Crystal Controller Documentation

Release 1.0.1

Josep Sampé, Gerard París, Edgar Zamora-Gómez, Raúl Gracia

Oct 06, 2018

Contents

Crystal is a transparent, dynamic and open Software-Defined Storage (SDS) system for OpenStack Swift.

This documentation is generated by the Sphinx toolkit and lives in the source tree.

CHAPTER 1

Introduction

1.1 Crystal Overview

Crystal is a transparent, dynamic and open Software-Defined Storage (SDS) system for OpenStack Swift.

As depicted in the figure below, Crystal separates high level policies from the mechanisms that implement them at the data plane, to avoid hard-coding the policies in the system itself. To do so, it uses three main abstractions: *filter*, *metric*, and *controller*.

- A **filter** is a piece of programming logic that a system administrator can inject into the data plane to perform custom computations. In Crystal, this concept includes from arbitrary computations on object requests, such as compression or encryption, to resource management such as bandwidth differentiation.
- A **metric** has the role to automate the execution of filters based on the information accrued from the system. There are two types of information sources. A first type that corresponds to the real-time measurements got from the running workloads, like the number of GET operations per second of a tenant or the IO bandwidth allocated to a data container. As with filters, a fundamental feature of workload metrics is that they can be deployed at runtime. A second type of source is the metadata from the objects themselves. Such metadata is typically associated with read and write requests and includes properties like the size or type of data objects.
- The **controller** is the algorithm that manages the behavior of the data plane based on monitoring metrics. A controller may contain a very simple rule to enforce compression filter on a tenant, or it may execute a complex bandwidth differentiation algorithm requiring global visibility of the cluster.

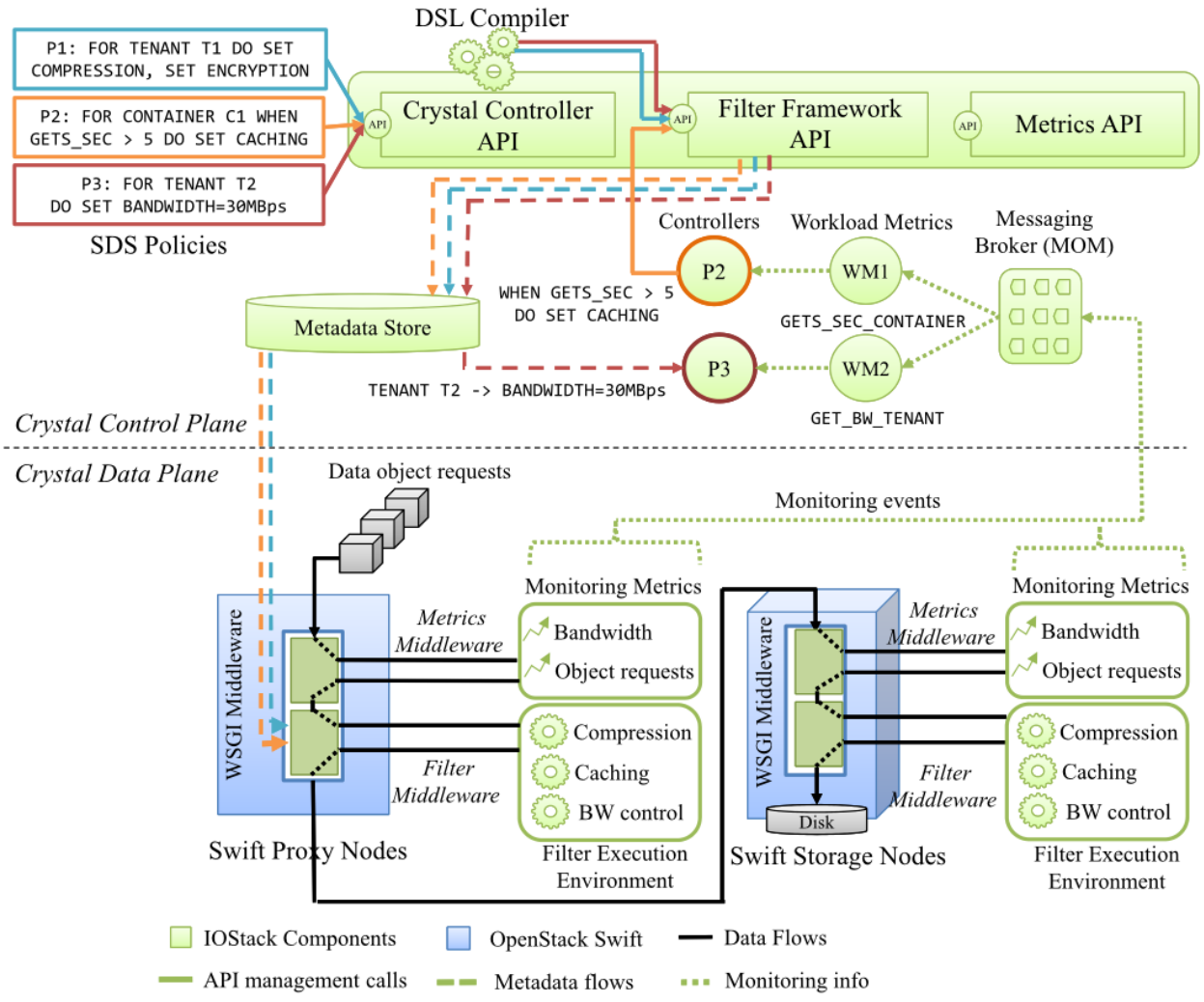


Fig. 1: Crystal Architecture

CHAPTER 2

Crystal components

2.1 Controller

The Crystal Controller is the Software-Defined-Storage (SDS) REST API in the IOStack architecture. It is a Django project that implements the REST API needed to handle filters, storlets and policies on top of Openstack Swift object-storage system. This API also includes a set of python processes who use the PyActive middleware, an Object Oriented implementation of the Actor model.

This part allows to create simple policies using a DSL (integrated in the Crystal Controller API) and to deploy them as an actor process, who analyze the system data thanks to the monitoring system, and allows to set or remove filters to tenants depending on the established policy.

2.2 Filter Middleware

The filter middleware is a Swift middleware that executes storage filters that intercept object flows to run computations or perform transformations on them.

Two differentiated kinds of filters are supported:

- Storlet filters: Java classes that implement the IStorlet interface and are able to intercept and modify the data flow of GET/PUT requests in a secure and isolated manner.
- Native filters: python classes that can intercept GET/PUT requests at all the possible life-cycle stages offered by Swift.

As depicted in the diagram above, filters can be installed both in the proxy and the object storage server. Several filters can be executed for the same request (e.g. compression+encryption). The execution order of filters can be configured and does not depend on the kind of filter: a storlet can be applied before a native filter and vice versa.

Filter classes must be registered through Crystal controller API, that also provides means to configure the filter pipeline and to control the server where they will be executed.

A convenient web dashboard is also available to simplify Crystal controller API calls.

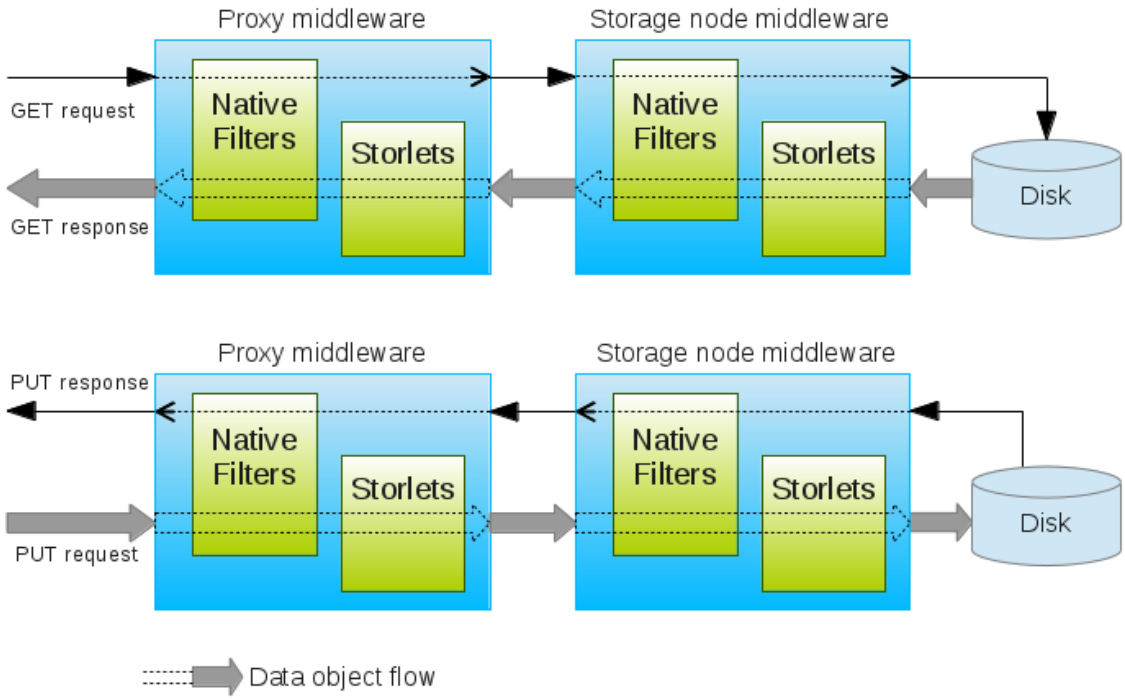


Fig. 1: Crystal filters

There is a repository that includes some filter samples for compression, encryption, caching, bandwidth differentiation, ...

2.2.1 Storlet filters

In order to use Storlet filters, it is necessary to install the storlet engine to Swift.

The code below is an example of a storlet filter:

```
public class ExampleStorlet implements IStorlet {
    /**
     * The invoke method intercepts the data flow of GET/PUT requests,␣
↪offering
     * the input and output stream to perform calculations/modifications.
     */
    @Override
    public void invoke(ArrayList<StorletInputStream> inStreams,
            ArrayList<StorletOutputStream> outStreams,
            Map<String, String> parameters,
            StorletLogger logger) throws StorletException {

        StorletInputStream sis = inStreams.get(0);
        InputStream is = sis.getStream();
        HashMap<String, String> metadata = sis.getMetadata();

        StorletObjectOutputStream sos =␣
↪(StorletObjectOutputStream)outStreams.get(0);
        OutputStream os = sos.getStream();
        sos.setMetadata(metadata);

        # The parameters map contains:
        # 1) the special parameter "reverse", that is "True" when the␣
↪filtering process
        #    should be reversed (e.g. decompression in the compression␣
↪filter)
        # 2) Other custom parameters that can be configured through the␣
↪Controller API
        String reverse = parameters.get("reverse");
        String customParam = parameters.get("custom_param");

        byte[] buffer = new byte[65536];
        int len;

        try {
            while((len=is.read(buffer)) != -1) {

                // ...
                // Filter code should be placed here to run calculations on␣
↪data
                // or perform modifications
                // ...

                os.write(buffer, 0, len);
            }
            is.close();
            os.close();
        } catch (IOException e) {
```

(continues on next page)

(continued from previous page)

```
            logger.emitLog("Example Storlet - raised IOException: " + e.
↪getMessage());
        }
    }
}
```

The *StorletInputStream* is used to stream object's data into the storlet. An instance of the class is provided whenever the Storlet gets an object as an input. Practically, it is used in all storlet invocation scenarios to stream in the object's data and metadata. To consume the data call *getStream()* to get a *java.io.InputStream* on which you can just *read()*. To consume the metadata call the *getMetadata()* method.

In all invocation scenarios the storlet is called with an instance of *StorletObjectOutputStream*. Use the *setMetadata()* method to set the Object's metadata. Use *getStream()* to get a *java.io.OutputStream* on which you can just *write()* the content of the object. Notice that *setMetadata()* must be called. Also, it must be called before writing the data.

The *StorletLogger* class supports a single method called *emitLog()*, and accepts a String.

For more information on writing and deploy Storlets, please refer to Storlets documentation.

2.2.2 Native filters

The code below is an example of a native filter:

```
class NativeFilterExample(object):

    def __init__(self, global_conf, filter_conf, logger):
        # The constructor receives the configuration parameters and the␣
↪logger
        self.logger = logger

    # This method is called by the middleware to allow filters to intercept␣
↪GET/PUT requests life-cycle
    def execute(self, req_resp, crystal_iter, request_data):
        method = request_data['method']

        if method == 'get':
            if isinstance(req_resp, Request):
                # ...
                # Filter code for GET requests should be placed here
                # ...
            elif isinstance(req_resp, Response):
                # ...
                # Filter code for GET responses should be placed here (the␣
↪response includes
                # the object data in this phase)
                # ...
        elif method == 'put':
            if isinstance(req_resp, Request):
                # ...
                # Filter code for PUT requests should be placed here (the␣
↪request includes
                # the object data in this phase)
                # ...
            elif isinstance(req_resp, Response):
                # ...
                # Filter code for PUT responses should be placed here
```

(continues on next page)

(continued from previous page)

```
                # ...

        return crystal_iter
```

The *execute()* method is called by the middleware at all life-cycle stages of the request/response. The *req_resp* parameter can be the swift.common.swob.Request or swift.common.swob.Response depending on the life-cycle phase the method is called. Upon registering the filter through Crystal controller, you can specify which server and life-cycle phase the filter will be called at, depending on the type of required computation or data-manipulation. For example, a caching filter should be executed at proxy servers, intercepting both the PUT and GET requests before reaching the object server (at request phase).

The *crystal_iter* parameter is an iterator of the data stream to be processed. The *execute()* method must return the *crystal_iter* or a modified data stream because the successive filters must receive an iterator to operate correctly, in turn.

The *request_data* parameter is a dictionary that contains the following keys:

- *'app'*: *'proxy-server'* or *'object-server'*
- *'api_version'*: the Swift API version
- *'account'*: the tenant name
- *'container'*: the container name
- *'object'*: the object name
- *'method'*: *'put'* or *'get'*

2.3 Metric Middleware

Crystal Metric Middleware is a middleware for OpenStack Swift that dynamically manages monitoring metrics.

Crystal Metric Middleware is an extension point of Crystal: new metrics can be added in order to provide more information to the controller. The github repository includes some metric samples like the number of GET/PUT operations, the number of active operations, the bandwidth used or the request performance.

The code below is an example of a simple metric that counts GET requests:

```
from crystal_metric_middleware.metrics.abstract_metric import AbstractMetric

# The metric class inherits from AbstractMetric
class GetOps(AbstractMetric):

    # This method must be implemented because is the main interception point.
    def execute(self):
        # Setting the type of the metric. The metric internal value is reset␣
↪when it is
        # published (at periodic intervals)
        self.type = 'stateless'

        # Checking if it's a GET operation and an object request
        if self.method == "GET" and self._is_object_request():
            # register_metric(key, value) increments by one the current␣
↪metric (GetOps)
            # for the current target tenant
            self.register_metric(self.account, 1)
```

(continues on next page)

(continued from previous page)

```
        return self.response
```

The code below is an example of a bandwidth metric for GET object requests:

```
class GetBw(AbstractMetric):

    def execute(self):
        self.type = 'stateless'

        if self.method == "GET" and self._is_object_request():
            # By calling _intercept_get(), all read chunks will enter on_
↪read method
            self._intercept_get()

        # If the request is intercepted with _intercept_get() it is␣
↪necessary to return the response
        return self.response

    def on_read(self, chunk):
        # In this case, the metric counts the number of bytes
        mbytes = (len(chunk)/1024.0)/1024
        self.register_metric(self.account, mbytes)
```

The next example shows a metric that counts active PUT requests:

```
class PutActiveRequests(AbstractMetric):

    def execute(self):
        # The type is stateful: the internal value is never reset
        self.type = 'stateful'

        if self.method == "PUT" and self._is_object_request():
            self._intercept_put()
            # Incrementing the metric (a new active PUT request has been␣
↪intercepted)
            self.register_metric(self.account, 1)

        return self.request

    def on_finish(self):
        # Decrementing the metric (the PUT request has just finished)
        self.register_metric(self.account, -1)
```

Metric classes (inheriting from AbstractMetric) must implement the *execute()* method and can implement the optional *on_read()* and *on_finish()* methods.

- *execute()*: This method is the main interception point. If the metric needs access to the data flow or needs to know when the request has finished, it has to call *self._intercept_get()* or *self._intercept_put()*.
- *on_read()*: this method is called if the metric has previously called *self._intercept_get()* or *self._intercept_put()*. All read chunks will enter this method.
- *on_finish()*: this method is called if the metric has previously called *self._intercept_get()* or *self._intercept_put()*. This method is called once the request has been completed.

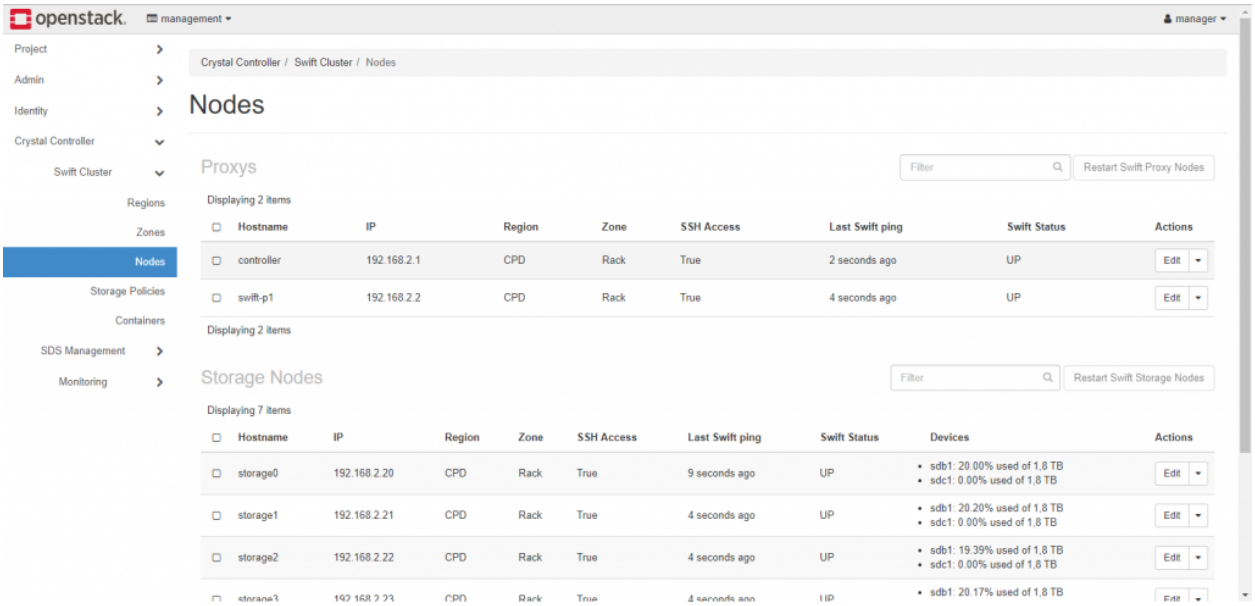
There are three types of metrics supported:

- *stateless*: the default type. When the metric is published (typically at one second intervals), the internal value is reset to 0.
- *stateful*: when the metric is published, the internal value is not reset. Thus, the value can be incremented/decremented during the next intervals. In a previous example, a stateful metric is used to count the active PUT requests: whenever a request is intercepted the metric value is incremented, and when the request finishes the metric value is decremented. At periodic intervals the metric value is published, showing how many concurrent requests are being served at a given instant.
- *force*: this type of metric is published directly after the call to register_metric instead of being published at intervals.

Metric classes must be registered and enabled through Crystal controller API. A convenient web dashboard is also available to simplify these API calls.

2.4 Dashboard

Crystal provides a user-friendly dashboard to manage policies, filters and workload metrics. The dashboard is completely integrated in the OpenStack Horizon project. Moreover, Crystal integrates advanced monitoring analysis tools for administrators in order to explore the behavior tenants, containers and even objects in the system and devise appropriate policies.



2.4.1 Demo videos

The following videos demonstrate some uses of Crystal SDS-Dashboard:

- Crystal - My first storage policy: This tutorial will teach you how to write a storage policy with Crystal and install a storage filter. We show the how this enables dynamic reconfiguration of OpenStack Swift, which can be exploited to optimize storage workloads.
- Crystal - Playing with Dynamic Storage Automation Policies: In this video we show how to use dynamic storage automation policies that are triggered by workload monitoring metrics.

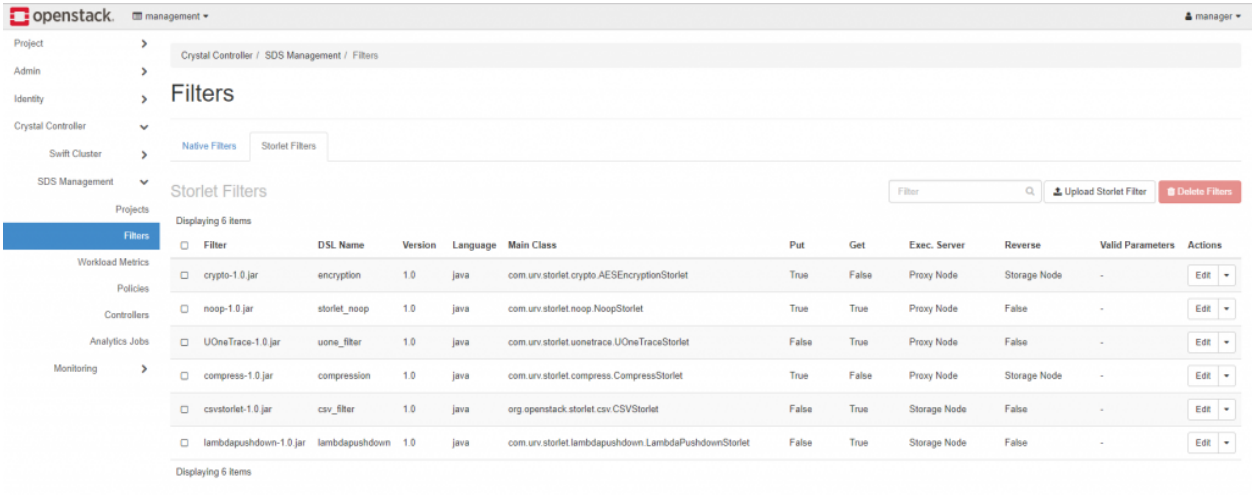


Fig. 2: SDS Administration

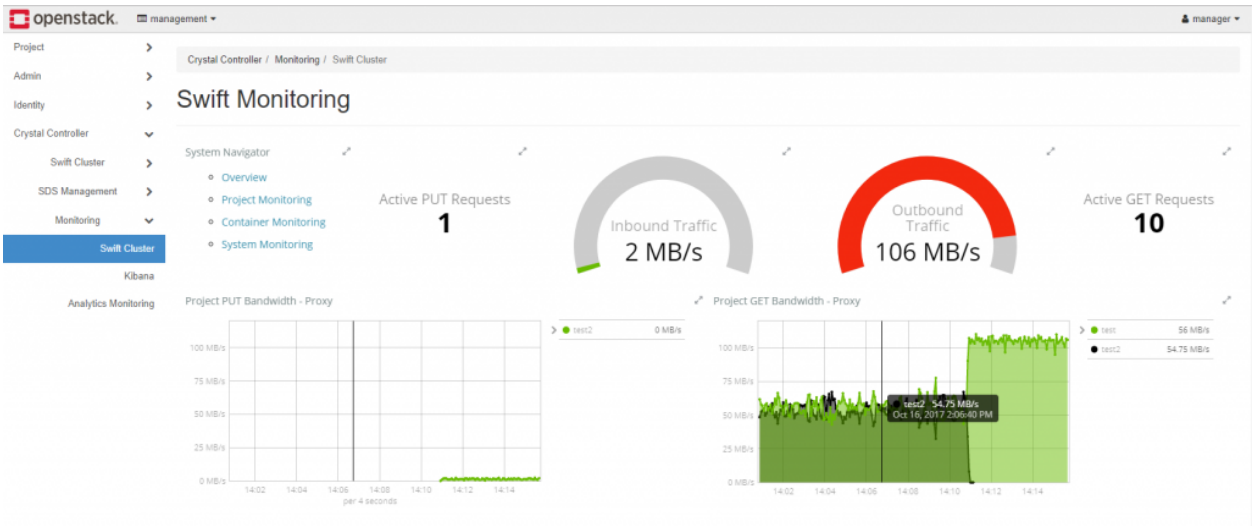


Fig. 3: Storage Monitoring

- Crystal - Multi-tenant Bandwidth Differentiation: In this video we show how Crystal can provide bandwidth differentiation in a multi-tenant OpenStack Swift deployment.

2.4.2 Source code

Dashboard source code is available in the following repository branch: SDS-Dashboard source code

CHAPTER 3

Controller API specification

3.1 Authentication & Error handling

3.1.1 Authentication

After successfully receiving the credentials from keystone, it is necessary that all API calls will be authenticated. To authenticate the calls, it needs to add the header explained in the next table:

OAuth PARAMETER	DESCRIPTION
X-Auth-Token	Admin token obtained after a successful authentication in keystone.

3.1.2 Error handling

Errors are returned using standard HTTP error code syntax.

Any additional info is included in the body of the return call, JSON-formatted.

CODE	DESCRIPTION
400	Bad input parameter. Error message should indicate which one and why.
401	Authorization required. The presented credentials, if any, were not sufficient to access the folder resource. Returned if an application attempts to use an access token after it has expired.
403	Forbidden. The requester does not have permission to access the specified resource.
404	File or folder not found at the specified path.
405	Request method not expected (generally should be GET or POST).
5xx	Server error

3.2 Controller API

3.2.1 Workload metrics

Add a workload metric

An application can registry a metric workload by issuing an HTTP POST request. The application needs to provide the metric workload metadata like json format.

Request

URL structure The URL is **/registry/metrics**

Method POST

Request Query arguments JSON input that contains a dictionary with the following keys:

FIELD	DESCRIPTION
name	The name is the keyword to be used in condition clauses of storage policy definitions. Workload metric names should be unique and self-descriptive to ease the design of storage policies.
network_location	This requires the metadata information of a workload metric to provide the network location to reach the process and obtain the computed metric.
type	Workload metric's metadata should define the type of metric produces, such as integer or a boolean, to enable the DSL syntax checker to infer if values in condition clauses belong to the appropriate type.

HTTP Request Example

```
POST /registry/metrics

{
"name": "put_active_requests",
"network_location": "tcp://127.0.0.1:6899/registry.policies.metrics.swift_
↪metric/SwiftMetric/put_active_requests",
"type": "integer"
}
```

Response

Response example

```
Response <201>
TODO
```

Get all workload metrics

An application can get all the metrics registered by issuing an HTTP GET request.

Request

URL structure The URL is **/registry/metrics**

Method GET

HTTP Request Example

```
GET /registry/metrics
```

Response

Response example

```
Response <201>
[{
"name": "put_active_requests",
"network_location": "tcp://127.0.0.1:6899/registry.policies.metrics.swift_
↪metric/SwiftMetric/put_active_requests",
"type": "integer"
},{
"name": "get_active_requests",
"network_location": "tcp://127.0.0.1:6899/registry.policies.metrics.swift_
↪metric/SwiftMetric/get_active_requests",
"type": "integer"
}]
```

Update a workload metric

An application can update the metadata of a workload metric by issuing an HTTP PUT request.

Request

URL structure The URL is **/registry/metrics/{metric_name}**

Method PUT

HTTP Request Example

```
PUT /registry/metrics/put_active_requests
{
  "network_location": "tcp://192.168.1.5:6899/registry.policies.metrics.
↪swift_metric/SwiftMetric/put_active_requests"
}
```

Response

Response example

```
HTTP/1.1 201 CREATED
```

Get metric metadata

An application can ask for a workload metric metadata by issuing an HTTP GET request.

Request

URL structure The URL is **/registry/metrics/{metric_name}**

Method GET

HTTP Request Example

```
GET /registry/metrics/put_active_requests
```

Response

Response example

```
HTTP/1.1 200 OK

{
"name": "put_active_requests",
"network_location": "tcp://127.0.0.1:6899/registry.policies.metrics.swift_
↪metric/SwiftMetric/put_active_requests",
"type": "integer"
}
```

Delete a workload metric

An application can delete a workload metric by issuing an HTTP DELETE request.

Request

URL structure The URL is **/registry/metrics/{metric_name}**

Method DELETE

HTTP Request Example

```
DELETE /registry/metrics/put_active_requests
```

Response

Response example

```
HTTP/1.1 204 NO CONTENT
```

3.2.2 Filters

Register a filter

An application can register a filter by issuing an HTTP POST request. The application needs to provide the filter metadata in json format.

Request

URL structure The URL is **/registry/filters**

Method POST

Request Query arguments JSON input that contains a dictionary with the following keys:

FIELD	DESCRIPTION
name	Filter names should be unique and self-descriptive to ease the design of storage policies.
identifier	The identifier of the previously uploaded filter.
activation_url	Different filter types may have distinct calls from the SDS Controller API viewpoint, we need to provide the base URL to be used to trigger the filter activation.
valid_parameters	Dictionary where the keys are the parameters accepted by the filter, and the values are the type (i.e. boolean, integer) of each parameter.

HTTP Request Example

```
POST /registry/filters

{
"name":"compression",
"identifier":2,
"activation_url":"http://sds_controller/filters/1",
"valid_parameters":{"param1":"bool", "param2":"integer"}
}
```

Response

Response example

```
HTTP/1.1 201 CREATED
```

Get all registered filters

An application can get all registered filters by issuing an HTTP PUT request.

Request

URL structure The URL is **/registry/filters**

Method GET

HTTP Request Example

```
GET /registry/filters
```

Response

Response example

```
HTTP/1.1 200 OK

[{
  "name":"compression",
  "identifier":2,
  "activation_url":"http://sds_controller/filters/1",
  "valid_parameters":{"param1":"bool", "param2":"integer"}
},{
  "name":"compression_gzip",
  "identifier":2,
  "activation_url":"http://sds_controller/filters/1",
  "valid_parameters":{"param1":"bool", "param2":"integer"}
}]
```

Update a registered filter

An application can update the metadata of a registered filter by issuing an HTTP PUT request.

Request

URL structure The URL is **/registry/filters/{filter_name}**

Method PUT

HTTP Request Example

```
PUT /registry/filters/compression

{
"activation_url":"http://sds_controller/filters/2"
}
```

Response

Response example

```
HTTP/1.1 201 CREATED
```

Get registered filter metadata

An application can ask for a filter metadata by issuing an HTTP GET request.

Request

URL structure The URL is **/registry/filters/{filter_name}**

Method GET

HTTP Request Example

```
GET /registry/filters/compression
```

Response

Response example

```
HTTP/1.1 200 OK

{
  "name":"compression",
  "identifier":2,
  "activation_url":"http://sds_controller/filters/2",
  "valid_parameters":{"param1":"bool", "param2":"integer"}
}
```

Delete a registered filter

An application can delete a registered filter by issuing an HTTP DELETE request.

Request

URL structure The URL is **/registry/filters/{filter_name}**

Method DELETE

HTTP Request Example

```
DELETE /registry/filters/compress
```

Response

Response example

```
HTTP/1.1 204 NO CONTENT
```

3.2.3 Projects group

Add a projects group

An application can registry a projects group by issuing an HTTP POST request. The application needs to provide the project identifiers in a json array.

Request

URL structure The URL is **/registry/gtenants**

Method POST

Request Query arguments JSON input that contains an array of project identifiers.

HTTP Request Example

```
POST /registry/gtenants

[
"111456789abcdef",
"222456789abcdef",
"333456789abcdef",
]
```

Response

Response example

```
HTTP/1.1 201 CREATED
```

Get all projects groups

An application can get all projects groups registered by issuing an HTTP GET request.

Request

URL structure The URL is **/registry/gtenants**

Method GET

HTTP Request Example

```
GET /registry/gtenants
```

Response

Response example

```
HTTP/1.1 200 OK
{
  "2": [
    "000456789abcdef",
    "888456789abcdef",
    "999456789abcdef"
  ],
  "3": [
    "111456789abcdef",
    "222456789abcdef",
    "333456789abcdef"
```

(continues on next page)

(continued from previous page)

```
  ]
}
```

Get projects of a group

An application can get all tenants of a group registered by issuing an HTTP GET request.

Request

URL structure The URL is **/registry/gtenants/{gtenant_id}**

Method GET

HTTP Request Example

```
GET /registry/gtenants/3
```

Response

Response example

```
Response <201>
[
    "111456789abcdef",
    "222456789abcdef",
    "333456789abcdef"
]
```

Update members of a projects group

An application can modify the members of a group by issuing an HTTP PUT request.

Request

URL structure The URL is **/registry/gtenants/{gtenant_id}**

Method PUT

HTTP Request Example

```
PUT /registry/gtenants/2

[
"111456789abcdef",
"222456789abcdef"
]
```

Response

Response example

```
HTTP/1.1 201 CREATED
```

Delete a projects group

An application can delete a projects group by issuing an HTTP DELETE request.

Request

URL structure The URL is **/registry/gtenants/{gtenant_id}**

Method DELETE

HTTP Request Example

```
DELETE /registry/gtenants/2
```

Response

Response example

```
HTTP/1.1 204 NO CONTENT
```

Delete a member of a projects group

An application can delete a member of a projects group by issuing an HTTP DELETE request.

Request

URL structure The URL is **/registry/gtenants/{gtenant_id}/tenants/{project_id}**

Method DELETE

HTTP Request Example

```
DELETE /registry/gtenants/2/tenants/111456789abcdef
```

Response

Response example

```
HTTP/1.1 204 NO CONTENT
```

3.2.4 Object type

Create an object type

An application can registry an object type by issuing an HTTP POST request. The application needs to provide the json dictionary with the name of the object type and the file extensions.

Request

URL structure The URL is **/registry/object_type**

Method POST

Request Query arguments JSON input that contains a dictionary with the following keys:

FIELD	DESCRIPTION
name	The name of the object type.
types_list	An array of file extensions.

HTTP Request Example

```
POST /registry/object_type

{
"name": "DOCS",
"types_list": ["doc","docx","xls","txt"]
}
```

Response

Response example

```
HTTP/1.1 201 CREATED
```

Get all object types

An application can obtain all registered object types by issuing an HTTP GET request.

Request

URL structure The URL is **/registry/object_type**

Method GET

HTTP Request Example

```
GET /registry/object_type
```

Response

Response example

```
HTTP/1.1 200 OK

[
{
"name": "DOCS",
"types_list": ["doc","docx","xls","txt"]
},
{
"name": "PICS",
"types_list": ["jpg","jpeg","png","gif"]
}
]
```

Get extensions of an object type

An application can obtain the extensions list of a particular object type by issuing an HTTP GET request.

Request

URL structure The URL is **/registry/object_type/{object_type_name}**

Method GET

HTTP Request Example

```
GET /registry/object_type/PICS
```

Response

Response example

```
HTTP/1.1 200 OK

{
"name": "PICS",
"types_list": ["jpg","jpeg","png","gif"]
}
```

Update extensions of an object type

An application can update an object type by issuing an HTTP PUT request.

Request

URL structure The URL is **/registry/object_type/{object_type_name}**

Method PUT

Request Query arguments JSON input that contains an array of file extensions.

HTTP Request Example

```
PUT /registry/object_type/PICS

["jpg","jpeg","png","gif","bmp"]
```

Response

Response example

```
HTTP/1.1 201 CREATED
```

Delete an object type

An application can delete an object type by issuing an HTTP DELETE request.

Request

URL structure The URL is **/registry/object_type/{object_type_name}**

Method DELETE

HTTP Request Example

```
DELETE /registry/object_type/PICS

["jpg","jpeg","png","gif","bmp"]
```

Response

Response example

```
HTTP/1.1 200 OK
```

3.2.5 Metric modules

Upload a metric module

An application can upload a metric module by issuing an HTTP POST request. The application needs to provide the metric module data like a QueryDict with a key 'file' containing the upload file and a key 'metadata' containing a JSON object with metric module metadata. **media_type:** *multipart/form-data*

Request

URL structure The URL that represents the metric module data resource. The URL is

/registry/metric_module/data

Method POST

Request Headers

The request header includes the following information:

FIELD	DESCRIPTION
X-Auth-Token	Token to authenticate to OpenStack Swift as an **Admin**
enctype	The content type and character encoding of the response. The content type must be **multipart/form-data**.

The **metadata** parameter is a JSON object with the following fields:

FIELD	DESCRIPTION
class_name	The main class of the metric module to be created.
in_flow	Boolean indicating whether the metric applies to input flow.
out_flow	Boolean indicating whether the metric applies to output flow.
execution_server	'object' or 'proxy' depending on the server the metric should run on.
enabled	Boolean indicating whether the metric module should be enabled or not.

HTTP Request Example

```
POST /registry/metric_module/data

"media_type":"multipart/form-data"
file=<file get_active_requests.py>
metadata={
"class_name": "GetActiveRequests",
"in_flow": True,
"out_flow": False,
"execution_server": "proxy",
"enabled": True
}
```

Response

Response example

```
HTTP/1.1 201 CREATED

{
"id": 1,
"metric_name": "get_active_requests",
"class_name": "GetActiveRequests",
"in_flow": True,
"out_flow": False,
"execution_server": "proxy",
"enabled": True
}
```

Get all metrics modules

An application can get all metric modules by issuing an HTTP GET request.

Request

URL structure The URL that represents the metric module data resource. The URL is **/registry/metric_module**

Method GET

HTTP Request Example

```
GET /registry/metric_module
```

Response

Response example

```
HTTP/1.1 200 OK

[
{
"id": 1,
"metric_name": "get_active_requests",
"class_name": "GetActiveRequests",
"in_flow": True,
"out_flow": False,
"execution_server": "proxy",
"enabled": True
},
{
"id": 2,
"metric_name": "get_bw",
"class_name": "GetBw",
"in_flow": True,
"out_flow": False,
"execution_server": "proxy",
"enabled": True
}
]
```

Get a metric module

An application can get a metric module info by issuing an HTTP GET request.

Request

URL structure The URL that represents the metric module data resource. The URL is **/registry/metric_module/{metric_module_id}**

Method GET

HTTP Request Example

```
GET /registry/metric_module/1
```

Response

Response example

```
HTTP/1.1 200 OK

{
"id": 1,
"metric_name": "get_active_requests",
"class_name": "GetActiveRequests",
"in_flow": True,
"out_flow": False,
"execution_server": "proxy",
"enabled": True
}
```

Update a metric module

An application can update a metric module metadata by issuing an HTTP PUT request.

Request

URL structure The URL that represents the metric module data resource. The URL is **/registry/metric_module/{metric_module_id}**

Method PUT

HTTP Request Example

```
PUT /registry/metric_module/1

{
"class_name": "GetActiveRequests",
"in_flow": True,
"out_flow": False,
"execution_server": "proxy",
"enabled": False
}
```

Response

Response example

```
HTTP/1.1 200 OK
```

Delete a metric module

An application can delete a metric module metadata by issuing an HTTP DELETE request.

Request

URL structure The URL that represents the metric module data resource. The URL is **/registry/metric_module/{metric_module_id}**

Method DELETE

HTTP Request Example

```
DELETE /registry/metric_module/1
```

Response

Response example

```
HTTP/1.1 204 NO CONTENT
```

3.2.6 DSL Policies

List all static policies

An application can get all static policies sorted by execution order by issuing an HTTP GET request.

Request

URL structure The URL that represents the static policy resource. The URL is **/registry/static_policy**

Method GET

HTTP Request Example

```
GET /registry/static_policy
```

Response

Response example

```
HTTP/1.1 200 OK

[
{
"id": "1",
"target_id": "1234567890abcdef",
"target_name": "tenantA",
"filter_name": "compression-1.0.jar",
"object_type": "",
"object_size": "",
"execution_server": "proxy",
"execution_server_reverse": "proxy",
"execution_order": "1",
"params": ""
},
```

(continues on next page)

(continued from previous page)

```
{
"id": "2",
"target_id": "1234567890abcdef",
"target_name": "tenantA",
"filter_name": "encryption-1.0.jar",
"object_type": "",
"object_size": "",
"execution_server": "proxy",
"execution_server_reverse": "proxy",
"execution_order": "2",
"params": ""
},
]
```

Add a static policy

An application can add a new static policy by issuing an HTTP POST request.

Request

URL structure The URL that represents the static policy resource. The URL is **/registry/static_policy**

Method POST /registry/static_policy

Request Body The request body is a text/plain input with one or various DSL rules separated by newlines. Refer to *Crystal DSL Grammar* for a detailed explanation of Crystal DSL.

HTTP Request Example

```
Content-Type: text/plain
POST /registry/static_policy

FOR TENANT:1234567890abcdef DO SET compression
```

Response

Response example

```
HTTP/1.1 201 CREATED
```

Get a static policy

An application can get all static policies sorted by execution order by issuing an HTTP GET request.

Request

URL structure The URL that represents the static policy resource. The URL is **/registry/static_policy/{project_id}:{policy_id}**

Method GET

HTTP Request Example

```
GET /registry/static_policy/1234567890abcdef:1
```

Response

Response example

```
HTTP/1.1 200 OK

{
"id": "1",
"target_id": "1234567890abcdef",
"target_name": "tenantA",
"filter_name": "compression-1.0.jar",
"object_type": "",
"object_size": "",
"execution_server": "proxy",
"execution_server_reverse": "proxy",
"execution_order": "1",
"params": ""
}
```

Update a static policy

An application can update the static policy metadata by issuing an HTTP PUT request.

Request

URL structure The URL that represents the static policy resource. The URL is **/registry/static_policy/{project_id}:{policy_id}**

Method PUT

HTTP Request Example

In the following example, a put request is issued to change the execution server of the policy to object server:

```
PUT /registry/static_policy/1234567890abcdef:1

{
"execution_server": "object",
"execution_server_reverse": "object"
}
```

Response

Response example

```
HTTP/1.1 201 CREATED
```

Delete a static policy

An application can delete a static policy by issuing an HTTP DELETE request.

Request

URL structure The URL that represents the static policy resource. The URL is **/registry/static_policy/{project_id}:{policy_id}**

Method DELETE

HTTP Request Example

```
DELETE /registry/static_policy/1234567890abcdef:1
```

Response

Response example

```
HTTP/1.1 204 NO CONTENT
```

List all dynamic policies

An application can get all dynamic policies by issuing an HTTP GET request.

Request

URL structure The URL that represents the dynamic policy resource. The URL is **/registry/dynamic_policy**

Method GET

HTTP Request Example

```
GET /registry/dynamic_policy
```

Response

Response example

```
HTTP/1.1 200 OK

[
{
"id": "3",
"policy": "FOR TENANT:d70b71fc4c02466bb97544bd2c7c0932 DO SET compression",
"condition": "put_ops<3",
"transient": True,
"policy_location": "tcp://127.0.0.1:6899/registry.policies.rules.rule_
↪transient/TransientRule/policy:3",
"alive": True
},
```

(continues on next page)

(continued from previous page)

```
{
...
}
]
```

Add a dynamic policy

An application can add a new dynamic policy by issuing an HTTP POST request.

Request

URL structure The URL that represents the dynamic policy resource. The URL is **/registry/dynamic_policy**

Method POST /registry/dynamic_policy

Request Body The request body is a text/plain input with one or various DSL rules separated by newlines. Refer to [Crystal DSL Grammar](/doc/api_dsl.md) for a detailed explanation of Crystal DSL.

HTTP Request Example

```
Content-Type: text/plain
POST /registry/dynamic_policy

FOR TENANT:1234567890abcdef WHEN put_ops < 3  DO SET compression
```

Response

Response example

```
HTTP/1.1 201 CREATED
```

Delete a dynamic policy

An application can delete a dynamic policy by issuing an HTTP DELETE request.

Request

URL structure The URL that represents the dynamic policy resource. The URL is **/registry/dynamic_policy/{policy_id}**

Method DELETE

HTTP Request Example

```
DELETE /registry/dynamic_policy/3
```

Response

Response example

```
HTTP/1.1 204 NO CONTENT
```

3.3 Filters API

3.3.1 Filters

List Filters

An application can list the Filters by issuing an HTTP GET request.

Request

URL structure The URL that represents the filter data resource. The URL is **/filters**

Method GET

HTTP Request Example

```
GET /filters
```

Response

Response example

```
HTTP/1.1 200 OK

[{
"id":"1",
"filter_name": "compression-1.0.jar",
"filter_type":"storlet",
"main":"com.example.StorletCompression",
"is_pre_put":"False",
"is_post_put":"False",
"is_pre_get":"False",
"is_post_get":"False",
"execution_server":"proxy",
"execution_server_reverse":"proxy",
"interface_version":"1.0",
"object_metadata":"",
"dependencies":"",
"has_reverse":"False",
},
{
...
}]
```

Create a filter

An application can create a filter by issuing an HTTP POST request. The application needs to provide the filter metadata in json format. The binary file will be uploaded after this call, first it must upload the metadata fields.

Request

URL structure The URL that represents the filter data resource. The URL is **/filters**

Method POST

Request Query arguments JSON input that contains a dictionary with the following keys:

FIELD	DESCRIPTION
filter_type	The type of filter. Supported values: "storlet", "native" or "global"
interface_version	Currently we have a single version "1.0"
dependencies	A comma separated list of dependencies. Default: " "
object_metadata	Currently, not in use, but must appear. Use an empty value ""
main	The name of the class that implements the IStorlet API.
is_pre_put	Boolean to indicate that the filter will be executed before put requests reach the storage.
is_post_put	Boolean to indicate that the filter will be executed after put requests reach the storage.
is_pre_get	Boolean to indicate that the filter will be executed before get requests reach the storage.
is_post_get	Boolean to indicate that the filter will be executed after get requests reach the storage.
has_reverse	Boolean to indicate whether the filter has a reverse action, like compression/decompression.
execution_server	The execution server for this filter: "proxy" or "object"
execution_server_reverse	The reverse execution server for this filter: "proxy" or "object"
execution_order	This parameter can only be sent if filter_type is "global". An integer indicating the execution order of global filters.
enabled	This parameter can only be sent if filter_type is "global". A boolean to indicate if the filter is enabled.

HTTP Request Example

```
POST /filters

{
"filter_type": "storlet",
"interface_version": "1.0",
"dependencies": "",
"object_metadata": "",
"main": "com.example.StorletMain",
"is_pre_put": "False",
"is_post_put": "False",
"is_pre_get": "False",
"is_post_get": "False",
"has_reverse": "False",
"execution_server": "proxy",
"execution_server_reverse": "proxy"
}
```

Response

Response example

```
Response <201>
{
"id":1345,
"filter_type": "storlet",
"interface_version": "1.0",
"dependencies": "",
"object_metadata": "",
"main": "com.example.StorletMain",
"is_pre_put": "False",
"is_post_put": "False",
"is_pre_get": "False",
"is_post_get": "False",
"has_reverse": "False",
"execution_server": "proxy",
"execution_server_reverse": "proxy"
}
```

Upload a filter data

An application can upload a filter data by issuing an HTTP PUT request. The application needs to provide the filter data like a QueryDicct with a single key "file" containing the upload file. **media_type:** *multipart/form-data*

Request

URL structure The URL that represents the filter data resource. The URL is **/filters/{filter_id}/data**.

Method PUT

Request Headers The request header includes the following information:

FIELD	DESCRIPTION
X-Auth-Token	Token to authenticate to OpenStack Swift as an **Admin**
enctype	The content type and character encoding of the response. The content type must be **multipart/form-data**.

HTTP Request Example

```
PUT /filters/1345/data
"media_type":"multipart/form-data"
{"file":<binary file>} (QueryDict)
```

Delete a filter

An application can delete a filter by issuing an HTTP DELETE request. This call deletes the filter from SDS Controller store, **BUT** it does not undeploy this filter from OpenStack Swift. Therefore, some users could maintain activated this filter in their account after delete the filter from SDS Controller.

Request

URL structure The URL that represents the filter data resource. The URL is **/filters/{filter_id}**

Method DELETE

HTTP Request Example

```
DELETE /filters/1345
```

Get filter metadata

An application can ask for the filter metadata by issuing an HTTP GET request.

Request

URL structure The URL that represents the filter data resource. The URL is **/filters/{filter_id}**

Method GET

HTTP Request Example

```
GET /filters/1
```

Response

Response example

```
HTTP/1.1 200 OK

{
"id":"1",
"filter_name": "compression-1.0.jar",
"filter_type":"storlet",
"main":"com.example.StorletCompression",
"is_pre_put":"False",
"is_post_put":"False",
"is_pre_get":"False",
"is_post_get":"False",
"execution_server":"proxy",
"execution_server_reverse":"proxy",
"interface_version":"1.0",
"object_metadata":"",
"dependencies":"",
"has_reverse":"False",
}
```

Update filter metadata

An application can update the filter metadata by issuing an HTTP PUT request.

Request

URL structure The URL that represents the filter data resource. The URL is **/filters/{filter_id}**

Method PUT

Request Body JSON input that contains a dictionary with the following keys:

FIELD	DESCRIPTION
interface_version	Currently we have a single version "1.0"
dependencies	A comma separated list of dependencies. Default: " "
object_metadata	Currently, not in use, but must appear. Use an empty value ""
main	The name of the class that implements the IStorlet API.
is_pre_put	Boolean to indicate that the filter will be executed before put requests reach the storage.
is_post_put	Boolean to indicate that the filter will be executed after put requests reach the storage.
is_pre_get	Boolean to indicate that the filter will be executed before get requests reach the storage.
is_post_get	Boolean to indicate that the filter will be executed after get requests reach the storage.
has_reverse	Boolean to indicate whether the filter has a reverse action, like compression/decompression.
execution_server	The execution server for this filter: "proxy" or "object"
execution_server_reverse	The reverse execution server for this filter: "proxy" or "object"
execution_order	This parameter can only be sent if filter_type is "global". An integer indicating the execution order of global filters.
enabled	This parameter can only be sent if filter_type is "global". A boolean to indicate if the filter is enabled.

HTTP Request Example

```
PUT filters/1

{
"filter_type": "storlet",
"interface_version": "1.0",
"dependencies": "",
"object_metadata": "",
"main": "com.example.StorletMain",
"is_pre_put": "False",
"is_post_put": "False",
"is_pre_get": "False",
"is_post_get": "False",
"has_reverse": "False",
"execution_server": "proxy",
"execution_server_reverse": "proxy"
}
```

Response

Response example

```
HTTP/1.1 200 OK
```

Deploy a filter

An application can deploy a filter to Swift by issuing an HTTP PUT request. This operation creates a static policy

Request

URL structure The URL that represents the deployment of a filter to a project. The URL is **/filters/{project}/deploy/{filter_id}/**

Similarly, to deploy a filter to a project and a container, the URL is **/filters/{project}/{container}/deploy/{filter_id}/**

Method PUT

Request Body JSON input that contains a dictionary with the following keys:

FIELD	DESCRIPTION
params	The parameters needed by the filter execution. These parameters are codified as query string.
object_type	String. The type of objects the filter will be applied to.
object_size	String. The size of objects the filter will be applied to.

HTTP Request Example

```
Content-Type: application/json
PUT /filters/4f0279da74ef4584a29dc72c835fe2c9/deploy/3

{
"params":"select=user_id, type=string",
"object_type":"DOCS",
"object_size":">2000",
}
```

Response

The response is the id of the new static policy associated with the filter deployment.

Response example

```
HTTP/1.1 201 Created

1
```

Undeploy a Filter

An application can undeploy the filter from a Swift project by issuing an HTTP PUT request.

Request

URL structure The URL that represents the filter data resource. The URL is **/filters/{project}/undeploy/{filter_id}/**

Method PUT

HTTP Request Example

```
Content-Type: application/json
PUT /filters/4f0279da74ef4584a29dc72c835fe2c9/undeploy/3
```

3.3.2 Dependencies

Create a Dependency

An application can create a Dependency by issuing an HTTP POST request. The application needs to provide the Dependency metadata like json format. The binary file will be uploaded after this call, first it must upload the metadata fields.

Request

URL structure The URL that represents the filter dependencies resource. The URL is **/filters/dependencies.**

Method POST

Request Body JSON input that contains a dictionary with the following keys:

FIELD	DESCRIPTION
name	The name of the dependency to be created. It is a unique field.
version	While the engine currently does not parse this header, it must appear.
permissions	An optional metadata field, where the user can state the permissions given to the dependency when it is copied to the Linux container. This is helpful for binary dependencies invoked by the filter. For a binary dependency once can specify: "0755"

HTTP Request Example

```
POST /filters/dependencies

{
"name":"DependencyName",
"version":"1.0",
"permissions":"0755"
}
```

Response

Response example

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Content-Length: 248

{
"id":1345,
"name":"DependencyName",
"version":"1.0",
"permissions":"0755"
}
```

Upload a Dependency Data

An application can upload a Dependency data by issuing an HTTP PUT request. The application needs to provide the dependency data like a QueryDict with a single key "file" containing the upload file. **media_type:** *multipart/form-data*

Request

URL structure The URL that represents a filter dependency resource. The URL is **/filters/dependencies/{dependency_id}/data**

Method PUT

HTTP Request Example

```
PUT /filters/dependencies/:dependency_id/data
"media_type":"multipart/form-data"

{"file":<binary file>} (QueryDicct)
```

Response

Response example

```
HTTP/1.1 200 OK
```

Delete a Dependency

An application can delete a Dependency by issuing an HTTP DELETE request. This call delete the Dependency from Swift and SDS Controller.

Request

URL structure The URL that represents a filter dependency resource. The URL is **/filters/dependencies/{dependency_id}**

Method DELETE

HTTP Request Example

```
DELETE /filters/dependencies/:dependency_id
```

Get Dependency metadata

An application can ask for the Dependency metadata by issuing an HTTP GET request.

Request

URL structure The URL that represents a filter dependency resource. The URL is **/filters/dependencies/{dependency_id}**

Method GET

HTTP Request Example

```
GET /filters/dependencies/:dependency_id
Content-Type: application/json; charset=UTF-8
```

Response

Response example

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8

{
"id":1345,
"name":"DependencyName",
"version":"1.0",
"permissions":"0755"
}
```

List Dependencies

An application can list the Dependencies by issuing an HTTP GET request.

Request

URL structure The URL that represents filter dependencies resource. The URL is **/filters/dependencies**

Method GET

HTTP Request Example

```
Content-Type: application/json; charset=UTF-8
GET /filters
```

Response

Response example

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Content-Length: 248

[
{
"id":1345,
"name":"DependencyName",
"version":"1.0",
"permissions":"0755"
},{
"id":1345,
"name":"DependencyName",
"version":"1.0",
```

(continues on next page)

(continued from previous page)

```
"permissions":"0755"
},{
"id":1345,
"name":"DependencyName",
"version":"1.0",
"permissions":"0755"
}
]
```

Update Dependency metadata

An application can update the Dependency metadata by issuing an HTTP PUT request.

Request

URL structure The URL that represents a filter dependency resource. The URL is **/filters/dependencies/{dependency_id}**

Method PUT

Request Query arguments JSON input that contains a dictionary with the following keys:

FIELD	DESCRIPTION
name	The name of the dependency to be created. It is a unique field.
version	While the engine currently does not parse this header, it must appear.
permissions	An optional metadata field, where the user can state the permissions given to the dependency when it is copied to the Linux container. This is helpful for binary dependencies invoked by the filter. For a binary dependency once can specify: “0755”

HTTP Request Example

```
PUT /filters/dependencies/123
{
"name":"DependencyName",
"version":"1.0",
"permissions":"0755"
}
```

Response

Response example

```
HTTP/1.1 200 OK
Content-Type: application/json; charset=UTF-8
Content-Length: 248

{
"id":1345,
"name":"DependencyName",
"version":"1.0",
```

(continues on next page)

(continued from previous page)

```
"permissions":"0755"
}
```

Deploy Dependency

An application can deploy a Dependency to an account to Swift by issuing an HTTP PUT request.

Request

URL structure The URL that represents the a filter dependency resource. The URL is **/filters/dependencies/{project}/deploy/{dependency_id}/**

Method PUT

HTTP Request Example

```
PUT /filters/dependencies/4f0279da74ef4584a29dc72c835fe2c9/deploy/3
```

Undeploy Dependency

An application can undeploy the Dependency of an account from Swift by issuing an HTTP PUT request.

Request

URL structure The URL that represents the filter dependency data resource. The URL is **/filters/dependencies/{project}/undeploy/{dependency_id}/**

Method PUT

HTTP Request Example

```
PUT /filters/dependencies/4f0279da74ef4584a29dc72c835fe2c9/undeploy/3
```

List deployed Dependencies of an Account

An application can list all the deployed Dependencies of an account to Swift by issuing an HTTP GET request.

Request

URL structure The URL that represents the filter dependencies data resource. The URL is **/filters/dependencies/{project}/deploy/**

Method GET

HTTP Request Example

```
GET /filters/dependencies/123/deploy/
```

3.3.3 SLO info

Get all SLOs

An application can return all the SLO information about all projects by issuing an HTTP GET request.

Request

URL structure The URL that represents the SLO information about all projects. The URL is **/filters/slos/**

Method GET

HTTP Request Example

```
GET /filters/slos/
```

Response

Response example

```
HTTP/1.1 200 OK

[
{"dsl_filter": "bandwidth",
"slo_name": "get_bw",
"target": "AUTH_0123456789abcdef#0",
"value": "20"},
{"dsl_filter":"bandwidth",
"slo_name":"put_bw",
"target":"AUTH_0123456789abcdef#0",
"value":"24"}
]
```

Get a SLO

An application can return the SLO information by issuing an HTTP GET request.

Request

URL structure The URL that represents the SLO information for a filter, slo_name and target. The URL is **/filters/slo/{dsl_filter_keyword}/{slo_name}/{target}**

Method GET

HTTP Request Example

```
GET /filters/slo/bandwidth/get_bw/AUTH_0123456789abcdef%230
```

Response

Response example

```
HTTP/1.1 200 OK

{
"dsl_filter":"bandwidth",
"slo_name":"get_bw",
"target":"AUTH_0123456789abcdef#0",
"value":"20"
}
```

Create a SLO

An application can create a SLO for a filter, slo_name and target by issuing an HTTP POST request.

Request

URL structure The URL to create a SLO for the selected filter, slo_name and target is **/filters/slos** with a body containing a JSON object.

Method POST

HTTP Request Example

```
PUT /filters/slos/
{
"dsl_filter": "bandwidth",
"slo_name": "get_bw",
"target": "AUTH_0123456789abcdef#0",
"value": "80"
}
```

Response

Response example

```
HTTP/1.1 201 CREATED
```

Edit a SLO

An application can modify the assigned value for a filter, slo_name and target by issuing an HTTP PUT request.

Request

URL structure The URL that represents the SLO information to edit. The URL is **/filters/slo/{dsl_filter_keyword}/{slo_name}/{target}** with a body containing a JSON object.

Method PUT

HTTP Request Example

```
PUT /filters/slo/bandwidth/get_bw/AUTH_0123456789abcdef%230
{
"value": "100"
}
```

Response

Response example

```
HTTP/1.1 201 CREATED
```

Delete a SLO

An application can delete an SLO by issuing an HTTP DELETE request.

Request

URL structure The URL that represents the SLO information to delete. The URL is **/filters/slo/{dsl_filter_keyword}/{slo_name}/{target}**

Method DELETE

HTTP Request Example

```
DELETE /filters/slo/bandwidth/get_bw/AUTH_0123456789abcdef%230
```

Response

Response example

```
HTTP/1.1 204 NO CONTENT
```

3.4 Swift API

3.4.1 Enable SDS for a project

An application can enable Crystal-SDS for a particular project by issuing an HTTP POST request.

Request

URL structure The URL that represents the tenant resource is **/swift/tenants**

Method POST

Request Query arguments JSON input that contains a dictionary with the following keys:

FIELD	DESCRIPTION
tenant_name	The project name

HTTP Request Example

```
POST /swift/tenants

{
"tenant_name" : "tenantA"
}
```

Response

Response example

```
201 CREATED
```

3.4.2 Get a storage policies list

An application can retrieve a storage policies list by issuing a HTTP GET request.

Request

URL structure The URL that represents the storage policies resource is **/swift/storage_policies**

Method GET

HTTP Request Example

```
GET /swift/storage_policies
```

Response

Response example

```
200 OK

[
{"default":"no","name":"s5y6","policy_type":"replication","id":"4"},
{"default":"yes","name":"allnodes","policy_type":"replication","id":"0"},
{"default":"no","name":"s0y1","policy_type":"replication","id":"2"},
{"default":"no","name":"storage4","policy_type":"replication","id":"1"},
{"default":"no","name":"s3y4","policy_type":"replication","id":"3"}
]
```

3.4.3 Create a new storage policy

An application can create a new ring & storage policy by issuing an HTTP POST request.

Request

URL structure The URL that represents the storage policies resource is **/swift/spolicies**

Method POST

Request Query arguments JSON input that contains a dictionary with the following keys:

FIELD	DESCRIPTION
storage_node	A dictionary of location keys and weight values. Location key example: r1z1-{storage_node_management_interface_ip_address}:6000/{device_name}
policy_id	The unique ID to identify a policy
name	The name of the policy
partitions	Number of partitions
replicas	Number of replicas. For Erasure coding storage policies, replicas must be equal to the sum of ex_num_data_fragments and ec_num_parity_fragments.
time	Time (in hours) between moving a partition more than once
ec_type	Optional (only for Erasure Coding storage policies). This specifies the EC scheme that is to be used. Chosen from the list of EC backends provided vy PyECLib
ec_num_data_fragments	Optional (only for Erasure Coding storage policies). The total number of fragments that will be comprised of data.
ec_num_parity_fragments	Optional (only for Erasure Coding storage policies). The total number of fragments that will be comprised of parity.
ec_object_segment_size	Optional (only for Erasure Coding storage policies). The amount of data that will be buffered up before feeding a segment into the encoder/decoder.

HTTP Request Example

```
POST /swift/spolicies

{
"storage_node": {"r1z1-192.168.1.5:6000/sdb1":"200", "r1z1-192.168.1.6:6000/
↪sdb2":"200"},
"policy_id": 5,
"name": "ec104",
"partitions": 4,
"replicas": 14,
"time": 1
"ec_type": "liberasurecode_rs_vand",
"ec_num_data_fragments": 10,
"ec_num_parity_fragments": 4,
"ec_object_segment_size": 1048576
}
```

Response

Response example

```
201 CREATED
```

3.4.4 Locality

An application can ask for the location of specific account/container/object by issuing an HTTP GET request.

Request

URL structure The URL that represents the locality resource is **/swift/locality/{project}/{container}/{swift_object}**

Note: It's mandatory to enter the **project** parameter. **container** and **swift_object** are optional.

Method GET

HTTP Request Example

```
Content-Type: application/json

GET /swift/locality/AUTH_151542dfdsa541asd455fasf1/test1/file.txt
```

Response

Response example

```
200 OK
{
  "headers": {
    "X-Backend-Storage-Policy-Index": "0"
  },
  "endpoints": [
    "http://127.0.0.1:6010/sdb1/867/AUTH_151542dfdsa541asd455fasf1/test1/
↪file.txt",
    "http://127.0.0.1:6020/sdb2/867/AUTH_151542dfdsa541asd455fasf1/test1/
↪file.txt",
    "http://127.0.0.1:6040/sdb4/867/AUTH_151542dfdsa541asd455fasf1/test1/
↪file.txt"
  ]
}
```

3.5 Crystal DSL Grammar

3.5.1 Overview

	FOR *[TARGET]*	WHEN *[TRIGGER]*	DO *[ACTION]*
P1	**TENANT** T1	OBJECT_TYPE=DOCS	**SET** COMPRESSION **WITH** PARAM1=LZ4.**SET** ENCRYPTION
P2	**CONTAINER** C1	GETS_SEC > 5 **AND** OBJECT_SIZE<10M	**SET** CACHING **ON PROXY TRANSIENT**
P3	**TENANT** T2		**SET** BANDWIDTH **WITH** PARAM1=30MBps

Storage automation policies SLO oriented policies

Fig. 1: Crystal DSL structure

SDS policies are means of defining storage services or objectives to be achieved by the system. Crystal DSL hides the complexity of low-level policy enforcement, thus achieving simplified storage administration. The structure of our DSL is as follows:

Target: The target of a policy represents the recipient of a policy's action (e.g., filter enforcement) and it is mandatory to specify it on every policy definition. In our DSL design, targets can be tenants (projects), containers or even individual data objects.

Trigger (optional): Dynamic storage automation policies are characterized by the trigger clause. A policy may have one or more trigger clauses—separated by AND/OR operands—that specify the workload-based situation that will

trigger the enforcement of a filter on the target. Condition clauses consist of inspection triggers, operands (e.g, >, <, =) and values.

Action: The action clause of a SDS policy defines how a filter should be executed on an object request once the policy takes place. To this end, the action clause may accept parameters after the WITH keyword in form of key/value pairs that will be passed as input to the filter execution. Retaking the example of a compression filter, we may decide to enforce compression using a gzip or an lz4 engine, and even to decide the compression level of these engines. In Crystal, filters can be executed at the proxy or at storage nodes (i.e., stages). Our DSL enables to explicitly control the execution stage of a filter in the action clause with the ON keyword. For instance, in P1 at the previous figure we may want to execute compression on the proxy to save up bandwidth (ON PROXY) and encrypt the compressed data object on the storage nodes (ON STORAGE_NODE).

Moreover, dynamic storage automation policies can be persistent or transient; a persistent action means that once the policy is triggered the filter enforcement remains indefinitely (default), whereas actions to be executed only during the period where the condition is satisfied are transient (keyword TRANSIENT).

3.5.2 Examples

Apply the compression filter to all objects of tenant '1234567890abcdef':

```
FOR TENANT:1234567890abcdef DO SET compression
```

Apply caching on the proxy server to all objects of the container 'container1' of tenant '1234567890abcdef':

```
FOR CONTAINER:1234567890abcdef/container1 DO SET caching ON PROXY
```

Apply the caching filter to all objects of tenant '1234567890abcdef' when there are more than 10 GET operations per second (the filter remains indefinitely):

```
FOR TENANT:1234567890abcdef WHEN get_ops > 10  DO SET caching
```

Apply the caching filter to all objects of tenant '1234567890abcdef' only while there are more than 10 GET operations per second:

```
FOR TENANT:1234567890abcdef WHEN get_ops > 10  DO SET caching TRANSIENT
```

Apply a filter pipeline to all objects of tenant '1234567890abcdef'. For PUT operations, the first filter is compression (with a parameter) and the second one is encyption. For GET operations, filters are applied in reverse order:

```
FOR TENANT:1234567890abcdef DO SET compression WITH param1=lz4, SET encryption
```

Apply the compression filter to all objects of tenant '1234567890abcdef' that belong to the object type 'DOCS':

```
FOR TENANT:1234567890abcdef DO SET compression TO OBJECT_TYPE=DOCS
```

Apply the compression filter to all objects of tenant '1234567890abcdef' that are greater than 1024 bytes:

```
FOR TENANT:1234567890abcdef DO SET compression TO OBJECT_SIZE>1024
```

Apply the compression filter to all objects of tenant '1234567890abcdef' only when the request includes headers with parameters to invoke the compression filter:

```
FOR TENANT:1234567890abcdef DO SET compression CALLABLE
```

3.5.3 Grammar

Crystal DSL Grammar in Extended Backus–Naur Form (EBNF):

```
rule = 'FOR', target, ['WHEN', condition list], 'DO', action list, ['TO', object␣
↪list] ;

target = ( tenant | container | object target | tenant group ) ;

tenant = 'TENANT:', alphanums word ;

container = 'CONTAINER:', alphanums word, '/', alphanumshyphens word ;

object target = 'OBJECT:', alphanums word, '/', alphanumshyphens word, '/',␣
↪alphanumshyphens word ;

tenant group = 'G:', nums word ;

condition list = condition, { logical operator, condition list} ;

logical operator = ( 'AND' | 'OR' ) ;

condition = metric, operand, number ;

operand = ( '<' | '>' | '==' | '!=' | '<=' | '>=' ) ;

action list = action, { ',', action list} ;

action = ( 'SET' | 'DELETE' ), filter, { 'WITH', params list }, { 'ON', server }, {
↪'TRANSIENT' }, { 'CALLABLE' } ;

params list = param, { ',', params list } ;

param = alphanumsunderscore word, '=', alphanumsunderscore word ;

server = ( 'PROXY' | 'OBJECT' ) ;

object list = object param, { ',', object list } ;

object param =  object type | object size | ( object type, ',', object size ) | (␣
↪object size, ',', object type ) ;

object type = 'OBJECT_TYPE', '=', alphanums word ;

object size = 'OBJECT_SIZE', operand, number ;

alphanums word = alphanums, { alphanums } ;

alphas word = alphas, { alphas } ;

alphanumshyphens word = alphanumshyphens, { alphanumshyphens } ;

alphanumsunderscore word = alphanumsunderscore, { alphanumsunderscore } ;

nums word = nums, { nums } ;

alphanums = ? all alphanumeric characters: all lowercase and uppercase letters and␣
↪decimal digits ? ;
```

(continues on next page)

(continued from previous page)

```
alphas = ? all lowercase and uppercase letters ? ;

alphanumshyphens = ? all alphanumeric characters plus hyphen and underscore ? ;

alphanumsunderscore = ? all alphanumeric characters plus underscore ? ;

nums = ? all decimal digits ? ;

number = ? A floating point literal ? ;

metric = ? One of the registered metrics ?

filter = ? One of the registered filters ?
```

CHAPTER 4

Indices and tables

- genindex
- search