Curator

Release 0.1

Red Hat

Aug 10, 2022

CONTENTS

Curator (/kyoorādr/) is an air-gapped infrastructure consumption analysis project for OpenShift Container Platform.

Check out the *About* section for further information, including how to installation the project.

Note: This project is under active development.

CHAPTER

ONE

CONTENTS

1.1 About

1.1.1 Operator Curator

Operator Curator is an air-gapped infrastructure consumption analysis tool for the Red Hat OpenShift Container Platform. Curator retrieves infrastructure utilization for the OpenShift Platform using Operator koku-metrics and provides users the ability to query the infrastructure utilization based on time period, namespace, and infrastructure parameters.

Users can generate periodic standard and custom reports on infrastructure utilization, which are optionally delivered through automated emails. Curator also provides APIs to query the information utilization data that is stored in a database in the OpenShift cluster and it can also be used to feed data collected to any infrastructure billing system or business intelligence system. Additionally, Curator also provides administrators of the OpenShift cluster the option to back up their cluster infrastructure consumption data to S3-compatible storage.

You need to have administrator access to an OpenShift v.4.5+ cluster to deploy Operator Curator. For more information on the prerequisites, please view the Requirements section. Once deployed, all the authorized users and systems will be able to view the infrastructure utilization of OpenShift.

1.1.2 Operator Koku-metrics

Operator Koku-metrics can be used to obtain Red Hat OpenShift Container Platform (OCP) usage data and upload it to koku. The Koku-metrics operator utilizes Golang to collect usage data from an OCP cluster installation. You must have access to an OpenShift v.4.5+ cluster.

1.2 Usage

1.2.1 Use Cases

With Operator Curator,

Red Hat OpenShift Platform Administrators can:

- know the OpenShift cluster infrastructure utilization for the entire cluster.
- view infrastructure utilization for any individual namespace.
- view aggregated infrastructure utilization for a group of namespaces.
- notice the infrastructure utilization trend over time.
- get weekly, daily and monthly reports for the OpenShift infrastructure utilization.

- query Curator API to get the latest OpenShift infrastructure utilization.
- back up the infrastructure utilization data on S3-compatible storage.

Analysts can:

- pass the infrastructure usage data to an internal billing system.
- export the infrastructure usage data in CSV format to any tool.

1.3 Features

1.3.1 Periodic Reports

Using Curator, an administrator of OCP can generate daily, weekly and monthly infrastructure utilization reports generated in the form of a CSV file. Administrators will have the ability to set the report start time so that a daily report can be generated after 24 hours, a weekly report can be generated after 7 days and a monthly report can be generated after 30 days.

1.3.2 Custom reports

Using Curator, an administrator of OCP can also generate infrastructure utilization report between a start time and end time using the Curator API to generate the following reports:

- A report for the entire OCP cluster.
- A report for a specific namespace.
- An aggregated report for a group of namespaces.
- A report which contains only specific parameters.

1.3.3 Data pipelining

OCP administrators can run scripts that are packaged in the Curator to stream the infrastructure utilization data collected by Curator to their database for any internal billing machine.

1.3.4 Mailing services (Optional)

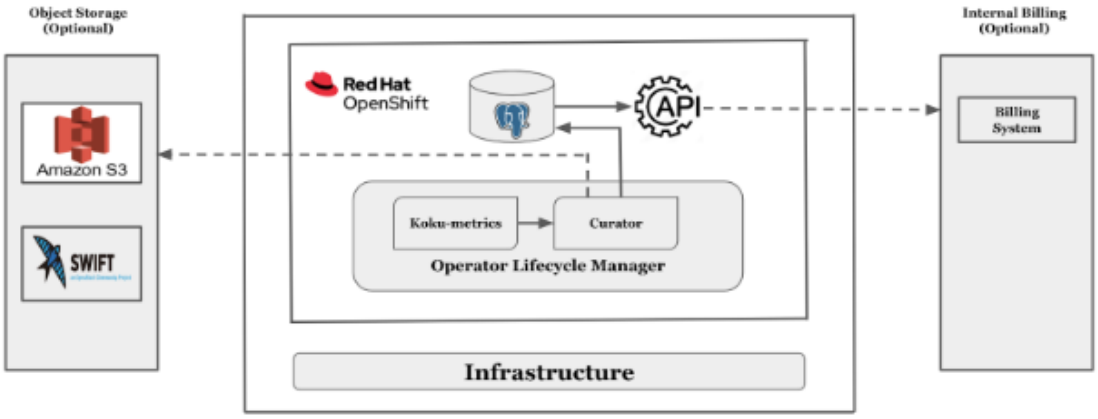
Using Curator, OCP administrators will have the ability to automatically send reports that are generated to any email addresses.

1.3.5 S3 backup (Optional)

OCP administrators will have the ability to backup the cluster infrastructure utilization data in a S3-compatible object storage.

1.4 System Design

1.4.1 System Architecture



1.4.2 Dependencies

The install and deploy Operator Curator, the following prerequisites need to be met:

- Admin access to OpenShift 4.5+
- Install Koku Metrics Operator
- You need to install koku-metrics-operator either via operator-hub or command line.
- We are currently using koku-metrics-operator version 0.9.8 with koku-metric config specified here.
- Install PostgreSQL Image
- Need to have a Postgres database up and running in a cluster to store all information.
- We are currently using postgres version 10.X.

1.5 Requirements

To run Curator Operator, the users are expected to meet the prerequisites listed below.

1.5.1 Prerequisite

Admin access to OpenShift 4.5+

Currently, we are using **Operator-SDK v1.17.0, Go v1.17.6, Postgres v13.4, and koku-metrics-operator v1.1.7**

Install Koku Metrics Operator

- You need to install koku-metrics-operator either via operator-hub or command line.
- We are currently using koku-metrics-operator version 1.1.7 with koku-metric config specified here.
- Run koku-metrics-operator config

```
#Documentation/kokumetris-cfg.yaml
$ oc apply -f Documentation/kokumetris-cfg.yaml
```

Install PostgreSQL Image

- Need to have Postgres database up and running in a cluster to store all information.
- We are currently using Postgres version 13.4 and config file specified sample.

Create database secrets

- First, create project/namespace using below command or web console

```
$ oc new-project <project_name>
```

- We need to create base64 encoded database secrets. To encode a string to base64 from the command-line use the below command

```
$ echo -n "curator" | base64
```

- For creating secrets run below commmand. Sample file is here.

```
#Documentation/db-secret-file.yaml
$ oc apply -f Documentation/db-secret-file.yaml
```

1.6 Installation

1.6.1 Deploying the Operator

1. **Clone the operator curator** repository.

   Currently, we have two versions of the curator operator.

   version 1 (production branch) - Curator operator with basic features (Periodic Reports, Custom reports)

   version 2 (operator-additional-features branch) - Curator with additional features (Mailing services, S3 backup Optional)

2. **To build and deploy Operator you can use one of the two branches.**

   First, update a project/namespace in *config/default/kustomization.yaml*. This is where we are going to deploy Curator Operator.

   Before running the operator, the CRD must be registered with the Kubernetes apiserver:

```
#cd curator-operator/
#switch to production or operator-additional-features branch
#you can use quay.io or docker.io to build and push operator image
make install
make docker-build docker-push IMG=quay.io/<user-name>/<image-name>
make deploy IMG=quay.io/<user-name>/<image-name>
```

The operator curator is running but is not doing any work. We need to create a CR.

3. **Create below two custom resources**

- FetchData custom resource to fetch data from koku-metrics-operator. Sample CR
- Report custom resource to generate the automatic report. Sample CR

```
# update the cronjobNamespace field to the correct namespace where you␣
↪installed koku-metrics-operator
oc apply -f <cr-file-path>
```

1.6.2 Uninstall a CustomResourceDefinition

```
#cd curator-operator/
make uninstall
```

When you uninstall a CRD, the server will uninstall the RESTful API endpoint and delete all custom objects stored in it.

1.6.3 Undeploy the Operator

```
#cd curator-operator/
make undeploy
```

The above command will delete everything including the project.

1.7 API

API Deployment

Deploy the API to Openshift

1 Download raw Koku-Metric-Operator reports for given time frame

```
oc port-forward <curator-operator-pod> 5000:8082
curl "http://localhost:5000/download?start=2022-04-09%2020:00:00&
↪end=2022-04-11%2021:32:23" -o <report-folder-name>
```

`start` and `end` parameters will be cast to PostgreSQL timestamp. Therefore multiple formats are supported.

Downloaded report follows similar structure of a Koku-Metrics-Operator uploaded report.

```
<start>-<end>-koku-metrics.tar.gz
├── <start>-<end>-koku-metrics.0.csv
├── <start>-<end>-koku-metrics.1.csv
├── <start>-<end>-koku-metrics.2.csv
└── <start>-<end>-koku-metrics.3.csv
```

2 Create a sample `Report` by defining the parameters

- reportingEnd: Required, RFC 3339 Datetime.
- reportingStart: Optional, RFC 3339 Datetime.
- reportPeriod: Optional, String. One of Day, Week, Month. Report period N = 1, 7, 30 days.
- namespace: Optional, String. Show report for namespace only. If omitted, show report for all namespace.
- metricsName: Optional, array of string. Show report aforementioned metrics only. If not mentioned, show a report for all metrics.

  Valid options are:

  - pod
  - pod_usage_cpu_core_seconds
  - pod_request_cpu_core_seconds
  - pod_limit_cpu_core_seconds
  - pod_usage_memory_byte_seconds
  - pod_limit_memory_byte_seconds
  - pod_request_memory_byte_seconds
  - node_capacity_cpu_cores
  - node_capacity_cpu_core_seconds
  - node_capacity_memory_bytes
  - node_capacity_memory_byte_seconds

Method 1: Time frame report

Provide parameter for both `reportingStart` and `reportingEnd`. (`reportPeriod` will be ignored if provided)

Result report contains raw CPU and memory metrics for time frame [`reportingStart`, `reportingEnd`) in project `namespace` (if provided).

```yaml
# config/samples/curator_v1alpha1_reportapi.yaml
apiVersion: curator.operatefirst.io/v1alpha1
kind: ReportAPI
namespace: report-system
metadata:
  name: timeframe-report-sample
spec:
  reportingStart: "2022-07-18T00:00:00Z"
  reportingEnd: "2022-07-20T00:00:00Z"  # prevents Reports targeting␣
↪future time
```

(continues on next page)

(continued from previous page)

```
  namespace: koku-metrics-operator
  metricsName: ["pod", "pod_request_cpu_core_seconds"]
```

Method 2: Standard daily, weekly, monthly report

Provide parameter for both `reportPeriod` and `reportingEnd`.

Result report contains raw CPU and memory metrics for the past N days until reportingEnd (including reportingEnd) in project `namespace` (if provided).

```
# config/samples/curator_v1alpha1_reportapi.yaml
apiVersion: curator.operatefirst.io/v1alpha1
kind: ReportAPI
namespace: report-system
metadata:
  name: daily-report-sample
spec:
  reportingEnd: "2022-07-20T00:00:00Z"  # prevents Reports targeting␣
↪future time
  reportPeriod: Day
  namespace: koku-metrics-operator
  metricsName: ["pod", "pod_request_cpu_core_seconds"]
```

Create one of the two Reports above you just defined:

```
oc project curator-operator-system
# Using project "curator-operator-system" on server ...
oc apply -f config/samples/curator_v1alpha1_reportapi.yaml
```

Access the Report by identifying Report by name and namespace it was created. For example, to access `daily-report-sample` on namespace `curator-operator-system`:

```
oc port-forward <curator-operator pod> 5000:8082
curl -XGET "http://localhost:5000/report?reportName=daily-report-
↪sample&reportNamespace=curator-operator-system"
```

1.8 Development and testing guide

1.8.1 Summary

This document covers the details related to contributing to this project. Details such as modifying the code, testing this modification, and debugging the issues will be listed in this document. If you are facing issues that are beyond the scope of this document and would like to discuss them with the Curator Team, you are more than welcome to raise an issue on our GitHub issue tracker.

1.8.2 Introduction

In this section, let's dive into the internals of the operator which explains the crux behind its working and its components. The curator operator runs on a distroless image which has only GO-related packages available in its core. Please refer to the architecture section to get a better understanding of the operator internals. The operator consists of 3 different controllers, API controller, Report Controller, and FetchData controller.

API Controller - It servers the API requests made to the controller to get the reports.

Report Controller - This runs the cron job based on the configuration(Daily, Weekly, and Monthly).

FetchData Controller - Fetches the data from Koku-metrics-operator, suggested running every 6 hours.

1.8.3 How to contribute to this project

This will serve as a kick-start guide for anyone who is willing to contribute to this project. Any changes, bug fixes or enhancements are always welcome.

Adding new Custom resources(CRs)

Custom resources help in modifying the way our system behaves up to a certain extent. Variables such as DB connections, email accounts and cron job timing can be set via the CRs.

New CRs can be added to the existing `api/version/<name>_types.go` files or by creating a new controller which creates a new `api/version/<name>_types.go`

This document might be helpful when dealing with CR validation: Custom Resource Validation

Once the new CRs are added or the existing CRs are modified make sure to run the below command to register these CRs and create the manifests.

```
make manifests

make generate
```

These parameters can be accessed in your controller code and can be used for any logical implementation. Check the existing code to find examples in the `controller/*` directory.

Modifying DB

Most of the core logic for report generation is built around the DB function and tables. You may want to have a tweak at it. All the DB-related manipulations exist in `internal/db/psqldb.go` there are queries that create a table and a query to create a function.

`generate_report(frequency_ text)` function carries the logic for fetching from the DB tables and generating the reports. Making modifications to this will change the the logic for generating new reports.

This logic is run when the controller is set up for the first time, hence creating DB tables and functions on the attached Database. If you run into issues while deploying this code, check the debugging section for any help.

Creating new controllers

For some new additional features it might be handy to create a new controller, for this there 2 important things that have to be in place:

- Custom resource definitions (API)
- Controller logic (Controller)

As this is a generic topic, hence attaching a link on how to create a new API group and a controller: Create a new API and Controller

The containers

Some of the logic exists inside a container that runs when we either call FetchData or the Report controllers. This project relies on 2 containers which are below:

- FetchData: `docker.io/surbhi0129/crd_unzip:latest`
- Email Reporter: `docker.io/rav28/email_controller:v0.0.12`

You can always pull these containers, modify their logic and update the latest container path in the controller code.

1.8.4 Testing

Currently, as there are no unit tests or testing frameworks it's important to test the sanity of the updated code for every PR. Below are a few steps or tests that can be run on the project to verify that nothing breaks.

Testing Day reports

Day reports are currently run at a particular time(currently at 8:05 PM EST) but for testing the changes its not practical to wait so long, hence change the time in the `controllers/report_controller.go`. Setting values like "*/2 * * *" would run the reports every 2 minutes.

Tip: use https://crontab.guru/ to get cron job configuration.

Note: Running Day/Month/Weekly reports keeps updating the reports_human table with duplicate data, hence test it only in a controlled DB environment.

Steps to test day reports

```
make manifests

make generate

make install

make docker-build IMG=docker.io/user/name:version #suggest you to use this␣
↪format when developing

make docker-push IMG=docker.io/user/name:version

make deploy IMG=docker.io/user/name:version
```

(continues on next page)

(continued from previous page)

```
oc apply -f path_to_report_yaml/curator_v1alpha1_report.yaml
```

Verify the logs in the Day report container in the `koku-metrics-operator` namespace. Also, verify the reports_human table in the DB.

Expected Result: `generate_report` function should be called with the right frequency and the `reports_human` table should be populated with new data.

Database creation

The below steps can be used to verify the DB creation in the psql. Deploying the operator connects the DB and deploys the Tables and Functions automatically.

Test cases:

Test: Create a DB with no tables and deploy the operator. Expected Result: All tables and routines should be created in the DB.

Test: Create a DB with a few tables and deploy the operator.

Expected Result: Pre-existing tables will not be modified but new tables and functions will be created.

Test: Create a DB with all the tables and no function and deploy the operator.

Expected Result: No new tables will be created but only the function will be created.

Test: Create a DB with all tables and a function with wrong return parameters.

Expected Result: Operator will throw an error and fail the deployment as the function has the wrong return parameters.

Test: Create a DB with all tables and correct functions and deploy the operator.

Expected Result: No changes in the DB.

Fetch Data controller

FetchData controller has a minimal job of fetching the data from the koku-metrics operator and populating the curator DB logs tables.

Steps:

Run all the commands until `make deploy` post that runs the below command to launch the cron job.

```
oc apply -f path_to_report_yaml/curator_v1alpha1_fetchdata.yaml
```

Note: It is recommended to run this cronJob once in 6 hours as the koku-metrics-operator updates it only once in 6 hours. But for testing, you could run it instantly.

Expected Results: logs_0, logs_1, and logs_2 tables will be populated with the latest data from koku-metrics-operator.

Sanity of the data generated

Any changes to the report_human table will need some sanity testing to be done in order to verify the results.

Sanity test can be done by querying Prometheus using PromQL queries.

Below are few sample queries which are used currently:

```
Login to openshift console -> Administrator -> Observe -> Metrics -> Run the
query
```

Table 1: PromQL queries for Report Parameters

Report Parameter	PromQL Query
Pods CPU request - Prometheus Report (Millicore)	sum(sum(kube_pod_container_resource_requests{resource='cpu',namespa * on(pod, namespace) group_left kube_pod_status_phase{phase='Running'}) without (container, instance, uid))
Pods memory request total - Prometheus Report (MB)	sum(sum(kube_pod_container_resource_requests{resource='memory',nam * on(pod, namespace) group_left kube_pod_status_phase{phase='Running'})) without (container, instance, uid) /1024 /1024
Volume storage request - Prometheus Report (GB)	sum(kube_persistentvolumeclaim_resource_requests_storage_bytes * on(persistentvolumeclaim, namespace) group_left(volumename) kube_persistentvolumeclaim_info{volumename != ''}) BY (namespace)
Volume storage usage - Prometheus Report (GB)	sum(kubelet_volume_stats_used_bytes * on(persistentvolumeclaim, namespace) group_left(volumename) kube_persistentvolumeclaim_info{volumename != '', namespace = <namespace>})
Volume storage request - Prometheus Report (GB)	sum(kubelet_volume_stats_capacity_bytes * on(persistentvolumeclaim, namespace) group_left(volumename) kube_persistentvolumeclaim_info{volumename != '', namespace = "koku-metrics-operator"})

1.9 FAQs

1.9.1 1. What are the report parameters?

Below is the list of report parameters and their significance in the reports. The reports are stored in the `reports_human` table.

Table 2: Report Parameters

Report Parameter	Definition	Data Representation
frequency	Frequency at which the report is generated	Day, Week or Month
Interval_start	Time interval from when the data collection was started	Time stamp with time zone
Interval_end	Time interval when the data collection was ended	Time stamp with time zone
namespace	The namespace to which the resource belongs to	String
pods_avg_usage_cpu_core_total	Average usage of the CPU cores in that namespace	Millicore
pods_request_cpu_core_total	Total CPU request made by the namespace	Millicore
pods_avg_usage_memory_total	Average memory used by a particular namespace	Megabytes[MB]
pods_request_memory_total	Total memory requested by the namespace	Megabytes[MB]
pods_limit_memory_total	Maximum limit for memory in that namespace	Megabytes[MB]
volume_storage_request_total	Total volume requested by the namespace	Gigabytes[GB]
persistent_volume_claim_capacity_total	Total capacity of the PVC in that namespace	Gigabytes[GB]
persistent_volume_claim_usage_total	Average usage of the PVC in a particular namespace	Gigabytes[GB]

1.9.2 2. How to check for reports?

There are a few things that you should verify before checking for reports or logs:

- Verify the pod is in a completed state(This ensures that there was no error when the pod ran)
- Check for logs in the pod(This should show some values)

Once these 2 checks are done you can connect to a pod by either an OpenShift terminal or by port forwarding the pod. The PSQL DB should have 3 log tables and 1 reports_human table which will be populated with data.