and cross-team understanding.

From Reactive to Proactive

Armed with detailed data and insights, teams can proactively address potential issues, optimize configurations, and streamline development processes.

3.0.2 Value

The Ops Catalog ecosystem delivers both developer satisfaction and broader team benefits:

Accelerated Development

Less time wasted on hunting information and manual tasks means faster time to market.

Improved Quality

Standardized data and potential automation reduce errors and inconsistencies.

Heightened Collaboration

The shared resource breaks down silos and encourages knowledge sharing.

Boosted Morale

Empowered developers focused on building, not bureaucracy, are happier developers.

3.0.3 Takeaways

Specification-based Collaboration

Ops Catalog is not just a catalog; it's a dynamic collaborative solution! Empower your DevOps teams with the freedom to automate workflows, boost visibility, and supercharge efficiency.

API-First, Open-Standard Catalog

Building a modern internal SaaS platform? Ops Catalog is your go-to companion! With an API-first approach, it provides flexibility and automation.

Discover, Manage, and Provision with Ease

Ops Catalog simplifies the complex. Effortlessly discover, manage, and provision software assets, putting you in control. No more headaches – just streamlined operations.

Federated Data Management and Powerful APIs

We believe in the power of collaboration. Ops Catalog fosters federated data management and equips you with robust APIs, ensuring seamless connectivity and accessibility. Owners own and manage their data.

Tailored for Your Team

Ops Catalog seamlessly integrates into your organizational setup, offering unparalleled flexibility. Run multiple Ops Catalogs without sacrificing visibility – each team can have its own, and a single catalog can effortlessly aggregate data from others. It's the agility your operations crave and the simplicity your teams deserve. Elevate your organizational dynamics with Ops Catalog – where adaptability meets simplicity! 🚀💻✨

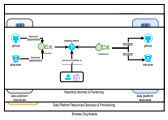
Built-in Gamification

We're making work fun! Ops Catalog introduces gamification to keep your teams engaged and motivated. Watch productivity soar as tasks become challenges and milestones turn into victories.

4. Use Cases

Few Possible Use Cases for DevOps and Self-Service Platforms:

4.0.1 Featured

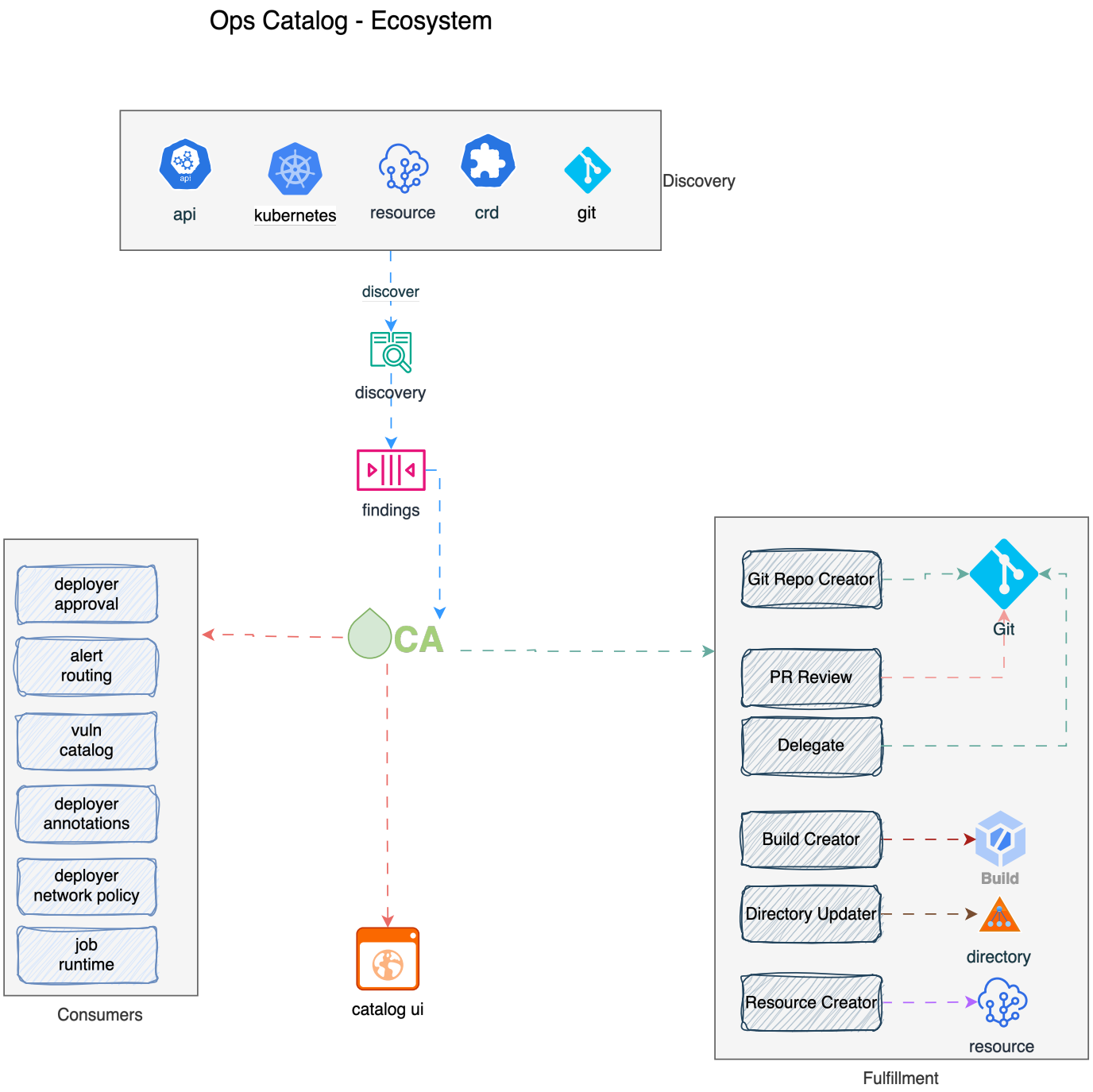


Resources

4.0.2 Other Use Cases

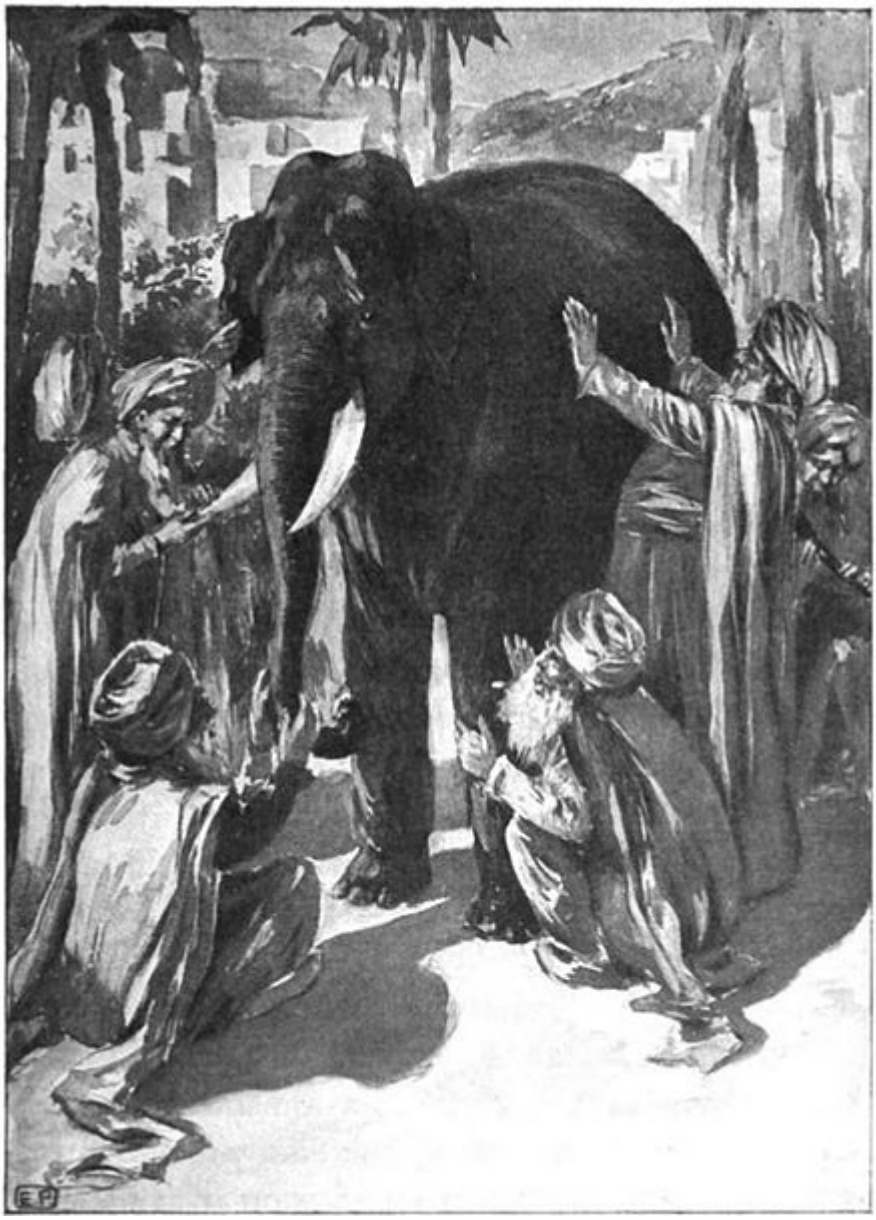
Area	Use Case	Purpose	Owner
Audit	Audit Report	Generate Inventory List and Licenses	CISO
Prod Support	Issue Assignment	Assign production issues to the correct owners	SRE
Operations	Manage Certificates	Manage Certificate lifecycle, provisioning and renewal alerts	Prod Support/ SRE
Deployment	Config Mgmt	Use Attributes shared by owners as config for deployment	DevOps
Release	Approval Checks	Seek approval from actual owners of the item being deployed	DevOps
Operations	Annotate Workloads	Annotate workloads to have rich information at runtime - log dashboards, deployment objects	DevOps
Operations	Create Data Resources	Create Resources like Database Schemas, Search Indices, Kafka Topics, Buckets	Data Platform Owners
Operations	Disover Platform Resources	Discover Container and Platform Resources like Deployments, Endpoints, IAM, Security Groups	Platform Owners
Operations	Browse Team Assets	Browse All types of Assets and Identify Owners, Discover Links including local runs and environments at team or division level	DevOps
Operations	Browse Org Assets	Browse All types of Assets and Identify Owners, Discover Links including local runs and environments at organisation level	Infra Teams
Code Factory	Repository Provisioning	Do not wait weeks for some admin to create repositories for you	Dev Experience
Build	Register New Builds*	Register New Builds when catalog has a new entry	Dev Experience
Data Engineering	Job DAGs	View info on Job Dags	Data Team
Release	Impact Assessment*	Calculate true impact and assign tasks to each team for PVT	Release Manager
Project Mgmt	Ticketing	Assign tickets to owners and teams for specific topics	Chief Scrum Master
IAM	User Mgmt*	Discover or Manager User, Groups, Roles	IAM
IAM	User Mgmt*	Discover or Manager User, Groups, Roles	IAM

* - in roadmap or being implemented. Reach out if interested to help with requirement and/or code.


Ops Catalog - Ecosystem
api
kubernetes
resource
crd
git
Discovery
discover
discovery
findings
CA
deployer approval
alert routing
vuln catalog
deployer annotations
deployer network policy
job runtime
Consumers
catalog ui
Git Repo Creator
Git
PR Review
Delegate
Build Creator
Build
Directory Updater
directory
Resource Creator
resource
Fulfillment

5. What is it?

Understanding a comprehensive system like Ops Catalog can be like the fable of the blind men and the elephant. Different people grasp different aspects, and while no single perspective is entirely wrong, it's the synthesis of these views that reveals the whole picture. This page explores common perceptions...



CMDB for Developers

It is a CMDB but for Developers. This highlights its role in providing a structured, queryable inventory of operational components, tailored to developer needs for understanding and interacting with the tech landscape.

Partial View: Registry for Microservices

While it can act as a registry for microservices by cataloging their metadata and endpoints, its scope is much broader, encompassing various types of resources, components, and their relationships.

Catalog API for Backstage

It serves as an excellent backend or catalog API for developer portals like Backstage, providing the structured data needed to populate such UIs.

Misconception: Place to Store Deployment Config

It's not primarily a system for storing raw deployment configurations (like K8s YAMLs or Helm charts). While it might link to such configs or catalog their existence, its focus is on metadata and relationships, not direct config management.

Partial View: Place to Lookup Ownership Data

Yes, it's a great place to look up ownership data for services, resources, and components. This is a key feature, but not its sole purpose.

Misconception: Creates Git Repositories

Ops Catalog itself does not typically create git repositories. It catalogs metadata about various entities, which might include information *about* git repositories, but it's not a Git repository management tool.

Misconception: Tool to Manage Database Schemas

While it might store metadata *about* databases (e.g., type, location, owner), it is not designed to manage or enforce database schemas directly.

Partial View: Helps Find Kubernetes Workloads

It can definitely help find and understand Kubernetes workloads by cataloging their metadata, relationships, and ownership. However, it's not limited to K8s and can catalog a wide variety of other components.

Partial View: Collects AWS Resources

It can collect and catalog AWS resources (or GCP, Azure, etc.), providing a unified view. But it's platform-agnostic and designed to handle resources from any environment.

Misconception: Can Provision Kafka Topics

While Ops Catalog can trigger automated workflows (e.g., via fulfillment engines) that *could* lead to provisioning Kafka topics, the catalog itself is not the provisioning engine. It holds the 'what' and 'why', and can initiate the 'how'.

Partial View: API Server for Listing Resources I Own

It does provide an API to list resources you own, which is a valuable feature for visibility and management. But it also serves broader organizational needs for understanding the entire tech ecosystem.

Aggregator to View All Resources of the Company

This is a strong understanding. Ops Catalog can aggregate information from various sources and discovery mechanisms to provide a comprehensive, company-wide view of technological assets and their relationships.

6. Ops Catalog Examples

There are a bunch of example configurations available in this project to run Ops Catalog with various extensions.

6.1 Running Ops Catalog Api

Ensure you have Docker/Nerdctl installed and execute the following:

```
git clone git@github.com:ops-catalog/examples.git
cd examples
```

You can follow this in examples repository as well.

6.2 Recipes

The recipes discussed here can be used for reference to try out a specific use case.

Recipe Name	Goal
simple	run a basic catalog API with static data
minimal	run ui,catalog and postgres
ssl	run ui, catalog and postgres. Postgres TLS is enabled.
edge	run ui and multiple catalog instances. One acts as aggregator and another as edge
discovery	finds resources in targets like databases, airflow, k8s, messaging servers
fulfillment	provisions resources against databases, message servers

A run.sh file is provided to make it easier to run various docker compose files. Usage documentation for run.sh is shared below.

```
$ ./run.sh                                                                                                    i
a recipe can be run like this:

  ./run.sh recipe <name>
  ./run.sh down

Recipe command runs one of the available recipes.

name:
-----
simple      - a catalog server with sample catalog items loaded from ./datasets/catalog/my-account
minimal     - a basic setup with catalog, ui, postgres and some seed data
ssl         - a basic setup with catalog, ui and postgres with TLS
edge        - aggregator catalog discovers data from edge catalog
discovery   - performs discovery against few targets like postgres, cassandra, airflow, k8s, kafka and presents in UI
fulfillment - provisions resources in catalog which are not yet in target servers

Down command shuts down the running containers.
```

6.2.1 Recipe: simple

This is probably the simplest use case. We are mounting catalog items into a folder and serving it via catalog instance. This minimilastic config can be queried and filtered through a HTTP API call.

To run this recipe, invoke

```
./run.sh recipe simple
```

Docker containers started with ./run.sh command can be stopped using `./run.sh down`

Since this step only requires a single docker container to run, it can also be invoked directly like so:

```
docker run \
-v $(pwd)/datasets:/opt/app/datasets \
-v $(pwd)/recipes/simple:/opt/app/config \
-e CONFIG_FILE=/opt/app/config/config.yaml -p 8080:8080 \
-it opscatalog/catalog:latest
```

Test Access

Load the following link to view catalog item:

Link	Description
http://localhost:8080/api/catalog	Catalog Listing
http://localhost:8080/api/catalog?kind=Resource	List Resources Only
http://localhost:8080/api/catalog?kind=Component	List Components Only

6.2.2 Recipe: minimal

If you have resource constraint, you can run selected profile as well and accordingly update engines list in setup/containers/ops-catalog/conf/discovery.yaml or fulfillment.yaml

We can run the command like below to bring up ops catalog API, a stand-in for objects in a Kubernetes cluster, a postgres instance with two databases and a basic UI.

```
./run.sh recipe minimal
```

To test discovery and fulfillment, create a new schema against the running postgres.

```
docker exec -it postgres psql -Upostgres -dservicing
servicing=# CREATE SCHEMA IF NOT EXISTS refdata;
servicing=# \q
```

Also drop a new file into the mix

```
cat > datasets/my-account/merchant-schema.yaml <<EOF
apiVersion: "v1"
kind: Resource
class: Schema
metadata:
  name: "merchants"
  description: "Merchant Tables are hosted in this schema"
  license: "private"
dependencies:
  upstream: []
  providedBy: postgres.pg-2
  triggers: []
classification:
    tag: ["transaction", "customer"]
    domain: "transaction"
    team: "transaction"
    capability: "onlinebanking"
    businessUnit: "retail"
EOF
```

Once the discovery and fulfillment loop is complete, there should be two new items in the catalog. The schema called merchant should be visible in postgres as well.

```
docker exec -it postgres psql -Upostgres -dpreferences
preferences=# select catalog_name, schema_name from information_schema.schemata;
preferences=# \q
```

Check the catalog entry via api calls. The default refresh frequency is served data is set at 5 minutes which you can change by updating ops-catalog/conf/*.yaml

```
http://localhost:8080/api/catalog?name=merchants
http://localhost:8080/api/catalog?name=servicing.refdata
```

If you list files under datasets/discovered-items you should also see few new folders depending on what discovery engines were enabled. In the case of minimal profile, you will see k8s and postgres folders populated with catalog items.

UI

Ops Catalog can provide data to your existing backstage instance. Navigate to http://localhost:7007/catalog to see Ops Catalog objects within backstage UI.

Tearing Down

Each Recipe run can be cleaned up by running `./run.sh down`

6.2.3 Recipe: ssl

To run with SSL enabled, set SSL_STATE to on in your .dockerenv or one of the env files. A .ssl file is provided for reference and this setup can be run with SSL by executing:

```
./run.sh recipe ssl
```

6.2.4 Recipe: edge

A number of catalog instances can be run in federated mode. This allows for local teams to run their own edge catalog for their unique workflows. A more global catalog instance can then act as aggregator to compose data from all catalogs.

You can simulate this by running the following:

```
./run.sh recipe edge
```

6.2.5 Recipe: discovery

This compose file also runs a standalone kafka, postgres and cassandra and seeds them with initial objects so they can be collected by the discovery module.

```
./run.sh recipe discovery
```

6.2.6 Recipe: fulfillment

The compose file is somewhat similar to discovery as it is now writing back to the targets.

```
./run.sh recipe fulfillment
```

6.3 Cleanup Activity

To cleanup what we did just then, run the following command to remove all running containers associated with this project.

```
./run.sh down
```

or

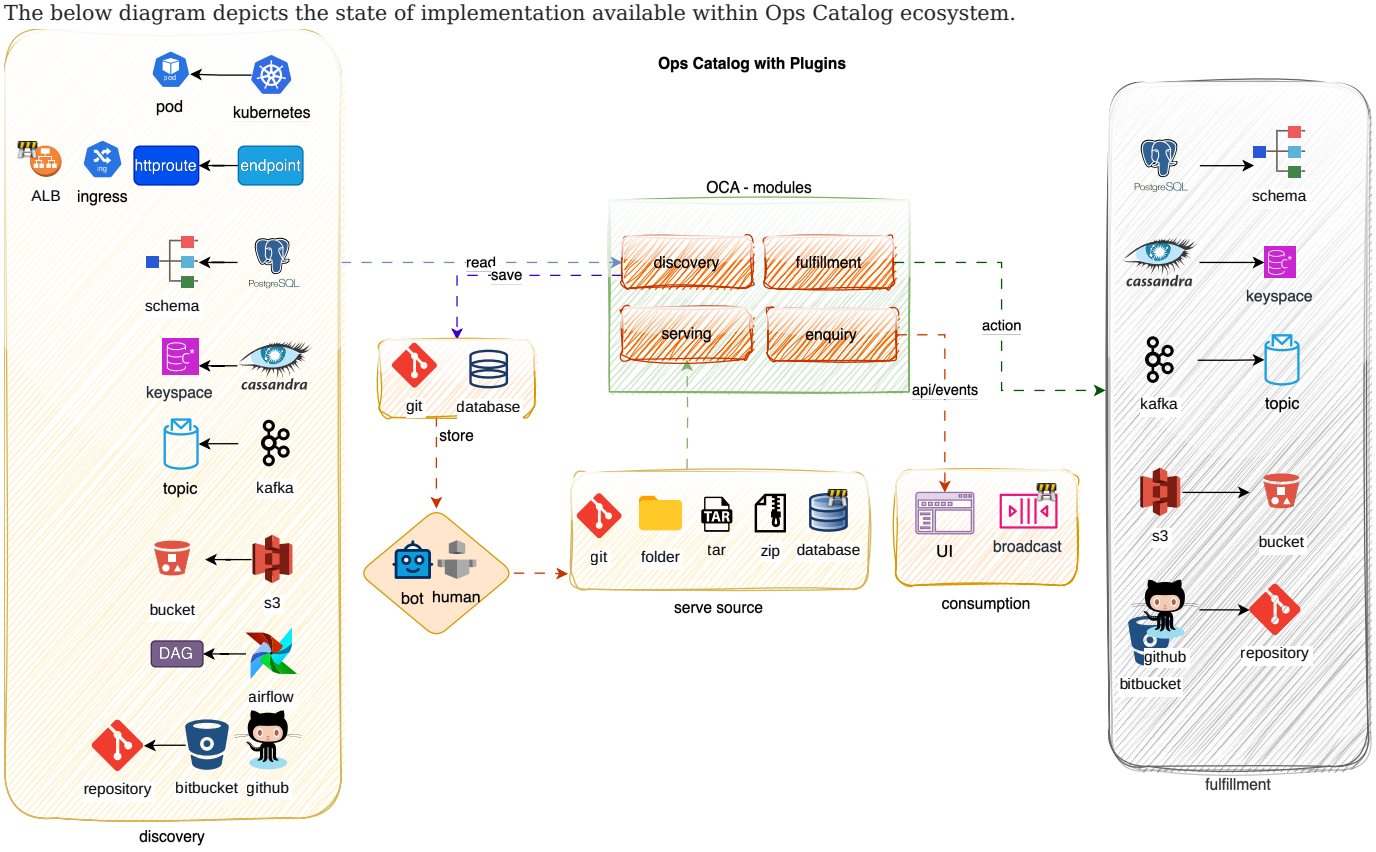
```
docker compose down
```

7. Ecosystem

There are four modules within Ops Catalog.

Module	Purpose
Discovery	gather data from various runtimes
Fulfillment	create resources or perform action on targets
Serving	read from various sources and provide catalog data
Enquiry	display in reference UI and broadcast events

The below diagram depicts the state of implementation available within Ops Catalog ecosystem.



8. Specification

8.1 Structure

The major components

There are eight major categories of attributes in a catalog item.

Category	Description
Metadata	Catalog Item name, a few key attributes and various options to add additional data in the form of labels and annotations.
Contact	Ownership and Collaborator details along with role they play for the catalog item.
Properties	Properties associated with catalog item
Links	Collection of external links by intent - eg source, quality, test, build
Dependencies	Information on upstream, downstream and dependency on resources
Classification	A collection of attributes to categorise the catalog item
Runtime	Information on entry points, key URIs and augmentation of discovered data
Audit	Audit Log about information captured in this catalog item

VERSIONING

Ops Catalog Spec uses the kind and version attribute made popular by Kubernetes to specify the object type and apiVersion.

```
apiVersion: "v1"
kind: Component
metadata: {}
```

METADATA

The metadata section of the document contains the following key attributes.

Attribute Name	Description
name	Name of the catalog item
description	Text describing the catalog item
labels	A map structure that can hold custom attributes to tag this artifact with
tier	This attribute records the priority of the catalog entry in terms of urgency values Tier 1-5
layer	This attribute denotes the catalog runtime tier
annotations	A map structure that can hold additional enrichment data
logo	Catalog item logo name or location
contact	Id that can be used to contact the owners of this catalog item for support
license	Applicable License for this catalog item e.g. Apache 2.0, Private, Commercial, GPL, BSD etc

`tier`, `contact` and `layer` are automatically copied by the implementation as attributes under labels. This helps pass this information to other DevOps processes (eg. Deployment) and the same information can be used for search.

The below yaml snippet shows an example metadata section of a catalog item.

```yaml
metadata:
  name: "user-service"
  description: "This microservice provides customer information"
  labels:
    internet-facing: true
  annotations: { }
  tier: 1
  layer: web
  language: go
  logo: "user-service.svg"
  contact: "user-service@example.io"
  license: "private"
```

CONTACT

There can be a number of contacts, a team or an individual can play with regards to the maintenance of a project. While there can be a single owner from accountability perspective, a catalog item can have many contributors playing different roles.

Few roles such as owner, contributors, support are explicitly laid out and additional roles can be configured as approvers by specifying the intent.

The `id` attribute holds the value of a contact ID of a team or individual using the following well known ID formats.

Id Type	Example	Fully Qualified Example
username	@user1	id://user1, ad://1600920
team	web-team	team://web-team
group	[approvergroup]	group://approvergroup
email	web@company.com	email://web@company.com
chat channel	#slackChannel	slack://slackChannel, mattermost://channel2
phone	+6189209999	tel://+6189209999, mob://+6189209999

As there can be a variety of ID providers and chat providers, an `IDProviderConfig` object can be specified to specify the defaults.

The below yaml snippet shows contact configuration example for a catalog item.

```yaml
contact:
  owner:
    id: "@user1"
  contributors:
    - id: "web-team"
    - id: "web@example.com"
  support:
    - id: "#slackChannel"
    - id: "mattermost://web-support"
    - id: "+6189209999"
  participants:
    - id: "[CodeApprovers]"
      intent: "approvers"
    - id: "some-team"
      intent: "maintenance"
    - id: "leadership"
      intent: "stakeholders"
```

PROPERTIES

While catalog metadata could be used to store associated properties, a dedicated section helps with clarity and access. It is because of this reason, the recommended way to store properties is by using the `properties` attribute.

To ensure compatibility with industry trend around specifying config under annotations, this version of the specification also supports the annotation mode.

The following seven namespaces are recognized and this means that the API and tooling around the specification should support operations for these namespaces:

Namespaces	Description
lifecycle	List of config properties describing the catalog item's lifecycle and actions
build	List of config properties used by build tools
dev	List of attributes for dev mode like local runs, tests and one-liner command
operations	List of attributes associated with runtime state of this catalog item
preferences	List of flags one might require for this catalog item
resources	A resource namespace can be used to highlight resource specific requirements
custom	Where basic tooling support is required for uncategorised properties

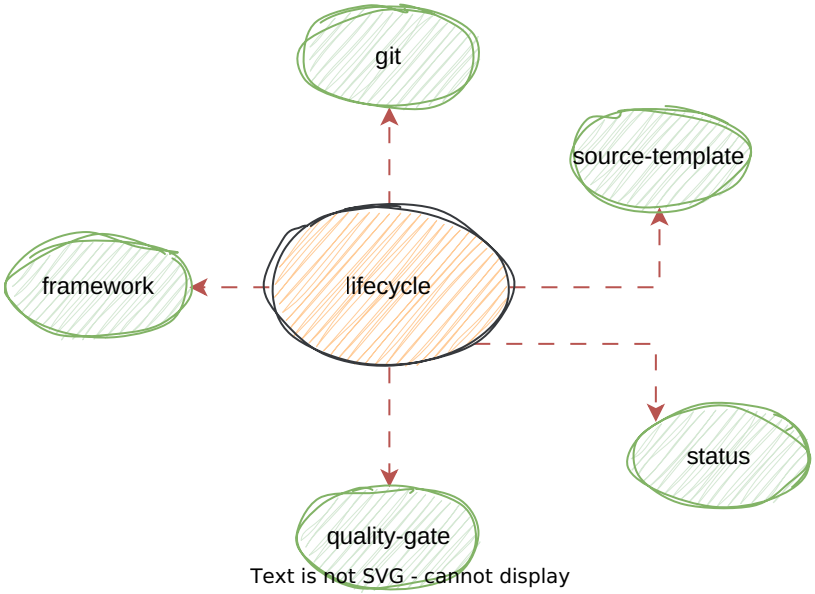
The below snippet shows properties provided as annotation under each recognised namespace. The annotation keys are prefixed with .ops.catalog/

```
metadata:
  annotations:
    lifecycle.ops.catalog/git: "managed"
    lifecycle.ops.catalog/status: "active"
    lifecycle.ops.catalog/source-template: "microservices-starter"
    lifecycle.ops.catalog/quality-gate: "true"
    lifecycle.ops.catalog/framework: "spring-boot"
    lifecycle.ops.catalog/language: "java"

    build.ops.catalog/docker: "true"
    build.ops.catalog/gradle-wrapper: "true"
    build.ops.catalog/command: "./gradlew clean build"

    dev.ops.catalog/quickstart: "./gradlew runApp"
    dev.ops.catalog/local-run: "docker-compose up -d"
    dev.ops.catalog/test: "./gradlew clean build"

    operations.ops.catalog/idempotent: yes
    operations.ops.catalog/cron: "0 * * * *"

    resources.ops.catalog/profile: "cpu-medium"
    resources.ops.catalog/deploy: { cpu: 0.5, memory: "512m", class: "medium" }

    preferences.ops.catalog/show-login: "false"
    custom.ops.catalog/http-requests: {"timeout": 10s, "verify-tls": "true"}
```

Specification using properties attribute.

```
properties:
  lifecycle:
    git: "managed"
    status: "active"
    source-template: "microservices-starter"
    quality-gate: "true"
  dev:
    quickstart: "./gradlew runApp"
    local-run: "docker-compose up -d"
  build:
    docker: "true"
    gradle-wrapper: "true"
    command: "./gradlew clean build"
  operations:
    idempotent: "yes"
    cron: "0 * * * *"
  preferences:
    show-login: false
  custom:
    http-requests:
      timeout: 10s
      verify-tls: true
```

In the diagram below, attributes such as source template and status are properties under the lifecycle scope.



If there is a conflict, configuration specified under properties attribute will override the ones in annotations. While the implementation can perform a merge operation, it is recommended to stick to one of the two approaches to avoid surprises.

LINKS

An item can have many different concerns housed in other systems. We can collect required information using the link object array.

To allow for extension and multiple links for similar intent, the type attribute can have classifier suffix as shown in the table below.

Link Type	Description
source	Specify location from which source can be retrieved
artifact	Location of the artifact. When a classifier is not provided, it will be estimated.
artifact/image	Location of the container Image
artifact/jar	Jar Artifact
contract	Location where contract is located. When a classifier is not provided, it will be estimated.
contract/api	API contract location
contract/data	Data contract location
build	Location of the build associated with this catalog item
docs	Various docs associated with the catalog item
dashboard	Dashboards associated with this catalog item
dashboard	Observability Dashboards associated with this catalog item
dashboard/logs	Dashboards associated with this catalog item
dashboard/metrics	Dashboards associated with this catalog item
alerts	Alert Spec associated with this catalog item
chat	Chat channel link if available
precondition	Precondition Contract or Spec for this catalog item to operate

A yaml snippet showing links configuration for a catalog item.

```yaml
links:
  - type: source
    url: git@github.com:owner/abc.git
  - type: artifact/image
    url: docker://owner/abc
  - type: artifact/jar
    url: https://artifactory/owner/abc
  - type: contract
    url: https://site/contract.json
  - type: build
    url: https://jenkins/owner/example/build/
  - type: chat
    url: https://slack/myteam
  - type: docs
    url: https://readthedocs.org/example
  - type: dashboard
    url: https://perf-dashboard/
```

DEPENDENCIES

Catalog Items often have dependencies on other items within a setup. An attribute `upstream` keeps track of what this application ends up consuming data from. `downstream` keeps track of what this item calls to share data. Dependencies attribute captures information about resources or technology this catalog item relies on usually using upstream and downstream data.

```yaml
dependencies:
  upstream: []
  downstream: []
  triggers: []
```

Similarly, triggers is an array of entries which are dependent on the execution of this. `triggers` may be useful in a pipeline or to configure watch notifications upon changes.

CLASSIFICATION

Classification attributes help us filter or group assets based on their type and category they belong to. Additionally, they can be tagged to extend further to capture cross-cutting concerns or a general topic.

Let's establish the definition of what key terms like domain, capability mean in the context of ops catalog.

Capability

A group of product features that collectively provide a business level function. Examples of capabilities can be things like Operations, Credit Risk Assessment, Onboarding, Card Payment, Data Analytics, Client Experience.

Domain

Domain usually covers a specific area or features that a software solution aims to provide. Catalog items can be loosely clustered by domain in which a set of experts usually perform operations to minimise context switching and maximise efficiency.

The classification node consists of the following attributes:

Name	Description
tag	An array of labels to apply to this catalog item for filtering, search, grouping purpose
domain	Domain of which this catalog item is a member of
capability	A specific product capability this catalog item is a part of
businessUnit	Business Division to which this catalog item belongs

The table below shows various kind, type and additional attributes associated with each type.

Kind	Type	Attributes
Component	App, Job	all
Store	Disk, SFTP, NFS, Bucket, Database, Messaging	all
Pipeline	Pipeline, Workflow	all, +graph
Resource	Schema, Keyspace, Collection, Index, Topic, Queue, Repository	providedBy attribute under dependencies pins resource to a provider
Endpoint	HTTPRoute, TCPRoute, Ingress, ALB	all
Appliance	VM, Hardware, Machine, Image	-dependencies
Technology	Library, Framework, Language, Technology	-dependencies
SecurityRule	Firewall, SecurityGroup	+rule - from address, to address, port, toport
SecurityRuleGroup	Firewall, SecurityGroup list	+rule[]
Infrastructure	Infrastructure	+contains - appliance, store, component
Environment	Environment	all, lifecycle.ops.catalog/schedule, lease
Capability	Capability	contains - component
Service	SaaS, Jira, Confluence, Bitbucket, Kubernetes, AD	all
User	User	all, lifecycle.ops.catalog/id, +membership generated
Group	Group	all, +members, +roles
Role	Role	all, +permissions
Provider	IDProvider, DirectoryProvider, *Provider	all, +attachment to pick the service to use

The yaml representation of the classification object can be like this:

```yaml
kind: Component
classification:
    type: "App"
    tag: ["payment", "customer"]
    domain: "payment"
    team: "paypaynow"
    capability: "onlinebanking"
    businessUnit: "retail"
```

RUNTIME

While it may seem like something that is relevant for applications, it is equally important to store or discover runtime properties of an asset. Data such as entrypoint, liveness probe URLs and IP addresses, can be collected to drive automation for deployment configurations and other automation tasks.

The table below shows the various intents and usage

Intent	Description
entrypoint	Usually the first location used to access an asset
readiness	Location which can be accessed to check if the catalog item is ready
liveness	Location which can be accessed to check if the catalog item is alive
correctness	Location which can be accessed to verify that the catalog item is healthy and behaving as expected
address	Location where the asset is, usually an IP address but could be other values as well

List of environments where the catalog item is deployed or available.

```
runtime:
    endpoint:
      - intent: entrypoint
        location: "http://host/app"
      - intent: readiness
        location: "/ready"
      - intent: liveness
        location: "/live"
      - intent: correctness
        location: "/correct"
      - intent: "address"
        location: "10.0.0.3"
    environment:
      - prod
      - nonprod
      - test
```

Implementations may choose to store this level of data in a separate storage system and show this detail only when retrieving the catalog item.

Normalisation

An implementation of the operations catalog is advised to perform following optimisations to make the search and access process simpler.

COPYING KEY ATTRIBUTES TO LABELS

Attributes such as layer, tier and contact can be copied to labels so the consumers can take advantage of built-in search and filter operations.

before:

```
metadata:
  layer: web
  tier: 1
  contact: "user-service@example.io"
```

after:

```
metadata:
  layer: web
  tier: 1
  contact: "user-service@example.io"
labels:
    core.ops.catalog/layer: web
    core.ops.catalog/tier: 1
    core.ops.catalog/contact: "user-service@example.io"
```

ADDING AUDIT

The catalog items may be loaded and updated at various intervals. This section shows the information on changes and source of change. This is a dynamic part of the specification and may not be populated immediately.

Source attribute tells us how the catalog item was added. It can be one of api,crd,git

```
audit:
  operations:
```

```
  - name: "updated"
    updated: "2023-07-01T10:00:00+1000"
    description: "App 1 caused dependency to be updated"
  source: api
```

ADDING ANNOTATIONS

The attributes associated with namespaces under properties will be copied and/or merged with annotations.

before:

```
properties:
  lifecycle:
    source-template: "microservices-starter"
```

after:

```
metadata:
  annotations:
    lifecycle.ops.catalog/source-template: "microservices-starter"
properties:
  lifecycle:
    source-template: "microservices-starter"
```

An Example

An example catalog item configuration looks like this:

```
apiVersion: "v1"
kind: Component
metadata:
  name: "template"
  description: "Template for a Component"
  logo: "component.png"
  contact: "template@ops.catalog"
  labels:
    tier: 1
    internet-facing: false
    layer: web
  annotations: {}
includes:
  - java-app
  - backend-team
contact:
  owner:
    id: "@user1"
  contributors:
    - id: "web-team"
  support:
    - id: "#template"
  approvers:
    - id: "[DevExperience]"
properties:
  lifecycle:
    status: "active"
links:
  - type: source
    url: git@github.com:owner/abc.git
  - type: artifact/image
    url: docker://owner/abc
  - type: artifact/jar
dependencies:
  upstream: []
  downstream: []
  triggers: []
runtime:
  endpoint:
    - intent: entrypoint
      location: "http://template/app"
    - intent: "readiness"
      location: "/ready"
  environment:
    - prod
    - nonprod
    - test
classification:
    type: ""
    tag: []
    domain: ""
    team: ""
    capability: ""
    businessUnit: ""
audit: {}
```

Includes

While it is nice to view all the attributes associated with a catalog item in a single response. Managing a denormalised entry can be a pain. Templates help solve refactor your config management further to keep it tidy. Certain traits can be captured as templates and included into each catalog entry.

The following attribute can are allowed in the template specification.

Attribute	Description	Action	Applicable
apiVersion	every item evolves separately		❌
kind	need to be specific for each item		❌
metadata	can merge as many attributes and can be shared	merged	✅
contact	merge is supported to allow teams to have contact in one place	merged	✅
properties	merge is supported to allow multiple components to reuse properties	merged	✅
links	links should be unique but there might be a case of reuse	merged	✅
dependencies	should be unique. very important so do not take shortcut on this one.		❌
classification	can be reused by apps in a domain, team	merged	✅
runtime	if using same framework, most attributes could be reused	merged	✅

Template will always be applied first. The more local declaration of attributes in each catalog entry will override any data template might have added. It also does not always make sense to overwrite or merge data when importing from template. The above table shows the merge strategy applied for each attribute that can be put in template.

8.2 Types

8.2.1 Component

There are two types of Components - App and Job. Most artifacts in a team might fall into this category especially if you are running a microservices architecture.

Here is an example Component object.

```
apiVersion: "v1"
kind: Component
class: App
includes:
  - microservices
metadata:
  name: "checks"
  description: "Customer Credit Check that calls Account check SaaS"
  license: "private"
  logo: "onboarding-check.png"
  contact: "onboarding@my-account.io"
  tier: "1"
dependencies:
  upstream: []
  downstream: ["check SaaS"]
  triggers: []
classification:
    tag: ["onboarding", "origination"]
    domain: "origination"
    team: "loaders"
    capability: "onboarding"
    businessUnit: "retail"
properties:
  dev:
    quickstart: "./gradlew runApp"
    local-run: "docker-compose up -d"
  operations:
    idempotent: "true"
contact:
  owner:
    id: "@user10"
  contributors:
    - id: "@user2"
  support:
    - id: "#credit-check"
  participants:
    - id: "[onboarding]"

links:
  - type: "source"
    url: "https://github.com/my-account/checks"
  - type: "build"
    url: "https://jenkins/my-account/checks/build"
  - type: "docs"
    url: "https://scrolls/my-account/checks/intro"
  - type: "artifact"
    url: "https://quay.io/my-account/checks"
```

8.2.2 Store

Objects cataloged as Store kind will have some capability to provide storage of data. The Kind `Store` has the following types under it: Disk, SFTP, NFS, StorageService (S3, GCS), Database (Postgres, Cassandra), Messaging (Kafka, JMS)

To avoid levels of indirection, the attribute `class` is directly set to one of the implementations for each category of product under Store. For instance, the below example uses `class: Kafka` instead of `class: Messaging` as there is no immediate benefit to keeping messaging. Perhaps, Category can be determined on the fly if and when that is required.

All attributes applicable to components are also valid for Storage objects.

```
apiVersion: "v1"
kind: Store
class: Kafka
metadata:
  name: "kafka"
  description: "SaaS service for log aggregation and analysis"
  license: "Apache 2.0"
dependencies:
  upstream: []
  downstream: []
  triggers: []
classification:
    tag: ["messaging", "stream"]
    domain: "storage"
    team: "keepers"
    capability: "Operations"
    businessUnit: "tech"
```

8.2.3 Resource

The kind `Resource` can have the following classes : Schema, Keyspace, Collection, Index, Topic, Queue, Repository, Certificate

An example resource definition for a Schema:

```
apiVersion: "v1"
kind: Resource
class: Schema
metadata:
  name: "onboarding"
  description: "Schema and DB connection used by onboarding use case"
  license: "private"
dependencies:
  upstream: []
  providedBy: postgres.pg-2
  triggers: []
classification:
    tag: ["origination", "onboarding", "customer"]
    domain: "origination"
    team: "loaders"
    capability: "origination"
    businessUnit: "mybusiness"
```

Note, resources can be linked back to the exact provider (eg Storage or other objects) by `dependencies.providedBy` attribute.

A Cassandra Keyspace resource would look like this:

```
apiVersion: v1
class: Keyspace
kind: Resource
metadata:
    name: notificationstore
    description: Cassandra Keyspace for tables in notificationstore
    labels: {}
    annotations:
        discovery.ops.catalog/replication: |
            {"class":"org.apache.cassandra.locator.NetworkTopologyStrategy","datacenter1":"1"}
    tier: ""
    contact: ""
contact:
    owner: null
dependencies:
    upstream: []
    downstream: []
    providedBy: cassandra.cassandra-1
classification:
    tag: []
    domain: storage
    team: datahoarders
    capability: dataretention
    businessUnit: "tech"
properties:
    lifecycle:
        replication:
            class: org.apache.cassandra.locator.NetworkTopologyStrategy
            datacenter1: "1"
includes:
    - internal
```

Note the attributes under lifecycle namespace which are relevant to Cassandra Keyspace Similarly, a certificate resource is represented like so where the lifecycle namespace contains attributes specific to certificates:

```
apiVersion: v1
class: Certificate
kind: Resource
metadata:
    name: ops-cert
    description: Tls Certificate http://k8s-read:6100
    labels: {}
    tier: ""
    contact: ""
classification:
    tag: []
    domain: platform
    team: devops
    capability: operations
    businessUnit: "tech"
properties:
    lifecycle:
        common-name: docs.ops-catalog.io
        created: "2024-01-18T11:25:20Z"
        expiry: "2024-02-17T11:25:20Z"
        issuer: sparkjob
```

```
        san:
            - docs.ops-catalog.io.default.svc
includes:
    - internal
```

8.2.4 Endpoint

Endpoint Kind is a catalog item type to store information about Access points like Ingress, HTTPRoute, TCPRoute, ALBs and similar loadbalancers.

```
apiVersion: v1
class: HttpRoute
kind: Endpoint
metadata:
    name: service-entrypoint
    description: Http Route in Kubernetes Cluster http://k8s.ops-catalog.io:80
    labels: {}
    annotations: {}
dependencies:
    upstream: []
    downstream:
        - k8s-read
        - account
classification:
    type: App
    tag: []
    domain: platform
    team: devops
    capability: operations
    businessUnit: "tech"
properties: {}
includes:
    - internal
runtime:
    endpoint:
        - intent: entrypoint
          location: k8s.ops-catalog.io
links:
    - type: definition
      url: http://k8s.ops-catalog.io:80/api/crd-instance?resource-group=gateway.networking.k8s.io&resource-type=httproutes&resource-
version=v1beta1&namespace=default&resource-name=k8s-read-http
```

Specific information about endpoint can be kept in extensions map which could be made available contextually.

8.2.5 Templates

TEMPLATES

Incorporating templates into a definition allows for the application of traits from various collections, promoting data reusability. This streamlined approach facilitates the modification of attributes in a centralized location. Templates exhibit hierarchy, enabling them to include traits from other templates, which may, in turn, depend on additional templates.

Essentially, templates share similarities with Catalog Items but are distinguished by having their kind set to `Template`.

The resolution of templates is the initial step, constructing a collection that can subsequently be applied in non-template declarations. To apply a template in a specific definition, the template name is provided in the includes field, which accepts an array of template names.

The order in which templates are included matters; the first template in the list is applied first. Subsequent templates override attributes imported later. Ultimately, the original declaration is applied to the effective structure resulting from the template merge, ensuring that any local declarations on the Catalog Item always take precedence over attributes borrowed from templates.

How do I create a template?

A template looks like a regular Catalog Item. This template `account-team` can now be used by other items (probably) all items in the domain of accounts.

This template also includes another template `internal`

```yaml
apiVersion: v1
kind: Template
metadata:
    name: account-team
classification:
    team: accountants
    domain: accounts
includes:
    - internal
```

Anything declared in internal will be overridden by attributes in account-team. Similarly, when this is included in the definition of a catalog item, the latter includes will override the attributes in this template. Ultimately, local declaration in catalog item will take precedence over any imported by includes.

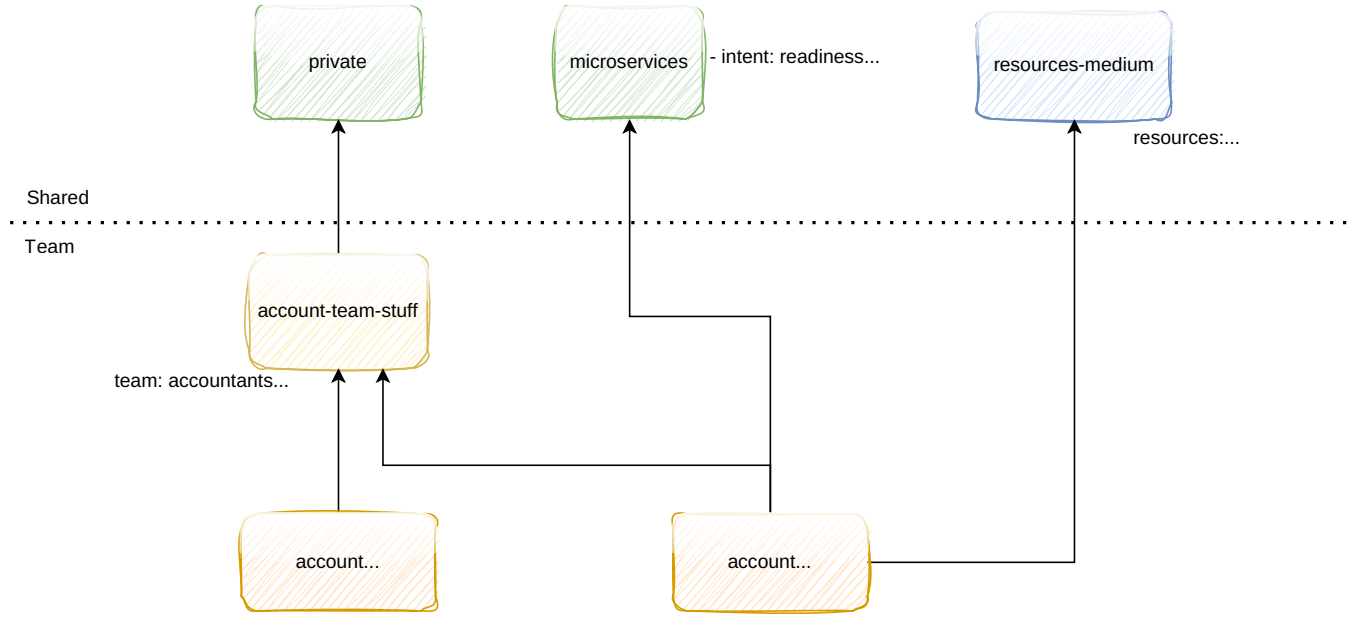


Figure: Organisation of Templates
Text is not SVG - cannot display

When do I use template?

You can use template to reduce the amount of configuration code you need to manage in many places. By grouping related attributes in a single template, you allow them to be shared within the domain and across the organisation.

How do I manage templates?

Templates may coexist with catalog item definitions, but it is advisable for templates to have a dedicated source separate from the main catalog. This separation aids in the meticulous management of crucial attributes, particularly those with a significant impact. Less frequently used or local definitions may be stored on a per-domain or per-team basis, providing a scalable approach as your team expands.

8.3 Catalog

8.3.1 Running

The Ops Catalog can be run in various modes controlled by settings in config file. The catalog app looks to load a config file provided as CONFIG_FILE environment variable. It defaults to a config.yaml in the same directory as the catalog binary.

A docker image is provided for Catalog Application.

`docker pull opscatalog/catalog:latest` You can build image locally via `make go-image`

Running from source is also possible by executing `go run cmd/catalog.go` or `make run-local`

A quickstart guide can be found here

8.3.2 Api Docs

8.3.3 Configuration

The ops catalog project reads configuration provided by file either loaded though CONFIG_FILE environment variable or config.yaml located in the same folder as the catalog binary.

The example config.yaml and the content of it are discussed here.

```
options:
  log: {}
  server: {}
  serve: {}
  fulfillment: {}
  discovery: {}
```

LOG

The log configuration node allows you to provide the minimum log level for the catalog application and the format to write the logs in.

```
  log:
    level: "info"
    format: "json"
```

Valid values for levels are DEBUG, INFO, WARN and ERROR. The application supports plain text and json log format. Default format for logging is json.

SERVER

The address value configured through the server.address attribute is provided as listen host to the go server.

Few example values can show you what is possible: :8080, 0.0.0.0:8080 or 127.0.0.1:8080, some-domain.com:8080

```
  server:
    address: ":8080"
```

SERVE

The source attribute in serve node is an array of various different source types to load ops catalog data from. It can be from a http resource or a git project or a folder. Each source can be individually enabled or disabled. Few source examples are provided below.

Thre refresh frequency configuration can be provided to periodically refresh the catalog data by reloading from sources.

The search.index attribute is an array which can be configured to provide indexing and filter criteria when looking up catalog items.

Because `team` is one of the attribute values for index, we can query the ops catalog with the query parameter of ?team=. Same applies for the other query parameters here.

```
  serve:
    search:
      index:
        - "layer"
        - "tier"
        - "team"
        - "domain"
        - "contact"
        - "type"
        - "language"
        - "name"
    refresh:
      frequency: 1m
    source:
    - location: ../specification/examples/my-account
      extension: ".yaml"
      name: "gitsync"
      enabled: true

    - location: "https://github.com/ops-catalog/discovery-sink.git"
      extension: ".yaml"
      name: "discovery-sink"
      enabled: true
      options:
        username: "skhatri"
```

```
        password: "file:./tmp/password"


    - location: "http://localhost:8000/my-account-catalog2.gz"
      extension: ".yaml"
      name: "gz-http-catalog-source"
      enabled: false
```

DISCOVERY

We try to use the same source object structure for both serving and discovery files. You can add as many discovery sources as you would like as well. Further to that, you can list engines you would like to discover against. Similarly, the refresh frequency attribute helps run the discovery process as frequently as you would like.

Once items are discovered, we would need to persist them somewhere. We can specify a target location to commit the changes to.

```
  discovery:
    source:
      - location: ../examples/datasets/discovery
        extension: ".yaml"
        name: "discover-1"
        enabled: true
    enabledEngines:
      - k8s
      - cassandra
      - kafka
      - airflow
      - postgres
      - aws
    refresh:
      frequency: 10m
    target:
      name: "sink"
      location: "https://github.com/ops-catalog/discovery-sink.git"
      options:
        username: "skhatri"
        password: "file:./tmp/ghkey"
```

If the discovery target and serving source are the same, the newly discovered objects will be available through catalog API as soon as the next serve refresh activity is run.

For teams, who would like to review what has been discovered, it is recommended to commit to a different branch and load serving data from a different branch. A PR review workflow can be added both to validate and to enrich data further before sharing with others.

FULFILLMENT

Fulfillment module requires access to catalog entries. Hence, it needs reference to catalog URL. It can either be referenced in-process via the value of "local" or a location url can be specified when running catalog and fulfillment module separately.

If you are also enabling discovery or adding discovery source in the settings file and would like to reuse the discovery configuration for fulfillment as well, it is possible to do so by enabling the flag `useDiscovery`

Use the target attribute to provide custom fulfillment targets.

```
  fulfillment:
    reference:
      catalog:
        #location - local, url
        location: "local"
      useDiscovery: true
      target:
        - name: broadcast
          location: "kafka://kafka.ops-catalog.io:9092/catalog-events"
          options:
            username: "kafka"
            password: "none"
    run:
      frequency: 2m
    enabledEngines:
      - cassandra
      - postgres
      - kafka
#     - broadcast
```

The fulfillment task can run periodically and can be specified through run.frequency attribute under fulfillment.

Finally, enabledEngines array can be configured to choose one or many targets against which you would like to provison catalog items.

A complete config.yaml is provided for your reference.

```yaml
options:

  log:
    level: "info"
    format: "json"
  server:
    address: ":8080"

  fulfillment:
    reference:
      catalog:
        location: "local"
      useDiscovery: true
      target:
        - name: broadcast
          location: "kafka://kafka.ops-catalog.io:9092/catalog-events"
          options:
            username: "kafka"
            password: "none"
    run:
      frequency: 2m
    enabledEngines:
      - cassandra
      - postgres
      - kafka
      - broadcast

  discovery:
    source:
      - location: ../specification/examples/discovery
        extension: ".yaml"
        name: "discover-1"
        enabled: true
    enabledEngines:
      - k8s
      - cassandra
      - kafka
      - airflow
      - postgres
      - aws
    refresh:
      frequency: 10m
    target:
      name: "sink"
      location: "https://github.com/ops-catalog/discovery-sink.git"
      options:
        username: "skhatri"
        password: "file:./tmp/ghkey"


  serve:
    search:
      index:
        - "layer"
        - "tier"
        - "team"
        - "domain"
        - "contact"
        - "type"
        - "language"
        - "name"
      parallel: 1
    refresh:
      frequency: 1m
    source:
    - location: ../specification/examples/my-account
      extension: ".yaml"
      name: "gitsync"
      enabled: true

    - location: "https://github.com/ops-catalog/discovery-sink.git"
      extension: ".yaml"
      name: "discovery-sink"
      enabled: true
      options:
        username: "skhatri"
        password: "file:./tmp/password"


    - location: "https://github.com/ops-catalog/specification.git"
      extension: ".yaml"
      name: "git-source"
      enabled: false
      options:
        username: "skhatri"
        password: "file:./tmp/password"
        subpath: examples/my-account
```

```yaml
  - location: "git@github.com:ops-catalog/specification.git"
    extension: ".yaml"
    name: "git-ssh-source"
    enabled: false
    options:
      keyfile: "./tmp/key"
      subpath: examples/my-account

  - location: "$HOME/dev/projects/product/opscatalog/my-account-catalog3.tar"
    extension: ".yaml"
    name: "tar-catalog-source"
    enabled: false

  - location: "$HOME/dev/projects/product/opscatalog/my-account-catalog.tar.gz"
    extension: ".yaml"
    name: "tar-gz-catalog-source"
    enabled: false

  - location: "$HOME/dev/projects/product/opscatalog/my-account-catalog2.gz"
    extension: ".yaml"
    name: "gzip-catalog-source"
    enabled: false

  - location: "$HOME/dev/projects/product/opscatalog/my-account-catalog.zip"
    extension: ".yaml"
    name: "zip-catalog-source"
    enabled: false

  - location: "http://localhost:8000/my-account-catalog.zip"
    extension: ".yaml"
    name: "zip-http-catalog-source"
    enabled: false

  - location: "http://localhost:8000/my-account-catalog2.gz"
    extension: ".yaml"
    name: "gz-http-catalog-source"
    enabled: false
```

8.4 Discovery

8.4.1 Specification

A discovery configuration has the type attribute which tells us the engine name that will be able to process it. The instance attribute contains a list of engine instances to scout for resources.

The majority of the information about all engines can be held in a grand total of these five properties under options attribute:

Attribute	Description
host	the server address
port	port to connect to
username	username for auth
password	password for auth
ssl	flag to specify whether TLS is required
cafile	Location of CA file, if not globally trusted and ssl is enabled
keyfile	Location of key file, required for client auth when ssl is enabled
certfile	Location of cert file, required for client auth when ssl is enabled

There are other attributes which are contextual and they are listed here:

Attribute	Description	Applicable to
database	Database Name	Postgres
org	Organisation owning the repositories in scope	Github
project	Project owning the repositories in scope	Bitbucket
use-hints	Flag used to determine whether object enrichment should be done by reading comments, tags, properties etc	Databases,Git Service,Object Store
provider	Name of the provider if multiple provider exist for a type - eg Github, Bitbucket	

There are times when you would like to include or exclude certain objects from being discovered. It is possible to specify a regular expression for both under `filter` attribute.

If you want to provide owner information for discovered items, you can provide the classification attributes like in the example below.

```
kind: Discovery
metadata:
  name: "kafka-discovery"

type: kafka

instance:
  - name: "kafka-1"
    filter:
      excludes: [".*"]
      includes: ["account.*", "deployments.*", "aws-sts.*"]
    enabled: true
    includes:
      - internal
    options:
      host: "kafka.ops-catalog.io"
      port: "9092"
      username: "kafka"
      password: "file:./tmp/kafkapassword"
```

```
      ssl: "false"
      use-hints: "true"
    classification:
      team: "keepers"
      domain: "storage"
      type: "Topic"
      capability: "operations"
      businessUnit: "tech"
    duplicatesStrategy: "ignore"
```

Similarly, you can specify the template names as includes attributes. The template names will be applied to all discovered items. Note that the classification attributes will be preferred over what is included by templates.

To allow maximum flexibility, item level overrides are possible to specify on discovered items in the form of tags, annotations or comments.

The below table summarises the ways to override for each resource type

Resource Engine	Override Approach
Postgres	Comments on Schema
Cassandra	Comments on catalog table in each keyspace
Kafka	None
S3	S3 Object Tags
SNS	SNS Object Tags
SQS	SQS Object Tags
Lambda	Lambda Tags
Kubernetes Workloads	Annotations
Endpoints in Gateway API & Kubernetes	Annotations
Airflow	Tags Array in Dags
Git	Message specified in tag named catalog
ALB	Object Tags

In Scenarios where it is not possible to read further enrichment hints from the above locations, especially true for databases and Git, if running with restricted permissions, one can disable lookup by specifying `use-hints: "false"`. When not provided, it use-hints defaults to true.

Duplicates can be handled by providing a strategy value at instance level.

```
instance:
  ...
  duplicatesStrategy: ""
```

Following are the possible values for `duplicatesStrategy`

Strategy	Description
skip	Drop the discovered item
ignore	Ignore and add discovered item to the catalog
mergeLeft	Use existing catalog object as the base and override with attributes found in discovery
merge	Shorthand for mergeLeft
mergeRight	Use discovered object as the base and override with attributes found in catalog
replace	Replace discovered item with original item from the catalog

Discovery Instances are enabled by default. At times, we may want to disable a declared discovery instance and for this we have the `enabled` flag. Enabled flag can be evaluated at runtime as well with simple expressions. Few examples of valid expressions are listed below.

Expression	Meaning
enabled: "${env.SSL_STATE==on}"	Enable the discovery config when environment variable SSL_STATE is set to "on"
enabled: "${env.GOOS!=darwin}"	Only enable this discovery when not running on MacOS
enabled: "${on=on}"	It also handles text to text comparison. Useful when lhs or rhs the outcome of template substitution
enabled: true	A boolean static value to enable discovery config

See Example Postgres Discovery Config to check how expressions are used to enable different discovery targets.

Discovery Handler only supports equal and not equal checks currently.

8.4.2 Workloads

Component Discovery

Workloads can be discovered from a running Kubernetes instance. We make use of a lightweight Kubernetes Object Query Service that gives us metadata on deployments and statefulsets.

Here is a sample discovery config to retrieve data from Kubernetes excluding few workloads while annotating all of them with team data.

```
apiVersion: "v1"
kind: Discovery
metadata:
  name: "k8s-discovery"

type: k8s
instance:
  - name: "k8s"
    filter:
      excludes: ["local-path-.*", "metrics-.*", "coredns", "envoy.*gateway.*"]
    includes:
      - internal
    options:
      host: "k8s-read.ops-catalog.io"
      port: "80"
      ssl: "false"
      secretPrivateKey: "file:./some-location"
      secretPublicKey: "file:./some-public-key"
    classification:
      team: "devops"
      domain: "platform"
      type: "App"
      capability: "operations"
      businessUnit: "tech"
```

k8s-read performs an encryption on TLS certificates for a public key provided in the request header. The private and public keys can be configured with the option attributes `secretPrivateKey` and `secretPublicKey`

8.4.3 Resources

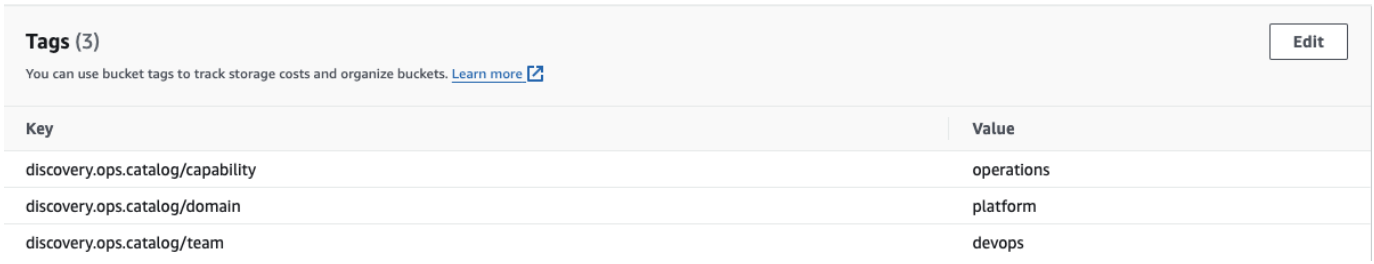
Resources can be discovered from a whole lot of targets like Object Store, Databases, Kafka etc. We share the configuration for each type of discovery below:

AWS RESOURCES

Here is a configuration to retrieve buckets from S3.

```
apiVersion: "v1"
kind: Discovery
metadata:
  name: "aws-discovery"

type: aws
instance:
  - name: "s3-1"
    filter:
      excludes: [".*"]
      includes: ["spark.*", "gpg.*", ".*biller", "aws-sts.*"]
      resources: ["s3"]
    includes:
      - internal
      - data-platform
    classification:
      team: "datahoarders"
      domain: "storage"
      capability: "dataretention"
      businessUnit: "tech"
    options:

      host: "s3.ap-southeast-2.amazonaws.com"
      port: "443"

      username: "file:./tmp/access_key"
      password: "file:./tmp/secret_key"
      region: "ap-southeast-2"

      ssl: "true"
```

This is just like any discovery config, the only difference is the region information. Discovery annotations can be retrieved from storage resources like S3 buckets.



The other supported resource types in AWS are EKS (via Kubernetes Discovery), Lambda, SNS Topics and SQS Queues.

AIRFLOW DAGS

Airflow Dags discovery requires airflow instance to have REST API enabled.

```
apiVersion: "v1"
kind: Discovery
metadata:
  name: "airflow-discovery"

type: airflow

instance:
  - name: "airflow-1"
    includes:
      - internal
    options:
      host: "airflow.ops-catalog.io"
      port: "8280"
      username: "admin"
      password: "file:./tmp/airflow"
      ssl: "false"
    classification:
      team: "dataplatform"
      domain: "jobs"
```

```
      capability: "jobs"
      businessUnit: "tech"
```

Object level enrichment overrides will be applied from tags array found in Dags API

KAFKA TOPICS

Here is a config for Kafka discovery

```
apiVersion: "v1"
kind: Discovery
metadata:
  name: "kafka-discovery"

type: kafka

instance:
  - name: "kafka-1"
    includes:
      - internal
    options:
      host: "kafka.ops-catalog.io"
      port: "9092"
      ssl: "false"
    classification:
      team: "keepers"
      domain: "storage"
      capability: "operations"
      businessUnit: "tech"
```

POSTGRES SCHEMA

Similarly, a postgres example is presented here

```
apiVersion: "v1"
kind: Discovery
metadata:
  name: "pg-discovery"

type: postgres
instance:
  - name: "pg-1"
    includes:
      - internal
    classification:
      team: "datahoarders"
      domain: "storage"
      type: "Schema"
      capability: "dataretention"
      businessUnit: "tech"
    options:
      host: "postgres.ops-catalog.io"
      port: "5432"
      username: "postgres"
      password: "file:./tmp/pgpassword"
      ssl: "false"
      database: "servicing"

  - name: "pg-2"
    options:
      host: "postgres.ops-catalog.io"
      port: "5432"
      username: "postgres"
      password: "file:./tmp/pgpassword"
      ssl: "false"
      database: "preferences"
```

Note you can have as many instance configs as you like. Here we are discovering schemas from two different databases inside the same database host. We are enriching one discovery with ownership information whereas the other one is not enriched at discovery time.

Schema level overrides can be applied in the form of Postgres Schema comments.

Example comment:

```
COMMENT ON
    SCHEMA refdata IS '{
    "discovery.ops.catalog/skip": "false",
    "discovery.ops.catalog/team": "notifiers",
    "discovery.ops.catalog/domain": "jobs",
    "discovery.ops.catalog/capability": "datamart",
    "discovery.ops.catalog/includes": "data-stuff,internal"
  }';
```

CASSANDRA KEYSPACES

Cassandra Keyspace discovery config is very similar to other discovery types.

```
apiVersion: "v1"
kind: Discovery
metadata:
  name: "cassandra-discovery"

type: cassandra

instance:
  - name: "cassandra-1"
    options:
      host: "cassandra.ops-catalog.io"
      port: "9042"
      username: "cassandra"
      password: "file:./tmp/casspassword"
      ssl: "false"
    includes:
      - internal
    classification:
      team: "datahoarders"
      domain: "storage"
      type: "Keyspace"
      capability: "dataretention"
      businessUnit: "tech"
```

It is not possible to add comment to Cassandra Keyspace. Hence we look for a table called catalog and read its comment if available to perform item level override.

```
CREATE TABLE catalog (
    ID text primary key,
    CONTENT BLOB
) WITH
comment = '{
  "discovery.ops.catalog/skip": "false",
  "discovery.ops.catalog/team": "account",
  "discovery.ops.catalog/domain": "onboarding",
  "discovery.ops.catalog/capability": "onboarding",
  "discovery.ops.catalog/includes": "onboarding-stuff,internal"
}';
```

GIT REPOSITORIES

Given that a Repository source is an attribute contained by a Catalog Item representing an app, it is possible that you want to only include select git projects as catalog item (ones which are used by GitOps).

```
apiVersion: "v1"
kind: Discovery
metadata:
  name: "github"

type: git

instance:
  - name: "personal-github"
    includes:
      - internal
    filter:
      includes: [".*by-example", ".*playground"]
    options:
      host: "api.github.com"
      port: "443"
      username: "file:./tmp/githubusername"
      password: "file:./tmp/githubpassword"
      ssl: "true"
    classification:
      team: "devexp"
      domain: "platform"
      capability: "operations"
      businessUnit: "tech"
    duplicatesStrategy: "ignore"

  - name: "org-github"
    includes:
      - internal
    filter:
      excludes: [".*"]
      includes: [".*examples", ".*specification"]
    options:
      host: "api.github.com"
      port: "443"
      username: "file:./tmp/githubusername"
      password: "file:./tmp/githubpassword"
      ssl: "true"
      org: "ops-catalog"
```

```
      use-hints: "false"
    classification:
      team: "devexp"
      domain: "platform"
      capability: "operations"
      businessUnit: "tech"
    duplicatesStrategy: "ignore"
```

For the second github instance, we are choosing not to look for enrichment data in a special tag as we provide the attribute `use-hints: "false"`

It is also possible to store all Git Repositories in Ops Catalog. A recommended setup is to run separate instance of OpsCatalog for the purpose of Discovering and Provisioning all Git repositories for your organisation.

Further enrichment hints can be provided to Git Discovery module by creating a "tag" by the name "catalog".

```
 git tag -m'{
  "discovery.ops.catalog/skip": "false",
  "discovery.ops.catalog/team": "data",
  "discovery.ops.catalog/domain": "jobs",
  "discovery.ops.catalog/capability": "analytics",
  "discovery.ops.catalog/includes": "data-stuff,internal"
}' catalog
git push --tags
```

8.4.4 Endpoint

Endpoints can be found in many different targets like Cloud LB, Kubernetes Ingress, Gateway API etc.

KUBERNETES

When you configure Kubernetes for workload discovery, you get Endpoints like Ingress, HTTPRoute, TCPRoute for free.

Discovery Annotations under `Metadata.Annotations` are recognised and processed by discovery agent for endpoints.

Eligible annotations start with `discovery.ops-catalog.io/`

```
kind: Ingress
metadata:
  annotations:
    discovery.ops-catalog.io/contact: '@user2'
    discovery.ops-catalog.io/domain: platform
    discovery.ops-catalog.io/includes: internal,platform
    discovery.ops-catalog.io/skip: "false"
    discovery.ops-catalog.io/team: devops
```

8.5 Fulfillment

8.5.1 Specification

Fulfillment process requires access to catalog as well as targets against which the fulfillment is to be applied.

We can reuse the discovery config as targets and we can either access in-process catalog API or a remote one.

Refer to configuration guide on how to add targets.

While most fulfillment targets will create resources or perform some provisioning action, the system also has a broadcast plugin that will send fulfillment data to a resource target. Only kafka is supported today for this mode.

```
fulfillment:
    reference:
      target:
        - name: broadcast
          location: "kafka://kafka.ops-catalog.io:9092/catalog-events"
          options:
            username: "kafka"
            password: "none"
    enabledEngines:
        - broadcast
```

With a setup like above, each fulfillment action is represented as a JSON payload and forwarded to the topic catalog-events.

8.6 Scorecard

Scorecard

Ops Catalog uses a series of checks to provide a score for each catalog item. This can serve as a guide to improving and refining the data that is contributed by owners of each catalog item.

Each catalog item has a grade that is determined by the score. Possible grades are A+, A, B, C, D, E, F.

The score node is attached to each catalog item where you can see both the label and value.

```
    "score": {
        "value": 64,
        "label": "D"
      }
```

The debt section of the catalog item response has additional information on how points were deducted.

```
    {
      "debt": {
        "entries": [
          {
            "name": "Tier information should be populated",
            "description": "1 point(s) deducted",
            "severity": "medium"
          },
          {
            "name": "A valid contact is recommended",
            "description": "2 point(s) deducted",
            "severity": "medium"
          }
        ]
      }
    }
```

9. Examples

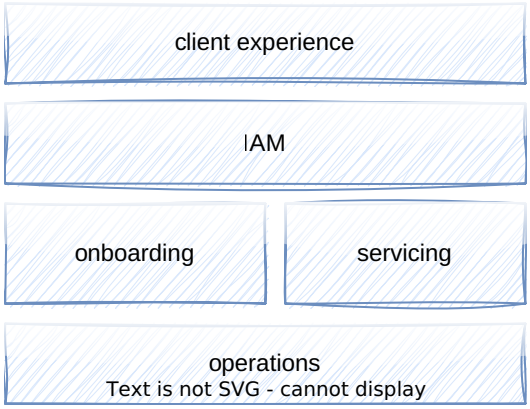
9.1 My Account

Examples

Following are the use cases which have been adapted to follow the operations catalog specification.

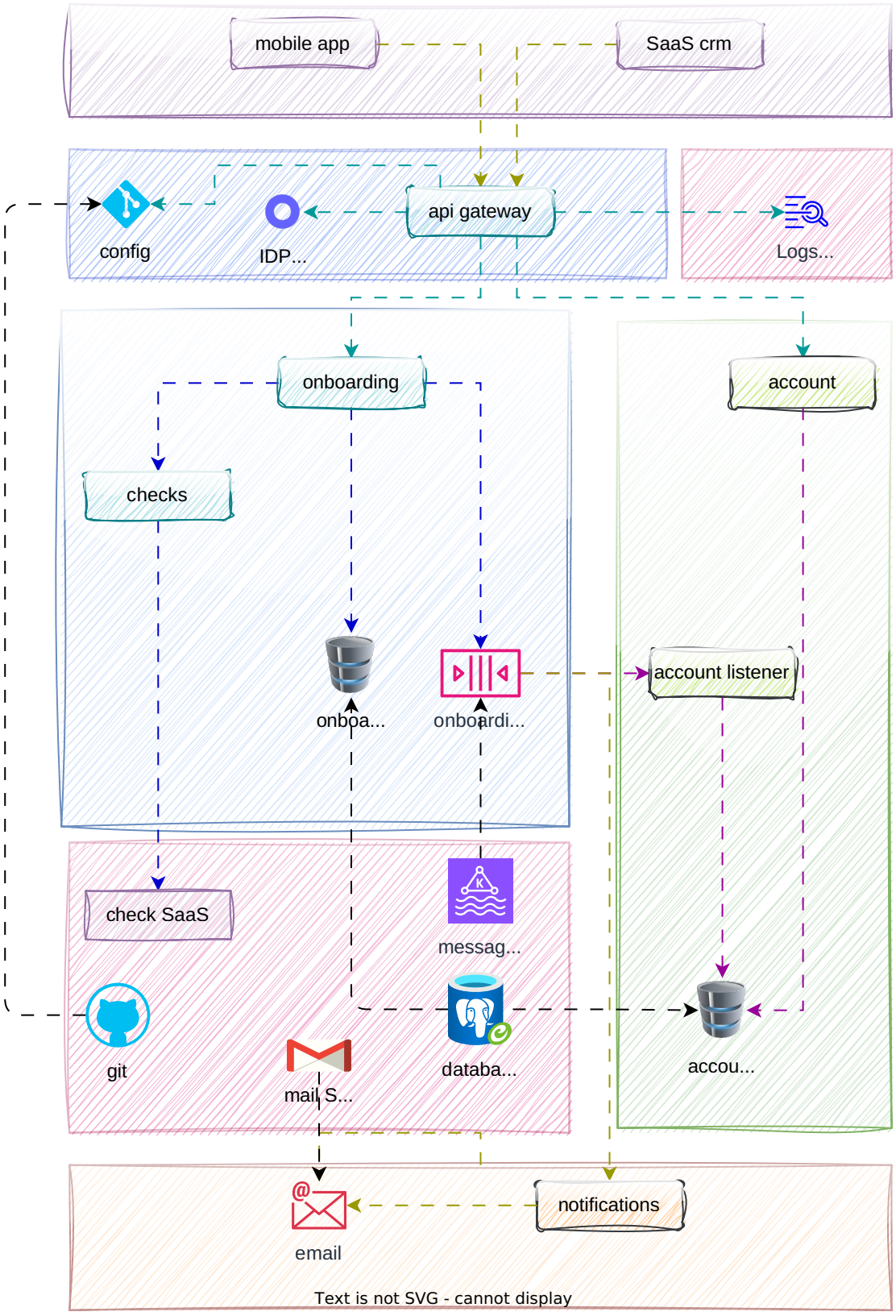
USE CASE 1

The first use case is a generic account management architecture where there are bunch of services, SaaS services, databases and queue. In an online system, there could be few capabilities like - Client Experience, Onboarding, Correspondence, Operations, Servicing



Similarly, multiple domains have been introduced in this architecture to show involvement of various teams to fulfill a typical account opening scenario. Namely, the domains used in this example are infra, preferences, storage, account, origination, gateway and frontend. A capability can have multiple domain. A team can also service multiple capabilities and look after multiple domains.

My Account Architecture



Catalog Item	Kind	Item Type	Domain	Team	Capability
mobile app	Component	App	servicing	frontend	Client Experience
crm	Service	SaaS	servicing	frontend	Client Experience
api gateway	Component	App	gateway	guards	iam
iDP	Service	SaaS	gateway	guards	iam
api config	Resource	Repository	gateway	guards	iam
logs	Service	SaaS	infra	platform	Operations
onboarding	Component	App	origination	loaders	Onboarding
checks	Component	App	origination	loaders	Onboarding
checks SaaS	Service	SaaS	origination	platform	Onboarding
account	Component	App	account	accountants	Servicing
account listener	Component	App	account	accountants	Servicing
notifications	Component	App	preferences	alerters	Correspondence
email	Resource	Mailbox	preferences	alerters	Correspondence
onboarding queue	Resource	Queue	origination	loaders	Onboarding
onboarding database	Resource	Schema	origination	loaders	Onboarding
account database	Resource	Schema	account	accountants	Servicing
mail SaaS	Service	SaaS	infra	platform	Operations
database server	Store	Database	storage	keepers	Operations
messaging service	Store	Messaging	storage	keepers	Operations
git service	Appliance	Git	infra	platform	Operations

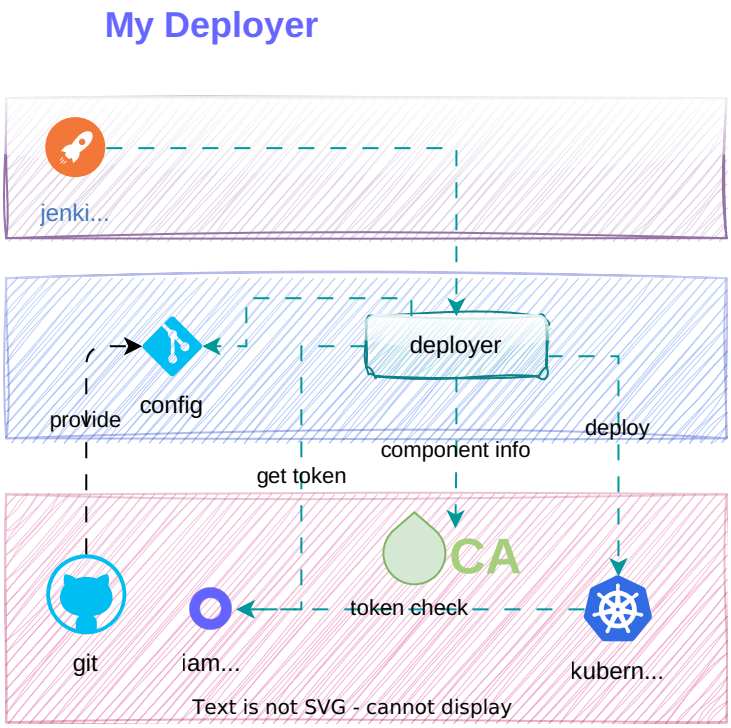
9.2 Deployer

Examples

Following are the use cases which have been adapted to follow the operations catalog specification.

USE CASE 2

This use case shows few components in devops space used to perform deployment using tools like Jenkins, Git and Kubernetes.



Multiple domains have been introduced in this architecture to show involvement of various teams to fulfill a typical deployment setup.

Catalog Item	Kind	Class	Domain	Team	Capability
platform-config	Resource	Repository	tools	platform	Continuous Deployment
onboarding-config	Resource	Repository	tools	loaders	Continuous Deployment
deployer	Component	App	tools	platform	Continuous Deployment
iam Service	Appliance	SaaS	iam	guards	iam
jenkins	Appliance	CD	tools	tinkerers	Continuous Deployment
git	Appliance	Git	tools	tinkerers	Operations
kubernetes	Appliance	Kubernetes	infrastructure	platform	Operations
oca	Component	App	tools	tinkerers	Operations

10. Integrations

10.1 Backstage

Ops-Catalog can be used from right within Backstage by adding a Catalog mapping plugin like so:

CREATE PLUGIN WITH CATALOGBUILDER

path: packages/backend/src/plugins/catalog.ts

```
import { CatalogBuilder } from '@backstage/plugin-catalog-backend';
import { PluginEnvironment } from '../types';
import { OpsProvider } from './OpsProvider';
import { OpsProcessor } from './OpsProcessor';

export default async function createPlugin(
  env: PluginEnvironment,
): Promise<any> {
  const builder = CatalogBuilder.create(env);
  const frobs = new OpsProvider(env);
  builder.addEntityProvider(frobs);
  builder.addProcessor(new OpsProcessor());
  const { processingEngine, router } = await builder.build();
  await processingEngine.start();
  await env.scheduler.scheduleTask({
    id: 'run_ops_refresh',
    fn: async () => {
      await frobs.run()
    },
    frequency: { minutes: 2 },
    timeout: { minutes: 1 },
  });
  return router;
}
```

CREATE PROCESSOR

path: packages/backend/src/plugins/OpsProcessor.ts

```
import {
    Entity,
  } from '@backstage/catalog-model';
  import {
    CatalogProcessor,
    CatalogProcessorEmit,
  } from '@backstage/plugin-catalog-node';
  import { LocationSpec } from '@backstage/plugin-catalog-common';


  export class OpsProcessor implements CatalogProcessor {
    getProcessorName(): string {
      return 'OpsProcessor';
    }

    private readonly validators = [];

    async validateEntityKind(entity: Entity): Promise<boolean> {
      return entity.apiVersion === "v1";
    }

    async postProcessEntity(
      entity: Entity,
      _location: LocationSpec,
      emit: CatalogProcessorEmit,
    ): Promise<Entity> {

      return entity;
    }
  }
```

CREATE MAPPING

path: packages/backend/src/plugins/OpsProvider.ts

```
import {
  ANNOTATION_LOCATION,
  ANNOTATION_ORIGIN_LOCATION,
} from '@backstage/catalog-model';
import {
  EntityProvider,
  EntityProviderConnection,
```

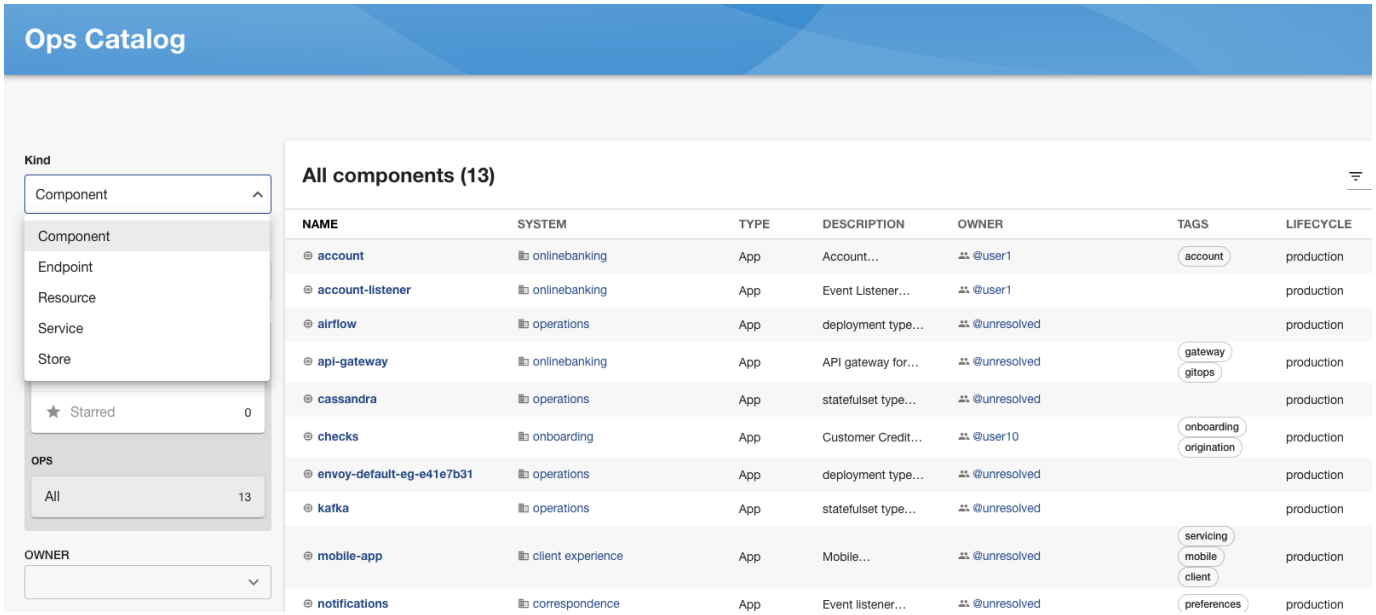
```
} from '@backstage/plugin-catalog-node';
import { PluginEnvironment } from '../types';

export class OpsProvider implements EntityProvider {
  private connection?: EntityProviderConnection;
  private env: PluginEnvironment;

  public constructor(env: PluginEnvironment) {
    this.env = env;
  }
  getProviderName(): string {
    return `ops-catalog`;
  }

  public async connect(connection: EntityProviderConnection): Promise<void> {
    this.connection = connection;
  }

  async run(): Promise<void> {
    if (!this.connection) {
      throw new Error('DB not initilized');
    }
    const catalogObj: any[] = [];
    const catalogResponse = await fetch('http://localhost:8080/api/catalog');
    const d = await catalogResponse.json();

    for (const entity of d.data) {
      let linkSource = entity.links || [];
      const links = linkSource.map((link:{url:string, type:string}) => {
        return {
          url: link.url,
          title: link.type
        }
      });
      let apiVersion = 'backstage.io/v1alpha1';
      let kind = entity.kind;
      switch(kind) {
        case 'Resource':
          break;
        case 'Component':
          break;
        default:
          apiVersion = entity.apiVersion;
      }
      let dependenciesSource = entity.dependencies || { downstream: [], upstream: []};
      let dependencies = (
        dependenciesSource.downstream || []).concat(dependenciesSource.upstream || []
        );

      let name = `${entity.metadata.name.replaceAll(" ", "")}`;
      let owner = '@backstage/maintainers';
      if (entity.contact.owner) {
        owner = `${entity.contact.owner.id}`;
      }
      let annotations = entity.metadata.annotations || {};
      annotations[ANNOTATION_LOCATION] ='ops:https://ops-catalog.io/';
      annotations[ANNOTATION_ORIGIN_LOCATION]='ops:https://ops-catalog.io/';

      let tags = entity.classification.tag || [];
      let sourceLabels = entity.metadata.labels || {};
      let labels = new Map(Object.entries(sourceLabels)
      .map(([k,v]) => [k,String(v).replaceAll(" ", "").replaceAll("@", "_at_")]))
      let system = entity.classification.capability || "unknown";
      const catalogEntity = {
        kind: kind,
        apiVersion: apiVersion,
        metadata: {
          description: `${entity.metadata.description}`,
          annotations: annotations,
          labels: labels,
          links,
          // name of the entity
          name: `${name}`,
          title: `${name}`,
          tags: tags,
        },
        spec: {
          type: `${entity.class}`,
          lifecycle: 'production',
          owner: owner,
          profile: {
            displayName: `${entity.metadata.name}`,
            email: '',
            picture: '',
          },
          dependsOn: dependencies,
          memberOf: [],
          system: `${system}`
        },
      };

      catalogObj.push(catalogEntity);
    }
```

```
    await this.connection?.applyMutation({
      type: 'full',
      entities: catalogObj.map(entity => {
        return { entity, locationKey: 'ops:https://ops-catalog.io/' };
      }),
    });
  }
}
```

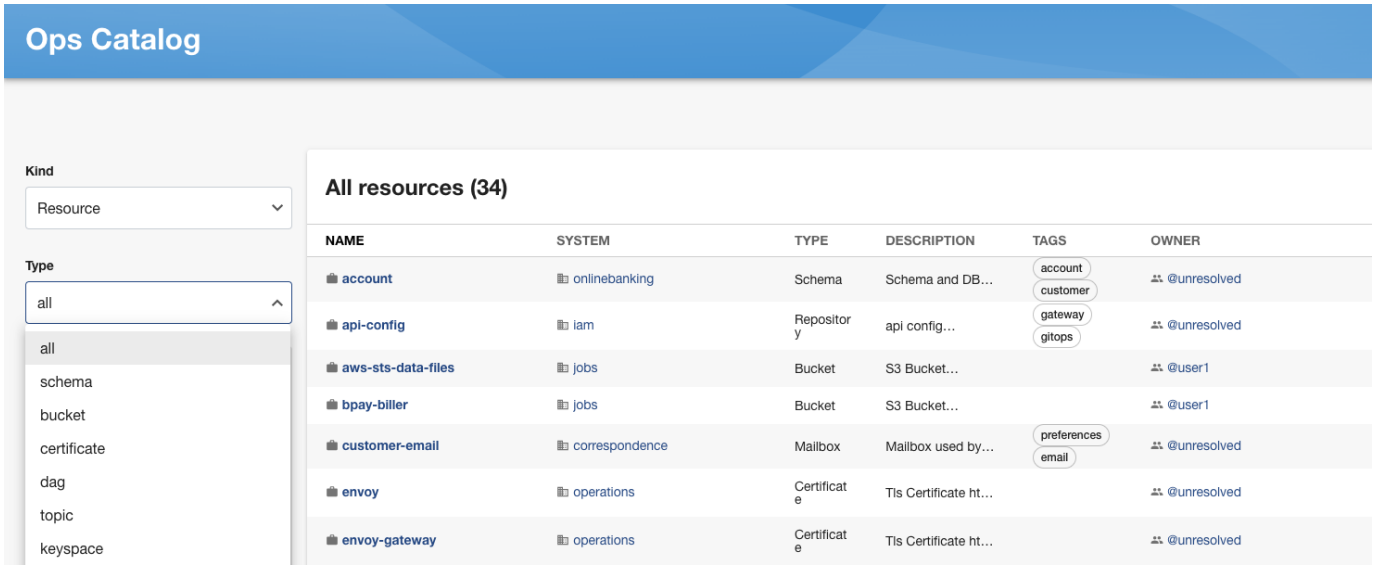
UI

This will show new Kinds in backstage and the relationship defined in ops-catalog will be visible there too.

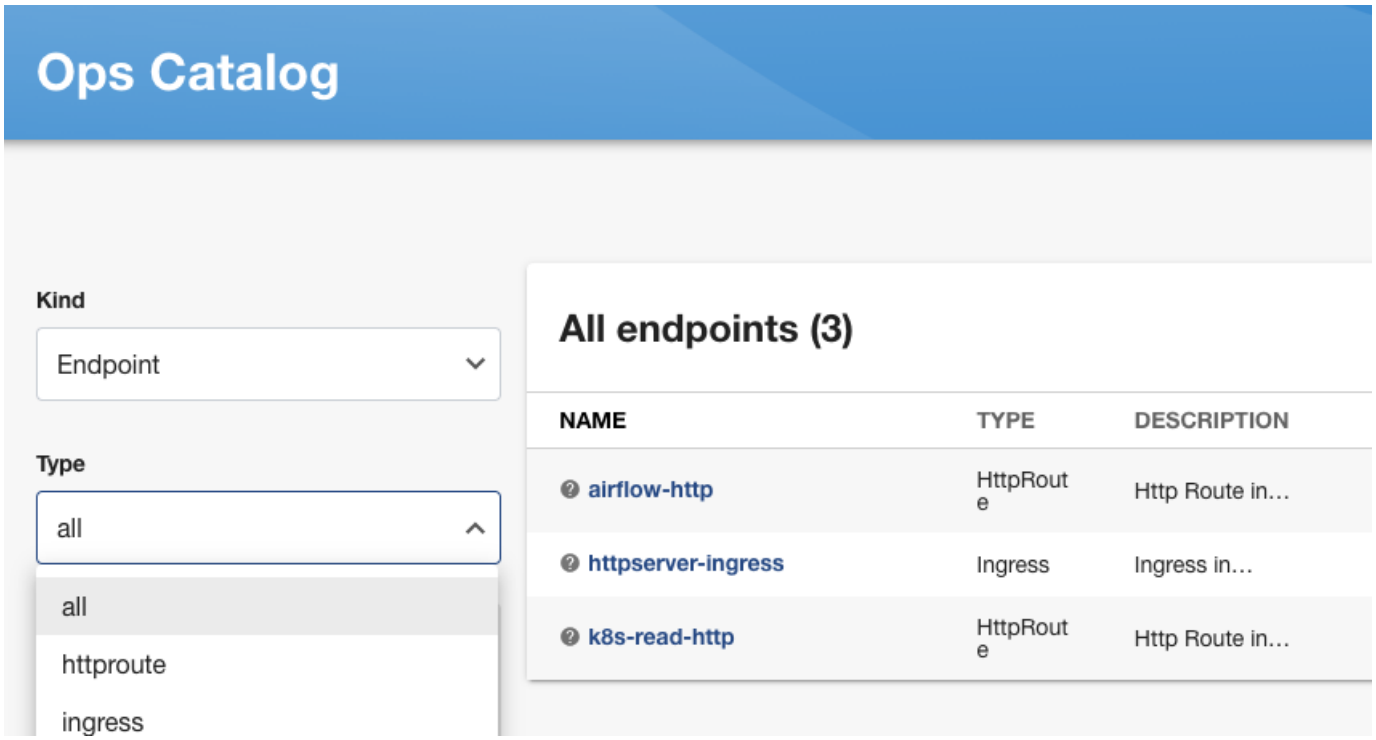
Component View



Resource View



Endpoint View



11. FAQs

11.0.1 FAQs

IS CATALOG SERVER ESSENTIAL?

While Catalog Server serves as a reference implementation, its purpose is to foster collaboration in the DevOps space regarding the standardized capture of assets and attributes. You have the flexibility to select and adopt functionalities that align with their needs. Contributing custom implementations to the community would be mutually beneficial, enhancing the collective understanding and utility of the system.

WHY DO I NEED API?

You can avoid performing dry run. It becomes a shared asset which many tools can use for their operations.

HOW DO I HANDLE ENVIRONMENT SPECIFIC PROPERTIES?

To address environment-specific properties, consider incorporating them into templates with selectors, enabling the activation of specific templates based on the relevant environment. Alternatively, you may opt to place these properties in separate sources, allowing for controlled inclusion—particularly beneficial in environments requiring independent instances for each specific setting.