Red Hat 3scale API Management 2.5

Administering the API Gateway

Intermediate to advanced goals to manage your installation.

Last Updated: 2019-07-31

Red Hat 3scale API Management 2.5 Administering the API Gateway

Intermediate to advanced goals to manage your installation.

Legal Notice

Abstract

This guide provides the information regarding configuration tasks, which can be performed after the basic installation.

Table of Contents

PREFACE . . . 5

CHAPTER 1. OPERATING APICAST . . . 6
1.1. MAPPING RULES 6
1.2. MAPPING RULES WORKFLOW 6
1.3. HOST HEADER 7
1.4. PRODUCTION DEPLOYMENT 7
1.5. PUBLIC BASE URL 8
1.6. PROTECTING YOUR API BACKEND 8
1.7. USING APICAST WITH PRIVATE APIS 8

CHAPTER 2. OPERATING DOCKER-CONTAINERIZED ENVIRONMENTS . . . 10
2.1. TROUBLESHOOTING APICAST ON THE DOCKER-CONTAINERIZED ENVIRONMENT 10
2.1.1. Cannot connect to the Docker daemon error 10
2.1.2. Basic Docker command-line interface commands 10

CHAPTER 3. ADVANCED APICAST CONFIGURATION . . . 11
3.1. DEFINE A SECRET TOKEN 11
3.2. CREDENTIALS 11
3.3. ERROR MESSAGES 12
3.4. CONFIGURATION HISTORY 13
3.5. DEBUGGING 14
3.6. PATH ROUTING 14

CHAPTER 4. APICAST POLICIES . . . 16
4.1. APICAST STANDARD POLICIES 16
4.1.1. 3scale Auth Caching policy 17
4.1.2. 3scale Batcher policy 18
4.1.3. Anonymous Access policy 19
4.1.4. CORS Request Handling policy 20
4.1.5. Echo policy 21
4.1.6. Edge Limiting policy 22
4.1.6.1. Types of limits 22
4.1.6.2. Limit definition 23
4.1.6.3. Liquid templating 24
4.1.6.4. Applying conditions 24
4.1.6.5. Configuring the store 25
4.1.6.6. Error handling 25
4.1.7. Header Modification policy 25
4.1.8. IP Check policy 27
4.1.9. Liquid Context Debug policy 28
4.1.10. Logging policy 28
4.1.11. OAuth 2.0 Token Introspection policy 29
4.1.12. Referrer policy 32
4.1.13. RH-SSO/Keycloak Role Check policy 32
4.1.14. Routing policy 34
4.1.14.1. Routing rules 34
4.1.14.2. Request path rule 34
4.1.14.3. Header rule 34
4.1.14.4. Query argument rule 35
4.1.14.5. JWT claim rule 35
4.1.14.6. Multiple operations rule 36

4.1.14.7. Combining rules 37
4.1.14.8. Catch-all rules 38
4.1.14.9. Supported operations 39
4.1.14.10. Liquid templating 39
4.1.14.11. Set the host used in the Host header 40
4.1.15. Prometheus Metrics 40
4.1.16. SOAP policy 42
4.1.17. TLS Client Certificate Validation policy 43
4.1.17.1. About TLS Client Certificate Validation policy 43
4.1.17.2. Setting up APIcast to work with TLS Client Certificate Validation 43
4.1.17.2.1. Prerequisites: 44
4.1.17.2.2. Setting up APIcast to work with the policy 44
4.1.17.3. Configuring TLS Client Certificate Validation in your policy chain 45
4.1.17.3.1. Prerequisites 45
4.1.17.3.2. Configuring the policy 45
4.1.17.4. Verifying functionality of the TLS Client Certificate Validation policy 45
4.1.17.4.1. Prerequisites: 46
4.1.17.4.2. Verifying policy functionality 46
4.1.17.5. Removing a certificate from the whitelist 46
4.1.17.5.1. Prerequisites 46
4.1.17.5.2. Removing a certificate 46
4.1.17.6. Reference material 46
4.1.18. Upstream policy 46
4.1.19. URL Rewriting policy 47
4.1.19.1. Commands for rewriting the path 48
4.1.19.2. Commands for rewriting the query string 48
4.1.20. URL Rewriting with Captures policy 50
4.2. ENABLING A STANDARD POLICY 51
4.3. CREATING CUSTOM APICAST POLICIES 51
4.4. ADDING CUSTOM POLICIES TO APICAST 51
4.4.1. Adding custom policies to the built-in APIcast 52
4.4.2. Adding custom policies to APIcast on another OpenShift Container Platform 53
4.5. CREATING A POLICY CHAIN IN 3SCALE API MANAGEMENT 53
4.6. CREATING A POLICY CHAIN JSON CONFIGURATION FILE 54

CHAPTER 5. INTEGRATING A POLICY CHAIN WITH APICAST NATIVE DEPLOYMENTS 56
5.1. USING VARIABLES AND FILTERS IN POLICIES 56

CHAPTER 6. APICAST ENVIRONMENT VARIABLES 58
APICAST_BACKEND_CACHE_HANDLER 58
APICAST_CONFIGURATION_CACHE 58
APICAST_CONFIGURATION_LOADER 58
APICAST_CUSTOM_CONFIG 58
APICAST_ENVIRONMENT 58
APICAST_EXTENDED_METRICS 59
APICAST_LOG_FILE 59
APICAST_LOG_LEVEL 59
APICAST_ACCESS_LOG_FILE 59
APICAST_OIDC_LOG_LEVEL 59
APICAST_MANAGEMENT_API 59
APICAST_MODULE 60
APICAST_PATH_ROUTING 60
APICAST_POLICY_LOAD_PATH 60

APICAST_PROXY_HTTPS_CERTIFICATE_KEY 60
APICAST_PROXY_HTTPS_CERTIFICATE 60
APICAST_PROXY_HTTPS_PASSWORD_FILE 60
APICAST_PROXY_HTTPS_SESSION_REUSE 61
APICAST_REPORTING_THREADS 61
APICAST_RESPONSE_CODES 61
APICAST_SERVICES_LIST 61
APICAST_SERVICE_${ID}_CONFIGURATION_VERSION 61
APICAST_WORKERS 61
BACKEND_ENDPOINT_OVERRIDE 62
OPENSSL_VERIFY 62
RESOLVER 62
THREESCALE_CONFIG_FILE 62
THREESCALE_DEPLOYMENT_ENV 62
THREESCALE_PORTAL_ENDPOINT 62
OPENTRACING_TRACER 63
OPENTRACING_CONFIG 63
OPENTRACING_HEADER_FORWARD 63
APICAST_HTTPS_PORT 63
APICAST_HTTPS_CERTIFICATE 63
APICAST_HTTPS_CERTIFICATE_KEY 63
all_proxy, ALL_PROXY 63
http_proxy, HTTP_PROXY 64
https_proxy, HTTPS_PROXY 64
no_proxy, NO_PROXY 64

PREFACE

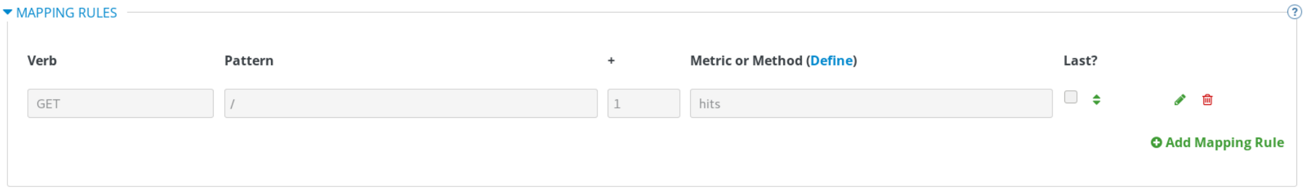
This guide will help you to apply intermediate to advanced configuration features to your 3scale API Management installation. For basic details regarding installation, refer to Installing 3scale API Management.

CHAPTER 1. OPERATING APICAST

This section describes the concepts to consider when working with advanced configurations for APICast.

1.1. MAPPING RULES

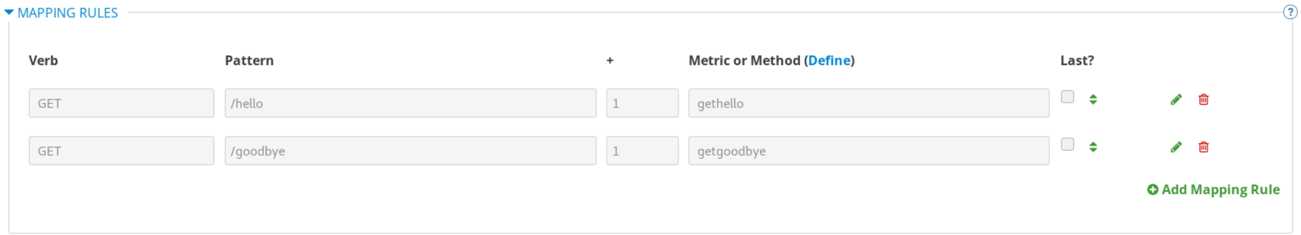
The following is an example of a mapping rule:



Verb	Pattern	+	Metric or Method (Define)	Last?
GET	/	1	hits	□

This rule means that any **GET** request that starts with / will increment the metric **hits** by 1. This mapping rule will match any request to your API. Most likely you will change this rule since it is too generic.

The mapping rules define which metrics (and methods) you want to report depending on the requests to your API. For instance, below you can see the rules for the Echo API that serves us as another example:



Verb	Pattern	+	Metric or Method (Define)	Last?
GET	/hello	1	gethello	□
GET	/goodbye	1	getgoodbye	□

The matching of the rules is done by prefix and can be arbitrarily complex (the notation follows OpenAPI and ActiveDocs specifications)

- You can do a match on the path over a literal string: /**hello**
- Mapping rules can contain named wildcards: /**{word}**

This rule will match anything in the placeholder **{word}**, making requests like /**morning** match the rule.

Wildcards can appear between slashes or between slash and dot.

- Mapping rules can also include parameters on the query string or in the body: /**{word}?value={value}**

APIcast will try to fetch the parameters from the query string when it's a GET and from the body when it's a POST, DELETE, PUT.

Parameters can also have named wildcards.

By default, all mapping rules are evaluated from first to last, according to the sort you specified. If you add a rule /**v1** to the example on the figure above, it would be matched for the requests whose path starts with /v1, considering if it is either /**v1**/**word** or /**v1**/**sentence**.

1.2. MAPPING RULES WORKFLOW

The intended workflow to define mapping rules is as follows:

- You can add new rules by clicking the **Add Mapping Rule** button. Then you select an HTTP method, a pattern, a metric (or method) and finally its increment. When you are done, click **Update & Test Staging Configuration**to apply the changes.
- Mapping rules will be grayed out on the next reload to prevent accidental modifications.
- To edit an existing mapping rule you must enable it first by clicking the pencil icon on the right.
- To delete a rule click on the trash icon.
- Modifications and deletions will be saved when you hit the **Update & Test Staging Configuration** button.

In addition to this workflow, you have two features:

- To sort mapping rules, you can drag and drop them using the green arrows next to the checkbox of each mapping rule in the **Last?** column. The specified sort is saved in the database and it is kept in the content of the proxy configuration specifications after you click the **Update & test in Staging Environment** button.
- You can select the **Last?** checkbox to stop processing other mapping rules. For example, if you have the following mapping rules defined in **API Integration Settings** and you have different metrics associated to each rule:

```
(get) /path/to/example/search
(get) /path/to/example/{id}
```

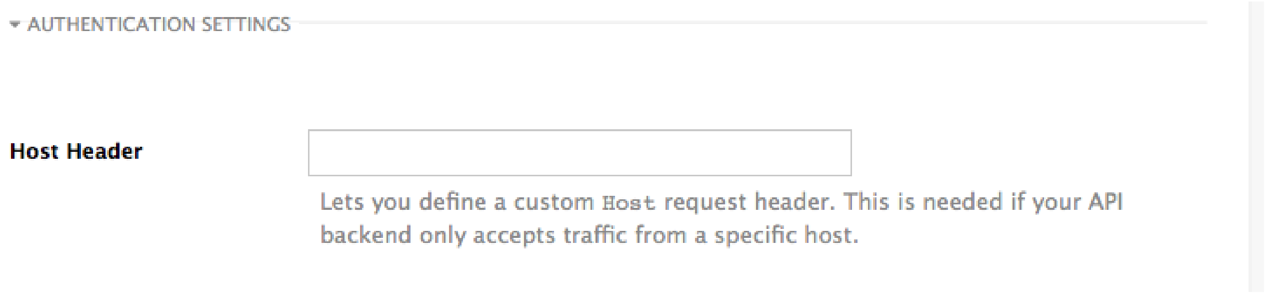
When calling with: **(get) /path/to/example/search**, it will stop processing the remaining mapping rules and incrementing their metrics after the rule is matched.

For more advanced configuration options, you can check the APIcast advanced configuration tutorial.

1.3. HOST HEADER

This option is only needed for those API backends that reject traffic unless the **Host** header matches the expected one. In these cases, having a gateway in front of your API backend will cause problems since the **Host** will be the one of the gateway, e.g. **xxx-yyy.staging.apicast.io**

To avoid this issue you can define the host your API backend expects in the **Host Header** field in the Authentication Settings, and the hosted APIcast instance will rewrite the host.



▾ AUTHENTICATION SETTINGS

Host Header

Lets you define a custom `Host` request header. This is needed if your API backend only accepts traffic from a specific host.

1.4. PRODUCTION DEPLOYMENT

Once you have configured your API integration and verified it is working in the Staging environment, you can go ahead with one of the APIcast production deployments.

At the bottom of the Integration page you will find the *Production* section. You will find two fields here: the *Private Base URL*, which will be the same as you configured in the *Staging* section, and the *Public Base URL*.

1.5. PUBLIC BASE URL

The **Public Base URL** is the URL, which your developers will use to make requests to your API, protected by 3scale. This will be the URL of your APIcast instance.

If you are using one of the Self-managed deployment options, you can choose your own Public Base URL for each one of the environments provided (staging and production), on a domain name you are managing. Note that this URL should be different from the one of your API backend, and could be something like **https://api.yourdomain.com:443**, where **yourdomain.com** is the domain that belongs to you. After setting the Public Base URL make sure you save the changes and, if necessary, promote the changes in staging to production.

Please note that APIcast will only accept calls to the hostname which is specified in the Public Base URL. For example, for the Echo API example used above, if we specify **https://echo-api.3scale.net:443** as the Public Base URL, the correct call would be be:

```
curl "https://echo-api.3scale.net:443/hello?user_key=YOUR_USER_KEY"
```

In case you don't yet have a public domain for your API, you can also use the APIcast IP in the requests, but you still need to specify a value in the Public Base URL field (even if the domain is not real), and in this case make sure you provide the host in the Host header, for example:

```
curl "http://192.0.2.12:80/hello?user_key=YOUR_USER_KEY" -H "Host: echo-api.3scale.net"
```

If you are deploying on local machine, you can also just use "localhost" as the domain, so the Public Base URL will look like **http://localhost:80**, and then you can make requests like this:

```
curl "http://localhost:80/hello?user_key=YOUR_USER_KEY"
```

In case you have multiple API services, you will need to set this Public Base URL appropriately for each service. APIcast will route the requests based on the hostname.

1.6. PROTECTING YOUR API BACKEND

Once you have APIcast working in production, you might want to restrict direct access to your API backend without credentials. The easiest way to do this is by using the Secret Token set by APIcast. Please refer to the Advanced APIcast configuration for information on how to set it up.

1.7. USING APICAST WITH PRIVATE APIS

With APIcast it is possible to protect the APIs which are not publicly accessible on the Internet. The requirements that must be met are:

- APIcast self-managed must be used as the deployment option
- APIcast needs to be accessible from the public internet and be able to make outbound calls to the 3scale Service Management API
- the API backend should be accessible by APIcast

In this case you can set your internal domain name or the IP address of your API in the *Private Base URL* field and follow the rest of the steps as usual. Note, however, that you will not be able to take advantage of the Staging environment, and the test calls will not be successful, as the Staging APIcast instance is hosted by 3scale and will not have access to your private API backend). But once you deploy APIcast in your production environment, if the configuration is correct, APIcast will work as expected.

CHAPTER 2. OPERATING DOCKER-CONTAINERIZED ENVIRONMENTS

2.1. TROUBLESHOOTING APICAST ON THE DOCKER-CONTAINERIZED ENVIRONMENT

This section describes the most common issues that you can find when working with APIcast on a Docker-containerized environment.

2.1.1. *Cannot connect to the Docker daemon* error

The **docker: Cannot connect to the Docker daemon. Is the docker daemon running on this host?** error message may be because the Docker service hasn't started. You can check the status of the Docker daemon by running the **sudo systemctl status docker.service** command.

Ensure that you are run this command as the **root** user because the Docker containerized environment requires root permissions in RHEL by default. For more information, see here).

2.1.2. Basic Docker command-line interface commands

If you started the container in the detached mode (**-d** option) and want to check the logs for the running APIcast instance, you can use the **log** command: **sudo docker logs <container>**. Where ,**<container>** is the container name (*"apicast"* in the example above) or the container ID. You can get a list of the running containers and their IDs and names by using the **sudo docker ps** command.

To stop the container, run the **sudo docker stop <container>** command. You can also remove the container by running the **sudo docker rm <container>** command.

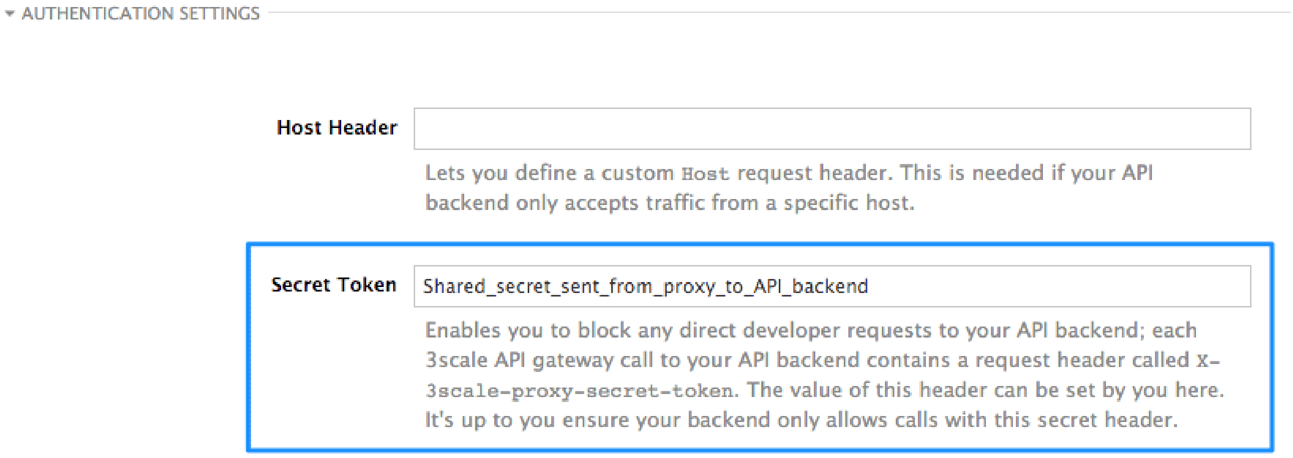
For more information on available commands, see Docker commands reference.

CHAPTER 3. ADVANCED APICAST CONFIGURATION

This section covers the advanced settings option of 3scale's API gateway in the staging environment.

3.1. DEFINE A SECRET TOKEN

For security reasons, any request from the 3scale gateway to your API backend contains a header called **X-3scale-proxy-secret-token**. You can set the value of this header in **Authentication Settings** on the Integration page.



▾ AUTHENTICATION SETTINGS

Host Header

Lets you define a custom Host request header. This is needed if your API backend only accepts traffic from a specific host.

Secret Token: Shared_secret_sent_from_proxy_to_API_backend

Enables you to block any direct developer requests to your API backend; each 3scale API gateway call to your API backend contains a request header called X-3scale-proxy-secret-token. The value of this header can be set by you here. It's up to you ensure your backend only allows calls with this secret header.

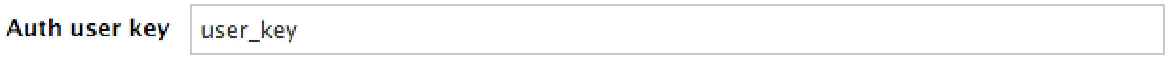
Setting the secret token acts as a shared secret between the proxy and your API so that you can block all API requests that do not come from the gateway if you do not want them to. This adds an extra layer of security to protect your public endpoint while you are in the process of setting up your traffic management policies with the sandbox gateway.

Your API backend must have a public resolvable domain for the gateway to work, so anyone who knows your API backend can bypass the credentials checking. This should not be a problem because the API gateway in the staging environment is not meant for production use, but it is always better to have a fence available.

3.2. CREDENTIALS

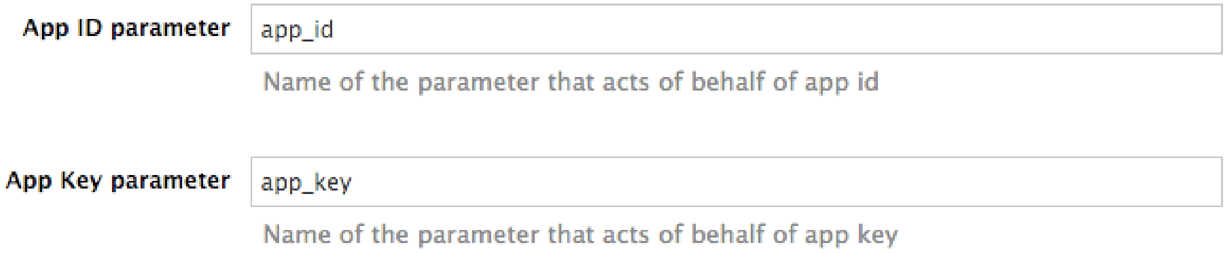
The API credentials within 3scale are either **user_key** or **app_id/app_key** depending on the authentication mode that you are using. OpenID Connect is valid for the API gateway in the staging environment, but it cannot be tested in the Integration page.

However, you might want to use different credential names in your API. In this case, you need to set custom names for the **user_key** if you are using the API key mode:



Auth user key: user_key

Alternatively, for the **app_id** and **app_key**:



App ID parameter: app_id

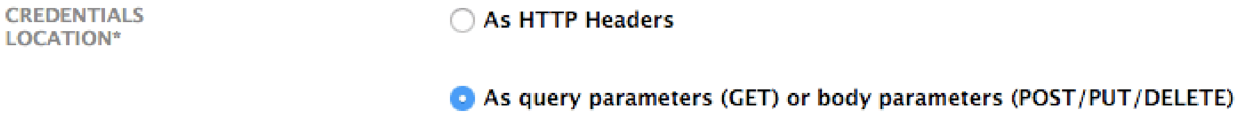
Name of the parameter that acts of behalf of app id

App Key parameter: app_key

Name of the parameter that acts of behalf of app key

For instance, you could rename **app_id** to **key** if that fits your API better. The gateway will take the name **key** and convert it to **app_id** before doing the authorization call to the 3scale backend. Note that the new credential name has to be alphanumeric.

You can decide if your API passes credentials in the query string (or body if not a GET) or in the headers.



CREDENTIALS LOCATION*

○ As HTTP Headers

● As query parameters (GET) or body parameters (POST/PUT/DELETE)

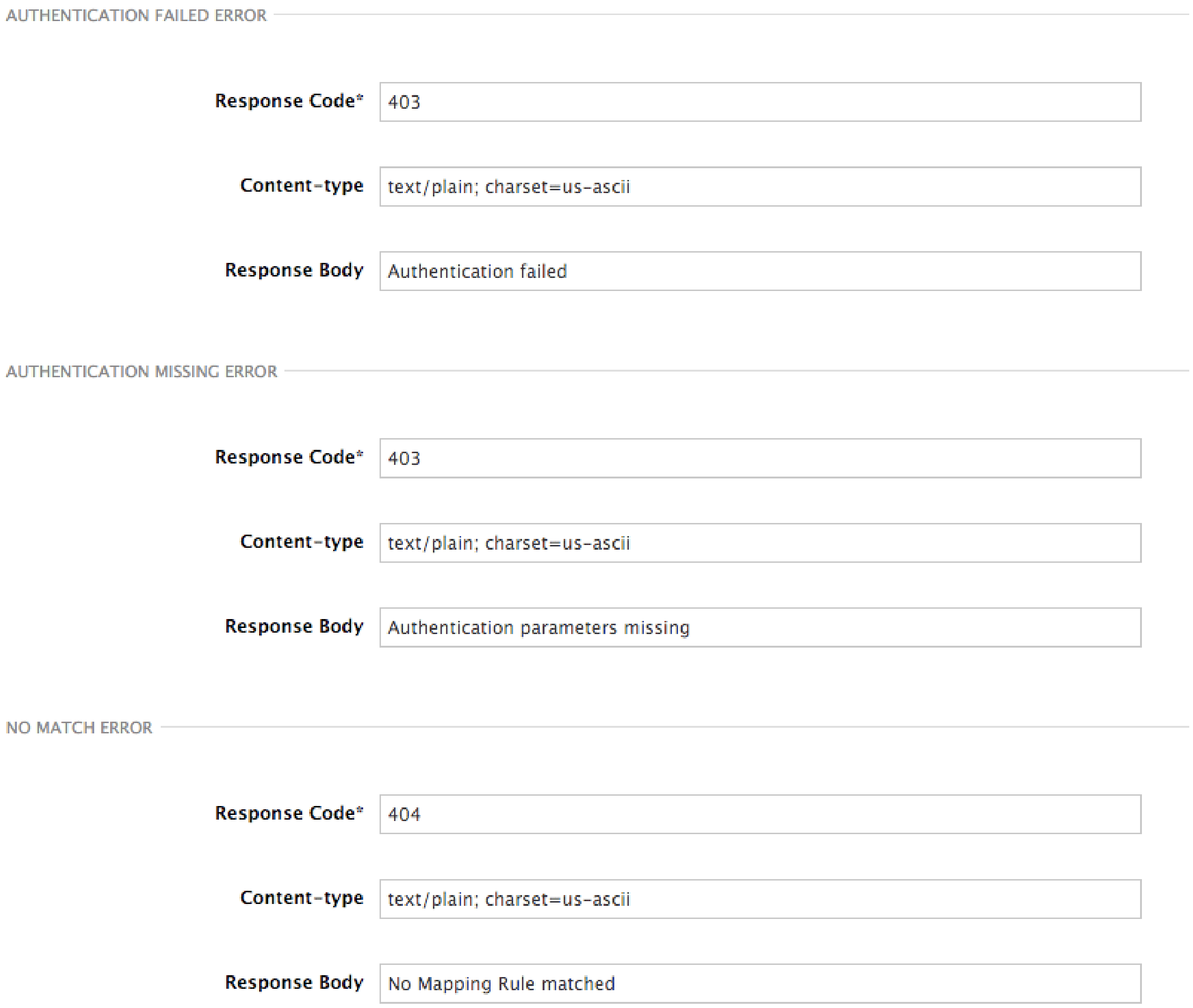
NOTE

APIcast normalizes header names when extracting credentials. This means they are case insensitive, and underscores and hyphens are treated equally. For example, if you set the App Key parameter as **App_Key**, other values such as **app-key** are also accepted as valid app key headers.

3.3. ERROR MESSAGES

Another important element for a full-fledged configuration is to define your own custom error messages.

It is important to note that the 3scale API gateway in the staging environment will do a pass of any error message generated by your API. However, since the management layer of your API is now carried out by the gateway, there are some errors that your API will never see because some requests will be terminated by the gateway.



AUTHENTICATION FAILED ERROR

Response Code* 403

Content-type text/plain; charset=us-ascii

Response Body Authentication failed

AUTHENTICATION MISSING ERROR

Response Code* 403

Content-type text/plain; charset=us-ascii

Response Body Authentication parameters missing

NO MATCH ERROR

Response Code* 404

Content-type text/plain; charset=us-ascii

Response Body No Mapping Rule matched

Following are some of the errors:

- Authentication missing: This error is generated whenever an API request does not contain any credentials. It occurs when users do not add their credentials to an API request.
- Authentication failed: This error is generated whenever an API request does not contain the valid credentials. It can be because the credentials are fake or because the application has been temporarily suspended.
- No match: This error means that the request did not match any mapping rule and therefore no metric is updated. This is not necessarily an error, but it means that either the user is trying random paths or that your mapping rules do not cover legitimate cases.
- Retry after: This error is triggered when an API request exceeds the rate limit. A Retry-After header is returned, with a status code of 429 and the number of seconds until the limit expires.

3.4. CONFIGURATION HISTORY

Every time you click the **Update & Test Staging Configuration**button, the current configuration is saved in a JSON file. The staging gateway will pull the latest configuration with each new request. For each environment, staging or production, you can see a history of all the previous configuration files.

Note that it is not possible to automatically roll back to previous versions. Instead a history of all your configuration versions with their associated JSON files is provided. Use these files to check what configuration you had deployed at any point of time. If you want to, you can recreate any deployments manually.

3.5. DEBUGGING

Setting up the gateway configuration is easy, but you may still encounter errors. In such cases, the gateway can return useful debug information to track the error.

To get the debugging information from APIcast, you must add the following header to the API request: **X-3scale-debug: {SERVICE_TOKEN}** with the service token corresponding to the API service that you are reaching to.

When the header is found and the service token is valid, the gateway will add the following information to the response headers:

```
X-3scale-matched-rules: /v1/word/{word}.json, /v1
X-3scale-credentials: app_key=APP_KEY&app_id=APP_ID
X-3scale-usage: usage%5Bversion_1%5D=1&usage%5Bword%5D=1
```

X-3scale-matched-rules indicates which mapping rules have been matched for the request in a comma-separated list.

The header **X-3scale-credentials** returns the credentials that were passed to 3scale backend.

X-3scale-usage indicates the usage that was reported to 3scale backend. **usage%5Bversion_1%5D=1&usage%5Bword%5D=1** is a URL-encoded **usage[version_1]=1&usage[word]=1** and shows that the API request incremented the methods (metrics) **version_1** and **word** by 1 hit each.

3.6. PATH ROUTING

APIcast handles all the API services configured on a 3scale account (or a subset of services, if the **APICAST_SERVICES_LIST** environment variable is configured). Normally, APIcast routes the API requests to the appropriate API service based on the hostname of the request, by matching it with the *Public Base URL* . The first service where the match is found is used for the authorization.

The Path routing feature allows using the same *Public Base URL* on multiple services and routes the requests using the path of the request. To enable the feature, set the **APICAST_PATH_ROUTING** environment variable to **true** or **1**. When enabled, APIcast will map the incoming requests to the services based on both hostname and path.

This feature can be used if you want to expose multiple backend services hosted on different domains through one gateway using the same *Public Base URL* . To achieve this you can configure several API services for each API backend (i.e. *Private Base URL*) and enable the path routing feature.

For example, you have 3 services configured in the following way:

- Service A Public Base URL: **api.example.com** Mapping rule: **/a**
- Service B Public Base URL: **api2.example.com** Mapping rule: **/b**
- Service C Public Base URL: **api.example.com** Mapping rule: **/c**

If path routing is **disabled** (**APICAST_PATH_ROUTING=false**), all calls to **api.example.com** will try to match Service A. So, the calls **api.example.com/c** and **api.example.com/b** will fail with a *"No Mapping Rule matched"* error.

If path routing is **enabled** (**APICAST_PATH_ROUTING=true**), the calls will be matched by both host and path. So:

- **api.example.com/a** will be routed to Service A
- **api.example.com/c** will be routed to Service C
- **api.example.com/b** will fail with "No Mapping Rule matched" error, i.e. it will NOT match Service B, as the *Public Base URL* does not match.

If path routing is used, you must ensure there is no conflict between the mapping rules in different services that use the same *Public Base URL* , i.e. each combination of method + path pattern is only used in one service.

CHAPTER 4. APICAST POLICIES

APIcast policies are units of functionality that modify how APIcast operates. Policies can be enabled, disabled, and configured to control how they modify APIcast. Use policies to add functionality that is not available in a default APIcast deployment. You can create your own policies, or use standard policies provided by Red Hat 3scale.

The following topics provide information about the standard APIcast policies, creating your own custom APIcast policies, and creating a policy chain.

- APIcast Standard Policies
- Creating custom APIcast policies
- Creating a policy chain in the AMP

Control policies for a service with a policy chain. Policy chains do the following:

- specify what policies APIcast uses
- provide configuration information for policies 3scale uses
- specify the order in which 3scale loads policies

NOTE

Red Hat 3scale provides a method for adding custom policies, but does not support custom policies.

In order to modify APIcast behavior with custom policies, you must do the following:

- Add custom policies to APIcast
- Define a policy chain that configures APIcast policies
- Add the policy chain to APIcast

4.1. APICAST STANDARD POLICIES

3scale API Management provides the following standard policies:

- Section 4.1.1, “3scale Auth Caching policy”
- Section 4.1.2, “3scale Batcher policy”
- Section 4.1.3, “Anonymous Access policy”
- Section 4.1.4, “CORS Request Handling policy”
- Section 4.1.5, “Echo policy”
- Section 4.1.6, “Edge Limiting policy”
- Section 4.1.7, “Header Modification policy”
- Section 4.1.9, “Liquid Context Debug policy”

- Section 4.1.10, “Logging policy”
- Section 4.1.11, “OAuth 2.0 Token Introspection policy”
- Section 4.1.12, “Referrer policy”
- Section 4.1.13, “RH-SSO/Keycloak Role Check policy”
- Section 4.1.14, “Routing policy”
- Section 4.1.16, “SOAP policy”
- Section 4.1.18, “Upstream policy”
- Section 4.1.19, “URL Rewriting policy”
- Section 4.1.20, “URL Rewriting with Captures policy”

You can enable and configure standard policies in the 3scale API Management.

4.1.1. 3scale Auth Caching policy

The 3scale Auth Caching policy caches authentication calls made to APIcast. You can select an operating mode to configure the cache operations.

3scale Auth Caching is available in the following modes:

1. Strict - Cache only authorized calls.

"Strict" mode only caches authorized calls. If a policy is running under the "strict" mode and if a call fails or is denied, the policy invalidates the cache entry. If the backend becomes unreachable, all cached calls are rejected, regardless of their cached status.

2. Resilient – Authorize according to last request when backend is down.

The "Resilient" mode caches both authorized and denied calls. If the policy is running under the "resilient" mode, failed calls do not invalidate an existing cache entry. If the backend becomes unreachable, calls hitting the cache continue to be authorized or denied based on their cached status.

3. Allow - When backend is down, allow everything unless seen before and denied.

The "Allow" mode caches both authorized and denied calls. If the policy is running under the "allow" mode, cached calls continue to be denied or allowed based on the cached status. However, any new calls are cached as authorized.

> IMPORTANT
>
> Operating in the "allow" mode has security implications. Consider these implications and exercise caution when using the "allow" mode.

4. None - Disable caching.

The "None" mode disables caching. This mode is useful if you want the policy to remain active, but do not want to use caching.

Configuration properties

property	description	values	required?
caching_type	The **caching_type** property allows you to define which mode the cache will operate in.	data type: enumerated string [resilient, strict, allow, none]	yes

Policy object example

```
{
  "name": "caching",
  "version": "builtin",
  "configuration": {
    "caching_type": "allow"
  }
}
```

For information on how to configure policies, see the Creating a policy chain section of the documentation.

4.1.2. 3scale Batcher policy

The 3scale Batcher policy provides an alternative to the standard APIcast authorization mechanism, in which one call to the 3scale backend (Service Management API) is made for each API request APIcast receives.

The 3scale Batcher policy reduces latency and increases throughput by significantly reducing the number of requests to the 3scale backend. In order to achieve this, this policy caches authorization statuses and batches usage reports.

When the 3scale Batcher policy is enabled, APIcast uses the following authorization flow:

1. On each request, the policy checks whether the credentials are cached:
   - If the credentials are cached, the policy uses the cached authorization status instead of calling the 3scale backend.
   - If the credentials are not cached, the policy calls the backend and caches the authorization status with a configurable Time to Live (TTL).
2. Instead of reporting the usage corresponding to the request to the 3scale backend immediately, the policy accumulates their usage counters to report them to the backend in batches. A separate thread reports the accumulated usage counters to the 3scale backend in a single call, with a configurable frequency.

The 3scale Batcher policy improves the throughput, but with reduced accuracy. The usage limits and the current utilization are stored in 3scale, and APIcast can only get the correct authorization status when making calls to the 3scale backend. When the 3scale Batcher policy is enabled, there is a period of time APIcast is not sending calls to 3scale. During this window, applications making calls might go over the defined limits.

Use this policy for high-load APIs if the throughput is more important than the accuracy of the rate limiting. The 3scale Batcher policy gives better results in terms of accuracy when the reporting frequency and authorization TTL are much less than the rate limiting period. For example, if the limits are per day and the reporting frequency and authorization TTL are configured to be several minutes.

The 3scale Batcher policy supports the following configuration settings:

- **auths_ttl**: Sets the TTL in seconds when the authorization cache expires.
  When the authorization for the current call is cached, APIcast uses the cached value. After the time set in the **auths_ttl** parameter, APIcast removes the cache and calls the 3scale backend to retrieve the authorization status.
- **batch_report_seconds**: Sets the frequency of batch reports APIcast sends to the 3scale backend. The default value is **10** seconds.

IMPORTANT

To use this policy, enable both the **3scale APIcast** and **3scale Batcher** policy in the policy chain.

4.1.3. Anonymous Access policy

The Anonymous Access policy exposes a service without authentication. It can be useful, for example, for legacy applications that cannot be adapted to send the authentication parameters. The Anonymous policy only supports services with API Key and App Id / App Key authentication options. When the policy is enabled for API requests that do not have any credentials provided, APIcast will authorize the calls using the default credentials configured in the policy. For the API calls to be authorized, the application with the configured credentials must exist and be active.

Using the Application Plans, you can configure the rate limits on the application used for the default credentials.

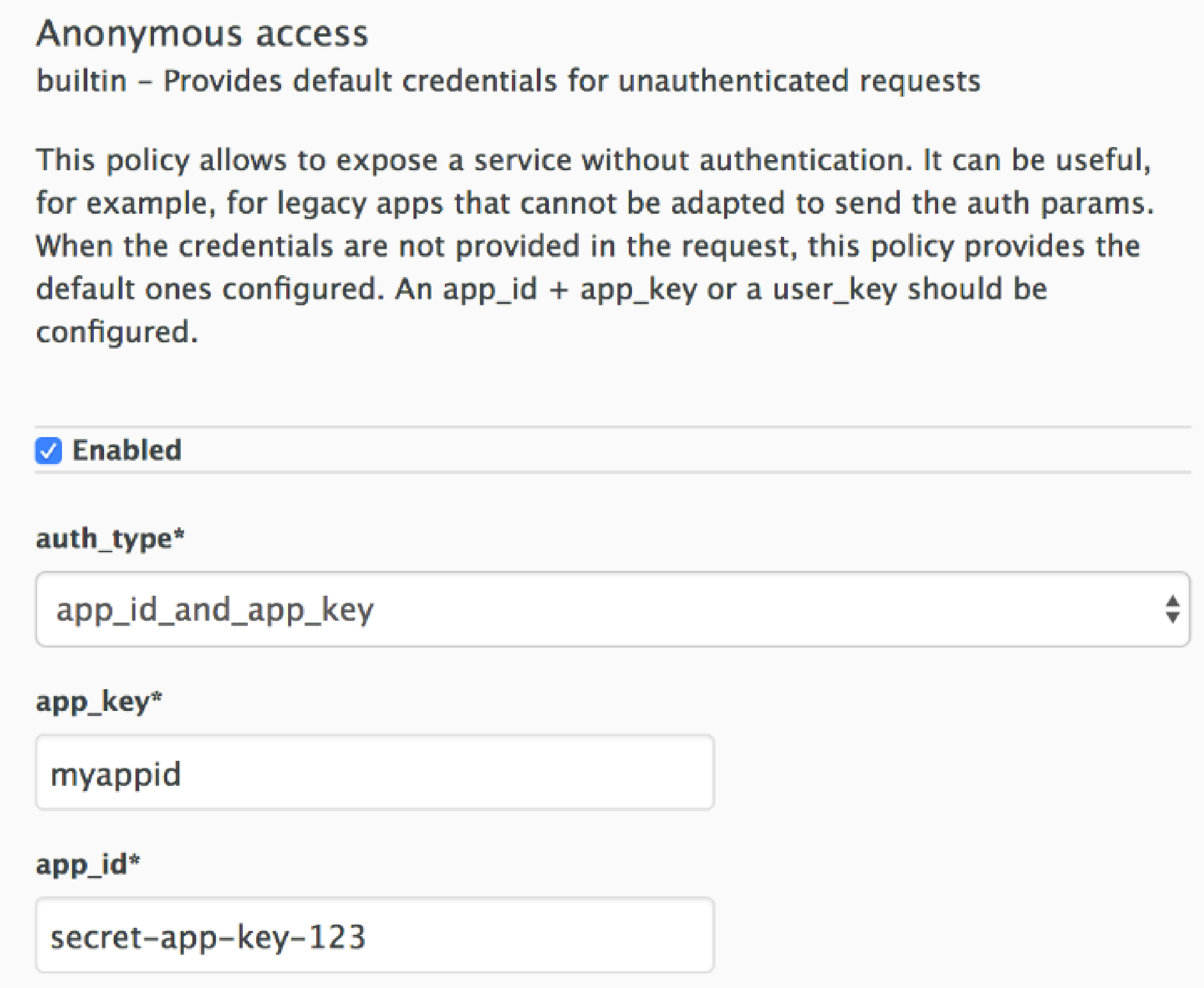
NOTE

You need to place the Anonymous Access policy before the APIcast Policy, when using these two policies together in the policy chain.

Following are the required configuration properties for the policy:

- **auth_type**: Select a value from one of the alternatives below and make sure the property corresponds to the authentication option configured for the API:
  - **app_id_and_app_key**: For App ID / App Key authentication option.
  - **user_key**: For API key authentication option.
- **app_id** (only for **app_id_and_app_key** auth type): The App Id of the application that will be used for authorization if no credentials are provided with the API call.
- **app_key** (only for **app_id_and_app_key** auth type): The App Key of the application that will be used for authorization if no credentials are provided with the API call.
- **user_key** (only for the **user_key** auth_type): The API Key of the application that will be used for authorization if no credentials are provided with the API call.

Figure 4.1. Anonymous Access Policy



Anonymous access

builtin - Provides default credentials for unauthenticated requests

This policy allows to expose a service without authentication. It can be useful, for example, for legacy apps that cannot be adapted to send the auth params. When the credentials are not provided in the request, this policy provides the default ones configured. An app_id + app_key or a user_key should be configured.

☑ Enabled

auth_type*

app_id_and_app_key

app_key*

myappid

app_id*

secret-app-key-123

4.1.4. CORS Request Handling policy

The Cross Origin Resource Sharing (CORS) request handling policy allows you to control CORS behavior by allowing you to specify:

- Allowed headers
- Allowed methods
- Allow credentials
- Allowed origin headers

The CORS request handling policy will block all unspecified CORS requests.

NOTE

You need to place the CORS Request Handling policy before the APIcast Policy, when using these two policies together in the policy chain.

Configuration properties

property	description	values	required?
allow_headers	The **allow_headers** property is an array in which you can specify which CORS headers APIcast will allow.	data type: array of strings, must be a CORS header	no
allow_methods	The **allow_methods** property is an array in which you can specify which CORS methods APIcast will allow.	data type: array of enumerated strings [GET, HEAD, POST, PUT, DELETE, PATCH, OPTIONS, TRACE, CONNECT]	no
allow_origin	The **allow_origin** property allows you to specify an origin domain APIcast will allow	data type: string	no
allow_credentials	The **allow_credentials** property allows you to specify whether APIcast will allow a CORS request with credentials	data type: boolean	no

Policy object example

```
{
  "name": "cors",
  "version": "builtin",
  "configuration": {
    "allow_headers": [
      "App-Id", "App-Key",
      "Content-Type", "Accept"
    ],
    "allow_credentials": true,
    "allow_methods": [
      "GET", "POST"
    ],
    "allow_origin": "https://example.com"
  }
}
```

For information on how to configure policies, refer to the Creating a policy chain section of the documentation.

4.1.5. Echo policy

The Echo policy prints an incoming request back to the client, along with an optional HTTP status code.

Configuration properties

property	description	values	required?
status	The HTTP status code the echo policy will return to the client	data type: integer	no
exit	Specifies which exit mode the echo policy will use. The **request** exit mode stops the incoming request from being processed. The **set** exit mode skips the rewrite phase.	data type: enumerated string [request, set]	yes

Policy object example

```
{
  "name": "echo",
  "version": "builtin",
  "configuration": {
    "status": 404,
    "exit": "request"
  }
}
```

For information on how to configure policies, refer to the Creating a policy chain section of the documentation.

4.1.6. Edge Limiting policy

The Edge Limiting policy aims to provide flexible rate limiting for the traffic sent to the backend API and can be used with the default 3scale authorization. Some examples of the use cases supported by the policy include:

- End-user rate limiting: Rate limit by the value of the "sub" (subject) claim of a JWT token passed in the Authorization header of the request (configured as **{{ jwt.sub }}**).
- Requests Per Second (RPS) rate limiting.
- Global rate limits per service: Apply limits per service rather than per application.
- Concurrent connection limit: Set the number of concurrent connections allowed.

4.1.6.1. Types of limits

The policy supports the following types of limits that are provided by the lua-resty-limit-traffic library:

- **leaky_bucket_limiters**: Based on the "leaky_bucket" algorithm that is based on the average number of requests plus a maximum burst size.

- **fixed_window_limiters**: Based on a fixed window of time (last X seconds).
- **connection_limiters**: Based on the concurrent number of connections.

You can scope any limit by service or globally.

4.1.6.2. Limit definition

The limits have a key that encodes the entities that are used to define the limit (an IP, a service, an endpoint, an ID, the value for a specific header, etc.). The Key is specified in the **key** parameter of the limiter.

key is an object that is defined by the following properties:

- **name**: It is the name of the key. It must be unique in the scope.
- **scope**: It defines the scope of the key. The supported scopes are:
    - Per service scope that affects one service (**service**).
    - Global scope that affects all the services (**global**).
- **name_type**: It defines how the "name" value will be evaluated:
    - As plain text (**plain**)
    - As Liquid (**liquid**)

Each limit also has some parameters that vary depending on their types:

- **leaky_bucket_limiters**: **rate**, **burst**.
    - **rate**: It defines how many requests can be made per second without a delay.
    - **burst**: It defines the amount of requests per second that can exceed the allowed rate. An artificial delay is introduced for requests above the allowed rate (specified by **rate**). After exceeding the rate by more requests per second than defined in **burst**, the requests get rejected.
- **fixed_window_limiters**: **count**, **window**. **count** defines how many requests can be made per number of seconds defined in **window**.
- **connection_limiters**: **conn**, **burst**, **delay**.
    - **conn**: Defines the maximum number of the concurrent connections allowed. It allows exceeding that number by **burst** connections per second.
    - **delay**: It is the number of seconds to delay the connections that exceed the limit.

Examples

1. Allow 10 requests per minute to service_A:

    ```
    {
      "key": { "name": "service_A" },
      "count": 10,
    ```

```
  "window": 60
}
```

2. Allow 100 connections with bursts of 10 with a delay of 1s:

```
{
  "key": { "name": "service_A" },
  "conn": 100,
  "burst": 10,
  "delay": 1
}
```

You can define several limits for each service. In case multiple limits are defined, the request can be rejected or delayed if at least one limit is reached.

4.1.6.3. Liquid templating

The Edge Limiting policy allows specifying the limits for the dynamic keys by supporting Liquid variables in the keys. For this, the **name_type** parameter of the key must be set to "liquid" and the **name** parameter can then use Liquid variables. Example: **{{ remote_addr }}** for the client IP address or **{{ jwt.sub }}** for the "sub" claim of the JWT token.

Example:

```
{
  "key": { "name": "{{ jwt.sub }}", "name_type": "liquid" },
  "count": 10,
  "window": 60
}
```

For more information about Liquid support, see Section 5.1, “Using variables and filters in policies” .

4.1.6.4. Applying conditions

Each limiter can have an optional condition that defines when the limiter must be applied. The condition is specified in the **condition** property of the limiter.

condition is defined by the following properties:

- **combine_op**. It is the boolean operator applied to the list of operations. The following two values are supported: **or** and **and**.
- **operations**. It is a list of conditions that need to be evaluated. Each operation is represented by an object with the following properties:
  - **left**: The left part of the operation.
  - **left_type**: How the **left** property is evaluated (plain or liquid).
  - **right**: The right part of the operation.
  - **right_type**: How the **right** property is evaluated (plain or liquid).

- **op**: Operator applied between the left and the right parts. The following two values are supported: **==** (equals) and **!=** (not equals).

Example:

```
"condition": {
  "combine_op": "and",
  "operations": [
    {
      "op": "==",
      "right": "GET",
      "left_type": "liquid",
      "left": "{{ http_method }}",
      "right_type": "plain"
    }
  ]
}
```

4.1.6.5. Configuring the store

By default, Edge Limiting policy uses OpenResty shared dictionary for the rate limiting counters. However, an external Redis server can be used instead of the shared dictionary. This can be useful when multiple APIcast instances are used. Redis server can be configured using the **redis_url** parameter.

4.1.6.6. Error handling

The limiters support the following parameters to configure how the errors are handled:

- **limits_exceeded_error** allows to configure the error status code and message that will be returned to the client when the configured limits are exceeded. The following parameters should be configured:
  - **status_code**: The status code of the request when the limits are exceeded. Default: **429**.
  - **error_handling**: How to handle the error.
    - **exit**: "Respond with an error".
    - **log**: "Let the request go through and only output logs"
- **configuration_error** allows to configure the error status code and message that will be returned to the client in case of incorrect configuration. The following parameters should be configured:
  - **status_code** : The status code when there is a configuration issue. Default: 500.
  - **error_handling**: How to handle the error.
    - **exit**: "Respond with an error".
    - **log**: "Let the request go through and only output logs".

4.1.7. Header Modification policy

The Header Modification policy allows you to modify the existing headers or define additional headers to add to or remove from an incoming request or response. You can modify both response and request headers.

The Header Modification policy supports the following configuration parameters:

- **request**: List of operations to apply to the request headers
- **response**: List of operations to apply to the response headers

Each operation consists of the following parameters:

- **op**: Specifies the operation to be applied. The **add** operation adds a value to an existing header. The **set** operation creates a header and value, and will overwrite an existing header's value if one already exists. The **push** operation creates a header and value, but will not overwrite an existing header's value if one already exists. Instead, **push** will add the value to the existing header. The **delete** operation removes the header.
- **header**: Specifies the header to be created or modified and can be any string that can be used as a header name (e.g. **Custom-Header**).
- **value_type**: Defines how the header value will be evaluated and can either be **plain** for plain text or **liquid** for evaluation as a Liquid template. For more information, see Section 5.1, "Using variables and filters in policies".
- **value**: Specifies the value that will be used for the header. For value type "liquid" the value should be in the format **{{ variable_from_context }}**. Not needed when deleting.

Policy object example

```
{
  "name": "headers",
  "version": "builtin",
  "configuration": {
    "response": [
      {
        "op": "add",
        "header": "Custom-Header",
        "value_type": "plain",
        "value": "any-value"
      }
    ],
    "request": [
      {
        "op": "set",
        "header": "Authorization",
        "value_type": "plain",
        "value": "Basic dXNlcm5hbWU6cGFzc3dvcmQ="
      },
      {
        "op": "set",
        "header": "Service-ID",
        "value_type": "liquid",
        "value": "{{service.id}}"
      }
      ]
    }
  }
```

For information on how to configure policies, see the Creating a policy chain section of the documentation.

4.1.8. IP Check policy

The IP Check policy is used to deny or allow requests based on a list of IPs.

Configuration properties

property	description	data type	required?
check_type	The **check_type** property has two possible values, **whitelist** or **blacklist**. **blacklist** will deny all requests from IPs on the list. **whitelist** will deny all requests from IPs *not* on the list.	string, must be either **whitelist** or **blacklist**	yes
ips	The **ips** property allows you to specify a list of IP addresses to whitelist or blacklist. Both single IPs and CIDR ranges can be used.	array of strings, must be valid IP addresses	yes
error_msg	The **error_msg** property allows you to configure the error message returned when a request is denied.	string	no
client_ip_sources	The **client_ip_sources** property allows you to configure how to retrieve the client IP. By default, the last caller IP is used. The other options are **X-Forwarded-For** and **X-Real-IP**.	array of strings, valid options are one or more of **X-Forwarded-For**, **X-Real-IP**, **last_caller**.	no

Policy object example

```
{
  "name": "ip_check",
  "configuration": {
    "ips": [ "3.4.5.6", "1.2.3.0/4" ],
    "check_type": "blacklist",
    "client_ip_sources": ["X-Forwarded-For", "X-Real-IP", "last_caller"],
    "error_msg": "A custom error message"
  }
}
```

For information on how to configure policies, refer to the Creating a policy chain section of the documentation.

4.1.9. Liquid Context Debug policy

NOTE

The Liquid Context Debug policy is meant only for debugging purposes in the development environment and not in production.

This policy responds to the API request with a JSON, containing the objects and values that are available in the context and can be used for evaluating Liquid templates. When combined with the 3scale APIcast or Upstream policy, Liquid Context Debug must be placed before them in the policy chain in order to work correctly. To avoid circular references, the policy only includes duplicated objects once and replaces them with a stub value.

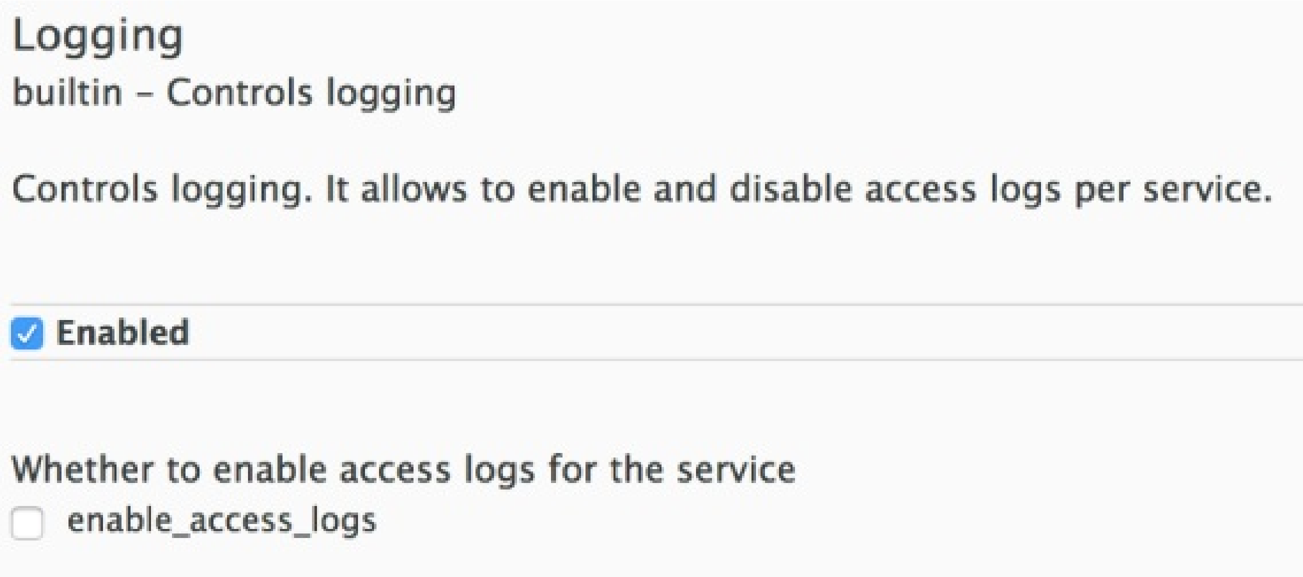
An example of the value returned by APIcast when the policy is enabled:

```
{
  "jwt": {
    "azp": "972f7b4f",
    "iat": 1537538097,
    ...
    "exp": 1537574096,
    "typ": "Bearer"
  },
  "credentials": {
    "app_id": "972f7b4f"
  },
  "usage": {
    "deltas": {
      "hits": 1
    },
    "metrics": [
      "hits"
    ]
  },
  "service": {
    "id": "2",
    ...
  }
  ...
}
```

4.1.10. Logging policy

The Logging policy allows enabling or disabling APIcast (NGINX) access logs for each API service individually. By default, this policy is not enabled in policy chains.

This policy only supports the **enable_access_logs** configuration parameter. To disable access logging for a service, enable the policy, unselect the **enable_access_logs** parameter and click the **Submit** button. To enable the access logs, select the **enable_access_logs** parameter or disable the Logging policy.



Logging

builtin - Controls logging

Controls logging. It allows to enable and disable access logs per service.

☑ Enabled

Whether to enable access logs for the service

□ enable_access_logs

You can combine the Logging policy with the global setting for the location of access logs. Set the **APICAST_ACCESS_LOG_FILE** environment variable to configure the location of APIcast access logs. By default, this variable is set to **/dev/stdout**, which is the standard output device. For further details about global APIcast parameters, see Chapter 6, *APIcast environment variables*.

4.1.11. OAuth 2.0 Token Introspection policy

The OAuth 2.0 Token Introspection policy allows validating the JSON Web Token (JWT) token used for services with the OpenID Connect authentication option using the Token Introspection Endpoint of the token issuer (Red Hat Single Sign-On).

APIcast supports the following authentication types in the **auth_type** field to determine the Token Introspection Endpoint and the credentials APIcast uses when calling this endpoint:

use_3scale_oidc_issuer_endpoint

With this setting, APIcast uses the client credentials (Client ID and Client Secret) and the Token Introspection Endpoint from the OpenID Connect Issuer setting configured on the Service Integration page.
APIcast discovers the Token Introspection endpoint from the **token_introspection_endpoint** field the **.well-known/openid-configuration** endpoint of the OpenID Connect issuer returns.

Example 4.1. Authentication type set touse_3scale_oidc_issuer_endpoint

The following is a configuration example if the authentication type is set to **use_3scale_oidc_issuer_endpoint**:

```
"policy_chain": [
…
  {
    "name": "apicast.policy.token_introspection",
    "configuration": {
      "auth_type": "use_3scale_oidc_issuer_endpoint"
    }
  }
…
],
```

client_id+client_secret

This option enables you to specify a different Token Introspection Endpoint, as well as the Client ID and Client Secret APIcast uses to request token information.
When using this option, set the following configuration parameters:

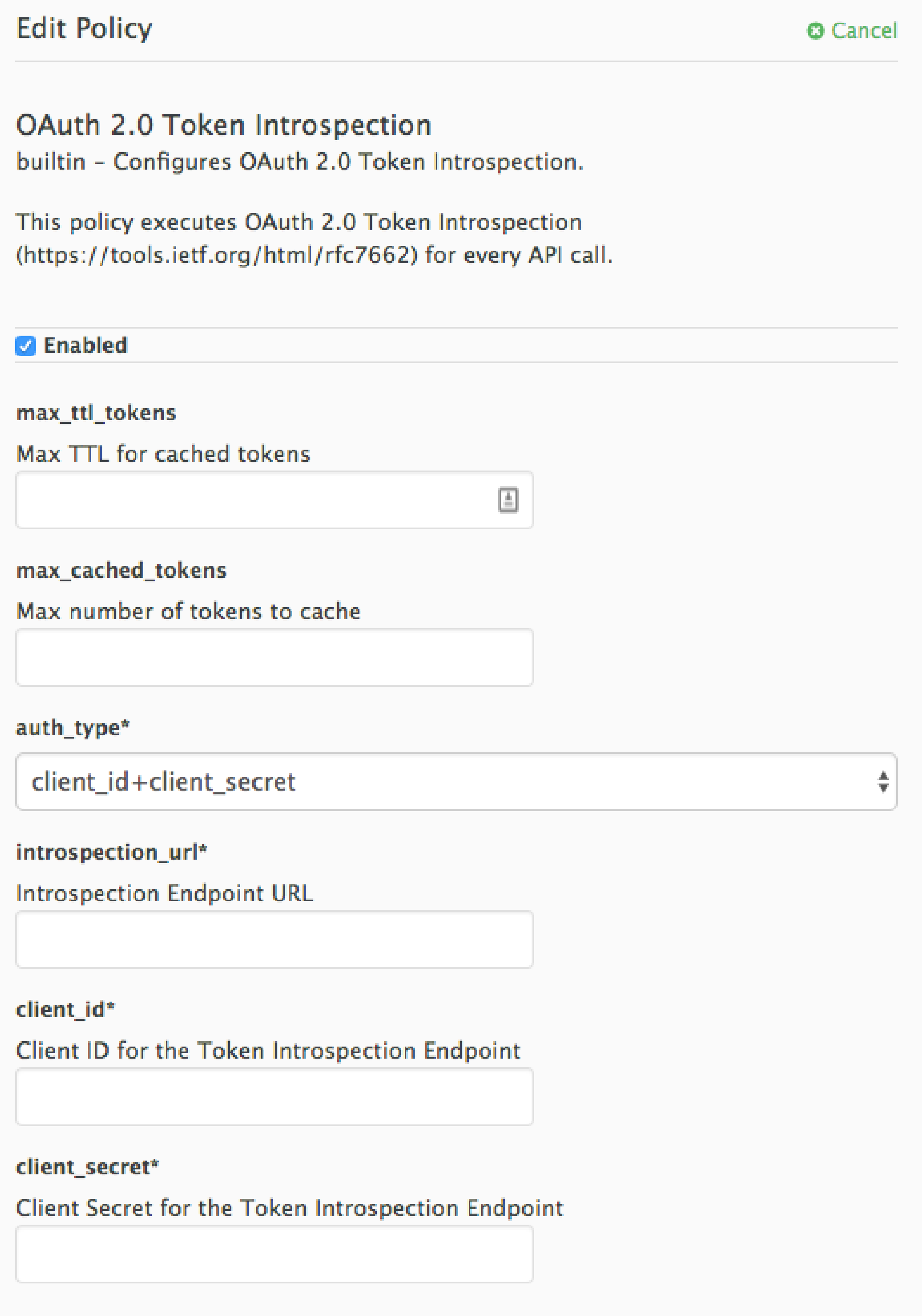
- **client_id**: Sets the Client ID for the Token Introspection Endpoint.
- **client_secret**: Sets the Client Secret for the Token Introspection Endpoint.
- **introspection_url**: Sets the Introspection Endpoint URL.

Example 4.2. Authentication type set toclient_id+client_secret

The following is a configuration example if the authentication type is set to **client_id+client_secret**:

```
"policy_chain": [
…
  {
    "name": "apicast.policy.token_introspection",
    "configuration": {
      "auth_type": "client_id+client_secret",
      "client_id": "myclient",
      "client_secret": "mysecret",
      "introspection_url": "http://red_hat_single_sign-on/token/introspection"
    }
  }
…
],
```

Regardless of the setting in the **auth_type** field, APIcast uses Basic Authentication to authorize the Token Introspection call (**Authorization: Basic *<token>*** header, where *<token>* is Base64-encoded *<client_id>*:*<client_secret>* setting).



Edit Policy

OAuth 2.0 Token Introspection

builtin - Configures OAuth 2.0 Token Introspection.

This policy executes OAuth 2.0 Token Introspection (https://tools.ietf.org/html/rfc7662) for every API call.

☑ Enabled

max_ttl_tokens

Max TTL for cached tokens

max_cached_tokens

Max number of tokens to cache

auth_type*

client_id+client_secret

introspection_url*

Introspection Endpoint URL

client_id*

Client ID for the Token Introspection Endpoint

client_secret*

Client Secret for the Token Introspection Endpoint

The response of the Token Introspection Endpoint contains the **active** attribute. APIcast checks the value of this attribute. Depending on the value of the attribute, APIcast authorizes or rejects the call:

- **true**: The call is authorized
- **false**: The call is rejected with the **Authentication Failed** error

The policy allows enabling caching of the tokens to avoid calling the Token Introspection Endpoint on every call for the same JWT token. To enable token caching for the Token Introspection Policy, set the **max_cached_tokens** field to a value from **0**, which disables the feature, and **10000**. Additionally, you can set a Time to Live (TTL) value from **1** to **3600** seconds for tokens in the **max_ttl_tokens** field.

4.1.12. Referrer policy

The Referrer policy enables the Referrer Filtering feature. When the policy is enabled in the service policy chain, APIcast sends the value of the **Referer** policy of the upcoming request to the Service Management API (AuthRep call) in the **referrer** parameter. For more information on how Referrer Filtering works, see the Referrer Filtering section in **Authentication Patterns**.

4.1.13. RH-SSO/Keycloak Role Check policy

This policy adds role check when used with the OpenID Connect authentication option. This policy verifies realm roles and client roles in the access token issued by Red Hat Single Sign-On. The realm roles are specified when you want to add role check to every client's resources or 3Scale.

Following are the two types of role checks that the **type** property specifies in the policy configuration:

- **whitelist** (default): When **whitelist** is used, APIcast will check if the specified scopes are present in the JWT token and will reject the call if the JWT doesn't have the scopes.
- **blacklist**: When **blacklist** is used, APIcast will reject the calls if the JWT token contains the blacklisted scopes.

It is not possible to configure both checks – **blacklist** and **whitelist** in the same policy, but you can add more than one instances of the **RH-SSO/Keycloak role check** policy to the APIcast policy chain.

You can configure a list of scopes via the **scopes** property of the policy configuration.

Each **scope** object has the following properties:

- **resource**: Resource (endpoint) controlled by the role. This is the same format as Mapping Rules. The pattern matches from the beginning of the string and to make an exact match you must append *$* at the end.
- **resource_type**: This defines how the **resource** value is evaluated.
  - As plain text (**plain**): Evaluates the **resource** value as plain text. Example: **/api/v1/products$**.
  - As Liquid text (**liquid**): Allows using Liquid in the **resource** value. Example: **/resource_{{ jwt.aud }}** manages access to the resource including the Client ID (contained in the JWT **aud** claim).
- **realm_roles**: Use it to check the realm role (see the Realm Roles in Red Hat Single Sign-On documentation).
  The realm roles are present in the JWT issued by Red Hat Single Sign-On.

```
"realm_access": {
  "roles": [
      "<realm_role_A>", "<realm_role_B>"
    ]
  }
```

The real roles must be specified in the policy.

```
"realm_roles": [
  { "name": "<realm_role_A>" }, { "name": "<realm_role_B>" }
]
```

Following are the available properties of each object in the **realm_roles** array:

- **name**: Specifies the name of the role.
- **name_type**: Defines how the name must be evaluated; it can be **plain** or **liquid** (works the same way as for the **resource_type**).
- **client_roles**: Use **client_roles** to check for the particular access roles in the client namespace (see the Client Roles in Red Hat Single Sign-On documentation).
  The client roles are present in the JWT under the **resource_access** claim.

```
  "resource_access": {
    "<client_A>": {
      "roles": [
        "<client_role_A>", "<client_role_B>"
      ]
    },
    "<client_B>": {
      "roles": [
        "<client_role_A>", "<client_role_B>"
      ]
    }
  }
```

Specify the client roles in the policy.

```
"client_roles": [
  { "name": "<client_role_A>", "client": "<client_A>" },
  { "name": "<client_role_B>", "client": "<client_A>" },
  { "name": "<client_role_A>", "client": "<client_B>" },
  { "name": "<client_role_B>", "client": "<client_B>" }
]
```

Following are the available properties of each object in the **client_roles** array:

- **name**: Specifies the name of the role.
- **name_type**: Defines how the **name** value must be evaluated; it can be **plain** or **liquid** (works the same way as for the **resource_type**).
- **client**: Specifies the client of the role. When it is not defined, this policy uses the **aud** claim as the client.
- **client_type**: Defines how the **client** value must be evaluated; it can be **plain** or **liquid** (works the same way as for the **resource_type**).

4.1.14. Routing policy

The Routing policy allows you to route requests to different target endpoints. You can define target endpoints and then you will be able to route incoming requests from the UI to those using regular expressions.

Routing is based on the following rules:

- Request path rule
- Header rule
- Query argument rule
- JSON Web Token (JWT) claim rule

When combined with the APIcast policy, the Routing policy should be placed before the APIcast one in the chain, as the two policies that comes first will output content to the response. When the second gets a change to run its content phase, the request will already be sent to the client, so it will not output anything to the response.

4.1.14.1. Routing rules

- If multiple rules exist, it will apply the first match. The rules will be ordered.
- If no rules apply, it will apply upstream, so the policy will work in combination with APIcast policy, which gets the upstream from the service config, which is a private base URL.

4.1.14.2. Request path rule

This is a configuration that routes to **http://example.com** when the path is **/accounts**:

```
{
  "name": "routing",
  "version": "builtin",
  "configuration": {
    "rules": [
      {
        "url": "http://example.com",
        "condition": {
          "operations": [
            {
              "match": "path",
              "op": "==",
              "value": "/accounts"
            }
          ]
        }
      }
    ]
  }
}
```

4.1.14.3. Header rule

This is a configuration that routes to **http://example.com** when the value of the header **Test-Header** is **123**:

```
{
 "name": "routing",
 "version": "builtin",
 "configuration": {
  "rules": [
   {
    "url": "http://example.com",
    "condition": {
     "operations": [
      {
       "match": "header",
       "header_name": "Test-Header",
       "op": "==",
       "value": "123"
      }
     ]
    }
   }
  ]
 }
}
```

4.1.14.4. Query argument rule

This is a configuration that routes to **http://example.com** when the value of the query argument **test_query_arg** is **123**:

```
{
 "name": "routing",
 "version": "builtin",
 "configuration": {
  "rules": [
   {
    "url": "http://example.com",
    "condition": {
     "operations": [
      {
       "match": "query_arg",
       "query_arg_name": "test_query_arg",
       "op": "==",
       "value": "123"
      }
     ]
    }
   }
  ]
 }
}
```

4.1.14.5. JWT claim rule

To route based on the value of a JWT claim, there needs to be a policy in the chain that validates the JWT and stores it in the context that the policies share.

This is a configuration that routes to **http://example.com** when the value of the JWT claim **test_claim** is **123**:

```
{
  "name": "routing",
  "version": "builtin",
  "configuration": {
    "rules": [
      {
        "url": "http://example.com",
        "condition": {
          "operations": [
            {
              "match": "jwt_claim",
              "jwt_claim_name": "test_claim",
              "op": "==",
              "value": "123"
            }
          ]
        }
      }
    ]
  }
}
```

4.1.14.6. Multiple operations rule

Rules can have multiple operations and route to the given upstream only when all of them evaluate to true (using the 'and' **combine_op**), or when at least one of them evaluates to true (using the 'or' **combine_op**). The default value of **combine_op** is 'and'.

This is a configuration that routes to **http://example.com** when the path of the request is **/accounts** and when the value of the header **Test-Header** is **123**:

```
{
  "name": "routing",
  "version": "builtin",
  "configuration": {
    "rules": [
      {
        "url": "http://example.com",
        "condition": {
          "combine_op": "and",
          "operations": [
            {
              "match": "path",
              "op": "==",
              "value": "/accounts"
            },
            {
              "match": "header",
              "header_name": "Test-Header",
              "op": "==",
              "value": "123"
            }
          ]
        }
      }
    ]
  }
}
```

This is a configuration that routes to **http://example.com** when the path of the request is **/accounts** or when the value of the header **Test-Header** is **123**:

```
{
  "name": "routing",
  "version": "builtin",
  "configuration": {
    "rules": [
      {
        "url": "http://example.com",
        "condition": {
          "combine_op": "or",
          "operations": [
            {
              "match": "path",
              "op": "==",
              "value": "/accounts"
            },
            {
              "match": "header",
              "header_name": "Test-Header",
              "op": "==",
              "value": "123"
            }
          ]
        }
      }
    ]
  }
}
```

4.1.14.7. Combining rules

Rules can be combined. When there are several rules, the upstream selected is one of the first rules that evaluates to true.

This is a configuration with several rules:

```
{
  "name": "routing",
  "version": "builtin",
  "configuration": {
    "rules": [
      {
        "url": "http://some_upstream.com",
        "condition": {
          "operations": [
            {
              "match": "path",
              "op": "==",
              "value": "/accounts"
            }
          ]
        }
      },
      {
        "url": "http://another_upstream.com",
        "condition": {
          "operations": [
            {
              "match": "path",
              "op": "==",
              "value": "/users"
            }
          ]
        }
      }
    ]
  }
}
```

4.1.14.8. Catch-all rules

A rule without operations always matches. This can be useful to define catch-all rules.

This configuration routes the request to **http://some_upstream.com** if the path is **/abc**, routes the request to **http://another_upstream.com** if the path is **/def**, and finally, routes the request to **http://default_upstream.com** if none of the previous rules evaluated to true:

```
{
  "name": "routing",
  "version": "builtin",
  "configuration": {
    "rules": [
      {
        "url": "http://some_upstream.com",
        "condition": {
          "operations": [
            {
              "match": "path",
              "op": "==",
              "value": "/abc"
            }
          ]
        }
      },
      {
        "url": "http://another_upstream.com",
        "condition": {
          "operations": [
            {
              "match": "path",
              "op": "==",
              "value": "/def"
            }
          ]
        }
      },
      {
        "url": "http://default_upstream.com",
        "condition": {
          "operations": []
        }
      }
    ]
  }
}
```

4.1.14.9. Supported operations

The supported operations are **==**, **!=**, and **matches**. The latter matches a string with a regular expression and it is implemented using ngx.re.match

This is a configuration that uses **!=**. It routes to **http://example.com** when the path is not **/accounts**:

```
{
  "name": "routing",
  "version": "builtin",
  "configuration": {
    "rules": [
      {
        "url": "http://example.com",
        "condition": {
          "operations": [
            {
              "match": "path",
              "op": "!=",
              "value": "/accounts"
            }
          ]
        }
      }
    ]
  }
}
```

4.1.14.10. Liquid templating

It is possible to use liquid templating for the values of the configuration. This allows you to define rules with dynamic values if a policy in the chain stores the key **my_var** in the context.

This is a configuration that uses that value to route the request:

```
{
  "name": "routing",
  "version": "builtin",
  "configuration": {
    "rules": [
      {
        "url": "http://example.com",
        "condition": {
          "operations": [
            {
              "match": "header",
              "header_name": "Test-Header",
              "op": "==",
              "value": "{{ my_var }}",
              "value_type": "liquid"
            }
          ]
        }
      }
    ]
  }
}
```

4.1.14.11. Set the host used in the Host header

By default, when a request is routed, the policy sets the Host header using the host of the URL of the rule that matched. It is possible to specify a different host with the **host_header** attribute.

This is a configuration that specifies **some_host.com** as the host of the Host header:

```
{
  "name": "routing",
  "version": "builtin",
  "configuration": {
    "rules": [
      {
        "url": "http://example.com",
        "host_header": "some_host.com",
        "condition": {
          "operations": [
            {
              "match": "path",
              "op": "==",
              "value": "/"
            }
          ]
        }
      }
    ]
  }
}
```

4.1.15. Prometheus Metrics

Prometheus is a stand-alone, open source systems monitoring and alerting toolkit. You can use Prometheus to visualize metrics and alerts for APIcast-managed API services.

Prometheus metrics availability

Prometheus metrics are only accessible with certain APIcast deployment options:

Deployment	Prometheus supported?	Availability?	Resources
APIcast Hosted	No	-	-
APIcast Built-in	Yes	Prometheus is supported on OpenShift 3.11+	Refer to the OpenShift Open Container Platform documentation for guidance on configuration.
APIcast Self-managed	Yes	Prometheus is supported if using OpenShift 3.11+. For Docker environments or other self-managed options, you will need to install and configure Prometheus yourself.	Refer to the OpenShift Open Container Platform documentation for guidance on configuration.

Prometheus metrics list

The following metrics are always available:

Metric	Description	Type	Labels
nginx_http_connections	Number of HTTP connections	gauge	state(accepted,active,handled,reading,total,waiting,writing)
nginx_error_log	APIcast errors	counter	level(debug,info,notice,warn,error,crit,alert,emerg)
openresty_shdict_capacity	Capacity of the dictionaries shared between workers	gauge	dict(one for every dictionary)
openresty_shdict_free_space	Free space of the dictionaries shared between workers	gauge	dict(one for every dictionary)
nginx_metric_errors_total	Number of errors of the Lua library that manages the metrics	counter	-

Metric	Description	Type	Labels
total_response_time_seconds	Time needed to sent a response to the client (in seconds)	histogram	service_id, service_system_name
upstream_response_time_seconds	Response times from upstream servers (in seconds)	histogram	service_id, service_system_name
upstream_status	HTTP status from upstream servers	counter	status, service_id, service_system_name
threescale_backend_calls	Authorize and report requests to the 3scale backend (Apisonator)	counter	endpoint(authrep, auth, report), status(2xx, 4xx, 5xx)

The following metrics are only available when using the 3scale Batcher policy:

Metric	Description	Type	Labels
batching_policy_auths_cache_hits	Hits in the auths cache of the 3scale batching policy	counter	-
batching_policy_auths_cache_misses	Misses in the auths cache of the 3scale batching policy	counter	-

Metrics with no value

If a metric has no value, the metric is hidden. For example, if **nginx_error_log** has no errors to report, the **nginx_error_log** metric is not displayed. It will only be visible once it has a value.

4.1.16. SOAP policy

The SOAP policy matches SOAP action URIs provided in the SOAPAction or Content-Type header of an HTTP request with mapping rules specified in the policy.

Configuration properties

property	description	values	required?

property	description	values	required?
pattern	The **pattern** property allows you to specify a string that APIcast will seek matches for in the SOAPAction URI.	data type: string	yes
metric_system_name	The **metric_system_name** property allows you to specify the 3scale backend metric with which your matched pattern will register a hit.	data type: string, must be a valid metric	yes

Policy object example

```
{
  "name": "soap",
  "version": "builtin",
  "configuration": {
    "mapping_rules": [
      {
        "pattern": "http://example.com/soap#request",
        "metric_system_name": "soap",
        "delta": 1
      }
    ]
  }
}
```

For information on how to configure policies, refer to the Creating a policy chain section of the documentation.

4.1.17. TLS Client Certificate Validation policy

4.1.17.1. About TLS Client Certificate Validation policy

With the TLS Client Certificate Validation policy, APIcast implements a TLS handshake and validates the client certificate against a whitelist. A whitelist contains certificates signed by the Certified Authority (CA) or just plain client certificates. In case of an expired or invalid certificate, the request is rejected and no other policies will be processed.

The client connects to APIcast to send a request and provides a Client Certificate. APIcast verifies the authenticity of the provided certificate in the incoming request according to the policy configuration. APIcast can also be configured to use a client certificate of its own to use it when connecting to the upstream.

4.1.17.2. Setting up APIcast to work with TLS Client Certificate Validation

APIcast needs to be configured to terminate TLS. Follow the steps below to configure the validation of client certificates provided by users on APIcast with the Client Certificate Validation policy.

4.1.17.2.1. Prerequisites:

- You need to have access to a 3scale installation.
- You need to wait for all the deployments to finish.

4.1.17.2.2. Setting up APIcast to work with the policy

To set up APIcast and configure it to terminate TLS, follow these steps:

1. You need to get the access token and deploy APIcast self-managed, as indicated in Deploying APIcast using the OpenShift template.

   NOTE

   APIcast self-managed deployment is required as the APIcast instance needs to be reconfigured to use some certificates for the whole gateway.

2. For testing purposes only, you can use the lazy loader with no cache and staging environment and **--param** flags for the ease of testing

   ```
   oc new-app -f https://raw.githubusercontent.com/3scale/3scale-amp-openshift-
   templates/master/apicast-gateway/apicast.yml --param CONFIGURATION_LOADER=lazy --
   param DEPLOYMENT_ENVIRONMENT=staging --param CONFIGURATION_CACHE=0
   ```

3. Generate certificates for testing purposes. Alternatively, for production deployment, you can use the certificates provided by a Certificate Authority.

4. Create a Secret with TLS certificates

   ```
   oc create secret tls apicast-tls
   --cert=ca/certs/server.crt
   --key=ca/keys/server.key
   ```

5. Mount the Secret inside the APIcast deployment

   ```
   oc set volume dc/apicast --add --name=certificates --mount-path=/var/run/secrets/apicast --
   secret-name=apicast-tls
   ```

6. Configure APIcast to start listening on port 8443 for HTTPS

   ```
   oc set env dc/apicast APICAST_HTTPS_PORT=8443
   APICAST_HTTPS_CERTIFICATE=/var/run/secrets/apicast/tls.crt
   APICAST_HTTPS_CERTIFICATE_KEY=/var/run/secrets/apicast/tls.key
   ```

7. Expose 8443 on the Service

   ```
   oc patch service apicast -p '{"spec":{"ports":[{"name":"https","port":8443,"protocol":"TCP"}]}}'
   ```

8. Delete the default route

```
oc delete route api-apicast-staging
```

9. Expose the **apicast** service as a route

```
oc create route passthrough --service=apicast --port=https --hostname=api-3scale-apicast-
staging.$WILDCARD_DOMAIN
```

> **NOTE**
>
> This step is needed for every API you are going to use and the domain changes for every API.

10. Verify that the previously deployed gateway works and the configuration was saved, by specifying [Your_user_key] in the placeholder.

```
curl https://api-3scale-apicast-staging.$WILDCARD_DOMAIN?user_key=[Your_user_key] -v
--cacert ca/certs/ca.crt
```

4.1.17.3. Configuring TLS Client Certificate Validation in your policy chain

4.1.17.3.1. Prerequisites

- You need 3scale login credentials.
- You need to have configured APIcast with the TLS Client Certificate Validation policy .

4.1.17.3.2. Configuring the policy

1. To add the TLS Client Certificate Validation policy to your API, follow the steps described in Enabling a standard Policy and choose TLS Client Certificate Validation.
2. Click the **TLS Client Certificate Validation** link.
3. To enable the policy, select the **Enabled** checkbox.
4. To add certificates to the whitelist, click the plus **+** icon.
5. Specify the certificate including **-----BEGIN CERTIFICATE-----** and **-----END CERTIFICATE-----**.
6. When you have finished setting up your API with TLS Client Certificate Validation, click **Update Policy**.

Additionally:

- You can add more certificates by clicking the plus **+** icon.
- You can also reorganize the certificates by clicking the up and down arrows.

To save your changes, click **Update & test in Staging Environment**

4.1.17.4. Verifying functionality of the TLS Client Certificate Validation policy

4.1.17.4.1. Prerequisites:

- You need 3scale login credentials.
- You need to have configured APIcast with the TLS Client Certificate Validation policy .

4.1.17.4.2. Verifying policy functionality

You can verify the applied policy by specifying **[Your_user_key]** in the placeholder.

```
curl https://api-3scale-apicast-staging.$WILDCARD_DOMAIN\?user_key\=[Your_user_key] -v --
cacert ca/certs/ca.crt --cert ca/certs/client.crt --key ca/keys/client.key

curl https://api-3scale-apicast-staging.$WILDCARD_DOMAIN\?user_key\=[Your_user_key] -v --
cacert ca/certs/ca.crt --cert ca/certs/server.crt --key ca/keys/server.key

curl https://api-3scale-apicast-staging.$WILDCARD_DOMAIN\?user_key\=[Your_user_key] -v --
cacert ca/certs/ca.crt
```

4.1.17.5. Removing a certificate from the whitelist

4.1.17.5.1. Prerequisites

- You need 3scale login credentials.
- You need to have set up APIcast with the TLS Client Certificate Validation policy .
- You need to have added the certificate to the whitelist, by configuring TLS Client Certificate Validation in your policy chain.

4.1.17.5.2. Removing a certificate

1. Click the **TLS Client Certificate Validation** link.
2. To remove certificates from the whitelist, click the **x** icon.
3. When you have finished removing the certificates, click **Update Policy**.

To save your changes, click **Update & test in Staging Environment**

4.1.17.6. Reference material

For more information about working with certificates, you can refer to Red Hat Certificate System .

4.1.18. Upstream policy

The Upstream policy allows you to parse the Host request header using regular expressions and replace the upstream URL defined in the Private Base URL with a different URL.

For Example:

A policy with a regex **/foo**, and URL field **newexample.com** would replace the URL **https://www.example.com/foo/123/** with **newexample.com**

Policy chain reference:

property	description	values	required?
regex	The **regex** property allows you to specify the regular expression that the Upstream policy will use when searching for a match with the request path.	data type: string, Must be a valid regular expression syntax	yes
url	Using the **url** property, you can specify the replacement URL in the event of a match. Note that the upstream policy does not check whether or not this URL is valid.	data type: string, ensure this is a valid URL	yes

Policy object example

```
{
  "name": "upstream",
  "version": "builtin",
  "configuration": {
    "rules": [
      {
        "regex": "^/v1/.*",
        "url": "https://api-v1.example.com",

      }
    ]
  }
}
```

For information on how to configure policies, refer to the Creating a policy chain section of the documentation.

4.1.19. URL Rewriting policy

The URL Rewriting policy allows you to modify the path of a request and the query string.

When combined with the 3scale APIcast policy, if the URL rewriting policy is placed before the 3scale APIcast policy in the policy chain, the APIcast mapping rules will apply to the modified path. If the URL rewriting policy is placed after APIcast in the policy chain, then the mapping rules will apply to the original path.

The policy supports the following two sets of operations:

- **commands**: List of commands to be applied to rewrite the path of the request.

- **query_args_commands**: List of commands to be applied to rewrite the query string of the request.

4.1.19.1. Commands for rewriting the path

Following are the configuration parameters that each command in the **commands** list consists of:

- **op**: Operation to be applied. The options available are: **sub** and **gsub**. The **sub** operation replaces only the first occurrence of a match with your specified regular expression. The **gsub** operation replaces all occurrences of a match with your specified regular expression. See the documentation for the sub and gsub operations.
- **regex**: Perl-compatible regular expression to be matched.
- **replace**: Replacement string that is used in the event of a match.
- **options** (optional): Options that define how the regex matching is performed. For information on available options, see the ngx.re.match section of the OpenResty Lua module project documentation.
- **break** (optional): When set to true (checkbox enabled), if the command rewrote the URL, it will be the last one applied (all posterior commands in the list will be discarded).

4.1.19.2. Commands for rewriting the query string

Following are configuration parameters that each command in the **query_args_commands** list consists of:

- **op**: Operation to be applied to the query arguments. The following options are available:
  - **add**: Add a value to an existing argument.
  - **set**: Create the arg when not set and replace its value when set.
  - **push**: Create the arg when not set and add the value when set.
  - **delete**: Delete an arg.
- **arg**: The query argument name that the operation is applied on.
- **value**: Specifies the value that is used for the query argument. For value type "liquid" the value should be in the format **{{ variable_from_context }}**. For the **delete** operation the value is not taken into account.
- **value_type** (optional): Defines how the query argument value is evaluated and can either be **plain** for plain text or **liquid** for evaluation as a Liquid template. For more information, see Section 5.1, “Using variables and filters in policies” . If not specified, the type "plain" is used by default.

Example

The URL Rewriting policy is configured as follows:

```
{
  "name": "url_rewriting",
    "version": "builtin",
    "configuration": {
      "query_args_commands": [
        {
          "op": "add",
          "arg": "addarg",
          "value_type": "plain",
          "value": "addvalue"
        },
        {
          "op": "delete",
          "arg": "user_key",
          "value_type": "plain",
          "value": "any"
        },
        {
          "op": "push",
          "arg": "pusharg",
          "value_type": "plain",
          "value": "pushvalue"
        },
        {
          "op": "set",
          "arg": "setarg",
          "value_type": "plain",
          "value": "setvalue"
        }
      ],
      "commands": [
        {
          "op": "sub",
          "regex": "^/api/v\\d+/",
          "replace": "/internal/",
          "options": "i"
        }
      ]
    }
```

The original request URI that is sent to the APIcast:

```
https://api.example.com/api/v1/products/123/details?
user_key=abc123secret&pusharg=first&setarg=original
```

The URI that APIcast sends to the API backend after applying the URL rewriting:

```
https://api-backend.example.com/internal/products/123/details?
pusharg=first&pusharg=pushvalue&setarg=setvalue
```

The following transformations are applied:

1. The substring **/api/v1/** matches the only path rewriting command and it is replaced by **/internal/**.
2. **user_key** query argument is deleted.
3. The value **pushvalue** is added as an additional value to the **pusharg** query argument.

4. The value **original** of the query argument **setarg** is replaced with the configured value **setvalue**.

5. The command **add** was not applied because the query argument **addarg** is not present in the original URL.

For information on how to configure policies, see the Creating a policy chain section of the documentation.

4.1.20. URL Rewriting with Captures policy

The URL Rewriting with Captures policy is an alternative to the Section 4.1.19, "URL Rewriting policy" policy and allows rewriting the URL of the API request before passing it to the API backend.

The URL Rewriting with Captures policy retrieves arguments in the URL and uses their values in the rewritten URL.

The policy supports the **transformations** configuration parameter. It is a list of objects that describe which transformations are applied to the request URL. Each tranformation object consist of two properties:

- **match_rule**: This rule is matched to the incoming request URL. It can contain named arguments in the **{nameOfArgument}** format; these arguments can be used in the rewritten URL. The URL is compared to **match_rule** as a regular expression. The value that matches named arguments must contain only the following characters (in PCRE regex notation): **[\w-.~%!$&'()*,;=@:]**. Other regex tokens can be used in the **match_rule** expression, such as **^** for the beginning of the string and **$** for the end of the string.

- **template**: The template for the URL that the original URL is rewritten with; it can use named arguments from the **match_rule**.

The query parameters of the original URL are merged with the query parameters specified in the **template**.

Example

The URL Rewriting with Captures is configured as follows:

```
{
  "name": "rewrite_url_captures",
  "version": "builtin",
  "configuration": {
    "transformations": [
      {
        "match_rule": "/api/v1/products/{productId}/details",
        "template": "/internal/products/details?id={productId}&extraparam=anyvalue"
      }
    ]
  }
}
```

The original request URI that is sent to the APIcast:

```
https://api.example.com/api/v1/products/123/details?user_key=abc123secret
```

The URI that APIcast sends to the API backend after applying the URL rewriting:

```
https://api-backend.example.com/internal/products/details?
user_key=abc123secret&extraparam=anyvalue&id=123
```

4.2. ENABLING A STANDARD POLICY

Perform the following steps to enable policies in the Admin Portal:

1. Log in to 3scale.
2. Navigate to the **API service**.
3. From **[your_API_name] > Integration > Configuration**, select **edit APIcast configuration**.
4. Under the **POLICIES** section, click **add policy**.
5. Select the policy you want to add and fill out the required fields.
6. Click the **Update and test in Staging Environment**button to save the policy chain.

4.3. CREATING CUSTOM APICAST POLICIES

You can create custom APIcast policies entirely or modify the standard policies.

In order to create custom policies, you must understand the following:

- Policies are written in Lua.
- Policies must adhere to and be placed in the proper file directory.
- Policy behavior is affected by how they are placed in a policy chain.
- The interface to add custom policies is fully supported, but not the custom policies themselves.

4.4. ADDING CUSTOM POLICIES TO APICAST

If you have created custom policies, you must add them to APIcast. How you do this depends on where APIcast is deployed.

You can add custom policies to the following APIcast self-managed deployments:

- APIcast built-in gateways as part of a 3scale on-premises deployment on OpenShift
- APIcast on OpenShift and the Docker containerized environment

You cannot add custom policies to APIcast hosted.

> **WARNING**
>
> Never make policy changes directly onto a production gateway. Always test your changes.

4.4.1. Adding custom policies to the built-in APIcast

To add custom APIcast policies to an on-premises deployment, you must build an OpenShift image containing your custom policies and add it to your deployment. Red Hat 3scale provides a sample repository you can use as a framework to create and add custom policies to an on-premises deployment.

This sample repository contains the correct directory structure for a custom policy, as well as a template which creates an image stream and BuildConfigs for building a new APIcast OpenShift image containing any custom policies you create.

> **WARNING**
>
> When you build **apicast-custom-policies**, the build process "pushes" a new image to the **amp-apicast:latest** tag. When there is an image change on this image stream tag (**:latest**), both the *apicast-staging* and the *apicast-production* tags, by default, are configured to automatically start new deployment. To avoid any disruptions to your production service (or staging, if you prefer) it is recommended to disable automatic deployment (*"Automatically start a new deployment when the image changes"* checkbox), or configure a different image stream tag for production (e.g. **amp-apicast:production**).

Follow these steps to add a custom policy to an on-premises deployment:

1. Fork the https://github.com/3scale/apicast-example-policy [public repository with the policy example] or create a private repository with its content. You need to have the code of your custom policy available in a Git repository for OpenShift to build the image. Note that in order to use a private Git repository, you must set up the secrets in OpenShift.

2. Clone the repository locally, add the implementation for your policy, and push the changes to your Git repository.

3. Update the **openshift.yml** template. Specifically, change the following parameters:

   a. **spec.source.git.uri: https://github.com/3scale/apicast-example-policy.git** in the policy BuildConfig – change it to your Git repository location.

   b. **spec.source.images[0].paths.sourcePath: /opt/app-root/policies/example** in the custom policies BuildConfig - change **example** to the name of the custom policy that you have added under the **policies** directory in the repository.

   c. Optionally, update the OpenShift object names and image tags. However, you must ensure that the changes are coherent (example: **apicast-example-policy** BuildConfig builds and pushes the **apicast-policy:example** image that is then used as a source by the **apicast-custom-policies** BuildConfig. So, the tag should be the same).

4. Create the OpenShift objects by running the command:

   ```
   oc new-app -f openshift.yml --param AMP_RELEASE=2.5.0
   ```

5. In case the builds do not start automatically, run the following two commands. In case you changed it, replace **apicast-example-policy** with your own BuildConfig name (e.g. **apicast-<name>-policy**). Wait for the first command to complete before you execute the second one.

   ```
   oc start-build apicast-example-policy
   oc start-build apicast-custom-policies
   ```

If the build-in APIcast images have a trigger on them tracking the changes in the **amp-apicast:latest** image stream, the new deployment for APIcast will start. After **apicast-staging** has restarted, go to the Integration page on the admin portal, and click the **Add Policy** button to see your custom policy listed. After selecting and configuring it, click **Update & test in Staging Environment**to make your custom policy work in the staging APIcast.

4.4.2. Adding custom policies to APIcast on another OpenShift Container Platform

You can add custom policies to APIcast on OpenShift Container Platform (OCP) by fetching APIcast images containing your custom policies from the the Integrated OpenShift Container Registry .

Add custom policies to APIcast on another OpenShift Container Platform

1. Add policies to APIcast built-in

2. If you are not deploying your APIcast gateway on your primary OpenShift cluster, establish access to the internal registry on your primary OpenShift cluster.

3. Download the 3scale 2.5 APIcast OpenShift template.

4. To modify the template, replace the default **image** directory with the full image name in your internal registry.

   ```
   image: <registry>/<project>/amp-apicast:latest
   ```

5. Deploying APIcast using the OpenShift template , specifying your customized image:

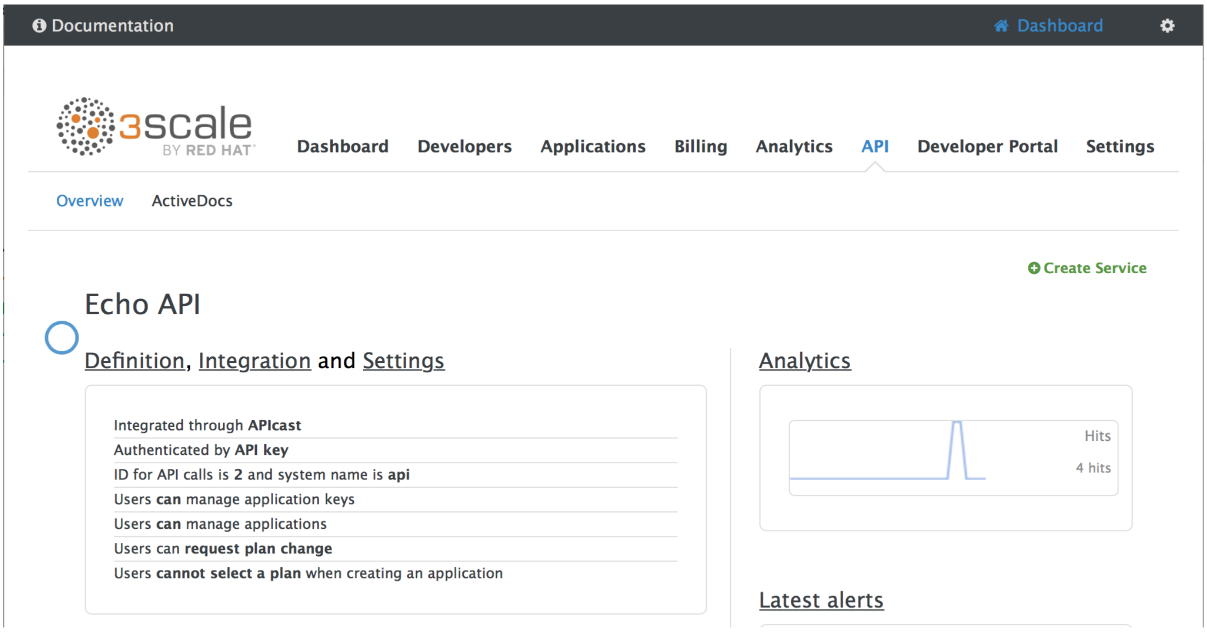
   ```
   oc new-app -f customizedApicast.yml
   ```

4.5. CREATING A POLICY CHAIN IN 3SCALE API MANAGEMENT

Create a policy chain in 3scale API Management as part of your APIcast gateway configuration. Follow these steps to modify the policy chain in the UI:

1. Log in to your AMP

2. Navigate to the API service



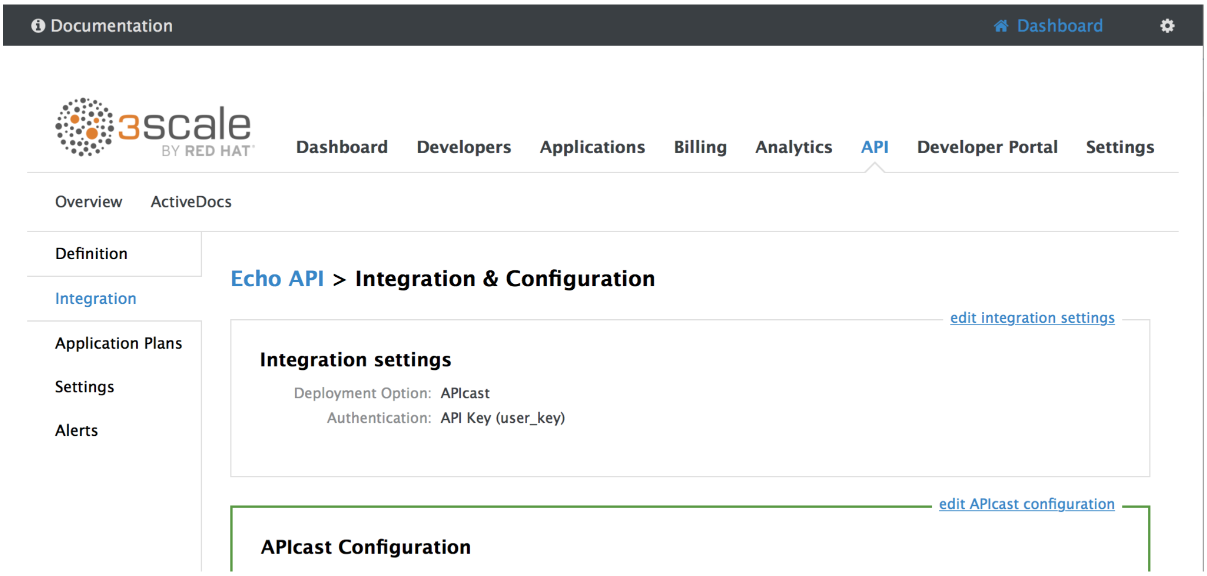
Echo API

Definition, Integration and Settings

- Integrated through **APIcast**
- Authenticated by **API key**
- ID for API calls is **2** and system name is **api**
- Users **can** manage application keys
- Users **can** manage applications
- Users can **request plan change**
- Users **cannot select a plan** when creating an application

Analytics

3. From **[your_API_name] > Integration > Configuration**, select **edit APIcast configuration**



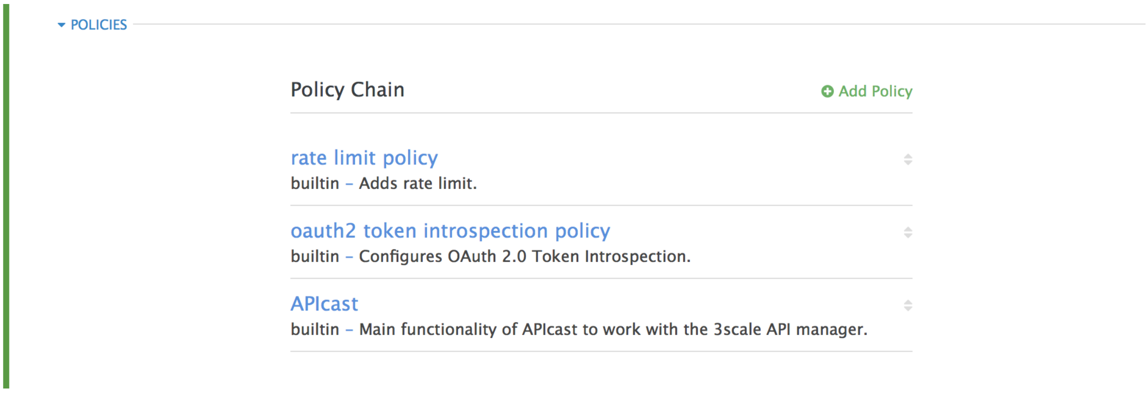
Echo API > Integration & Configuration

Integration settings

Deployment Option: APIcast
Authentication: API Key (user_key)

APIcast Configuration

4. Under the **POLICIES** section, use the arrow icons to reorder policies in the policy chain. Always place the **APIcast** policy last in the policy chain.



Policy Chain

- rate limit policy
  builtin – Adds rate limit.
- oauth2 token introspection policy
  builtin – Configures OAuth 2.0 Token Introspection.
- APIcast
  builtin – Main functionality of APIcast to work with the 3scale API manager.

5. Click the **Update and test in Staging Environment**button to save the policy chain

4.6. CREATING A POLICY CHAIN JSON CONFIGURATION FILE

If you are using a native deployment of APIcast, you can create a JSON configuration file to control your policy chain outside of the AMP.

A JSON configuration file policy chain contains a JSON array composed of the following information:

- the **services** object with an **id** value that specifies which service the policy chain applies to by number
- the **proxy** object, which contains the policy_chain and subsequent objects
- the **policy_chain** object, which contains the values that define the policy chain
- individual **policy** objects which specify both **name** and **configuration** data necessary to identify the policy and configure policy behavior

The following is an example policy chain for a custom policy **sample_policy_1** and the API introspection standard policy **token_introspection**:

```
{
  "services":[
    {
      "id":1,
      "proxy":{
        "policy_chain":[
          {
            "name":"sample_policy_1", "version": "1.0",
            "configuration":{
              "sample_config_param_1":["value_1"],
              "sample_config_param_2":["value_2"]
            }
          },
          {
            "name": "token_introspection", "version": "builtin",
            "configuration": {
              introspection_url:["https://tokenauthorityexample.com"],
              client_id:["exampleName"],
              client_secret:["secretexamplekey123"]
          },
          {
             "name": "apicast", "version": "builtin",
          }
        ]
      }
    }
  ]
}
```

All policy chains must include the built-in policy **apicast**. Where you place APIcast in the policy chain will affect policy behavior.

CHAPTER 5. INTEGRATING A POLICY CHAIN WITH APICAST NATIVE DEPLOYMENTS

For native APIcast deployments, you can integrate a custom policy chain by specifying a configuration file using the **THREESCALE_CONFIG_FILE** environment variable. The following example specifies the config file **example.json**:

```
THREESCALE_CONFIG_FILE=example.json bin/apicast
```

5.1. USING VARIABLES AND FILTERS IN POLICIES

Some Section 4.1, “APIcast standard policies” support Liquid templating that allows using not only plain string values, but also variables that are present in the context of the request.

To use a context variable, wrap its name in **{{** and **}}**, example: **{{ uri }}**. If the variable is an object, you can also access its attributes, for example: **{{ somevar.attr }}**.

Following are the standard variables that are available in all the policies:

- **uri**: The path of the request without query parameters (the value of the embedded NGINX variable **$uri**).
- **host**: The host of the request (the value of the embedded NGINX variable **$host**).
- **remote_addr**: The IP address of the client (the value of the embedded NGINX variable **$remote_addr**).
- **headers**: The object containing the request headers. Use **{{headers['Some-Header']}}** to get a specific header value.
- **http_method**: The request method: GET, POST, etc.

The variables are available in the **context** of the request. Policies can add extra variables to the context. These variables can be used by the same or other policies in the policy chain, provided that the phase where they are used is executed after the phase where the variable was added. It can also be the same phase if the variable is used in the policy that appears after the policy in which the variable was added.

Following are some examples of variables that the standard 3scale APIcast policy adds to the context:

- **jwt**: A parsed JSON payload of the JWT token (for OpenID Connect authentication).
- **credentials**: An object that holds the application credentials. Example: **"app_id": "972f7b4f"**, **"user_key": "13b668c4d1e10eaebaa5144b4749713f"**.
- **service**: An object that holds the configuration for the service that the current request is handled by. Example: the service ID would be available as {{ service.id }}.

For a full list of objects and values available in the context, see the Section 4.1.9, “Liquid Context Debug policy”).

The variables are used with the help of Liquid templates. Example: **{{ remote_addr }}**, **{{ headers['Some-Header'] }}**, **{{ jwt.aud }}**. The policies that support variables for the values have a special parameter, usually with the **_type** suffix (example: **value_type**, **name_type**, etc.) that accepts two values: "plain" for plain text and "liquid" for liquid template.

APIcast also supports Liquid filters that can be applied to the variables' values. The filters apply NGINX functions to the value of the Liquid variable.

The filters are placed within the variable output tag **{{ }}**, following the name of the variable or the literal value by a pipe character | and the name of the filter. Examples: **{{ 'username:password' | encode_base64 }}**, **{{ uri | escape_uri }}**.

Some filters do not require parameters, so you can use an empty string instead of the variable. Example: **{{ '' | utctime }}** will return the current time in UTC time zone.

Filters can be chained as follows: **{{ variable | function1 | function2 }}**. Example: **{{ '' | utctime | escape_uri }}**.

Following is the list of the available functions:

- escape_uri
- unescape_uri
- encode_base64
- decode_base64
- crc32_short
- crc32_long
- hmac_sha1
- md5
- md5_bin
- sha1_bin
- quote_sql_str
- today
- time
- now
- localtime
- utctime
- cookie_time
- http_time
- parse_http_time

CHAPTER 6. APICAST ENVIRONMENT VARIABLES

APIcast environment variables allow you to modify behavior for APIcast. The following values are supported environment variables:

NOTE

- Unsupported or deprecated environment variables are not listed
- Some environment variable functionality may have moved to APIcast policies

APICAST_BACKEND_CACHE_HANDLER

Values: strict | resilient

Default: strict

Deprecated: Use the Caching policy instead.

Defines how the authorization cache behaves when backend is unavailable. Strict will remove cached application when backend is unavailable. Resilient will do so only on getting authorization denied from backend.

APICAST_CONFIGURATION_CACHE

Values: a number

Default: 0

Specifies the interval (in seconds) that the configuration will be stored for. The value should be set to 0 (not compatible with boot value of **APICAST_CONFIGURATION_LOADER**) or more than 60. For example, if **APICAST_CONFIGURATION_CACHE** is set to 120, the gateway will reload the configuration from the API manager every 2 minutes (120 seconds). A value < 0 disables reloading.

APICAST_CONFIGURATION_LOADER

Values: boot | lazy

Default: lazy

Defines how to load the configuration. Boot will request the configuration to the API manager when the gateway starts. Lazy will load it on demand for each incoming request (to guarantee a complete refresh on each request **APICAST_CONFIGURATION_CACHE** should be 0).

APICAST_CUSTOM_CONFIG

Deprecated: Use policies instead.

Defines the name of the Lua module that implements custom logic overriding the existing APIcast logic.

APICAST_ENVIRONMENT

Default:

Value: string[:]

Example: production:cloud-hosted

Double colon (**:**) separated list of environments (or paths) APIcast should load. It can be used instead of **-e** or **---environment** parameter on the CLI and for example stored in the container image as default environment. Any value passed on the CLI overrides this variable.

APICAST_EXTENDED_METRICS

Default: false

Value: boolean

Example: "true"

Enables additional information on Prometheus metrics. The following metrics have the **service_id** and **service_system_name** labels which provide more in-depth details about APIcast:

- **total_response_time_seconds**
- **upstream_response_time_seconds**
- **upstream_status**

APICAST_LOG_FILE

Default: stderr

Defines the file that will store the OpenResty error log. It is used by **bin/apicast** in the **error_log** directive. Refer to NGINX documentation for more information. The file path can be either absolute, or relative to the prefix directory (apicast by default).

APICAST_LOG_LEVEL

Values: debug | info | notice | warn | error | crit | alert | emerg

Default: warn

Specifies the log level for the OpenResty logs.

APICAST_ACCESS_LOG_FILE

Default: stdout

Defines the file that will store the access logs.

APICAST_OIDC_LOG_LEVEL

Values: debug | info | notice | warn | error | crit | alert | emerg

Default: err

Allows to set the log level for the logs related to OpenID Connect integration.

APICAST_MANAGEMENT_API

Values:

- **disabled**: completely disabled, just listens on the port
- **status**: only the /**status**/ endpoints enabled for health checks
- **debug**: full API is open

The Management API is powerful and can control the APIcast configuration. You should enable the debug level only for debugging.

APICAST_MODULE

Default: apicast

Deprecated: Use policies instead.

Specifies the name of the main Lua module that implements the API gateway logic. Custom modules can override the functionality of the default **apicast.lua** module. See an example of how to use modules.

APICAST_PATH_ROUTING

Values:

- **true** or **1** for true
- **false**, **0** or empty for false

When this parameter is set to *true*, the gateway will use path-based routing in addition to the default host-based routing. The API request will be routed to the first service that has a matching mapping rule, from the list of services for which the value of the **Host** header of the request matches the *Public Base URL*.

APICAST_POLICY_LOAD_PATH

Default: **APICAST_DIR/policies**

Value: string[:]

Example: ~**/apicast/policies:$PWD/policies**

Double colon (**:**) separated list of paths where APIcast should look for policies. It can be used to first load policies from a development directory or to load examples.

APICAST_PROXY_HTTPS_CERTIFICATE_KEY

Default:

Value: string

Example: /home/apicast/my_certificate.key

The path to the key of the client SSL certificate.

APICAST_PROXY_HTTPS_CERTIFICATE

Default:

Value: string

Example: /home/apicast/my_certificate.crt

The path to the client SSL certificate that APIcast will use when connecting with the upstream. Notice that this certificate will be used for all the services in the configuration.

APICAST_PROXY_HTTPS_PASSWORD_FILE

Default:

Value: string

Example: /home/apicast/passwords.txt

Path to a file with passphrases for the SSL cert keys specified with **APICAST_PROXY_HTTPS_CERTIFICATE_KEY**.

APICAST_PROXY_HTTPS_SESSION_REUSE

Default: on

Values:

- **on**: reuses SSL sessions.
- **off**: does not reuse SSL sessions.

APICAST_REPORTING_THREADS

Default: 0

Value: integer >= 0

Experimental: Under extreme load might have unpredictable performance and lose reports.

Value greater than 0 is going to enable out-of-band reporting to backend. This is a new **experimental** feature for increasing performance. Client won't see the backend latency and everything will be processed asynchronously. This value determines how many asynchronous reports can be running simultaneously before the client is throttled by adding latency.

APICAST_RESPONSE_CODES

Values:

- **true** or **1** for true
- **false**, **0** or empty for false

Default: <empty> (*false*)

When set to true, APIcast will log the response code of the response returned by the API backend in 3scale. In some plans this information can later be consulted from the 3scale admin portal. Find more information about the Response Codes feature on the 3scale support site.

APICAST_SERVICES_LIST

Value: a comma-separated list of service IDs

Used to filter the services configured in the 3scale API Manager, and only use the configuration for specific services in the gateway, discarding those services' IDs that are not specified in the list. Service IDs can be found on the Admin Portal's Dashboard > **APIs** page, tagged as *ID for API calls* .

APICAST_SERVICE_${ID}_CONFIGURATION_VERSION

Replace **${ID}** with the actual Service ID. The value should be the configuration version you can see in the configuration history on the Admin Portal. Setting it to a particular version will prevent it from auto-updating and will always use that version.

APICAST_WORKERS

Default: auto

Values: *number* | auto

This is the value that will be used in the nginx **worker_processes** directive. By default, APIcast uses **auto**, except for the development environment where **1** is used.

BACKEND_ENDPOINT_OVERRIDE

URI that overrides backend endpoint from the configuration. Useful when deploying outside OpenShift deployed AMP. **Example**: **https://backend.example.com**.

OPENSSL_VERIFY

Values:

- **0**, **false**: disable peer verification
- **1**, **true**: enable peer verification

Controls the OpenSSL Peer Verification. It is off by default, because OpenSSL can't use system certificate store. It requires custom certificate bundle and adding it to trusted certificates.

It is recommended to use https://github.com/openresty/lua-nginx-module#lua_ssl_trusted_certificate and point to to certificate bundle generated by export-builtin-trusted-certs.

RESOLVER

Allows to specify a custom DNS resolver that will be used by OpenResty. If the **RESOLVER** parameter is empty, the DNS resolver will be autodiscovered.

THREESCALE_CONFIG_FILE

Path to the JSON file with the configuration for the gateway. The configuration can be downloaded from the 3scale admin portal using the URL: **<schema>://<admin-portal-domain>/admin/api/nginx/spec.json** (Example: **https://account-admin.3scale.net/admin/api/nginx/spec.json**).

When the gateway is deployed using Docker, the file has to be injected to the docker image as a read only volume, and the path should indicate where the volume is mounted, i.e. path local to the docker container.

You can find sample configuration files in examples folder.

It is **required** to provide either **THREESCALE_PORTAL_ENDPOINT** or **THREESCALE_CONFIG_FILE** (takes precedence) for the gateway to run successfully.

THREESCALE_DEPLOYMENT_ENV

Values: staging | production

Default: production

The value of this environment variable will be used to define the environment for which the configuration will be downloaded from 3scale (Staging or Production), when using new APIcast.

The value will also be used in the header **X-3scale-User-Agent** in the authorize/report requests made to 3scale Service Management API. It is used by 3scale just for statistics.

THREESCALE_PORTAL_ENDPOINT

URI that includes your password and portal endpoint in the following format: **<schema>://<password>@<admin-portal-domain>**. The **<password>** can be either the provider key or an access token for the 3scale Account Management API. <admin-portal-domain> is the URL used to log into the admin portal.

Example: **https://access-token@account-admin.3scale.net**.

When **THREESCALE_PORTAL_ENDPOINT** environment variable is provided, the gateway will download the configuration from 3scale on initializing. The configuration includes all the settings provided on the Integration page of the API(s).

It is **required** to provide either **THREESCALE_PORTAL_ENDPOINT** or **THREESCALE_CONFIG_FILE** (takes precedence) for the gateway to run successfully.

OPENTRACING_TRACER

Example: **jaeger**

This environment variable controls which tracing library will be loaded, right now, there's only one opentracing tracer available, **jaeger**.

If empty, opentracing support will be disabled.

OPENTRACING_CONFIG

This environment variable is used to determine the config file for the opentracing tracer, if **OPENTRACING_TRACER** is not set, this variable will be ignored.

Each tracer has a default configuration file: * **jaeger: conf.d/opentracing/jaeger.example.json**

You can choose to mount a different configuration than the provided by default by setting the file path using this variable.

Example: **/tmp/jaeger/jaeger.json**

OPENTRACING_HEADER_FORWARD

Default: **uber-trace-id**

This environment variable controls the HTTP header used for forwarding opentracing information, this HTTP header will be forwarded to upstream servers.

APICAST_HTTPS_PORT

Default: no value

Controls on which port APIcast should start listening for HTTPS connections. If this clashes with HTTP port it will be used only for HTTPS.

APICAST_HTTPS_CERTIFICATE

Default: no value

Path to a file with X.509 certificate in the PEM format for HTTPS.

APICAST_HTTPS_CERTIFICATE_KEY

Default: no value

Path to a file with the X.509 certificate secret key in the PEM format.

all_proxy, ALL_PROXY

Default: no value **Value:** string **Example:** **http://forward-proxy:80**

Defines a HTTP proxy to be used for connecting to services if a protocol-specific proxy is not specified. Authentication is not supported.

http_proxy, HTTP_PROXY

Default: no value **Value:** string **Example:** **http://forward-proxy:80**

Defines a HTTP proxy to be used for connecting to HTTP services. Authentication is not supported.

https_proxy, HTTPS_PROXY

Default: no value **Value:** string **Example:** **https://forward-proxy:443**

Defines a HTTP proxy to be used for connecting to HTTPS services. Authentication is not supported.

no_proxy, NO_PROXY

Default: no value **Value:** string\[,<string>\]; *Example: foo,bar.com,.extra.dot.com

Defines a comma-separated list of hostnames and domain names for which the requests should not be proxied. Setting to a single * character, which matches all hosts, effectively disables the proxy.