Red Hat 3scale API Management 2.12

Administering the API Gateway

Intermediate to advanced goals to manage your installation.

Last Updated: 2022-11-28

Red Hat 3scale API Management 2.12 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

MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright's message.

Administering the API Gateway helps you to apply intermediate to advanced configuration features to your 3scale installation. For basic details regarding installation, refer to Installing 3scale.

PART I. THE API GATEWAY

CHAPTER 1. INTRODUCTION TO ADVANCED OPERATION OF 3SCALE APICAST API GATEWAY

The introduction to advanced operation of 3scale APIcast will help you to adjust the configuration of access to your API.

1.1. PUBLIC BASE URL FOR CALLS TO 3SCALE APIS

The **Public Base URL** is the URL that your API consumers use to make requests to your API product, which is exposed publicly with 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 of the environments provided (staging and production) on a domain name you are managing. This URL should be different from the one for 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.

NOTE

The Public Base URL that you specify must use a port that is available in your OpenShift cluster. By default, the OpenShift router listens for connections only on the standard HTTP and HTTPS ports (80 and 443). If you want users to connect to your API over some other port, work with your OpenShift administrator to enable the port.

APIcast accepts calls to only the hostname specified in the Public Base URL. For example, if you specify **https://echo-api.3scale.net:443** as the Public Base URL, the correct call would be:

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

In case you do not have a public domain for your API, you can use the APIcast IP address in the requests, but you still need to specify a value in the *Public Base URL* field even if the domain is not real. 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"
```

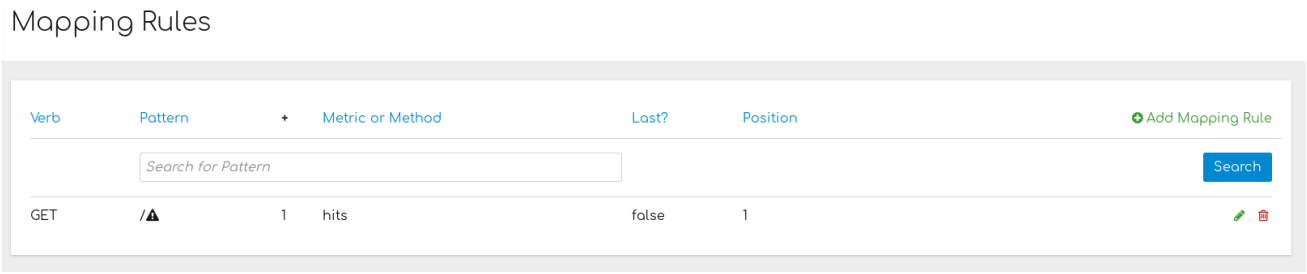
If you are deploying on a local machine, you can specify "localhost" as the domain, so the Public Base URL would look like **http://localhost:80**, and then you can make requests like this:

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

If you have multiple API products, set the Public Base URL appropriately for each product. APIcast routes the requests based on the hostname.

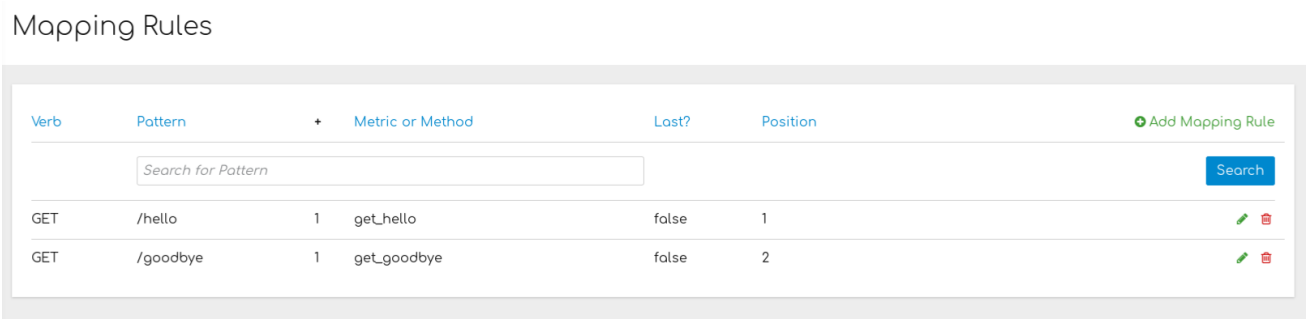
1.2. HOW APICAST APPLIES MAPPING RULES FOR CAPTURING USAGE OF 3SCALE APIS

Based on the requests to your API, mapping rules define the metrics or designate the methods for which you want to capture API usage. The following is an example of a mapping rule:



This rule means that any **GET** requests that start with / increment the metric **hits** by 1. This rule matches any request to your API. While this is a valid mapping rule, it is too generic and often leads to double counts if you add more specific mapping rules.

The following mapping rules for the Echo API show more specific examples:



Mapping rules work at the API product and API backend levels.

- Mapping rules at the product level.
  - The mapping rule takes precedence. This means that the product mapping rule is the first one to be evaluated.
  - The mapping rule is always evaluated, independent of which backend receives the redirected traffic.
- Mapping rules at the backend level.
  - When you add mapping rules to a backend, these are added to all the products bundling said backend.
  - The mapping rule is evaluated after the mapping rules defined at the product level.
  - The mapping rule is evaluated only if the traffic is redirected to the same backend the mapping rule belongs to.
  - The path of the backend for a product is automatically prepended to each mapping rule of the backend bundled to said product.

Example of mapping rules with products and backends

The following example shows mapping rules for a product with one backend.

- The **Echo API** backend:
  - Has the private endpoint: **https://echo-api.3scale.net**
  - Contains 2 mapping rules with the following patterns:

```
/hello
/bye
```

- The **Cool API** product:
  - Has this public endpoint: **https://cool.api**
  - Uses the **Echo API** backend with this routing path: **/echo**.
- Mapping rules with the following patterns are automatically part of the **Cool API** product:

```
/echo/hello
/echo/bye
```

  - This means that a request sent to the public URL **https://cool.api/echo/hello** is redirected to **https://echo-api.3scale.net/hello**.
  - Similarly, a request sent to **https://cool.api/echo/bye** redirects to **https://echo-api.3scale.net/bye**.

Now consider an additional product called **Tools For Devs** using the same **Echo API** backend.

- The **Tools For Devs** product:
  - Has this public endpoint: **https://dev-tools.api**
  - Uses the **Echo API** backend with the following routing path: **/tellmeback**.
- Mapping rules with the following patterns are automatically part of the **Tools For Devs** product:

```
/tellmeback/hello
/tellmeback/bye
```

  - Therefore, a request sent to the public URL **https://dev-tools.api/tellmeback/hello** is redirected to **https://echo-api.3scale.net/hello**.
  - Similarly, a request sent to **https://dev-tools.api/tellmeback/bye** redirects to **https://echo-api.3scale.net/bye**.
- If you add a mapping rule with the **/ping** pattern to the **Echo API** backend, both products - **Cool API** and **Tools For Devs**- are affected:
  - **Cool API** has a mapping rule with this pattern: **/echo/ping**.
  - **Tools For Devs** has a mapping rule with this pattern: **/tellmeback/ping**.

Matching of mapping rules

3scale applies mapping rules based on prefixes. The notation follows the OpenAPI and ActiveDocs specifications:

- A mapping rule must start with a forward slash (/).
- Perform a match on the path over a literal string, which is a URL, for example, **/hello**.
  - The mapping rule, once you have saved it, will cause requests to the URL string you have set and invoke metrics or methods you have defined around each mapping rule.

- Mapping rules can include parameters on the query string or in the body, for example, /**{word}?value={value}**).
- APIcast fetches the parameters in the following ways:
  - **GET** method: From the query string.
  - **POST**, **DELETE**, or **PUT** method: From the body.
- Mapping rules can contain named wildcards, for example, /**{word}**. This rule matches anything in the placeholder **{word}**, which makes requests such as /**morning** match the mapping rule. Wildcards can appear between slashes or between a slash and a dot. Parameters can also include wildcards.
- By default, all mapping rules are evaluated from first to last, according to the sort order you specified. If you add a rule /**v1**, it matches requests whose paths start with /**v1**, for example, /**v1**/**word** or /**v1**/**sentence**.
- You can add a dollar sign (**$**) to the end of a pattern to specify exact matching. For example, /**v1**/**word** matches only /**v1**/**word** requests, and does not match /**v1**/**word**/**hello** requests. For exact matching, you must also ensure that the default mapping rule that matches everything (/) has been disabled.
- More than one mapping rule can match the request path, but if none matches, the request is discarded with an HTTP 404 status code.

Mapping rules workflow

Mapping rules have the following workflow:

- You can define a new mapping rule at any time. See Defining mapping rules.
- Mapping rules are 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 the trash icon.
- All modifications and deletions are saved when you promote the changes in **Integration** > **Configuration**.

Stop other mapping rules

After processing one or more mapping rules, to stop processing other mapping rules, select **Last?** when creating a new mapping rule. For example, suppose you have the following mapping rules defined in **API Integration Settings** and you have different metrics associated with each rule:

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

When calling with **(get)** /**path**/**to**/**example**/**search**, after matching the rule, APIcast stops processing the remaining mapping rules and stops incrementing their metrics.

1.3. HOW APICAST HANDLES APIS THAT HAVE CUSTOM REQUIREMENTS

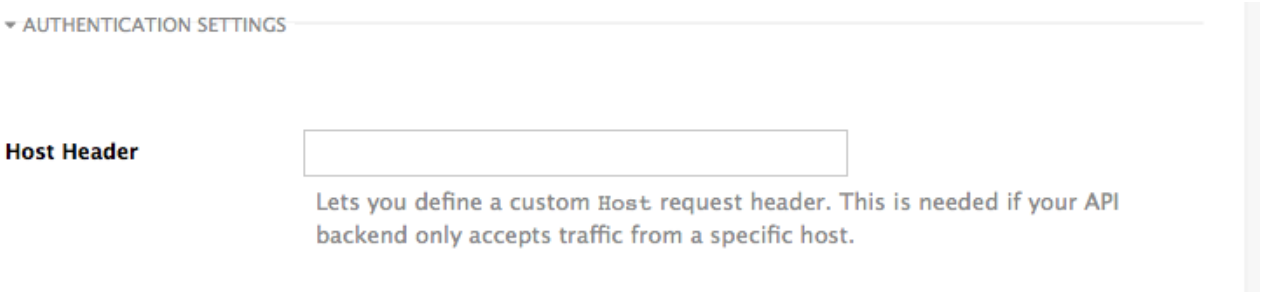
There are special cases that require custom APIcast configuration so that API consumers can successfully call the API.

Host header

This option is only needed for those API products that reject traffic unless the **Host** header matches the expected one. In these cases, having a gateway in front of your API product causes problems because the **Host** is the one of the gateway, for example, **xxx-yyy.staging.apicast.io**.

To avoid this issue, you can define the host your API product expects in the **Host Header** field in the *Authentication Settings*: **[Your_product_name] > Integration > Settings**

The result is that the hosted APIcast instance rewrites the host specification in the request call.



Protecting your API backend

After you have APIcast working in production, you might want to restrict direct access to your API product to only those calls that specify a secret token that you specify. Do this by setting the APIcast **Secret Token**. See Advanced APIcast configuration for information on how to set it up.

Using APIcast with private APIs

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

- Self-managed APIcast 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 product 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. However, doing this means that you cannot take advantage of the staging environment. Test calls will not be successful because the staging APIcast instance is hosted by 3scale, which does not have access to your private API backend. After you deploy APIcast in your production environment, if the configuration is correct, APIcast works as expected.

1.4. CONFIGURING APICAST TO USE OPENTRACING

OpenTracing is an API specification and method used to profile and monitor microservices. APIcast version 3.3 and later includes OpenTracing libraries and the Jaeger Tracer library.

Prerequisites

- Each external request must have a unique request ID attached. This is usually in an HTTP header.
- Each service must forward the request ID to other services.

- Each service must output the request ID in the logs.
- Each service must record additional information, such as the start and end time of the request.
- Logs must be aggregated, and provide a way to parse them via HTTP request ID.

Procedure

1. Ensure the **OPENTRACING_TRACER** environment variable is set to **jaeger**. If this is empty, OpenTracing is disabled.
2. Set the **OPENTRACING_CONFIG** environment variable to specify the default configuration file of your tracer. See the following example **jaeger.example.json** file.
3. Optional: Set the **OPENTRACING_HEADER_FORWARD** environment variable according to your OpenTracing configuration.

Verification

To test if the integration is properly working, check whether traces are reported in the Jaeger tracing interface.

Additional resources

- APIcast environment variables
- Upstream project for OpenTracing and Jaeger integration

1.5. INSTALLING JAEGER ON YOUR OPENSHIFT INSTANCE

3scale API providers can use OpenTracing with Jaeger to trace and troubleshoot calls to an API. To do this, install Jaeger on the OpenShift instance on which 3scale is running.

WARNING

Jaeger is a third-party component, which 3scale does not provide support for, with the exception of uses with APIcast. The following instructions are provided as a reference example only, and are not suitable for production use.

Procedure

1. Install the Jaeger all-in-one template in the current namespace:

```
oc process -f https://raw.githubusercontent.com/jaegertracing/jaeger-openshift/master/all-in-one/jaeger-all-in-one-template.yml | oc create -f -
```

2. Create a Jaeger configuration file **jaeger_config.json** and add the following:

```
{
    "service_name": "apicast",
```

```
    "disabled": false,
    "sampler": {
      "type": "const",
      "param": 1
    },
    "reporter": {
      "queueSize": 100,
      "bufferFlushInterval": 10,
      "logSpans": false,
      "localAgentHostPort": "jaeger-agent:6831"
    },
    "headers": {
      "jaegerDebugHeader": "debug-id",
      "jaegerBaggageHeader": "baggage",
      "TraceContextHeaderName": "uber-trace-id",
      "traceBaggageHeaderPrefix": "testctx-"
    },
    "baggage_restrictions": {
        "denyBaggageOnInitializationFailure": false,
        "hostPort": "127.0.0.1:5778",
        "refreshInterval": 60
    }
}
```

- A **sampler** constant of 1 indicates that you want to sample all requests.
- The location and queue size of the **reporter** are required.
- Under **headers**, the **TraceContextHeaderName** entry is required to track requests

3. Create a ConfigMap from your Jaeger configuration file and mount it into APIcast:

```
oc create configmap jaeger-config --from-file=jaeger_config.json
oc set volumes dc/apicast --add -m /tmp/jaeger/ --configmap-name jaeger-config
```

4. Enable OpenTracing and Jaeger with the configuration you just added:

```
oc set env deploymentConfig/apicast OPENTRACING_TRACER=jaeger
OPENTRACING_CONFIG=/tmp/jaeger/jaeger_config.json
```

5. Find the URL that the Jaeger interface is running on:

```
oc get route
(…) jaeger-query-myproject.127.0.0.1.nip.io
```

6. Open the Jaeger interface from the previous step, which shows data being populated from Openshift Health checks.

7. Add OpenTracing and Jaeger support to your backend APIs so that you can see the complete request trace. This varies in each back end, depending on the frameworks and languages used. As a reference example, see Using OpenTracing with Jaeger to collect Application Metrics in Kubernetes.

Additional resources

- Jaeger on OpenShift development setup
- Jaeger on OpenShift production setup
- Distributed tracing on OpenShift Service Mesh

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.

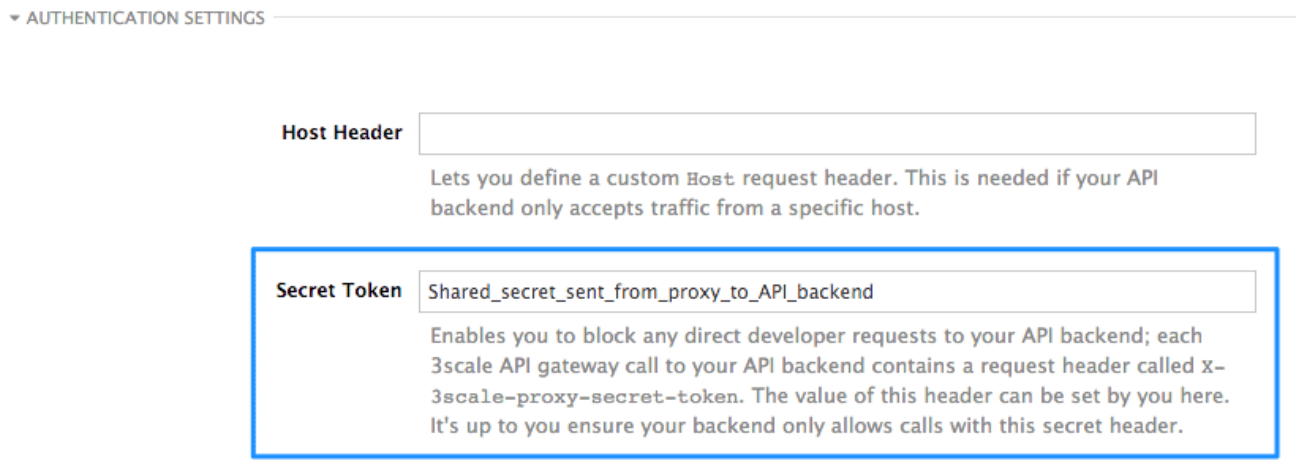
For more information on available commands, see Docker commands reference.

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.



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.

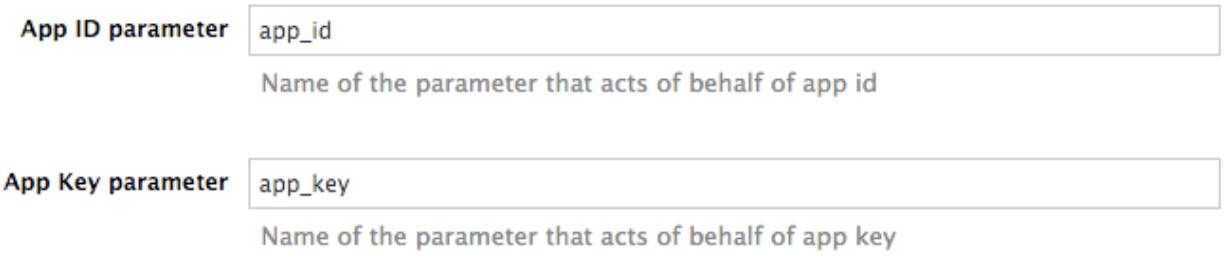
Your API backend must have a public resolvable domain for the gateway to work, so anyone who knows your API backend can bypass the credentials checking. This should not be a problem because the API gateway in the staging environment is not meant for production use, but it is always better to have a fence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

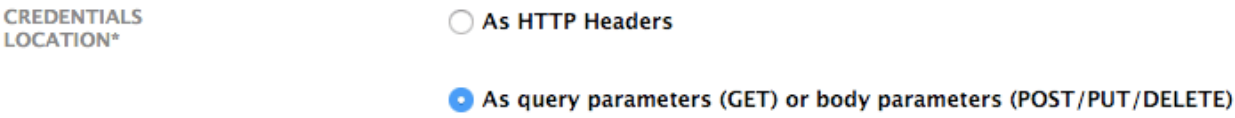
However, you might want to use different credential names in your API. In this case, you need to set custom names for the **user_key** if you are using the API key mode:

Alternatively, for the **app_id** and **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.



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. CONFIGURING ERROR MESSAGES

This section describes how to configure APIcast error messages.

As a proxy, 3scale APIcast manages requests in the following ways:

- If there are no errors, APIcast passes the request from the client to the API back end server, and returns the API response to the client without modifications. In case you want to modify the responses, you can use the Header Modification policy.
- If the API responds with an error message, such as **404 Not Found** or **400 Bad Request**, APIcast returns the message to the client. However, if APIcast detects other errors such as **Authentication missing**, APIcast sends an error message and terminates the request.

Hence, you can configure these error messages to be returned by APIcast:

- Authentication failed: This error means that the API request does not contain the valid credentials, whether due to fake credentials or because the application is temporarily suspended. Additionally, this error is generated when the metric is disabled, meaning its value is **0**.
- 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.
- 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.

- Usage limit exceeded: This error means that the client reached its rate limits for the requested endpoint. A client may reach more than one rate limit if the request matches multiple mapping rules.

To configure errors, follow these steps:

1. Navigate from **[Your_product_name] > Integration > Settings**
2. Under **Gateway response**, choose the error type you want to configure.
3. Specify values for these fields:
   - Response Code: The three-digit HTTP response code.
   - Content-type: The value of the **Content-Type** header.
   - Response Body: The value of the response message body.
4. To save your changes, click **Update Product**.

3.4. CONFIGURATION HISTORY

Every time you click **Promote v.[n] to Staging APIcast**, where **[n]** represents the version number, 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:

1. Navigate from **[Your_product_name] > Integration > Configuration**
2. Click the **Configuration history** link, located next to the environment of your interest: *Staging APIcast* or *Production APIcast*.

Note that it is not possible to automatically roll back to previous versions. Instead, you access to a history of all your configuration versions with their associated JSON files. 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 a policy chain, and creating custom APIcast policies.

4.1. STANDARD POLICIES TO CHANGE DEFAULT 3SCALE APICAST BEHAVIOR

3scale provides built-in, standard policies that are units of functionality that modify how APIcast processes requests and responses. You can enable, disable, or configure policies to control how they modify APIcast.

For details, see Enabling policies in the 3scale Admin Portal . 3scale provides the following standard policies:

- 3scale Auth Caching
- 3scale Batcher
- 3scale Referrer
- Anonymous Access
- Camel Service
- Conditional Policy
- Content Caching
- CORS Request Handling
- Custom Metrics
- Echo
- Edge Limiting
- Header Modification
- HTTP Status Code Overwrite
- HTTP2 Endpoint
- IP Check
- JWT Claim Check
- Liquid Context Debug
- Logging

- Maintenance Mode
- NGINX Filter
- OAuth 2.0 Mutual TLS Client Authentication
- OAuth 2.0 Token Introspection
- On Fail
- Proxy Service
- Rate Limit Headers
- Response Request Content Limits
- Retry
- RH-SSO/Keycloak Role Check
- Routing
- SOAP
- TLS Client Certificate Validation
- TLS Termination
- Upstream
- Upstream Connection
- Upstream Mutual TLS
- URL Rewriting
- URL Rewriting With Captures
- Websocket

4.1.1. Enabling policies in the 3scale Admin Portal

In the Admin Portal, you can enable one or more policies for each 3scale API product.

Prerequisites

- A 3scale API product

Procedure

1. Log in to 3scale.
2. In the Admin Portal dashboard, select the API product for which you want to enable the policy.
3. From **[your_product_name]**, navigate to **Integration > Policies**.
4. Under the **POLICIES** section, click **Add policy**.

5. Select the policy you want to add and enter values in any required fields.

6. Click **Update Policy Chain** to save the policy chain.

4.1.2. 3scale Auth Caching

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

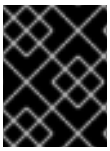
```
{
  "name": "caching",
  "version": "builtin",
```

```
  "configuration": {
    "caching_type": "allow"
  }
}
```

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

4.1.3. 3scale Batcher

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 that APIcast receives.



IMPORTANT

To use this policy, you must place **3scale Batcher** before the **3scale APIcast** policy in the policy chain.

The 3scale Batcher policy caches authorization statuses and batches usage reports, thereby significantly reducing the number of requests to the 3scale backend. With the 3scale Batcher policy you can improve APIcast performance by reducing latency and increasing throughput.

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 in which APIcast is not sending calls to 3scale. During this time 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.

- Set the **auths_ttl** parameter to a value other than **0**. Setting **auths_ttl** to a value of **0** would update the authorization counter the first time the request is cached, resulting in rate limits not being effective.

- **batch_report_seconds**: Sets the frequency of batch reports APIcast sends to the 3scale backend. The default value is **10** seconds.

4.1.4. 3scale Referrer

The 3scale Referrer policy enables the *Referrer Filtering* feature. When the policy is enabled in the service policy chain, APIcast sends the value of the 3scale Referrer policy to the *Service Management API* as an upwards *AuthRep* call. The value of the 3scale Referrer policy is sent in the **referrer** parameter in the call.

For more information on how *Referrer Filtering* works, see the Referrer Filtering section under **Authentication Patterns**.

4.1.5. Anonymous Access

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 Access policy supports services with only 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.

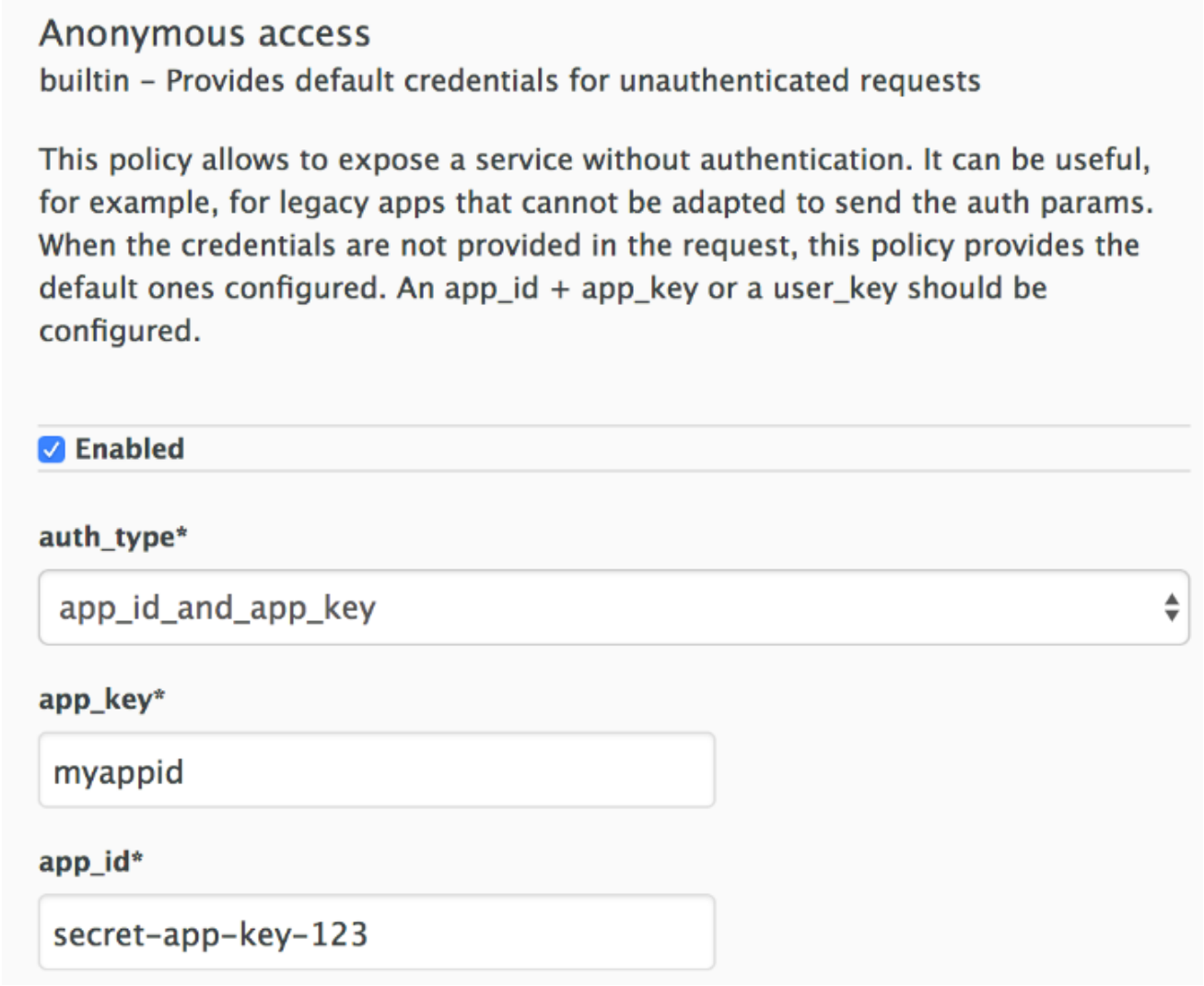
NOTE

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.

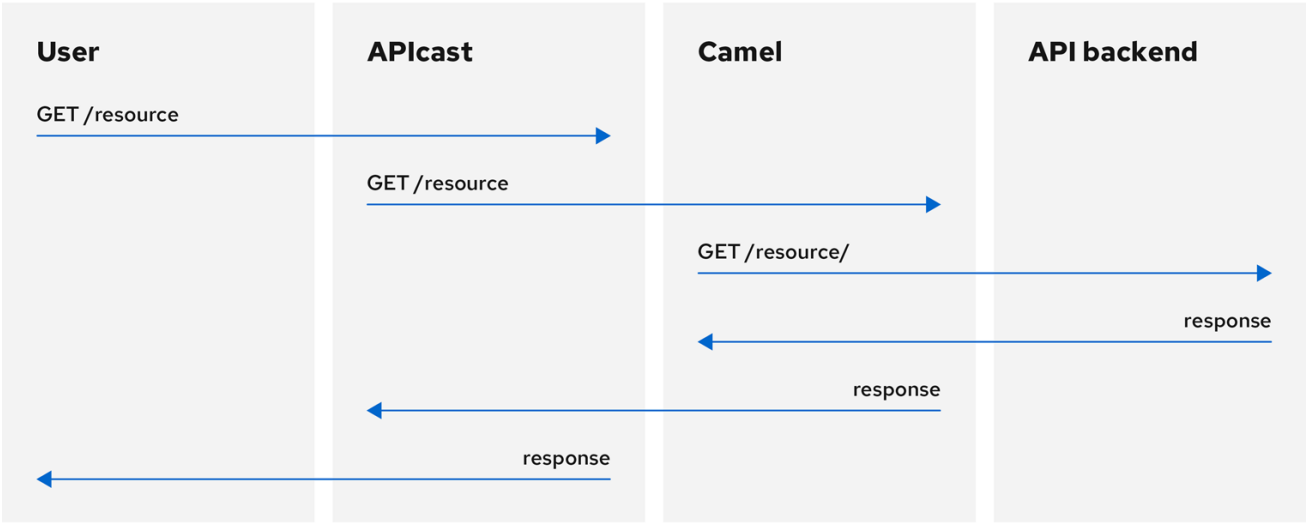
Figure 4.1. Anonymous Access policy



4.1.6. Camel Service

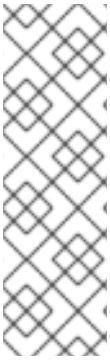
You can use the Camel Service policy to define an *HTTP proxy* where the 3scale traffic is sent over the defined Apache Camel proxy. In this case, Camel works as a reverse HTTP proxy, where APIcast sends the traffic to Camel, and Camel then sends the traffic on to the API backend.

The following example shows the traffic flow:



All APIcast traffic sent to the 3scale backend does not use the Camel proxy. This policy only applies to the Camel proxy and the communication between APIcast and API backend.

If you want to send all traffic through a proxy, you must use an **HTTP_PROXY** environment variable.



NOTE

- The Camel Service policy disables all load-balancing policies, and traffic is sent to the Camel proxy.
- If the **HTTP_PROXY**, **HTTPS_PROXY**, or **ALL_PROXY** parameters are defined, this policy overwrites those values.
- The proxy connection does not support authentication. You use the Header Modification policy for authentication.

Configuration

The following example shows the policy chain configuration:

```
"policy_chain": [
  {
    "name": "apicast.policy.apicast"
  },
  {
    "name": "apicast.policy.camel",
    "configuration": {
        "all_proxy": "http://192.168.15.103:8080/",
        "http_proxy": "http://192.168.15.103:8080/",
        "https_proxy": "http://192.168.15.103:8443/"
    }
  }
]
```

The **all_proxy** value is used if **http_proxy** or **https_proxy** is not defined.

Example use case

The Camel Service policy is designed to apply more fine-grained policies and transformation in 3scale using Apache Camel. This policy supports integration with Apache Camel over HTTP and HTTPS. For more details, see Chapter 6, *Transforming 3scale message content using policy extensions in Fuse* .

For details on using a generic HTTP proxy policy, see Section 4.1.25, "Proxy Service".

Example project

See the **camel-netty-proxy** example available from the Camel proxy policy on GitHub. This example project shows an HTTP proxy that transforms the response body from the API backend to uppercase.

4.1.7. Conditional Policy

The Conditional Policy is different from other APIcast policies as it contains a chain of policies. It defines a condition that is evaluated on each *nginx phase*, for example, *access*, *rewrite*, *log* and so on. When the condition is true, the Conditional Policy runs that phase for each of the policies that it contains in its chain.

IMPORTANT

The APIcast Conditional Policy is a Technology Preview feature only. Technology Preview features are not supported with Red Hat production service level agreements (SLAs) and might not be functionally complete. Red Hat does not recommend using them in production. These features provide early access to upcoming product features, enabling customers to test functionality and provide feedback during the development process. For more information about the support scope of Red Hat Technology Preview features, see Technology Preview Features Support Scope .

The following example assumes that the Conditional Policy defines the following condition: **the request method is POST**.

```
APIcast --> Caching --> Conditional --> Upstream

                    |
                    v

                Headers

                    |
                    v

              URL Rewriting
```

In this case, when the request is a **POST**, the order of execution for each phase will be the following:

1. APIcast
2. Caching
3. Headers
4. URL Rewriting
5. Upstream

When the request is not **POST**, the order of execution for each phase will be the following:

1. APIcast
2. Caching
3. Upstream

Conditions

The condition that determines whether to run the policies in the chain of the Conditional Policy can be expressed using *JSON* and uses liquid templating.

This example checks whether the request path is /**example_path**:

```
{
  "left": "{{ uri }}",
  "left_type": "liquid",
  "op": "==",
  "right": "/example_path",
  "right_type": "plain"
}
```

Both the left and right operands can be evaluated either as liquid or as plain strings. Plain strings are the default.

You can combine the operations with **and** or **or**. This configuration checks the same as the previous example plus the value of the **Backend** header:

```
{
  "operations": [
    {
      "left": "{{ uri }}",
      "left_type": "liquid",
      "op": "==",
      "right": "/example_path",
      "right_type": "plain"
    },
    {
      "left": "{{ headers['Backend'] }}",
      "left_type": "liquid",
      "op": "==",
      "right": "test_upstream",
      "right_type": "plain"
    }
  ],
  "combine_op": "and"
}
```

For more details see, policy config schema.

Supported variables in liquid

- uri
- host

- remote_addr
- headers['Some-Header']

The updated list of variables can be found here: ngx_variable.lua

This example executes the upstream policy when the **Backend** header of the request is *staging*:

```
{
  "name":"conditional",
  "version":"builtin",
  "configuration":{
    "condition":{
      "operations":[
        {
          "left":"{{ headers['Backend'] }}",
          "left_type":"liquid",
          "op":"==",
          "right":"staging"
        }
      ]
    },
    "policy_chain":[
      {
        "name":"upstream",
        "version": "builtin",
        "configuration":{
          "rules":[
            {
              "regex":"/",
              "url":"http://my_staging_environment"
            }
          ]
        }
      }
    ]
  }
}
```

4.1.8. Content Caching

The Content Caching policy allows you to enable and disable caching based on customized conditions. These conditions can only be applied on the client request, where upstream responses cannot be used in the policy.

When the Content Caching policy is in a policy chain, APIcast converts a **HEAD** request to a **GET** request before sending the request upstream. If you do not want this conversion, do not add the Content Caching policy to a policy chain.

If a cache-control header is sent, it will take priority over the timeout set by APIcast.

The following example configuration will cache the response if the Method is GET.

Example configuration

```
{
  "name": "apicast.policy.content_caching",
  "version": "builtin",
  "configuration": {
    "rules": [
      {
        "cache": true,
        "header": "X-Cache-Status-POLICY",
        "condition": {
          "combine_op": "and",
          "operations": [
            {
              "left": "{{method}}",
              "left_type": "liquid",
              "op": "==",
              "right": "GET"
            }
          ]
        }
      }
    ]
  }
}
```

Supported configuration

- Set the Content Caching policy to disabled for any of the following methods: **POST**, **PUT**, or **DELETE**.
- If one rule matches, and it enables the cache, the execution will be stopped and it will not be disabled. Sort by priority is important here.

Upstream response headers

The NGINX **proxy_cache_valid** directive information can only be set globally, with the **APICAST_CACHE_STATUS_CODES** and **APICAST_CACHE_MAX_TIME**. If your upstream requires a different behavior regarding timeouts, use the **Cache-Control** header.

4.1.9. CORS Request Handling

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

- Allowed headers
- Allowed methods
- Allowed origin headers
- Allowed credentials
- Max age

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
max_age	The **max_age** property allows you to set how long the results of a preflight request can be cached	data type: integer	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",
   "max_age" : 200
  }
}
```

For information about how to configure policies, see Modifying policy chains in the 3scale Admin Portal .

4.1.10. Custom Metrics

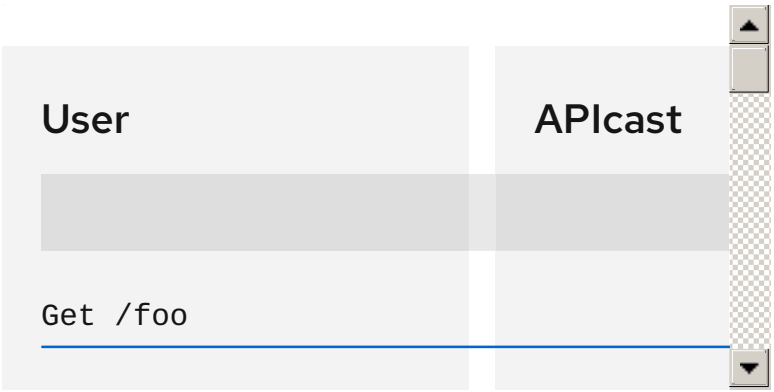
The Custom Metrics policy adds the availability to add metrics after the response sent by the upstream API. The main use case for this policy is to add metrics based on response code status, headers, or different NGINX variables.

Limitations of custom metrics

- When authentication happens before the request is sent to the upstream API, a second call to the back end will be made to report the new metrics to the upstream API.
- This policy does not work with batching policy.
- Metrics need to be created in the Admin Portal before the policy will push the metric values.

Examples for request flows

The following chart shows the request flow example of when authentication is not cached, as well as the flow when authentication is cached.



Configuration examples

This policy increments the metric error by the header increment if the upstream API returns a 400 status:

```
{
  "name": "apicast.policy.custom_metrics",
  "configuration": {
    "rules": [
      {
        "metric": "error",
        "increment": "{{ resp.headers['increment'] }}",
        "condition": {
          "operations": [
            {
```

```
            "right": "{{status}}",
            "right_type": "liquid",
            "left": "400",
            "op": "=="
          }
        ],
        "combine_op": "and"
      }
    }
  ]
 }
}
```

This policy increments the hits metric with the status_code information if the upstream API return a 200 status:

```
{
 "name": "apicast.policy.custom_metrics",
 "configuration": {
  "rules": [
   {
    "metric": "hits_{{status}}",
    "increment": "1",
    "condition": {
     "operations": [
      {
       "right": "{{status}}",
       "right_type": "liquid",
       "left": "200",
       "op": "=="
      }
     ],
     "combine_op": "and"
    }
   }
  ]
 }
}
```

4.1.11. Echo

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

property	description	values	required?
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 about how to configure policies, see the Creating a policy chain in 3scale section of the documentation.

4.1.12. Edge Limiting

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. This is 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.

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, which builds on the average number of requests plus a maximum burst size.
- **fixed_window_limiters**: Based on a fixed window of time: last n seconds.
- **connection_limiters**: Based on the concurrent number of connections.

You can scope any limit by service or globally.

Limit definition

The limits have a key that encodes the entities that are used to define the limit, such as an IP address, a service, an endpoint, an identifier, the value for a specific header, and other entities. This key is specified in the **key** parameter of the limiter.

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

- **name**: Defines the name of the key. It must be unique in the scope.
- **scope**: 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**: Definies how the **name** value is evaluated:
  - As plain text (**plain**)
  - As Liquid (**liquid**)

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

- **leaky_bucket_limiters**: **rate**, **burst**.
  - **rate**: Defines how many requests can be made per second without a delay.
  - **burst**: 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**: Defines the number of seconds to delay the connections that exceed the limit.

Examples

- Allow 10 requests per minute to service_A:

```
{
  "key": { "name": "service_A" },
  "count": 10,
  "window": 60
}
```

- Allow 100 connections with bursts of 10 with a delay of 1 second:

```
{
  "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.

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. For 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” .

Applying conditions

Each limiter must have a condition that defines when the limiter is applied. The condition is specified in the **condition** property of the limiter.

condition is defined by the following properties:

- **combine_op**: The boolean operator applied to the list of operations. Values of **or** and **and** are supported.
- **operations**: 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"
    }
  ]
}
```

Configuring storage of rate limit counters

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

Error handling

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

- **limits_exceeded_error**: Specifies 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**: Specifies how to handle the error, with following options:
    - **exit**: Stops processing request and returns an error message.
    - **log**: Completes processing request and returns output logs.
- **configuration_error**: Specifies 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**: Specifies how to handle the error, with following options:
    - **exit**: Stops processing request and returns an error message.
    - **log**: Completes processing request and returns output logs.

4.1.13. Header Modification

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 about how to configure policies, see the Creating a policy chain in 3scale section of the documentation.

4.1.14. HTTP Status Code Overwrite

As an API provider, you can add the HTTP Status Code Overwrite policy to an API product. This policy lets you change an upstream response code to a response code that you specify. 3scale applies the HTTP Status Code Overwrite policy to the response codes sent from the upstream service. In other words, when an API that 3scale exposes returns a code that does not fit your situation, you can configure the HTTP Status Code Overwrite policy to change that code to a response code that is meaningful for your application.

In a policy chain, any policies that produce response codes that you want to change must be before the HTTP Status Code Overwrite policy. If there are no policies that produce Status Codes that you want to change then the policy chain position of the HTTP Status Code Overwrite policy does not matter.

In the Admin Portal, add the HTTP Status Code Overwrite policy to a product's policy chain. In the policy chain, click the policy to specify the upstream response code that you want to change and the response code that you want returned instead. Click the plus sign for each additional upstream response code that you want to overwrite. For example, you could use the HTTP Status Code Overwrite policy to change upstream **201**, "Created", response codes, to **200**, "OK", response codes.

Another example of a response code that you might want to change is the response when a content limit is exceeded. The upstream might return **413**, payload too large, when a response code of **414**, request-URI too long, would be more helpful.

An alternative to adding the HTTP Status Code Overwrite policy in the Admin Portal is to use the 3scale API with a policy chain configuration file.

Example

The following JSON configuration in your policy chain configuration file would overwrite two upstream response codes.

```
{
  "name": "statuscode_overwrite",
  "version": "builtin",
  "configuration": {
    "http_statuses": [
      {
        "upstream": 200,
        "apicast": 201
      },
      {
        "upstream": 413,
        "apicast": 414
      }
    ]
  }
}
```

4.1.15. HTTP2 Endpoint

The HTTP2 Endpoint policy enables HTTP/2 protocol connections between consumer applications that send requests and APIcast. When the HTTP2 Endpoint policy is in a product's policy chain, the entire communications flow, from a consumer application that makes a request, to APIcast, to the upstream service, can use the HTTP/2 protocol.

When the HTTP2 Endpoint policy is in a policy chain:

- Request authentication must be by means of JSON web tokens or **App_ID** and **App_Key** pairs. API key authentication is not supported.
- The HTTP2 Endpoint policy must be before the 3scale APIcast policy.
- The upstream service's backends can implement HTTP/1.1 plaintext or Transport Layer Security (TLS).

- The policy chain must also include the TLS Termination policy.

4.1.16. IP Check

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 about how to configure policies, see the Creating a policy chain in 3scale section of the documentation.

4.1.17. JWT Claim Check

Based on JSON Web Token (JWT) claims, the JWT Claim Check policy allows you to define new rules to block resource targets and methods.

About JWT Claim Check policy

In order to route based on the value of a JWT claim, you need a policy in the chain that validates the JWT and stores the claim in the context that the policies share.

If the JWT Claim Check policy is blocking a resource and a method, the policy also validates the JWT operations. Alternatively, in case that the method resource does not match, the request continues to the backend API.

Example: In case of a GET request, the JWT needs to have the role claim as admin, if not the request will be denied. On the other hand, any non GET request will not validate the JWT operations, so POST resource is allowed without JWT constraint.

```
{
  "name": "apicast.policy.jwt_claim_check",
  "configuration": {
     "error_message": "Invalid JWT check",
     "rules": [
        {
           "operations": [
              {"op": "==", "jwt_claim": "role", "jwt_claim_type": "plain", "value": "admin"}
           ],
           "combine_op":"and",
           "methods": ["GET"],
           "resource": "/resource",
           "resource_type": "plain"
        }
     ]
  }
}
```

Configuring JWT Claim Check policy in your policy chain

To configure the JWT Claim Check policy in your policy chain:

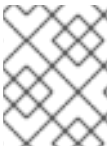
- You need to have access to a 3scale installation.
- You need to wait for all the deployments to finish.

Configuring the policy

1. To add the JWT Claim Check policy to your API, follow the steps described in Enabling policies in the 3scale Admin Portal and choose JWT Claim Check.

2. Click the **JWT Claim Check** link.
3. To enable the policy, select the **Enabled** checkbox.
4. To add rules, click the plus **+** icon.
5. Specify the **resource_type**.
6. Choose the operator.
7. Indicate the **resource** controlled by the rule.
8. To add the allowed methods, click the plus **+** icon.
9. Type the error message to show to the user when traffic is blocked.
10. When you have finished setting up your API with JWT Claim Check, click **Update Policy**. You can add more resource types and allowed methods by clicking the plus **+** icon in the corresponding section.
11. Click **Update Policy Chain** to save your changes.

4.1.18. Liquid Context Debug



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",
    ...
  }
  ...
}
```

4.1.19. Logging

The Logging policy has two purposes:

- To enable and disable access log output.
- To create a custom access log format for each service and be able to set conditions to write custom access log.

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 7, *APIcast environment variables*.

Additionally, the Logging policy has these features:

- This policy only supports the **enable_access_logs** configuration parameter.
- To enable the access logs, select the **enable_access_logs** parameter or disable the Logging policy.
- To disable access logging for an API:
  1. Enable the policy.
  2. Clear the **enable_access_logs** parameter
  3. Click the **Submit** button.
- By default, this policy is not enabled in policy chains.

4.1.19.1. Configuring the logging policy for all APIs

The APICAST_ENVIRONMENT can be used to load a configuration that makes the policy apply globally to all API products. The following is an example of how this can be achieved. APICAST_ENVIRONMENT is used to point to the path of a file, which depending on the type of deployment, template or operator, needs to be provided differently.

To configure the logging policy globally, consider the following depending on your deployment-type:

- For template-based deployments: it is a requirement to mount the file on the container via ConfigMap and VolumeMount.
- For 3scale operator-based deployments:
  - Previous to 3scale 2.11, it is a requirement to mount the file on the container via ConfigMap and VolumeMount.

    - As of 3scale 2.11, it is a requirement to use a secret referenced in the APIManager custom resource (CR).

- For the APIcast operator deployments:
    - Previous to 3scale 2.11 this could not be configured.
    - As of 3scale 2.11, it is a requirement to use a secret referenced in the APIManager custom resource (CR).
- For APIcast self-managed deployed on Docker, it is a requirement to mount the file on the container.

Logging options help to avoid issues with logs that are not correctly formatted in APIs.

The following is an example of a policy that loads in all services:

custom_env.lua file

```
local cjson = require('cjson')
local PolicyChain = require('apicast.policy_chain')
local policy_chain = context.policy_chain

local logging_policy_config = cjson.decode([[
{
  "enable_access_logs": false,
  "custom_logging": "\"{{request}}\" to service {{service.id}} and {{service.name}}"
}
]])

policy_chain:insert( PolicyChain.load_policy('logging', 'builtin', logging_policy_config), 1)

return {
  policy_chain = policy_chain,
  port = { metrics = 9421 },
}
```

4.1.19.1.1. Configuring the logging policy for all APIs by mounting the file on the container via ConfigMap and VolumeMount

1. Create a ConfigMap with the **custom_env.lua** file:

```
oc create configmap logging --from-file=/path/to/custom_env.lua
```

2. Mount a volume for the ConfigMap, for example for **apicast-staging**:

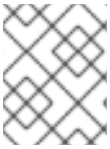
```
oc set volume dc/apicast-staging --add --name=logging --mount-path=/opt/app-root/src/config/custom_env.lua --sub-path=custom_env.lua -t configmap --configmap-name=logging
```

3. Set the environment variable:

```
oc set env dc/apicast-staging APICAST_ENVIRONMENT=/opt/app-root/src/config/custom_env.lua
```

4.1.19.1.2. Configuring the logging policy for all APIs using a secret referenced in the APIManager custom resource (CR)

From 3scale 2.11 in operator-based deployments, configure the logging policy as a secret and reference the secret in the APIManager custom resource (CR).



NOTE

The following procedure is valid for the 3scale operator only. You can however configure the APIcast operator in a similar way using these steps.

Prerequisites

- One or more custom environments coded with Lua.

Procedure

1. Create a secret with the custom environment content:

```
$ oc create secret generic custom-env --from-file=./custom_env.lua
```

2. Configure and deploy the APIManager CR with the APIcast custom environment:

   apimanager.yaml content:

```
apiVersion: apps.3scale.net/v1alpha1
kind: APIManager
metadata:
  name: apimanager-apicast-custom-environment
spec:
  apicast:
    productionSpec:
      customEnvironments:
        - secretRef:
            name: custom-env
    stagingSpec:
      customEnvironments:
        - secretRef:
            name: custom-env
```

3. Deploy the APIManager CR:

```
$ oc apply -f apimanager.yaml
```

If the secret does not exist, the operator marks the CR as failed. Changes to the secret will require a redeployment of the pod/container in order to reflect in APIcast.

Updating the custom environment

If you need to modify the custom environment content, there are two options:

- Recommended: Create another secret with a different name and update the APIManager CR field:

```
customEnvironments[].secretRef.name
```

The operator triggers a rolling update loading the new custom environment content.

- Update the existing secret content and redeploy APIcast turning **spec.apicast.productionSpec.replicas** or **spec.apicast.stagingSpec.replicas** to 0 and then back to the previous value.

4.1.19.1.3. Configuring the logging policy for all APIs for APIcast self-managed deployed on Docker

Run APIcast with this specific environment by mounting custom_env.lua using the following docker command:

```
docker run --name apicast --rm -p 8080:8080 \
    -v $(pwd):/config \
    -e APICAST_ENVIRONMENT=/config/custom_env.lua \
    -e THREESCALE_PORTAL_ENDPOINT=https://ACCESS_TOKEN@ADMIN_PORTAL_DOMAIN \
    quay.io/3scale/apicast:master
```

These are key concepts of the docker command to consider:

- Share the current Lua file to the container **-v $(pwd):/config**.
- Set the APICAST_ENVIRONMENT variable to the Lua file that is stored in the /**config** directory.

4.1.19.2. Examples of the logging policy

These are examples of the Logging policy, with the following caveats:

- If **custom_logging** or **enable_json_logs** property is enabled, default access log will be disabled.
- If **enable_json_logs** is enabled, the **custom_logging** field will be omitted.

Disabling access log

```
{
  "name": "apicast.policy.logging",
  "configuration": {
    "enable_access_logs": false
  }
}
```

Enabling custom access log

```
{
  "name": "apicast.policy.logging",
  "configuration": {
    "enable_access_logs": false,
    "custom_logging": "[{{time_local}}] {{host}}:{{server_port}} {{remote_addr}}:{{remote_port}} \"
{{request}}\" {{status}} {{body_bytes_sent}} ({{request_time}}) {{post_action_impact}}",
  }
}
```

Enabling custom access log with the service identifier

```
{
  "name": "apicast.policy.logging",
  "configuration": {
    "enable_access_logs": false,
    "custom_logging": "\"{{request}}\" to service {{service.id}} and {{service.serializable.name}}",
  }
}
```

Configuring access logs in JSON format

```
{
  "name": "apicast.policy.logging",
  "configuration": {
    "enable_access_logs": false,
    "enable_json_logs": true,
    "json_object_config": [
      {
        "key": "host",
        "value": "{{host}}",
        "value_type": "liquid"
      },
      {
        "key": "time",
        "value": "{{time_local}}",
        "value_type": "liquid"
      },
      {
        "key": "custom",
        "value": "custom_method",
        "value_type": "plain"
      }
    ]
  }
}
```

Configuring a custom access log only for a successful request

```
{
  "name": "apicast.policy.logging",
  "configuration": {
    "enable_access_logs": false,
    "custom_logging": "\"{{request}}\" to service {{service.id}} and {{service.name}}",
    "condition": {
      "operations": [
        {"op": "==", "match": "{{status}}", "match_type": "liquid", "value": "200"}
      ],
      "combine_op": "and"
    }
  }
}
```

Customizing access logs where the response status matches either **200** or **500**

```
{
  "name": "apicast.policy.logging",
  "configuration": {
    "enable_access_logs": false,
    "custom_logging": "\"{{request}}\" to service {{service.id}} and {{service.name}}",
    "condition": {
      "operations": [
        {"op": "==", "match": "{{status}}", "match_type": "liquid", "value": "200"},
        {"op": "==", "match": "{{status}}", "match_type": "liquid", "value": "500"}
      ],
      "combine_op": "or"
    }
  }
}
```

4.1.19.3. Additional information about custom logging

For custom logging, you can use Liquid templates with exported variables. These variables include:

- NGINX default directive variable: log_format. For example: **{{remote_addr}}**.
- Response and request headers:
  - **{{req.headers.FOO}}**: To get the FOO header in the request.
  - **{{res.headers.FOO}}**: To retrieve the FOO header on response.
- Service information, such as **{{service.id}}**, and all the service properties provided by these parameters:
  - THREESCALE_CONFIG_FILE
  - THREESCALE_PORTAL_ENDPOINT

4.1.20. Maintenance Mode

The Maintenance Mode policy allows you reject incoming requests with a specified status code and message. It is useful for maintenance periods or to temporarily block an API.

Configuration properties

The following is a list of possible properties and default values.

property	value	default	description
status	integer, *optional*	503	Response code
message	string, *optional*	503 Service Unavailable - Maintenance	Response message

Maintenance Mode policy example

```
{
```

```
  "policy_chain": [
    {"name": "maintenance-mode", "version": "1.0.0",
    "configuration": {"message": "Be back soon..", "status": 503} },
  ]
}
```

Apply maintenance mode for a specific upstream

```
{
    "name": "maintenance_mode",
    "version": "builtin",
    "configuration": {
        "message_content_type": "text/plain; charset=utf-8",
        "message": "Echo API /test is currently Unavailable",
        "condition": {
            "combine_op": "and",
            "operations": [
                {
                    "left_type": "liquid",
                    "right_type": "plain",
                    "op": "==",
                    "left": "{{ original_request.path }}",
                    "right": "/test"
                }
            ]
        },
        "status": 503
    }
}
```

For information about how to configure policies, see the Creating a policy chain in 3scale section of the documentation.

4.1.21. NGINX Filter

NGINX automatically checks some request headers and rejects requests when it cannot validate those headers. For example, NGINX rejects requests that have **If-Match** headers that NGINX cannot validate. If you want NGINX to skip validation of particular headers, add the NGINX Filter policy.

When you add the NGINX Filter policy, you specify one or more request headers for which you want NGINX to skip validation. For each header that you specify, you indicate whether or not to keep the header in the request. For example, the following JSON code adds the NGINX Filter policy so that it skips validation of **If-Match** headers but keeps **If-Match** headers in requests that are forwarded to the upstream server.

```
{ "name": "apicast.policy.nginx_filters",
  "configuration": {
    "headers": [
      {"name": "If-Match", "append": true}
    ]
  }
}
```

The next example also skips validation of **If-Match** headers but this code instructs NGINX to delete **If-Match** headers before sending requests to the upstream server.

```
{ "name": "apicast.policy.nginx_filters",
  "configuration": {
    "headers": [
      {"name": "If-Match", "append": false}
    ]
  }
}
```

Regardless of whether or not you append the specified header to the request that goes to the upstream server, you avoid an NGINX **412** response code when NGINX cannot validate a header that you specify.

IMPORTANT

Specifying the same header for the Header Modification policy and for the NGINX Filter policy is a potential source of conflict.

4.1.22. OAuth 2.0 Mutual TLS Client Authentication

This policy executes OAuth 2.0 Mutual TLS Client Authentication for every API call.

An example of the OAuth 2.0 Mutual TLS Client Authentication policy **JSON** is shown below:

```
{
  "$schema": "http://apicast.io/policy-v1/schema#manifest#",
  "name": "OAuth 2.0 Mutual TLS Client Authentication",
  "summary": "Configure OAuth 2.0 Mutual TLS Client Authentication.",
  "description": ["This policy executes OAuth 2.0 Mutual TLS Client Authentication ",
    "(https://tools.ietf.org/html/draft-ietf-oauth-mtls-12) for every API call."
  ],
  "version": "builtin",
  "configuration": {
    "type": "object",
    "properties": { }
  }
}
```

4.1.23. OAuth 2.0 Token Introspection

The OAuth 2.0 Token Introspection policy allows validating the JSON Web Token (JWT) token used for services with the OpenID Connect (OIDC) 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**: APIcast uses the client credentials, *Client ID* and *Client Secret*, as well as the Token Introspection Endpoint from the OIDC Issuer setting configured on the Service Integration page. APIcast discovers the Token Introspection endpoint from the **token_introspection_endpoint** field. This field is located in the **.well-known/openid-configuration** endpoint that is returned by the OIDC issuer.
  Authentication type set to **use_3scale_oidc_issuer_endpoint**:

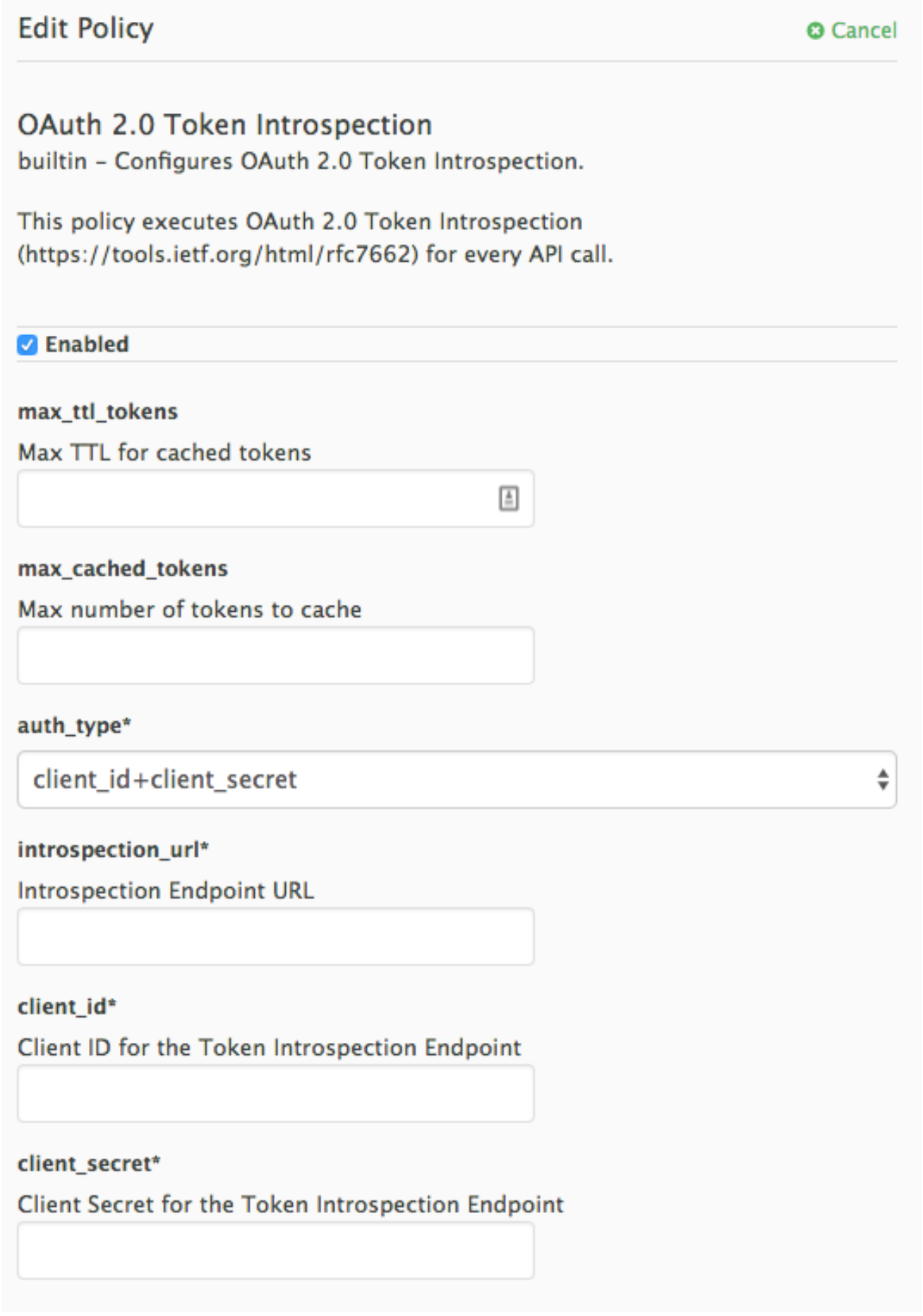
  ```
  "policy_chain": [
  …
  ```

```
  {
    "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.
    Authentication type 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).



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.

4.1.24. On Fail

As an API provider, you can add the On Fail policy to an API product. When the On Fail policy is in a policy chain and execution of a policy fails for a given API consumer request, APIcast does the following:

- Stops processing the request
- Returns the status code you specify to the application that sent the request

The On Fail policy is useful when APIcast cannot process a policy, perhaps because of an incorrect configuration or because of non-compliant code in a custom policy. Without the On Fail policy in the policy chain, APIcast skips a policy it cannot apply, processes any other policies in the chain, and sends the request to the upstream API. With the On Fail policy in the policy chain, APIcast rejects the request.

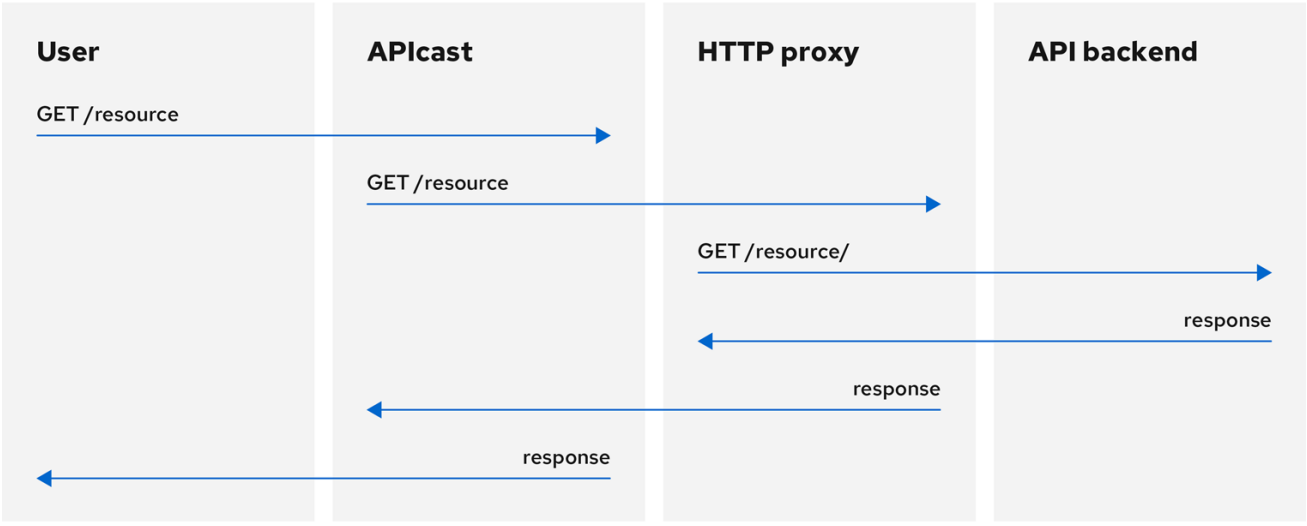
In a policy chain, the On Fail policy can be in any position.

In the Admin Portal, add the On Fail policy to a product's policy chain. In the policy chain, click the policy to specify the status code that you want APIcast to return when it applies the On Fail policy. For example, you could specify **400**, which indicates a bad request from the client.

4.1.25. Proxy Service

You can use the Proxy Service policy to define a generic *HTTP proxy* where the 3scale traffic will be sent using the defined proxy. In this case, the proxy service works as a reverse HTTP proxy, where APIcast sends the traffic to the HTTP proxy, and the proxy then sends the traffic on to the API backend.

The following example shows the traffic flow:



All APIcast traffic sent to the 3scale backend does not use the proxy. This policy only applies to the proxy and the communication between APIcast and API backend.

If you want to send all traffic through a proxy, you must use an **HTTP_PROXY** environment variable.

NOTE

- The Proxy Service policy disables all load-balancing policies, and traffic is sent to the proxy.
- If the **HTTP_PROXY**, **HTTPS_PROXY**, or **ALL_PROXY** parameters are defined, this policy overwrites those values.
- The proxy connection does not support authentication. You use the Header Modification policy for authentication.

Configuration

The following example shows the policy chain configuration:

```
"policy_chain": [
  {
    "name": "apicast.policy.apicast"
  },
  {
    "name": "apicast.policy.http_proxy",
    "configuration": {
        "all_proxy": "http://192.168.15.103:8888/",
        "https_proxy": "https://192.168.15.103:8888/",
        "http_proxy": "https://192.168.15.103:8888/"
    }
  }
]
```

The **all_proxy** value is used if **http_proxy** or **https_proxy** is not defined.

Example use case

The Proxy Service policy was designed to apply more fine-grained policies and transformation in 3scale using Apache Camel over HTTP. However, you can also use the Proxy Service policy as a generic HTTP proxy service. For integration with Apache Camel over HTTPS, see Section 4.1.6, “Camel Service”.

Example project

See the camel-netty-proxy example on GitHub. This project shows an HTTP proxy that transforms the response body from the API backend to uppercase.

4.1.26. Rate Limit Headers

The Rate Limit Headers policy adds **RateLimit** headers to response messages when your application subscribes to an application plan with rate limits. These headers provide useful information about the configured request quota limit and the remaining request quota and seconds in the current time window.

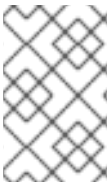
In the policy chain for a product, if you add the Rate Limit Headers policy it must be before the 3scale APIcast policy. If the 3scale APIcast policy is before the Rate Limit Headers policy then the Rate Limit Headers policy does not work.

RateLimit headers

The following **RateLimit** headers are added to each message:

- **RateLimit-Limit**: Displays the total request quota in the configured time window, for example, **10** requests.
- **RateLimit-Remaining**: Displays the remaining request quota in the current time window, for example, **5** requests.
- **RateLimit-Reset**: Displays the remaining seconds in the current time window, for example, **30** seconds. The behavior of this header is compatible with the **delta-seconds** notation of the **Retry-After** header.

By default, there are no rate limit headers in the response message when the Rate Limit Headers policy is not configured or when your application plan does not have any rate limits.



NOTE

If you are requesting an API metric with no rate limits but the parent metric has limits configured, the rate limit headers are still included in the response because the parent limits apply.

Additional resources

- Internet-Draft: RateLimit Header Fields for HTTP
- Configuring 3scale application plans and rate limits
- Configuring 3scale API metrics

4.1.27. Response/Request Content Limits

As an API provider, you can add the Response/Request Content Limits policy to an API product. This policy lets you limit the size of a request to an upstream API as well as the size of a response from an upstream API. Without this policy, the request/response size is unlimited.

This policy is helpful for preventing overloading of:

- A backend because it must act on a payload that is too large.
- An end-user (API consumer) because it receives more data than it can handle.

In a request or in a response, the **content-length** header is required for 3scale to apply the Response/Request Content Limits policy.

In the Admin Portal, after you add the Response/Request Content Limits policy to a product, click it to specify the limits in bytes. You can specify the request limit, or the response limit, or both. The default value, **0**, indicates an unlimited size.

Alternatively, you can add this policy by updating your policy chain configuration file, for example:

```
{
  "name": "apicast.policy.limits",
  "configuration":
  {
```

```
    "request": 100,
    "response": 100
  }
}
```

4.1.28. Retry

The Retry policy sets the number of retry requests to the upstream API. The retry policy is configured per service, so users can enable retries for as few or as many of their services as desired, as well as configure different retry values for different services.

IMPORTANT

As of 3scale 2.12, it is not possible to configure which cases to retry from the policy. This is controlled with the environment variable **APICAST_UPSTREAM_RETRY_CASES**, which applies retry requests to all services. For more on this, check out APICAST_UPSTREAM_RETRY_CASES.

An example of the retry policy **JSON** is shown below:

```
{
  "$schema": "http://apicast.io/policy-v1/schema#manifest#",
  "name": "Retry",
  "summary": "Allows retry requests to the upstream",
  "description": "Allows retry requests to the upstream",
  "version": "builtin",
  "configuration": {
    "type": "object",
    "properties": {
      "retries": {
        "description": "Number of retries",
        "type": "integer",
        "minimum": 1,
        "maximum": 10
      }
    }
  }
}
```

4.1.29. RH-SSO/Keycloak Role Check

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 (RH-SSO). The realm roles are specified when you want to add role check to every client resource of 3scale.

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

- **whitelist**: This is the 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 instance 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 containing the Client ID.
- **methods**: Use this parameter to list the allowed HTTP methods in APIcast, based on the user roles in RH-SSO. As examples, you can allow methods that have:
  - The **role1** realm role to access **/resource1**. For those methods that do not have this realm role, you need to specify the **blacklist**.
  - The **client1** role called **role1** to access **/resource1**.
  - The **role1** and **role2** realm roles to access **/resource1**. Specify the roles in **realm_roles**. You can also indicate the scope for each role.
  - The client role called **role1** of the application client, which is the recipient of the access token, to access **/resource1**. Use **liquid** client type to specify the JSON Web Token (JWT) information to the client.
  - The client role including the client ID of the application client, the recipient of the access token, to access **/resource1**. Use **liquid** client type to specify the JWT information to the **name** of the client role.
  - The client role called **role1** to access the resource including the application client ID. Use **liquid** client type to specify the JWT information to the **resource**.
- **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; the value can be **plain** or **liquid**. This 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; the value can be **plain** or **liquid**. This 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; The value can be **plain** or **liquid**. This works the same way as for the **resource_type**.

4.1.30. Routing

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

IMPORTANT

When you add the Routing policy to a policy chain, the Routing policy must always be immediately before the standard 3scale APIcast policy. In other words, there cannot be any policies between the Routing policy and the 3scale APIcast policy. This ensures correct APIcast output in the request that APIcast sends to the upstream API. Here are two examples of correct policy chains:

```
Liquid Context Debug
JWT Claim Check
Routing
3scale APIcast
```

```
Liquid Context Debug
Routing
3scale APIcast
JWT Claim Check
```

Routing rules

- If multiple rules exist, the Routing policy applies the first match. You can sort these rules.
- If no rules match, the policy will not change the upstream and will use the defined Private Base URL defined in the service configuration.

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"
            }
          ]
        }
      }
    ]
  }
}
```

-

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"
            }
          ]
        }
      }
    ]
  }
}
```

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"
            }
          ]
        }
      }
    ]
  }
}
```

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"
       }
      ]
     }
    }
   ]
  }
}
```

Multiple operations rule

Rules can have multiple operations and route to the given upstream only when all of them evaluate to true by using 'and' **combine_op**, or when at least one of them evaluates to true by using '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"
            }
          ]
        }
      }
    ]
  }
}
```

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"
            }
          ]
        }
      }
    ]
  }
}
```

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": []
      }
    }
  ]
 }
}
```

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"
            }
          ]
        }
      }
    ]
  }
}
```

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"
            }
          ]
        }
      }
    ]
  }
}
```

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": "/"
            }
          ]
        }
      }
    ]
  }
}
```

4.1.31. SOAP

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?
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, see the Creating a policy chain in 3scale section of the documentation.

4.1.32. TLS Client Certificate Validation

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.

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.

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

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

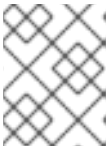
```
oc patch service apicast -p '{"spec":{"ports":[{"name":"https","port":8443,"protocol":"TCP"}]}}'
```

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
```

Configuring TLS Client Certificate Validation in your policy chain

To configure TLS Client Certificate Validation in your policy chain, you need 3scale login credentials. Also, you need to have configured APIcast with the TLS Client Certificate Validation policy .

1. To add the TLS Client Certificate Validation policy to your API, follow the steps described in Enabling policies in the 3scale Admin Portal 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 Policy Chain**.

Verifying functionality of the TLS Client Certificate Validation policy

To verify the functionality of the TLS Client Certificate Validation policy, you need 3scale login credentials. Also, you need to have configured APIcast with the TLS Client Certificate Validation policy .

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
```

Removing a certificate from the whitelist

To remove a certificate from the whitelist, 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 .

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 Policy Chain**.

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

4.1.33. TLS Termination

This section provides information about the Transport Layer Security (TLS) Termination policy: concepts, configuration, verification and file removal from the policy.

With the TLS Termination policy, you can configure APIcast to finish TLS requests for each API without using a single certificate for all APIs. APIcast pulls the configuration setting before establishing a connection to the client; in this way, APIcast uses the certificates from the policy and makes the TLS terminate. This policy works with these sources:

- Stored in the policy configuration.
- Stored on the file system.

By default, this policy is not enabled in policy chains.

Configuring TLS Termination in your policy chain

This section describes the prerequisites and steps to configure the TLS Termination in your policy chain, with Privacy Enhanced Mail (PEM) formatted certificates. Prerequisites are:

- Certificate issued by user
- A PEM-formatted server certificate
- A PEM-formatted certificate private key

Follow this procedure:

1. To add the TLS Termination policy to your API, follow the steps described in Enabling a standard Policy and choose TLS Termination.
2. Click the **TLS Termination** link.
3. To enable the policy, select the **Enabled** checkbox.
4. To add TLS certificates to the policy, click the plus **+** icon.
5. Choose the source of your certificates:
    - **Embedded certificate** is selected by default. Upload these certificates:
        - **PEM formatted certificate private key**. Click **Browse** to select and upload.

- **PEM formatted certificate**: Click **Browse** to select and upload.

- **Certificate from filesystem** - select and specify these certificate paths:
  - **Path to the certificate**
  - **Path to certificate private key**

6. When you have finished setting up your API with TLS Termination, 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 Policy Chain**.

Verifying functionality of the TLS Termination policy

You must have 3scale login credentials. You must have configured APIcast with the TLS Termination policy.

You can test in the command line if the policy works with the following command:

```
curl “${public_URL}:${port}/?user_key=${user_key}" --cacert ${path_to_certificate}/ca.pem -v
```

where:

- **public_URL**= The staging public base URL
- **port**= The port number
- **user_key**= The user key you want to authenticate with
- **path_to_certificate**= The path to the CA certificate in your local file system

Removing files from TLS Termination

This section describes the steps to remove the certificate and key files from the TLS Termination policy.

- You need 3scale login credentials.
- You need to have added the certificate to the policy, by configuring APIcast with the TLS Termination policy.

To remove a certificate:

1. Click the **TLS Termination** link.
2. To remove certificates and keys, click the **x** icon.
3. When you have finished removing the certificates, click **Update Policy**.

To save your changes, click **Update Policy Chain**.

4.1.34. Upstream

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, see the Creating a policy chain in 3scale section of the documentation.

4.1.35. Upstream Connection

The Upstream Connection policy allows you to change the default values of the following directives, for each API, depending on how you have configured the API backend server in your 3scale installation:

- **proxy_connect_timeout**

- **proxy_send_timeout**
- **proxy_read_timeout**

To configure the Upstream Connection policy:

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

Follow this procedure:

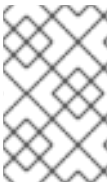
1. To add the Upstream Connection policy to your API, follow the steps described in Enabling policies in the 3scale Admin Portal and choose *Upstream Connection*.
2. Click the **Upstream Connection** link.
3. To enable the policy, select the **Enabled** checkbox.
4. Configure the options for the connections to the upstream:
   - **send_timeout**
   - **connect_timeout**
   - **read_timeout**
5. When you have finished setting up your API with Upstream Connection, click **Update Policy**.

To save your changes, click **Update Policy Chain**.

4.1.36. Upstream Mutual TLS

With the Upstream Mutual TLS policy, you can establish and validate mutual TLS connections between APIcast and upstream APIs based on the certificates set in the configuration.

When the **verify** field is enabled, the policy also verifies the server certificate from the upstream APIs. The **ca_certificates** contains a Privacy Enhanced Mail (PEM) formatted certificate, including the **-----BEGIN CERTIFICATE----- and -----END CERTIFICATE-----** that the APIcast uses to validate the server.



NOTE

You must enable the **verify** field and have **ca_certificates** filled for verification of the upstream API's certificate to take place. When the **verify** field is not enabled, only the check for the APIcast certificate at upstream APIs occurs.

To configure Upstream Mutual TLS in your policy chain, you need to have access to a 3scale installation.

1. To add the Upstream Mutual TLS policy to your API, follow the steps described in Enabling policies in the 3scale Admin Portal and choose *Upstream Mutual TLS*.
2. Click the **Upstream Mutual TLS** link.
3. To enable the policy, select the **Enabled** checkbox.
4. Choose a **Certificate type**:

- *path*: If you want to specify the path of a certificate, such as the one generated by OpenShift.
- *embedded*: If you want to use a third-party generated certificate, by uploading it from your file system.

5. In **Certificate**, specify the client certificate.
6. Indicate the key in **Certificate key**.
7. When you have finished setting up your API with Upstream Mutual TLS, click **Update Policy Chain**.

To promote your changes:

1. Go to **[Your_product] page > Integration > Configuration**
2. Under *APIcast Configuration*, click **Promote v# to Staging APIcast**

v# represents the version number of the configuration to be promoted.

Path configuration

Use the certificates path for OpenShift and Kubernetes secrets as follows:

```
{
  "name": "apicast.policy.upstream_mtls",
  "configuration": {
     "certificate": "/secrets/client.cer",
     "certificate_type": "path",
     "certificate_key": "/secrets/client.key",
     "certificate_key_type": "path"
  }
}
```

Embedded configuration

Use the following configuration for *http* forms and file upload:

```
{
  "name": "apicast.policy.upstream_mtls",
  "configuration": {
    "certificate_type": "embedded",
    "certificate_key_type": "embedded",
    "certificate": "data:application/pkix-cert;name=client.cer;base64,XXXXXXXXXXX",
    "certificate_key": "data:application/x-iwork-keynote-sffkey;name=client.key;base64,XXXXXXXX"
  }
}
```

For more details about the additional fields, **ca_certificates** and **verify** for Upstream Mutual TLS, policy config schema.

Additional considerations

The Upstream mutual TLS policy will overwrite **APICAST_PROXY_HTTPS_CERTIFICATE_KEY** and **APICAST_PROXY_HTTPS_CERTIFICATE** environment variable values. It uses the certificates set by the policy, so those environment variables will have no effect.

4.1.37. URL Rewriting

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 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.

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**: This is 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**: This is optional. When set to true with the checkbox enabled, if the command rewrote the URL, it will be the last one applied and all posterior commands in the list will be discarded.

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**: This is 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 in 3scale section of the documentation.

4.1.38. URL Rewriting with Captures

The URL Rewriting with Captures policy is an alternative to the URL Rewriting 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 policy 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
```

4.1.39. Websocket

The Websocket policy enables WebSocket protocol connections to upstream APIs. If you plan to enable the WebSocket protocol, consider the following:

- The WebSocket protocol does not support JSON Web Tokens.
- The WebSocket protocol does not allow additional headers.
- The WebSocket protocol is not part of the HTTP/2 standard.

For a given upstream API for which you enable WebSocket connections, you can define its backends as **http[s]** or **ws[s]**.

If you add the Websocket policy to a policy chain, ensure that the Websocket policy is before the 3scale APIcast policy.

4.2. POLICY CHAINS FROM 3SCALE STANDARD POLICIES

For each API product, you can specify a policy chain. A policy chain does the following:

- Specifies the policies that APIcast applies to requests.
- Provides configuration information for those policies.
- Determines the order in which APIcast applies policies.

To correctly order policies in a chain, it is important to understand how APIcast applies policies to API consumer requests.

4.2.1. How APIcast NGINX phases process 3scale policies

The 3scale API gateway, or APIcast, uses the NGINX proxy web server to apply policies. When APIcast receives a request from an API consumer, APIcast processes the request in an ordered series of NGINX phases. In each NGINX phase, APIcast can modify the original request by applying these policies:

- Policies in the upstream API policy chain. A policy chain is an ordered list of policies. By default, the policy chain for an upstream API includes the **3scale APIcast** policy. An API provider can add policies to the policy chain for a 3scale product. APIcast applies policies in an upstream API policy chain to API consumer requests sent to only that upstream API.

- Policies in the global 3scale policy chain. An API provider can set 3scale environment variables to update the global policy chain. APIcast applies policies in the global policy chain to all API consumer requests.

If the same policy is in an upstream API policy chain and in the global policy chain, the policy configuration in the upstream API policy chain has precedence.

After APIcast performs the processing required in all NGINX phases, APIcast sends the result in a request to the upstream API. Consequently, to achieve the desired behavior, it is important to understand the order in which NGINX phases process policies because processing can modify the API consumer request.

Order and description of NGINX phases

When APIcast receives a request from an API consumer, APIcast processes the request by applying the policies in the upstream API's policy chain and in the global policy chain. Each 3scale policy defines one or more functions. APIcast executes policy functions in an ordered series of NGINX phases. In each phase, NGINX runs any functions that are defined in the policies being applied and that specify execution in that phase. The following table lists the NGINX phases that run policy functions. Additional NGINX phases, not listed in this table, perform processing that is not affected by the order of policies in a policy chain.

NGINX phases in order	Description of processing in this phase
rewrite	Runs any functions that modify the request's target URI.
access	Runs any functions that verify the client's authorization to make the request.
content	Generates the request content to be sent to the upstream API. NGINX applies only one policy in the **content** phase. If more than one policy in a policy chain operates on request content NGINX applies only the policy that is earliest in the chain. This is important to understand because the builtin **3scale APIcast** policy is always in a policy chain and it requires NGINX processing in the **content** phase. For example, both the **3scale APIcast** policy and the **Upstream** policy update the request to specify the path for the upstream API. NGINX processes these functions in the **content** phase. If the **3scale APIcast** policy is before the **Upstream** policy then NGINX uses the configuration of the upstream API to add its path to the revised request. If the **Upstream** policy is before the **3scale APIcast** policy then NGINX evaluates the **Upstream** policy expression. When there is a match, NGINX changes the upstream API path accordingly in the revised request.
balancer	Runs any load balancing functions.
header_filter	Runs any functions that process the request header.
body_filter	Runs any functions that process the request body.
post_action	Runs any functions that process the request after NGINX has run functions on the header and body.

NGINX phases in order	Description of processing in this phase
log	Generates log information about the request.
metrics	Operates on any data that is received from the Prometheus endpoint.

Examples of NGINX phases that perform processing that is not affected by policy order:

- When APIcast starts, NGINX executes tasks associated with the **init** phase.
- When an APIcast worker starts, NGINX executes tasks associated with the **init_worker** phase.
- When APIcast terminates an HTTPS connection, NGINX executes tasks associated with the **ssl_certificate** phase.

Order in which NGINX runs policy functions

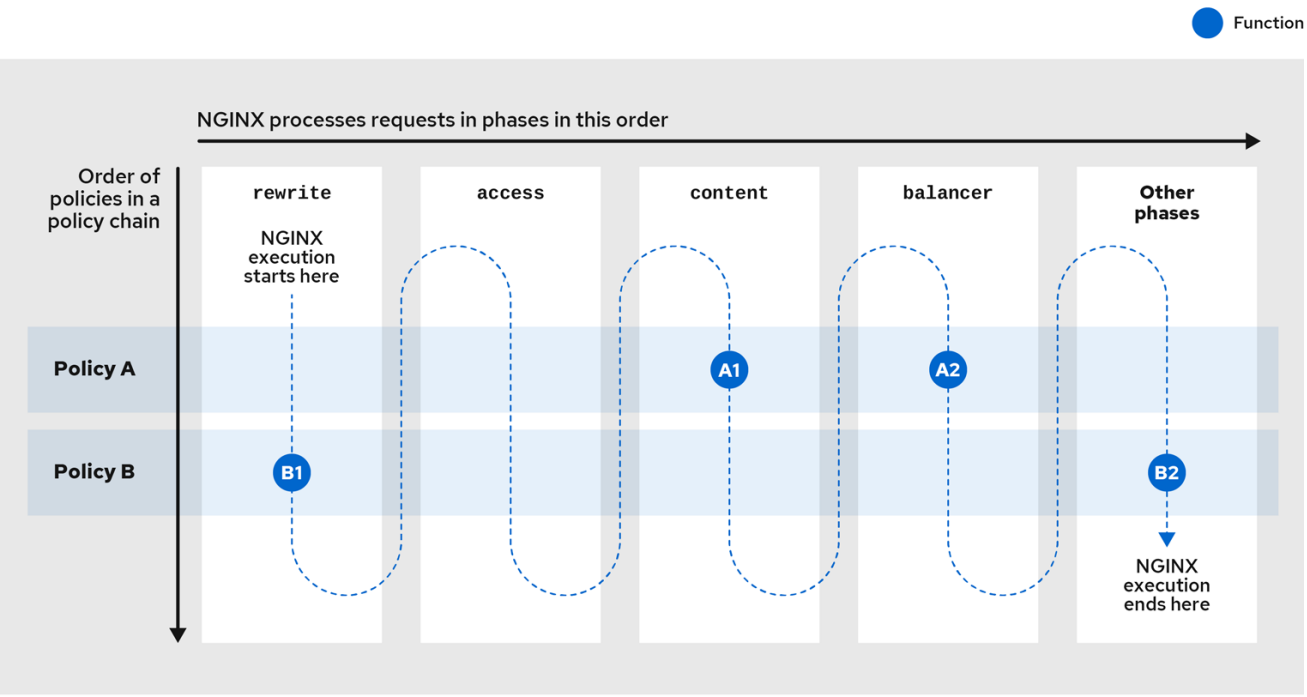
API providers can add one or more policies to a 3scale product to form a policy chain. In each phase, NGINX processes only those policy functions that specify execution in that phase. Each policy function specifies how APIcast should change its default behavior during processing in one NGINX phase. For example, in the **header_filter** phase, NGINX processes functions that specify **header_filter** and that presumably operate on request headers. In each phase, NGINX processes relevant functions in the order in which they are in the policy chain.

Policies can share data by means of a **context** object. Policies can read from and modify the **context** object in each phase.

The order in which NGINX executes policy functions depends on the following:

- The position of the policy in the policy chain
- The NGINX phase that processes a particular policy function

To obtain the desired behavior, you must correctly specify the policy chain order because the result of applying a policy can vary according to its place in a policy chain. The following diagram shows an example of the order in which NGINX applies policies.

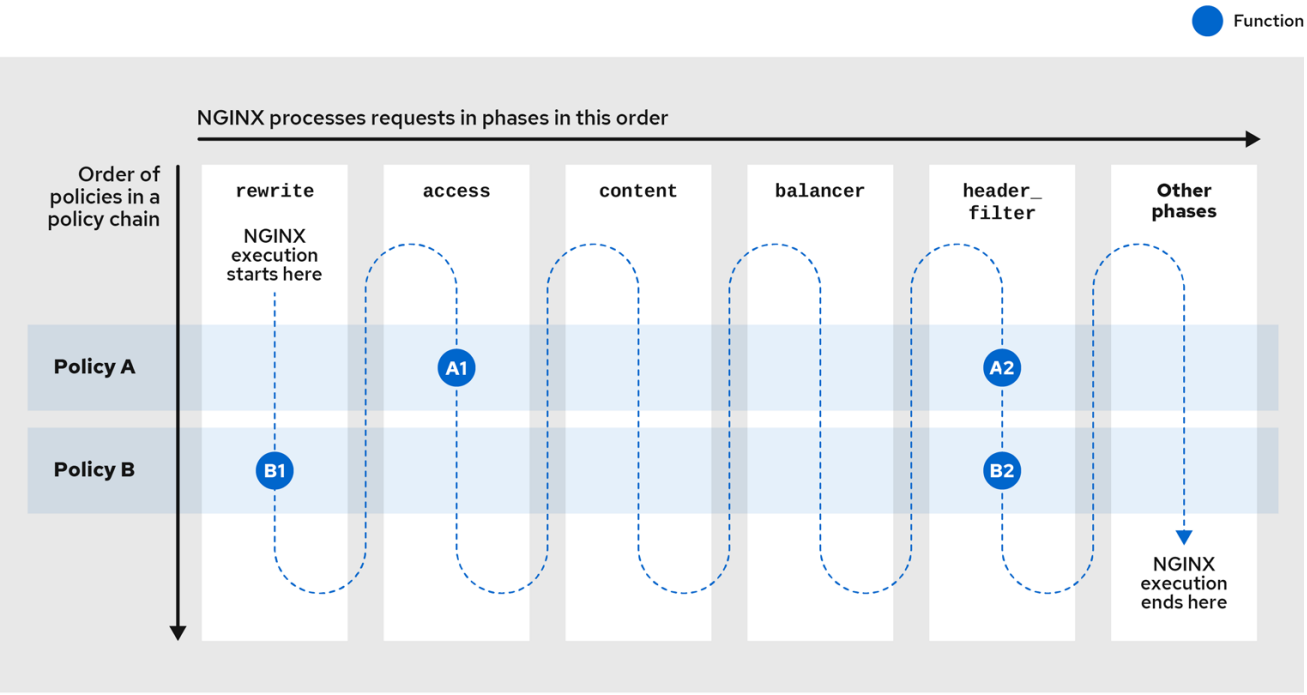


In the previous figure, policy **A** is first in the policy chain. However, NGINX processes a function in policy **B** first because that function is related to NGINX's first phase, the **rewrite** phase.

Now consider a product's policy chain that contains policy **A** and then policy **B** with these functions:

- Policy **A** specifies:
  - Function **A1** for NGINX to run in the **access** phase
  - Function **A2** for NGINX to run in the **header_filter** phase
- Policy **B** specifies:
  - Function **B1** for NGINX to run in the **rewrite** phase
  - Function **B2** for NGINX to run in the **header_filter** phase

The following figure shows the order in which NGINX runs the product's policy functions.



When APIcast receives a request for access to the upstream API exposed by this product, APIcast checks the product's policy chain and runs the functions as described in the following table:

NGINX phases in order	Functions that NGINX runs in this phase
rewrite	Runs the function **B1** that policy **B** specifies for the **rewrite** phase.
access	Runs the function **A1** that policy **A** specifies for **access** phase.
content	Neither policy **A** nor policy **B** specifies a function for execution in the **content** phase.
balancer	Neither policy **A** nor policy **B** specifies a function for execution in the **balancer** phase.
header_filter	The policy chain specifies policy **A** and then policy **B**. Consequently, this phase runs the function **A2** that policy **A** specifies for the **header_filter** phase and then runs the function **B2** that policy **B** specifies for the **header_filter** phase.
body_filter	Neither policy **A** nor policy **B** specifies a function for execution in this phase.
post_action	Neither policy **A** nor policy **B** specifies a function for execution in this phase.
log	Neither policy **A** nor policy **B** specifies a function for execution in this phase.

In this example, policy **A** is first in the policy chain but a function in policy **B** is the first function that NGINX runs. This is because policy **B** specifies a function **B1** that NGINX processes in the **rewrite** phase, which comes before the other phases.

For another example, consider this policy chain:

1. **URL Rewriting**
2. **3scale APIcast** (default policy assigned to all products)

The **URL Rewriting** policy modifies a request's target path. APIcast runs the **URL Rewriting** function in the **rewrite** phase. The **3scale APIcast** policy defines a function that APIcast runs in the **rewrite** phase as well as functions that APIcast runs in three other phases. When the **URL Rewriting** policy is first, the **3scale APIcast** policy applies mapping rules to the rewritten path. If the **3scale APIcast** policy is first and the **URL Rewriting** policy is second, the **3scale APIcast** policy applies mapping rules to the original path.

Additional resources

- NGINX phases that run 3scale standard policy functions
- 3scale standard policies and the NGINX phases that process them

4.2.2. Modifying policy chains in the 3scale Admin Portal

Modify a product's policy chain in the 3scale Admin Portal as part of your APIcast gateway configuration.

Procedure

1. Log in to 3scale.
2. Navigate to the API product you want to configure the policy chain for.
3. In **[your_product_name] > Integration > Policies**, click *Add policy*.
4. Under the **Policy Chain** section, use the arrow icons to reorder policies in the policy chain.
5. Click **Update Policy Chain** to save the policy chain.

Next steps

In the Admin Portal's left-side navigation panel, there is now a warning that indicates that there are **Configuration** changes that you have not promoted to APIcast. Promote the policy chain updates to Staging APIcast and test the update as needed. After confirming the desired behavior, promote the update to Production APIcast. If the **APICAST_CONFIGURATION_CACHE** environment variable is set to a number greater than zero (the default) it takes that number of seconds for APIcast to use the updated configuration.

4.2.3. Creating 3scale policy chains in JSON configuration files

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

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

- **services** object with an **id** value that specifies which service the policy chain applies to by number
- **proxy** object, which contains the **policy_chain** object and subsequent objects

- **policy_chain** object, which contains the values that define the policy chain
- Individual **policy** objects that 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 builtin policy **apicast**. Where you place the **apicast** policy in the policy chain affects policy behavior.

4.2.4. NGINX phases that run 3scale standard policy functions

The following table lists the main NGINX phases with the standard policies that define functions that NGINX runs in that phase. The table lists the phases in the order in which NGINX processes them.

A policy chain can contain more than one policy that NGINX processes in a particular phase. In this situation, ensure that the order of the policies in the chain is the correct order for processing the API request to obtain the desired result. The table lists the policies in alphabetical order.

NGINX phases in order	Standard policies that define functions that are processed in this phase

NGINX phases in order	Standard policies that define functions that are processed in this phase
rewrite	3scale APIcast 3scale Referrer Anonymous Access Echo Header Modification NGINX Filter SOAP Upstream URL Rewriting URL Rewriting with Captures Websocket
access	3scale APIcast 3scale Batcher Camel Proxy Content Caching Edge Limiting IP Check JWT Claim Check RH-SSO/Keycloak Role Check Maintenance Mode OAuth 2.0 Mutual TLS Client Authentication OAuth 2.0 Token Introspection Rate Limit Headers Response/Request Content Limits Routing TLS Client Certificate Validation Upstream
content	3scale APIcast Liquid Context Debug Rate Limit Headers Routing Upstream
balancer	Upstream Mutual TLS
header_filter	CORS Request Handling Header Modification Response/Request Content Limits HTTP Response Code Overwrite
body_filter	Response/Request Content Limits
post_action	3scale APIcast Custom metrics

NGINX phases in order	Standard policies that define functions that are processed in this phase
log	Edge Limiting Logging

Additional resources

- How APIcast NGINX phases process 3scale policies
- 3scale standard policies and the NGINX phases that process them

4.2.5. 3scale standard policies and the NGINX phases that process them

The following table lists the standard policies and the NGINX phase or phases that run that policy's function or functions. Use this table to correctly order policies in a policy chain to produce the correct request for the upstream API.

Standard policies	NGINX phases that run policy functions
3scale APIcast	**init** **rewrite** **access** **content** **post_action** APIcast applies the **3scale APIcast** policy to all requests.
Anonymous Access	**rewrite**
3scale Auth Caching	In a policy chain, the position of this policy does not matter.
3scale Batcher	**access**
3scale Referrer	**rewrite**
Camel Service	**access**
Conditional Policy	In a policy chain, the position of this policy does not matter.
Content Caching	**access**
CORS Request Handling	**header_filter**
Custom metrics	**post_action**
Echo	**rewrite**

Standard policies	NGINX phases that run policy functions
Edge Limiting	**access** **log**
Header Modification	**rewrite** **header_filter**
HTTP Response Code Overwrite	**header_filter**
IP Check	**access**
JWT Claim Check	**access**
Liquid Context Debug	**content**
Logging	**log**
Maintenance Mode	**access**
NGINX Filter	**rewrite**
OAuth 2.0 Mutual TLS Client Authentication	**access**
OAuth 2.0 Token Introspection	**access**
Proxy Service	In a policy chain, the position of this policy does not matter.
Rate Limit Headers	**access** +**content**
Response/Request Content Limits	**access** **header_filter** **body_filter**
Retry	In a policy chain, the position of this policy does not matter.
RH-SSO/Keycloak Role Check	**access**
Routing	**access** **content**
SOAP	**rewrite**
TLS Client Certificate Validation	**access**
TLS Termination	**ssl_certificate**

Standard policies	NGINX phases that run policy functions
Upstream	**rewrite** **access** **content**
Upstream Connection	In a policy chain, the position of this policy does not matter.
Upstream Mutual TLS	**balancer**
URL Rewriting	**rewrite**
URL Rewriting with Captures	**rewrite**
Websocket	**rewrite**

Additional resources

- How APIcast NGINX phases process 3scale policies
- NGINX phases that run 3scale standard policy functions

4.3. CUSTOM 3SCALE APICAST POLICIES

Configure custom policies to modify APIcast behavior. First, define a policy chain that configures APIcast policies, including your custom policies; then, add the policy chain to APIcast.

NOTE

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

Custom policies for APIcast depend on the configuration of your 3scale deployment:

- Add custom policies to these APIcast self-managed deployments: APIcast on OpenShift, and APIcast on the containerized environment you have installed.
- You cannot add custom policies to APIcast hosted.

WARNING

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

4.3.1. About custom policies for 3scale APIcast deployments

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

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.

4.3.2. Adding custom policies to 3scale embedded 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. 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, both the *apicast-staging* and the *apicast-production* tags, by default, are configured to automatically start new deployment. To avoid any disruptions to staging or to your production service disable automatic deployment by unselecting the *"Automatically start a new deployment when the image changes"* checkbox. Or, configure a different image stream tag for production, for example, **amp-apicast:production**.

Procedure

1. Create a **docker-registry** secret using the credentials you created in Creating a registry service account, following these considerations:
   - Replace **your-registry-service-account-username** with the username created in the format, **12345678|username**.
   - Replace **your-registry-service-account-password** with the password string below the username, under the *Token Information* tab.
   - Create a **docker-registry** secret for every new **namespace** where the image streams reside and which use *registry.redhat.io*.
     Run this command to create a **docker-registry** secret:

```
oc create secret docker-registry threescale-registry-auth \
  --docker-server=registry.redhat.io \
  --docker-username="your-registry-service-account-username" \
```

```
--docker-password="your-registry-service-account-password"
```

2. Fork the 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.

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

4. 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. For 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.

5. Create the OpenShift objects by running the command:

```
oc new-app -f openshift.yml --param AMP_RELEASE=2.12
```

6. 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, for example, **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 built-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, navigate to **Integration > Policies**, and click the **Add Policy** button to see your custom policy listed. After selecting and configuring it, click **Update Policy Chain** to make your custom policy work in the staging APIcast.

4.3.3. Adding custom policies to 3scale in 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 Integrated OpenShift Container Platform registry.

Procedure

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 2.12 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. Deploy APIcast using the OpenShift template , specifying your customized image:

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

NOTE

When custom policies are added to APIcast and a new image is built, those policies are automatically displayed as available in the Admin Portal when APIcast is deployed with the image. Existing services can see this new policy in the list of available policies, so it can be used in any policy chain.

When a custom policy is removed from an image and APIcast is restarted, the policy will no longer be available in the list, so you can no longer add it to a policy chain.

4.3.4. Including external Lua dependencies in 3scale custom policies

You can add an external Lua dependency to a custom policy so that APIcast can use a Lua library that is not yet in your 3scale image.

The procedure here shows you how to do this by using an example of a custom policy that transforms a response body from JSON to XML. The example custom policy requires the **xml2lua** XML parser, which is written in Lua. The complete example shows a short cut for building and testing but you cannot deploy your custom policy by following only the example procedure. To deploy a custom policy that has an external Lua dependency, you must perform the steps in this procedure as well as the procedure for Adding custom policies to 3scale in another OpenShift Container Platform .

The **JSON to XML** custom policy is only an example. It is not for use in a production environment.

Prerequisties

- A 3scale custom policy
- Access to an external Lua library

Procedure

1. In the directory that contains your custom policy, add a file that identifies the external Lua library.
   The name of the file must be **Roverfile**. In the **JSON to XML** custom policy example, **Roverfile** has this content:

```
luarocks {
 group 'production' {
  module { 'xml2lua' },
 }
}
```

   lua-rover is a wrapper around LuaRocks. **lua-rover** provides transitive locking for dependencies. LuaRocks is a package manager for Lua modules.

2. In the directory that contains your custom policy, add a **lua-rover** lock file.
The name of the file must be **Roverfile.lock**. In the **JSON to XML** custom policy example, **Roverfile.lock** has this content:

```
xml2lua 1.5-2||productionbash-4.4
```

Together, **Roverfile** and **Roverfile.lock** enable APIcast or the 3scale operator to fetch the dependent library.

3. In the file that defines your custom policy, add a line that specifies the Lua dependency. The **JSON to XML** custom policy example specifies this line:

```
local xml2lua = require("xml2lua")
```

4. In the Dockerfile that you use to build your custom policy, copy **Roverfile** and **Roverfile.lock**, and run **rover install**. The **JSON to XML** custom policy example adds these lines to its Dockerfile:

```
COPY Roverfile .
COPY Roverfile.lock .

RUN rover install --roverfile=/opt/app-root/src/Roverfile
```

Your Dockerfile can use APIcast or the 3scale operator to build the policy.

5. In the **Makefile** for your custom policy, specify the **build** target as you would for any custom policy.
For example, the **build** target might look like this:

```
TARGET_IMAGE="apicast/json_to_xml:latest"
# IP="http://localhost:8080"

build:
 docker build . --build-arg IMAGE=registry.redhat.io/3scale-amp2/apicast-gateway-rhel8:3scale2.12 -t $(TARGET_IMAGE)
```

Next steps

The remaining steps for deploying a custom policy that has an external Lua dependency are the same as they are for deploying other custom policies. That is, you need to push the image into your repository and replace the APIcast image with the one you just built.

Additional resources

- Adding custom policies to 3scale in another OpenShift Container Platform
- APIManager CRD Reference, APIcastSpec **image** parameter
- APIcast Custom Resource reference **image** parameter

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 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 with query parameters excluded from this path. The value of the embedded NGINX variable **$uri**.
- **host**: The host of the request, which is the value of the embedded NGINX variable **$host**.
- **remote_addr**: The IP address of the client, which is 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.

These standard variables are used in the context of the request, but policies can add more variables to the context. A phase refers to all the execution steps that APIcast has. Variables can be used by all the policies in the policy chain, provided these cases:

- Within the same phase, if the variable is added in the policy and then used in the following policy after the addition.
- If a variable is added in a phase, this variable can be used in the next phases.

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 Liquid context debug.

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, for example: **value_type**, **name_type**. 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 }}**, where **username:password** is a variable.
- **{{ 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. TRANSFORMING 3SCALE MESSAGE CONTENT USING POLICY EXTENSIONS IN FUSE

You can use Red Hat Fuse to create highly flexible policy extensions for Red Hat 3scale API Management. You can do this by creating policy extensions in Fuse on OpenShift and then configuring them as policies in the 3scale Admin Portal. Using an APIcast Camel proxy policy, you can perform complex transformations on request and response message content, for example, XML to JSON, which are implemented in the Apache Camel integration framework.

In addition, you can add or modify custom policy extensions dynamically in Camel, instead of rebuilding and redeploying a static APIcast container image. You can use any Camel Enterprise Integration Pattern (EIP) written in Camel Domain Specific Language (DSL) to implement an APIcast policy extension. This enables you to write policy extensions using a familiar programming language such as Java or XML. The example in this topic uses the Camel Netty4 HTTP component to implement the HTTP proxy in Java.

NOTE

This feature is not required if you are already using a Fuse Camel application in your 3scale API backend. In this case, you can use your existing Fuse Camel application to perform transformations.

Required software components

You must have the following Red Hat Integration components deployed on the same OpenShift cluster:

- Fuse on OpenShift 7.10
- 3scale On-premises 2.12
- APIcast embedded (default Staging and Production), or APIcast self-managed

You can deploy the custom Fuse policy in a different OpenShift project than 3scale, but this is not required. However, you must ensure that communication between both projects is possible. For details, see Configuring network policy with OpenShift SDN .

Additional resources

- Fuse on OpenShift Guide

6.1. INTEGRATING APICAST WITH APACHE CAMEL TRANSFORMATIONS IN FUSE

You can integrate APIcast with a transformation written as an Apache Camel application in Fuse on OpenShift. When the policy extension transformation is configured and deployed in 3scale, the 3scale traffic goes through the Camel policy extension, which transforms the message content. In this case, Camel works as a reverse HTTP proxy, where APIcast sends the 3scale traffic to Camel, and Camel then sends the traffic on to the API backend.

The example in this topic creates the HTTP proxy using the Camel Netty4 HTTP component:

- The request received over the HTTP proxy protocol is forwarded to the target service with the HTTP body converted to uppercase.

- The response from the target service is processed by converting it to uppercase and then returned to the client.
- This example shows the configuration required for HTTP and HTTPS use cases.

Prerequisites

- You must have Fuse on OpenShift 7.10 and 3scale 2.12 deployed on the same OpenShift cluster. For installation details, see:
  - Fuse on OpenShift Guide
  - Installing 3scale
- You must have cluster administrator privileges to install Fuse on OpenShift and 3scale and to create projects. However, you can create deployment configurations, deploy pods, or create services with edit access privileges per project.

Procedure

1. Write an Apache Camel application in Java using the Camel **netty4-http** component to implement the HTTP proxy. You can then use any Camel component to transform the message. The following simple example performs an uppercase transformation of the request and response from the service:

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.Locale;

import org.apache.camel.Exchange;
import org.apache.camel.Message;
import org.apache.camel.builder.RouteBuilder;
import org.apache.camel.model.RouteDefinition;

public class ProxyRoute extends RouteBuilder {

    @Override
    public void configure() throws Exception {
        final RouteDefinition from;
        if (Files.exists(keystorePath())) {
            from = from("netty4-http:proxy://0.0.0.0:8443?
ssl=true&keyStoreFile=/tls/keystore.jks&passphrase=changeit&trustStoreFile=/tls/keystore.jks"
); (1)
        } else {
            from = from("netty4-http:proxy://0.0.0.0:8080");
        }

        from
            .process(ProxyRoute::uppercase)
            .toD("netty4-http:"
                + "${headers." + Exchange.HTTP_SCHEME + "}://" (2)
                + "${headers." + Exchange.HTTP_HOST + "}:"
                + "${headers." + Exchange.HTTP_PORT + "}"
                + "${headers." + Exchange.HTTP_PATH + "}")
            .process(ProxyRoute::uppercase);
    }
```

```
    Path keystorePath() {
        return Path.of("/tls", "keystore.jks");
    }

    public static void uppercase(final Exchange exchange) { 3
        final Message message = exchange.getIn();
        final String body = message.getBody(String.class);
        message.setBody(body.toUpperCase(Locale.US));
    }

}
```

1. In this simple example, if your Java keystore file is mounted at **/tls/keystore.jks**, the listening port is set to **8443**.
2. When the Camel proxy policy is invoked by 3scale, the values for the **HTTP_SCHEME**, **HTTP_HOST**, **HTTP_PORT**, and **HTTP_PATH** headers are automatically set based on the values configured for the backend API in 3scale.
3. This simple example converts the message content to uppercase. You can perform more complex transformations on request and response message content, for example, XML to JSON, using Camel Enterprise Integration Patterns.

2. Deploy your Camel application on OpenShift and expose it as a service. For more details, see Creating and Deploying Applications on Fuse on OpenShift .

Additional resources

- Apache Camel Component Reference - Netty4 HTTP component

6.2. CONFIGURING AN APICAST POLICY EXTENSION CREATED USING APACHE CAMEL IN FUSE ON OPENSHIFT

After you have implemented the Apache Camel transformation using Fuse on OpenShift, you can use the 3scale Admin Portal to configure it as a policy extension in the APIcast policy chain.

The policy extension enables you to configure a 3scale product to use a Camel HTTP proxy. This service is used to send the 3scale traffic over the HTTP proxy to perform request–response modifications in a third-party proxy. In this case, the third-party proxy is Apache Camel implemented using Fuse on OpenShift. You can also configure APIcast to connect to the Camel HTTP proxy service securely using TLS.

NOTE

The policy extension code is implemented in an Apache Camel application in Fuse on OpenShift and cannot be modified or deleted from 3scale.

Prerequisites

- You must have Fuse on OpenShift 7.10 and 3scale 2.12 deployed on the same OpenShift cluster. For installation details, see:
  - Fuse on OpenShift Guide

- Installing 3scale

- You must have implemented an APIcast policy extension using an Apache Camel application in Fuse on OpenShift. See Section 6.1, "Integrating APIcast with Apache Camel transformations in Fuse"

- You must have deployed the Apache Camel application in an OpenShift pod and exposed it as a service. For details, see Creating and Deploying Applications on Fuse on OpenShift .

Procedure

1. In the 3scale Admin Portal, select **Integration** > **Policies**.

2. Select **POLICIES** > **Add policy** > **Camel Service**.

3. Enter the OpenShift routes used to connect to the Camel HTTP proxy service in the appropriate fields:

   - **https_proxy**: Connect to the Camel HTTP proxy using the **http** protocol and TLS port, for example:

     ```
     http://camel-proxy.my-3scale-management-project.svc:8443
     ```

   - **http_proxy**: Connect to the Camel HTTP proxy using the **http** protocol and port, for example:

     ```
     http://camel-proxy.my-3scale-management-project.svc:8080
     ```

   - **all_proxy**: Connect to the Camel HTTP proxy using the **http** protocol and port when the protocol is not specified, for example:

     ```
     http://camel-proxy.my-3scale-management-project.svc:8080
     ```

4. Promote the updated policy configuration to your staging or production environment. For example, click **Promote v. 3 to Staging APIcast**

5. Test the APIcast policy configuration using a 3scale **curl** command, for example:

   ```
   curl "https://testapi-3scale-apicast-staging.myuser.app.dev.3sca.net:443/?
   user_key=MY_USER_KEY" -k
   ```

   APIcast establishes a new TLS session for the connection to the Camel HTTP proxy.

6. Confirm that the message content has been transformed, which in this example means converted to uppercase.

7. If you wish to bypass APIcast and test the Camel HTTP proxy directly using TLS, you must use a custom HTTP client. For example, you can use the **netcat** command:

   ```
   $ print "GET https://mybackend.example.com HTTP/1.1\nHost:
   mybackend.example.com\nAccept: */*\n\n" | ncat --no-shutdown --ssl my-camel-proxy 8443
   ```

   This example creates an HTTP proxy request using the full URL after **GET**, and uses the **ncat --ssl** parameter to specify a TLS connection to the **my-camel-proxy** host on port **8443**.

NOTE

You cannot use **curl** or other common HTTP clients to test the Camel HTTP proxy directly because the proxy does not support HTTP tunneling using the **CONNECT** method. When using HTTP tunneling with **CONNECT**, the transport is end-to-end encrypted, which does not allow the Camel HTTP proxy to mediate the payload.

Additional resources

- Section 4.1.6, “Camel Service”

CHAPTER 7. 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

- all_proxy, ALL_PROXY
- APICAST_ACCESS_LOG_BUFFER
- APICAST_ACCESS_LOG_FILE
- APICAST_BACKEND_CACHE_HANDLER
- APICAST_CACHE_MAX_TIME
- APICAST_CACHE_STATUS_CODES
- APICAST_CONFIGURATION_CACHE
- APICAST_CONFIGURATION_LOADER
- APICAST_CUSTOM_CONFIG
- APICAST_ENVIRONMENT
- APICAST_EXTENDED_METRICS
- APICAST_HTTPS_CERTIFICATE
- APICAST_HTTPS_CERTIFICATE_KEY
- APICAST_HTTPS_PORT
- APICAST_HTTPS_VERIFY_DEPTH
- APICAST_LOAD_SERVICES_WHEN_NEEDED
- APICAST_LOG_FILE
- APICAST_LOG_LEVEL
- APICAST_MANAGEMENT_API
- APICAST_MODULE
- APICAST_OIDC_LOG_LEVEL
- APICAST_PATH_ROUTING
- APICAST_PATH_ROUTING_ONLY

- APICAST_POLICY_LOAD_PATH
- APICAST_PROXY_HTTPS_CERTIFICATE
- APICAST_PROXY_HTTPS_CERTIFICATE_KEY
- APICAST_PROXY_HTTPS_PASSWORD_FILE
- APICAST_PROXY_HTTPS_SESSION_REUSE
- APICAST_REPORTING_THREADS
- APICAST_RESPONSE_CODES
- APICAST_SERVICE_CACHE_SIZE
- APICAST_SERVICE_${ID}_CONFIGURATION_VERSION
- APICAST_SERVICES_LIST
- APICAST_SERVICES_FILTER_BY_URL
- APICAST_UPSTREAM_RETRY_CASES
- APICAST_WORKERS
- BACKEND_ENDPOINT_OVERRIDE
- HTTP_KEEPALIVE_TIMEOUT
- http_proxy HTTP_PROXY
- https_proxy HTTPS_PROXY
- no_proxy NO_PROXY
- OPENSSL_VERIFY
- OPENTRACING_CONFIG
- OPENTRACING_HEADER_FORWARD
- OPENTRACING_TRACER
- RESOLVER
- THREESCALE_CONFIG_FILE
- THREESCALE_DEPLOYMENT_ENV
- THREESCALE_PORTAL_ENDPOINT

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.

APICAST_ACCESS_LOG_BUFFER
Default: no value

Value: positive integer

Allows access log writes to be included in chunks of bytes. The result is fewer system calls, which improves the performance of the gateway.

APICAST_ACCESS_LOG_FILE
Default: stdout

Defines the file that will store the access logs.

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_CACHE_MAX_TIME
Default: 1m

Value: string

When the response is selected to be cached in the system, the value of this variable indicates the maximum time to be cached. If the cache-control header is not set, the time to be cached will be the defined one.

The format for this value is defined by the **proxy_cache_valid** NGINX directive.

This parameter is only used by the APIs that are using a content caching policy and the request is eligible to be cached.

APICAST_CACHE_STATUS_CODES
Default: 200, 302

Value: string

When the response code from upstream matches one of the status codes defined in this environment variable, the response content is cached in NGINX. The caching time depends on one of these values: headers cache time value, or the maximum time defined by the **APICAST_CACHE_MAX_TIME** environment variable.

This parameter is only used by the APIs that are using a content caching policy and the request is eligible to be cached.

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

APIcast should load a list of environments (or paths), separated by colons (**:**). This list 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_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.

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_VERIFY_DEPTH

Default: 1

Values: positive integers.

Defines the maximum length of the client certificate chain. If this parameter has **1** as its value, it is possible to include an additional certificate in the client certificate chain. For example, root certificate authority.

APICAST_LOAD_SERVICES_WHEN_NEEDED

Values:

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

Default: *false*

This option can be used when there are many services configured. However, its performance depends on additional factors, for example, the number of services, the latency between APIcast and the 3scale Admin Portal, and the Time To Live (TTL) of the configuration.

By default, APIcast loads all the services each time it downloads its configuration from the Admin Portal. When this option is enabled, the configurations uses lazy loading. APIcast will only load the ones configured for the host specified in the host header of the request.

NOTE

- The caching defined by **APICAST_CONFIGURATION_CACHE** applies.
- This option will be disabled when **APICAST_CONFIGURATION_LOADER** is **boot**.
- Not compatible with **APICAST_PATH_ROUTING**.

APICAST_LOG_FILE

Default: stderr

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

APICAST_LOG_LEVEL

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

Default: warn

Specifies the log level for the OpenResty logs.

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_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_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_PATH_ROUTING_ONLY

Values:

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

When this parameter is set to *true*, the gateway uses path-based routing and will not fallback to the default host-based routing. The API request is 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*.

This parameter has precedence over APICAST_PATH_ROUTING. If **APICAST_PATH_ROUTING_ONLY** is enabled, APIcast will only do path-based routing regardless of the value of **APICAST_PATH_ROUTING**.

APICAST_POLICY_LOAD_PATH

Default: **APICAST_DIR/policies**

Value: string[:]

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

A 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
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_CERTIFICATE_KEY
Default:

Value: string

Example: /home/apicast/my_certificate.key

The path to the key of the client SSL certificate.

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. For more information, see Setting up and evaluating the 3scale response codes log for your API .

APICAST_SERVICE_CACHE_SIZE

Values: integer >= 0

Default: 1000

Specifies the number of services that APIcast can store in the internal cache. A high value has a performance impact because Lua's **lru** cache will initialize all the entries.

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_SERVICES_LIST

Value: a comma-separated list of service IDs

The *APICAST_SERVICES_LIST* environment variable is used to filter the services you configure in the 3scale API Manager. This only applies the configuration for specific services in the gateway, discarding those service identifiers that are not specified in the list. You can find service identifiers for your product in the Admin Portal under **Products > [Your_product_name] > Overview**, then see **Configuration, Methods and Settings** and the *ID for API calls* .

APICAST_SERVICES_FILTER_BY_URL

Value: a PCRE (Perl Compatible Regular Expression) such as **.*.example.com**.

Filters the services configured in the 3scale API Manager.

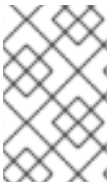
This filter matches with the *Public Base URL* , either staging or production. Services that do not match the filter are discarded. If the regular expression cannot be compiled, no services are loaded.

NOTE

If a service does not match but is **included** in the section called "**APICAST_SERVICES_LIST**", the service will not be discarded.

> **Example 7.1. A Regexp filter applied to backend endpoints**
>
> The Regexp filter **http://.*.foo.dev** is applied to the following backend endpoints:
>
> 1. http://staging.foo.dev
> 2. http://staging.bar.dev
> 3. http://prod.foo.dev
> 4. http://prod.bar.dev
>
> In this case, **1** and **3** are configured in APIcast and **2** and **4** are discarded.

APICAST_UPSTREAM_RETRY_CASES
Values: error | timeout | invalid_header | http_500 | http_502 | http_503 | http_504 | http_403 | http_404 | http_429 | non_idempotent | off



NOTE

This is only used when the retry policy is configured and specifies when a request to the upstream API should be retried. It accepts the same values as Nginx's PROXY_NEXT_UPSTREAM Module.

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**.

HTTP_KEEPALIVE_TIMEOUT
Default: 75 **Value**: positive integers **Example**: *1*

This parameter sets a timeout during which a keep-alive client connection will stay open on the server side. The zero value disables keep-alive client connections.

By default, the gateway keeps HTTP_KEEPALIVE_TIMEOUT disabled. This configuration allows the use of the keepalive timeout from NGINX whose default value is 75 seconds.

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.

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.

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.

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.

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. You must provide either THREESCALE_PORTAL_ENDPOINT or **THREESCALE_CONFIG_FILE** for the gateway to run successfully. From these two environment variables, **THREESCALE_CONFIG_FILE** takes precedence

To build the file with the configuration for the gateway, you have two alternatives depending on the number of services:

- *Option 1.* Download the configuration from the Admin Portal using the URL:

  ```
  <schema>://<admin-portal-domain>/admin/api/nginx/spec.json
  ```

  Consider the following:

  - The endpoint has limitations once 20 services have been exceeded.
  - You can see an example in: **https://account-admin.3scale.net/admin/api/nginx/spec.json**

- *Option 2.* Use the available 3scale API endpoints:

  - Either *Proxy Config Show* or *Proxy Config Show Latest*.

- Then, iterate over the services to build a configuration file. For this step, use the available 3scale API endpoints, or the equivalent 3scale toolbox commands.

When you deploy the gateway using a container image:

1. Configure the file to the image as a read-only volume.
2. Specify the path that indicates where you have mounted the volume.

You can find sample configuration files in examples folder.

THREESCALE_DEPLOYMENT_ENV

Values: staging | production

Default: production

The value of this environment variable defines the environment from which the configuration will be downloaded from; this is either 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 solely for statistics.

THREESCALE_PORTAL_ENDPOINT

URI that includes your password and portal endpoint in the following format:

<schema>://<password>@<admin-portal-domain>.

where:

- **<password>** can be either the provider key or an access token for the 3scale Account Management API.
- **<admin-portal-domain>** is the URL address to log into the 3scale Admin Portal.

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

When the **THREESCALE_PORTAL_ENDPOINT** environment variable is provided, the gateway downloads the configuration from 3scale on initializing. The configuration includes all the settings provided on the Integration page of the APIs.

You can also use this environment variable to create a single gateway with the Master Admin Portal.

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

CHAPTER 8. CONFIGURING APICAST FOR BETTER PERFORMANCE

This document provides general guidelines to debug performance issues in APIcast. It also introduces the available caching modes and explains how they can help in increasing performance, as well as details about profiling modes. The content is structured in the following sections:

8.1. GENERAL GUIDELINES

In a typical APIcast deployment, there are three components to consider:

- APIcast
- The 3scale back-end server that authorizes requests and keeps track of the usage
- The upstream API

When experiencing performance issues in APIcast:

- Identify the component that is responsible for the issues.
- Measure the latency of the upstream API, to determine the latency that APIcast plus the 3scale back-end server introduce.
- With the same tool you are using to run the benchmark, perform a new measurement but pointing to APIcast instead of pointing to the upstream API directly.

Comparing these results will give you an idea of the latency introduced by APIcast and the 3scale back-end server.

In a Hosted (SaaS) installation with self-managed APIcast, if the latency introduced by APIcast and the 3scale back-end server is high:

1. Make a request to the 3scale back-end server from the same machine where APIcast is deployed
2. Measure the latency.

The 3scale back-end server exposes an endpoint that returns the version: https://su1.3scale.net/status. In comparison, an authorization call requires more resources because it verifies keys, limits, and queue background jobs. Although the 3scale back-end server performs these tasks in a few milliseconds, it requires more work than checking the version like the **/status** endpoint does. As an example, if a request to **/status** takes around 300 ms from your APIcast environment, an authorization is going to take more time for every request that is not cached.

8.2. DEFAULT CACHING

For requests that are not cached, these are the events:

1. APIcast extracts the usage metrics from matching mapping rules.
2. APIcast sends the metrics plus the application credentials to the 3scale back-end server.
3. The 3scale back-end server performs the following:
    a. Checks the application keys, and that the reported usage of metrics is within the defined limits.
    b. Queues a background job to increase the usage of the metrics reported.
    c. Responds to APIcast whether the request should be authorized or not.
4. If the request is authorized, it goes to the upstream.

In this case, the request does not arrive to the upstream until the 3scale back-end server responds.

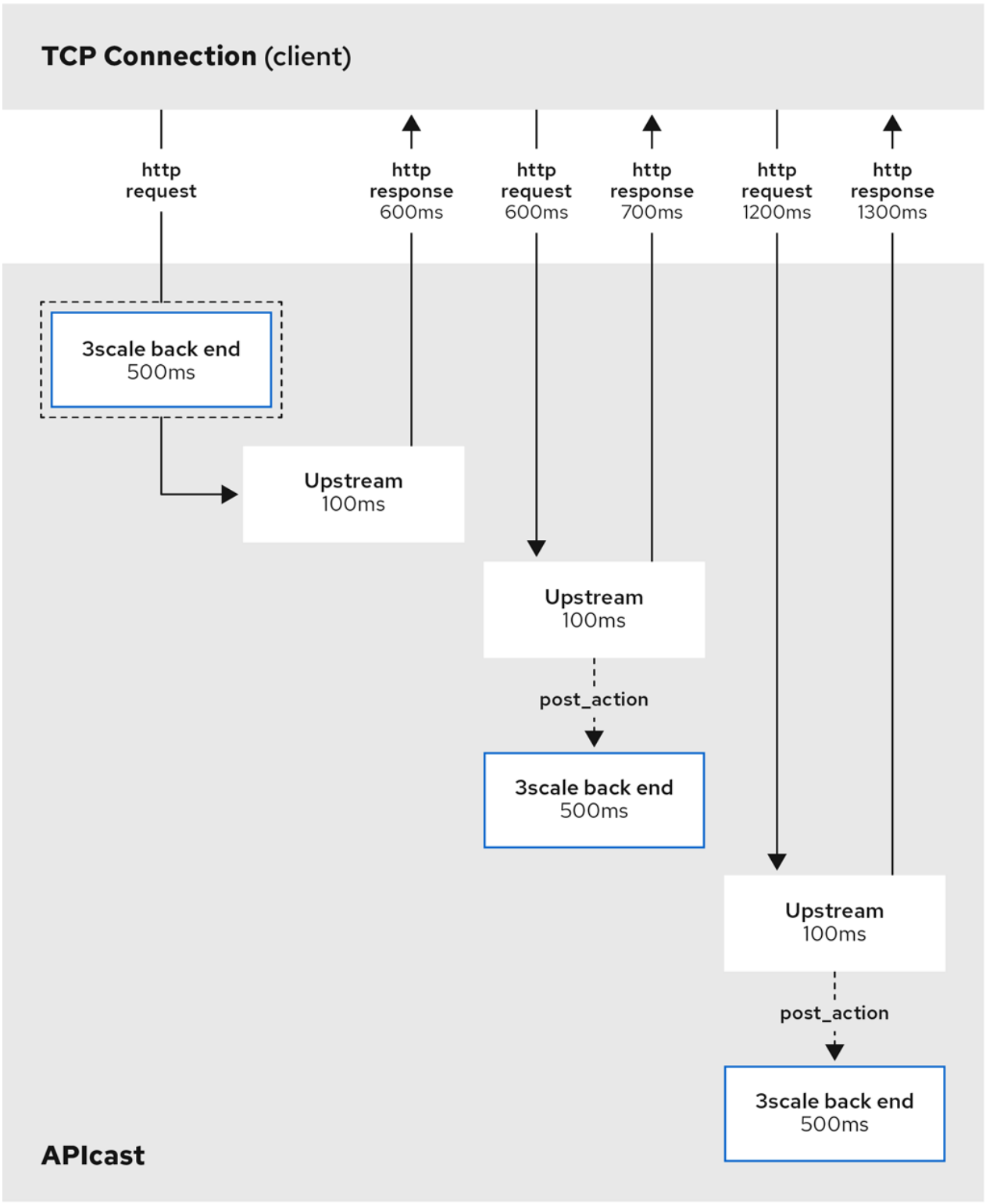
On the other hand, with the caching mechanism that comes enabled by default:

- APIcast stores in a cache the result of the authorization call to the 3scale back-end server if it was authorized.
- The next request with the same credentials and metrics will use that cached authorization instead of going to the 3scale back-end server.
- If the request was not authorized, or if it is the first time that APIcast receives the credentials, APIcast will call the 3scale back-end server synchronously as explained above.

When the authentication is cached, APIcast first calls the upstream and then, in a phase called *post action*, it calls the 3scale back-end server and stores the authorization in the cache to have it ready for the next request. Notice that the call to the 3scale back-end server does not introduce any latency because it does not happen in request time. However, requests sent in the same connection will need to wait until the *post action* phase finishes.

Imagine a scenario where a client is using *keep-alive* and sends a request every second. If the upstream response time is 100 ms and the latency to the 3scale back-end server is 500 ms, the client will get the response every time in 100 ms. The total of upstream response and the reporting would take 600 ms. That gives extra 400 ms before the next request comes.

The diagram below illustrates the default caching behavior explained.The behavior of the caching mechanism can be changed using the caching policy.



8.3. ASYNCHRONOUS REPORTING THREADS

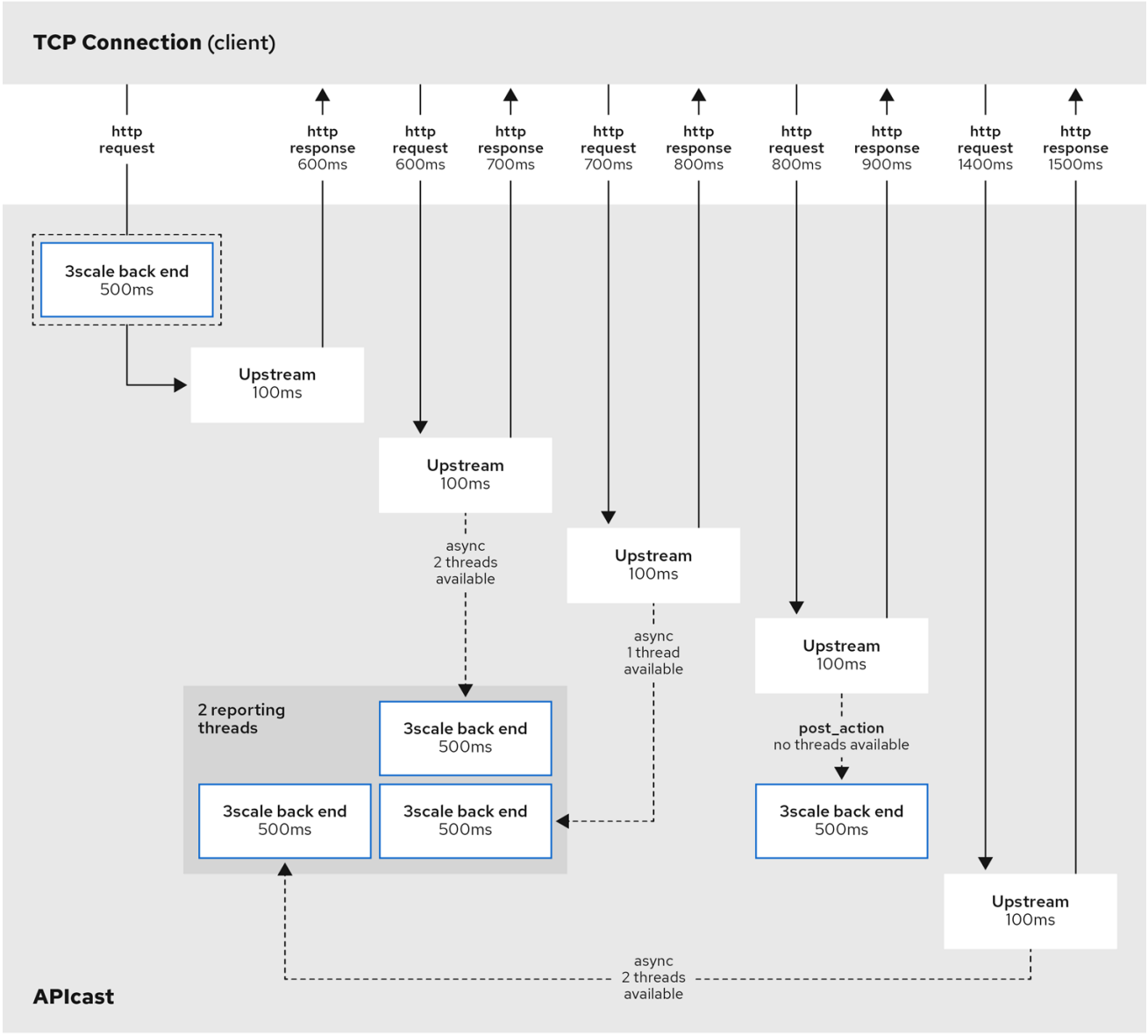
APIcast has a feature to enable a pool of threads that authorize against the 3scale back-end server. With this feature enabled, APIcast first synchronously calls the 3scale back-end server to verify the application and metrics matched by mapping rules. This is similar to when it uses the caching mechanism enabled by default. The difference is that subsequent calls to the 3scale back-end server are reported fully asynchronously as long as there are free reporting threads in the pool.

Reporting threads are global for the whole gateway and shared between all the services. When a second

TCP connection is made, it will also be fully asynchronous as long as the authorization is already cached. When there are no free reporting threads, the synchronous mode falls back to the standard asynchronous mode and does the reporting in the post action phase.

You can enable this feature using the APICAST_REPORTING_THREADS environment variable.

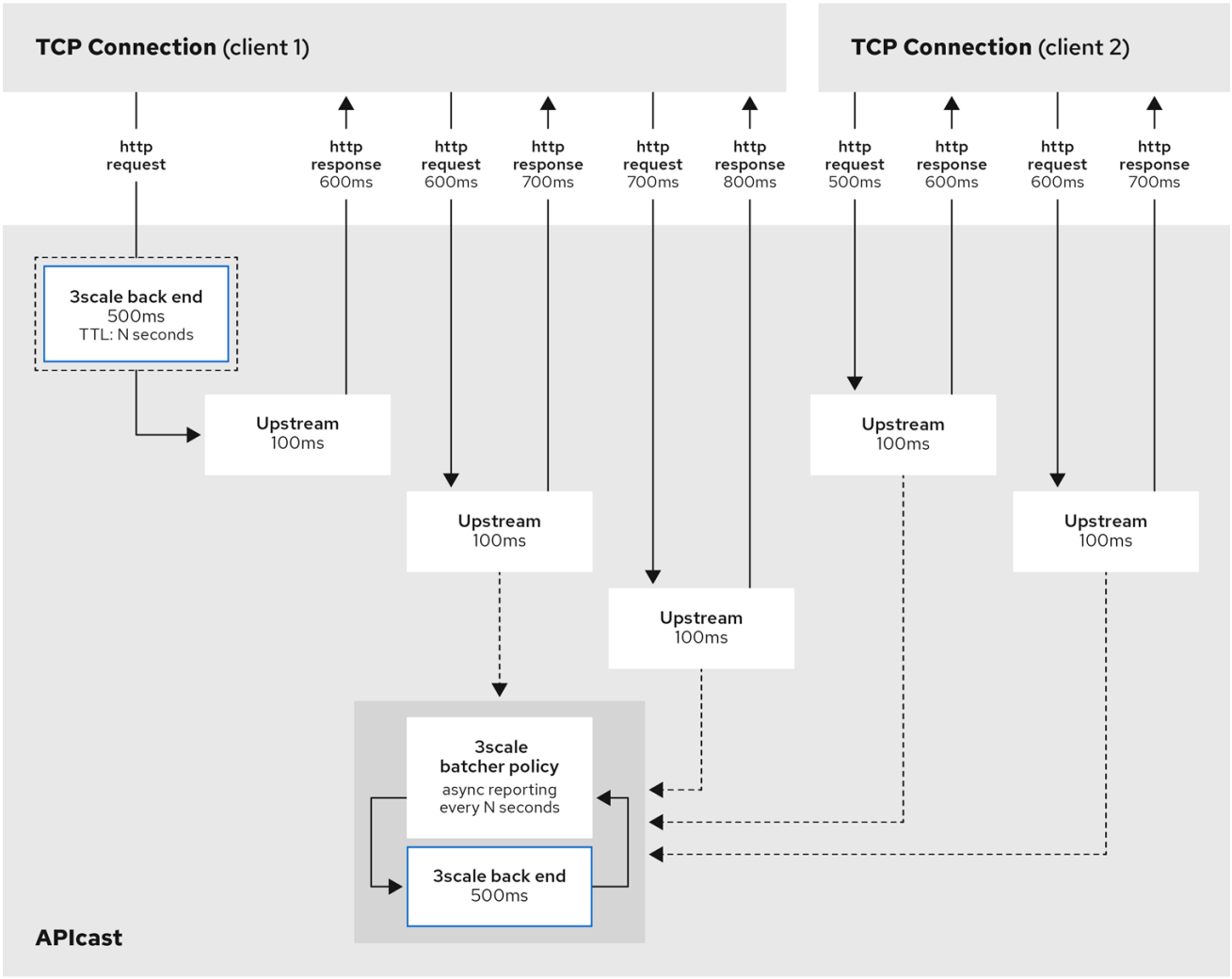
The diagram below illustrates how the asynchronous reporting thread pool works.



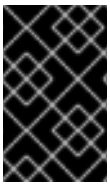
8.4. 3SCALE BATCHER POLICY

By default, APIcast performs one call to the 3scale back-end server for each request that it receives. The goal of the 3scale Batcher policy is to reduce latency and increase throughput by significantly reducing the number of requests made to the 3scale back-end server. In order to achieve that, this policy caches authorization statuses and batches reports.

See 3scale Batcher policy for details. The diagram below illustrates how the policy works.


TCP Connection (client 1)
TCP Connection (client 2)
http request
http response 600ms
http request 600ms
http response 700ms
http request 700ms
http response 800ms
http request 500ms
http response 600ms
http request 600ms
http response 700ms
3scale back end
500ms
TTL: N seconds
Upstream
100ms
Upstream
100ms
Upstream
100ms
Upstream
100ms
Upstream
100ms
3scale
batcher policy
async reporting
every N seconds
3scale back end
500ms
APIcast
3Scale_38_0819

CHAPTER 9. EXPOSING 3SCALE APICAST METRICS TO PROMETHEUS



IMPORTANT

For this release of 3scale, Prometheus installation and configuration are not supported. Optionally, you can use the community version of Prometheus to visualize metrics and alerts for APIcast-managed API services.

9.1. ABOUT PROMETHEUS

Prometheus is an open-source systems monitoring toolkit that you can use to monitor 3scale APIcast services deployed in the Red Hat OpenShift environment.

If you want to monitor your services with Prometheus, your services must expose a Prometheus endpoint. This endpoint is an HTTP interface that exposes a list of metrics and the current value of the metrics. Prometheus periodically scrapes these target-defined endpoints and writes the collected data into its database.

9.1.1. Prometheus queries

In the Prometheus UI, you can write queries in Prometheus Query Language (PromQL) to extract metric information. With PromQL, you can select and aggregate time series data in real time.

For example, you can use the following query to select all the values that Prometheus has recorded within the last 5 minutes for all time series that have the metric name **http_requests_total**:

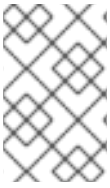
```
http_requests_total[5m]
```

You can further define or filter the results of a query by specifying a *label* (a key:value pair) for the metric. For example, you can use the following query to select all the values that Prometheus has recorded within the last 5 minutes for all time series that have the metric name **http_requests_total** and a **job** label set to **integration**:

```
http_requests_total{job="integration"}[5m]
```

The result of a query can either be shown as a graph, viewed as tabular data in Prometheus's expression browser, or consumed by external systems by using the Prometheus HTTP API. Prometheus provides a graphical view of the data. For a more robust graphical dashboard to view Prometheus metrics, Grafana is a popular choice.

You can also use the the PromQL language to configure alerts in the Prometheus alertmanager tool.



NOTE

Grafana is a community-supported feature. Deploying Grafana to monitor Red Hat 3scale products is not supported with Red Hat production service level agreements (SLAs).

9.2. APICAST INTEGRATION WITH PROMETHEUS

APIcast integration with Prometheus is available for the following deployment options:

- Self-managed APIcast - both with 3scale Hosted or On-premises API manager.
- Embedded APIcast in 3scale On-premises.

NOTE

APIcast integration with Prometheus is not available in hosted API manager and hosted APIcast.

By default, Prometheus can monitor the APIcast metrics listed in Table 9.2, "Prometheus Default Metrics for 3scale APIcast".

9.2.1. Additional options

Optionally, if you have cluster admin access to the OpenShift cluster, you can extend the **total_response_time_seconds**, **upstream_response_time_seconds**, and **upstream_status** metrics to include **service_id** and **service_system_name** labels. To extend these metrics, set the **APICAST_EXTENDED_METRICS** OpenShift environment variable to **true** with this command:

```
oc set env dc/apicast APICAST_EXTENDED_METRICS=true
```

If you use the 3scale Batcher policy (described in Section 4.1.3, "3scale Batcher"), Prometheus can also monitor the metrics listed in Table 9.3, "Prometheus Metrics for 3scale APIcast Batch Policy" .

NOTE

If a metric has no value, Prometheus hides the metric. For example, if **nginx_error_log** has no errors to report, Prometheus does not display the **nginx_error_log** metric. The **nginx_error_log** metric is only visible if it has a value.

Additional resources

For information about Prometheus, refer to Prometheus: Getting Started.

9.3. OPENSHIFT ENVIRONMENT VARIABLES FOR 3SCALE APICAST

To configure your Prometheus instance, you can set the OpenShift environment variable described in Table 9.1, "Prometheus Environment Variables for 3scale APIcast" .

Table 9.1. Prometheus Environment Variables for 3scale APIcast

Environment Variable	Description	Default

Environment Variable	Description	Default
APICAST_EXTENDED_METRICS	A boolean value that 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**	**false**

Additional resources

For information on setting environment variables, see the relevant OpenShift guides:

- OpenShift 4: Applications
- OpenShift 3.11: Developer Guide

For information about supported configurations, refer to the Red Hat 3scale API Management Supported Configurations page.

9.4. 3SCALE APICAST METRICS EXPOSED TO PROMETHEUS

After you set up Prometheus to monitor 3scale APIcast, by default it can monitor the metrics listed in in Table 9.2, "Prometheus Default Metrics for 3scale APIcast" .

The metrics listed in Table 9.3, "Prometheus Metrics for 3scale APIcast Batch Policy" are available only when you use the 3scale Batcher policy.

Table 9.2. Prometheus Default Metrics for 3scale APIcast

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)

Metric	Description	Type	Labels
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	*none*
total_response_time_seconds	Time needed to send a response to the client (in seconds) **Note**: To access the **service_id** and **service_system_name** labels, you must set the **APICAST_EXTENDED_METRICS** environment variable to **true** as described in Section 9.2, "APIcast integration with Prometheus".	histogram	**service_id**, **service_system_name**
upstream_response_time_seconds	Response times from upstream servers (in seconds) **Note**: To access the **service_id** and **service_system_name** labels, you must set the **APICAST_EXTENDED_METRICS** environment variable to **true** as described in Section 9.2, "APIcast integration with Prometheus".	histogram	**service_id**, **service_system_name**

Metric	Description	Type	Labels
upstream_status	HTTP status from upstream servers **Note**: To access the **service_id** and **service_system_name** labels, you must set the **APICAST_EXTENDED_METRICS** environment variable to **true** as described in Section 9.2, "APIcast integration with Prometheus".	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**)

Table 9.3. Prometheus Metrics for 3scale APIcast Batch Policy

Metric	Description	Type	Labels
batching_policy_auths_cache_hits	Hits in the auths cache of the 3scale batching policy	counter	*none*
batching_policy_auths_cache_misses	Misses in the auths cache of the 3scale batching policy	counter	*none*

PART II. API VERSIONING

CHAPTER 10. API VERSIONING

Red Hat 3scale API Management allows API versioning. You have three ways to version your API correctly when you manage your API with 3scale. The following methods are examples of how you could version your API within the 3scale Gateway, which provides extra features due to 3scale architecture.

10.1. GOAL

This guide is designed to give you enough information to implement an API versioning system within 3scale.

Suppose you have an API for finding songs. Users can search for their favorite songs by different keywords: artist, songwriter, song title, album title, and so on. Assume you had an initial version (v1) of the API and now you have developed a new, improved version (v2).

The following sections describe the three most typical ways of implementing an API versioning system using 3scale:

- URL versioning
- Endpoint versioning
- Custom header versioning

10.2. PREREQUISITES

- Complete the First steps with 3scale before using this quick start guide.

10.3. URL VERSIONING

If you have different endpoints for searching songs (by artist, by song title, and so on), with URL versioning you would include the API version as part of the URI, for example:

1. api.songs.com/v1/songwriter
2. api.songs.com/v2/songwriter
3. api.songs.com/v1/song
4. api.songs.com/v2/song
5. and so on

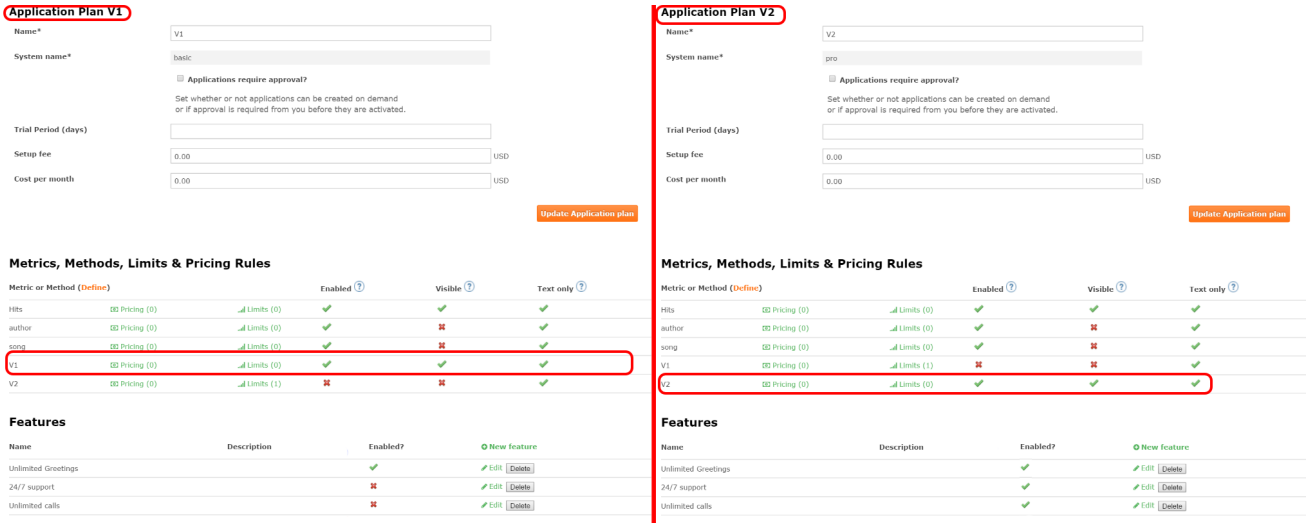
NOTE

When you use this method, you should have planned since v1 that you were going to version your API.

The 3scale Gateway would then extract the endpoint and the version from the URI. This approach allows you to set up application plans for any version/endpoint combination. You can then associate metrics with those plans and endpoints, and you can chart the usage for each endpoint on each version.

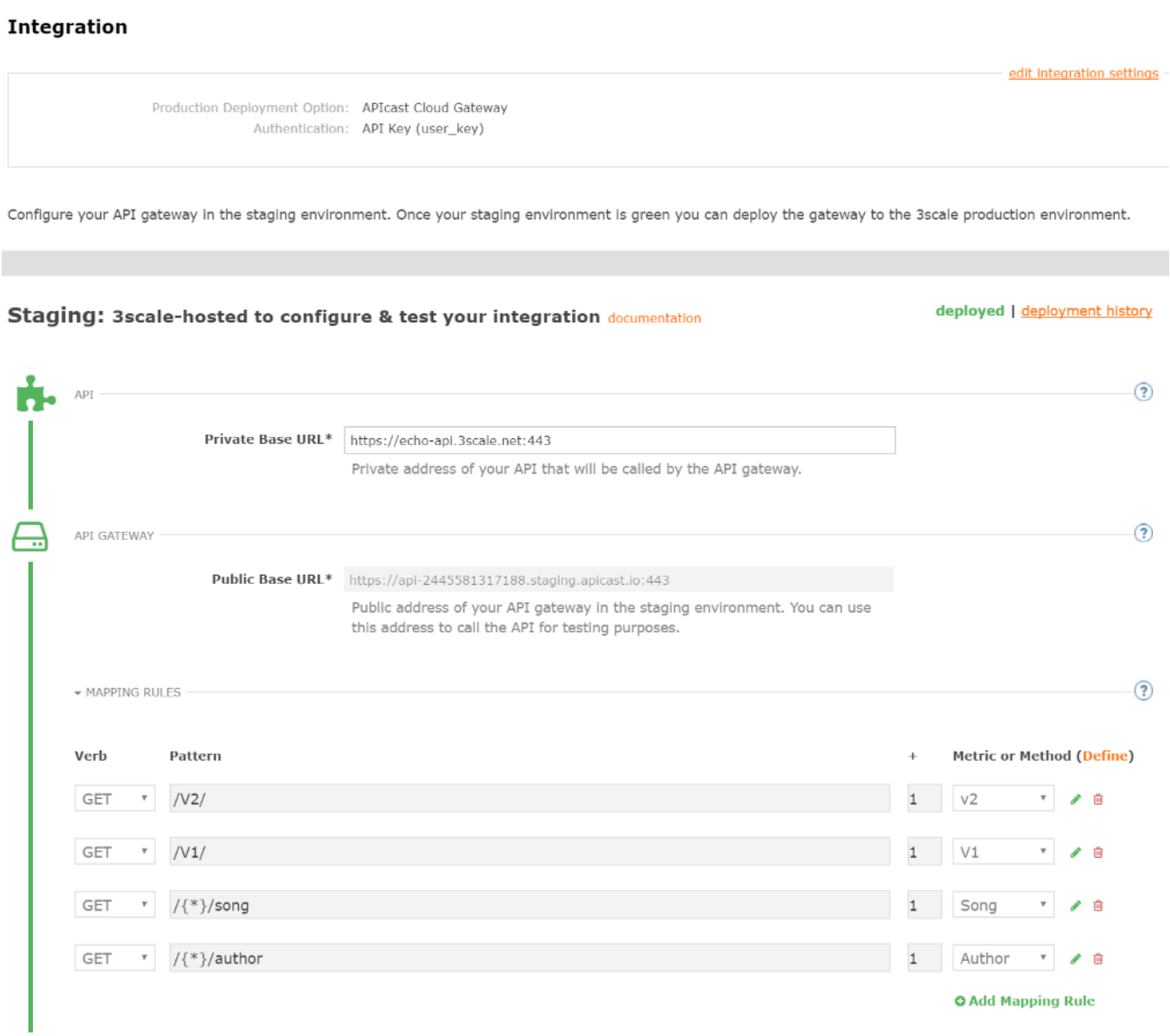
The following screen capture shows 3scale's flexibility.

Figure 10.1. Versioning Plan Feature



The only thing left to do is go to **[your_API_name] > Integration > Configuration**in your 3scale Admin Portal and map your URIs to your metrics, as shown in the following diagram.

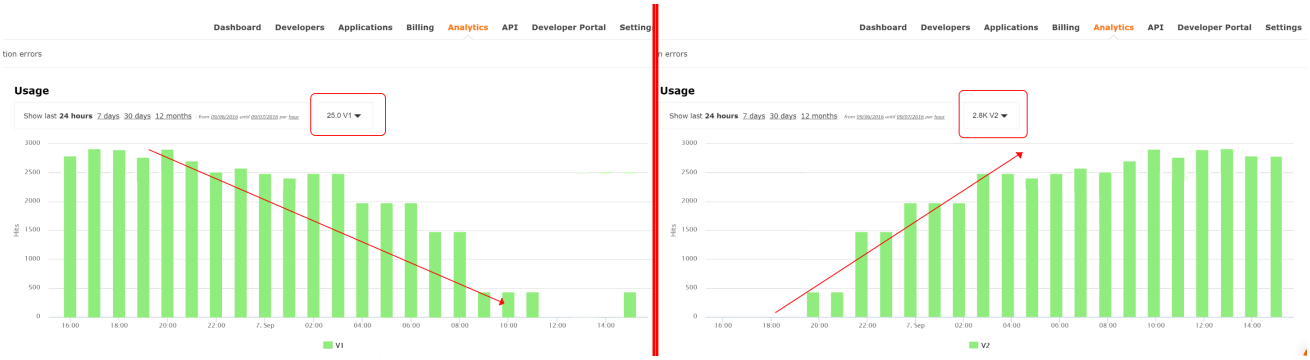
Figure 10.2. Mapping URIs to metrics



You now have two different versions of your API, each with different features enabled. You also have full control and visibility on their usage.

If you want to communicate to all of your users that they should move to the API v2, you can send an internal note asking them to do so. You can monitor who makes the move and see how the activity on v1 decreases while the activity on v2 increases. By adding the metric in your authorization calls to 3scale, you can see how much overall traffic is hitting v1 vs. v2 endpoints and get an idea of when it is safe to deprecate v1.

Figure 10.3. Versioning



If some users continue to use v1, you can filter out only those users to send another internal note about switching to v2.

3scale provides a three-step method for sending deprecation notices.

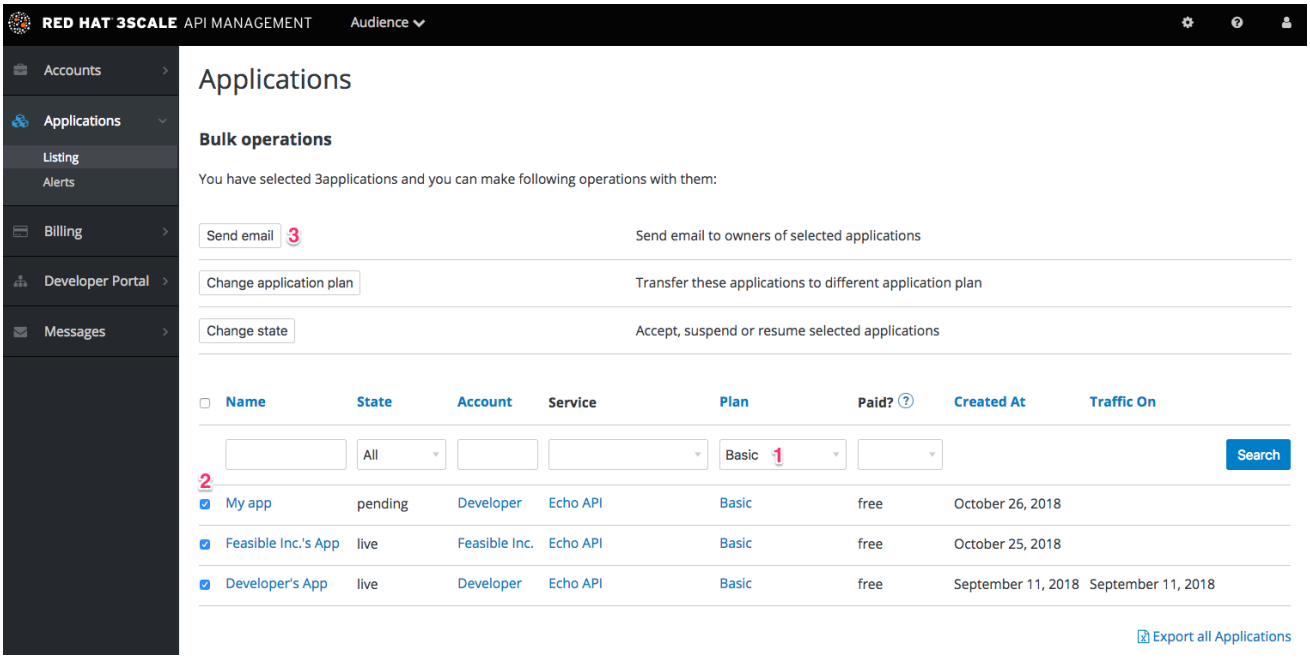
1. Navigate to **Audience > Applications > Listing**and filter the list by the application plan that you want to send the deprecation note and click **Search**.

2. Click the multiselector to select all of the applications for that particular version. New options display and allow you to perform bulk operations, such as **Send email**, **Change Application Plan**, and **Change State**.

3. Click **Send email** and follow the steps to send a deprecation notice to the owners of the selected applications.

The following image provides a visual reference.

Figure 10.4. Sending deprecation note



For each authrep call that is made to an endpoint, you authenticate only once but report twice: once for the endpoint and once for the API version. There is no double-billing because the call can be authenticated only one time. For each call you make to any endpoint of a specific API version, you aggregate the hits on a convenient metric named after the version number (v1, v2, and so on), which you can use to compare full version traffic with each other.

10.4. ENDPOINT VERSIONING

With endpoint versioning, you can have a different endpoint for each API version such as **api.cons.com/author_v1**. The gateway extracts the endpoint and the version from the endpoint itself. This method , as well as the previous method, allows the API provider to map external URLs to internal ones.

The endpoint versioning method can only be performed with the on-premise deployment method as it requires a URL rewrite using the LUA scripts that are provided as part of the on-premise configuration.

EXTERNAL		INTERNAL
api.songs.com/songwriter_v1	could be rewritten to	internal.songs.com/search_by_songwriter
api.songs.com/songwriter_v2	could be rewritten to	internal.songs.com/songwriter

Almost everything (mapping, application plans features, and so on.) works exactly the same as in the previous method.

10.5. CUSTOM HEADER VERSIONING

With custom header versioning, you use a header (that is, "x-api-version") instead of the URI to specify the version.

The gateway then extracts the endpoint from the path and the version from the header. Just as before, you can analyze and visualize any combination of path/version that you want. This approach has several inconveniences, regardless of the API management system you use. See API versioning methods, a brief reference for more information. Here are a few pointers on how 3scale works.

- Just like the previous method, custom header versioning can only be applied to on-premise hosted APIs because it requires some parsing/processing of the request headers to correctly route the authrep calls. This type of custom processing can only be done using Lua scripting.
- With this method, the fine-grained feature separation of the previous methods is much harder to achieve.
- The most important advantage of this method is that the URL and endpoints specified by the developers never change. When a developer wants to switch from one API version to another, they only have to change the header. Everything else works the same.

PART III. API AUTHENTICATION

CHAPTER 11. AUTHENTICATION PATTERNS

By the end of this tutorial you will know how to set the authentication pattern on your API and the effect that this has on applications communicating with your API.

Depending on your API, you may need to use different authentication patterns to issue credentials for access to your API. These can range from API keys to openAuth tokens and custom configurations. This tutorial covers how to select from the available standard Authentication Patterns.

11.1. SUPPORTED AUTHENTICATION PATTERNS

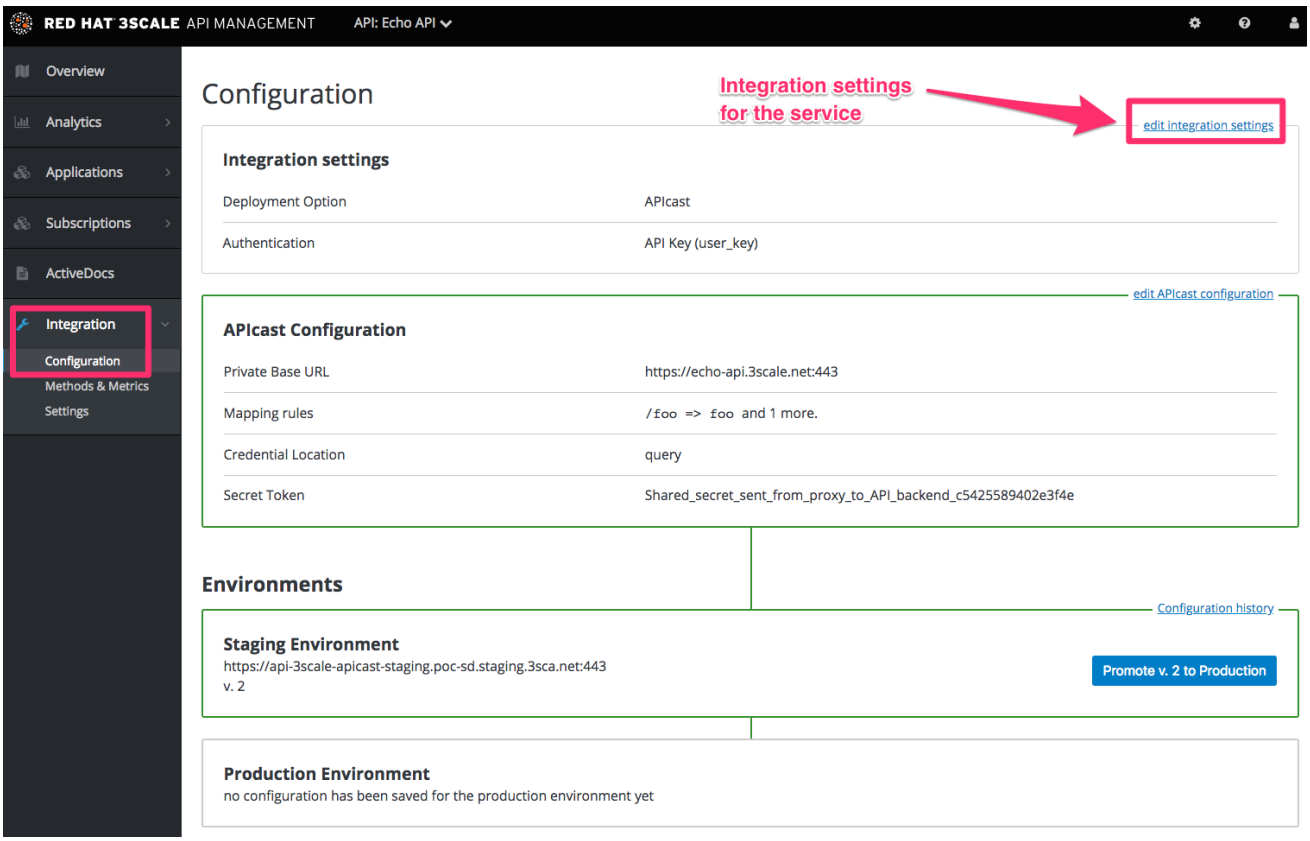
3scale supports the following authentication patterns out of the box:

- **Standard API Keys**: Single randomized strings or hashes acting as an identifier and a secret token.
- **Application Identifier and Key pairs**: Immutable identifier and mutable secret key strings.
- **OpenID Connect**

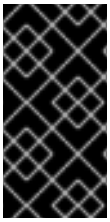
11.2. SETTING UP AUTHENTICATION PATTERNS

11.2.1. Select the authentication mode for your service

Navigate to the API service you want to work on (there may be only one service named API in which case select this). Go to the **Integration** section.



Each service that you operate can use a different authentication pattern, but only one pattern can be used per service.



IMPORTANT

You must not change the authentication pattern after the credentials have been registered because the behavior of the service may then become unpredictable. To change authentication patterns we recommend creating a new service and migrating customers.

11.2.2. Select the Authentication mode you want to use

To select an authentication mode, scroll to the AUTHENTICATION section. Here, you can choose one of the following options:

- API Key (user_key)
- App_ID and App_Key Pair
- OpenID Connect

11.2.3. Ensure your API accepts the correct types of credentials

Depending on the credential type chosen, you may need to accept different parameters in your API calls (key fields, IDs etc.). The names of these parameters may not be the same as those used internally at 3scale. The 3scale authentication will function correctly if the correct parameter names are used in calls to the 3scale backend.

11.2.4. Create an application to test credentials

To ensure that the credential sets are working, you can create a new application to issue credentials to use the API. Navigate to the Accounts area of your Admin Portal Dashboard, click the account you want to use and click **New application**.

Filling out the form and clicking save will create a new application with credentials to use the API. You can now use these credentials to make calls to your API and records will be checked against the list of applications registered in 3scale.

11.3. STANDARD AUTHENTICATION PATTERNS

3scale supports the authentication patterns detailed in the following sections.

11.3.1. API key

The simplest form of credential supported is the single API model. Here, each application with permissions on the API has a single (unique) long character string; example:

```
API-key = 853a76f7c8d5f4a1ee8bf10a4e0d1f13
```

By default, the name of the key parameter is **user_key**. You can use this label or choose another, such as **API-key**. If choosing another label, you need to map the value before you make the authorization calls to 3scale. The string acts as both, an identifier and a secret token, for use of the API. It is recommended that you use such patterns only in environments with low security requirements or with SSL security on API calls. Following are the operations that can be carried out on the token and application:

- Application Suspend: This suspends the applications access to the API and, in effect, all calls to the API with the relevant key will be suspended.

- Application Resume: Undoes the effect of an application suspend action.
- Key Regenerate: This action generates a new random string key for the application and associates it with the application. Immediately after this action is taken, calls with the previous token will cease to be accepted.

The latter action can be triggered from the API Administration in the Admin Portal and (if permitted) from the API Developers User console.

11.3.2. App_ID and App_Key pair

The API Key Pattern combines the identity of the application and the secret usage token in one token; however, this pattern separates the two:

- Each application using the API, issues an immutable initial identifier known as the **Application ID** (App ID). The App ID is constant and may or may not be secret.
- In addition, each application can have between one and five **Application Keys** (App_Keys). Each Key is associated directly with the App_ID and should be treated as secret.

```
app_id = 80a4e03 app_key = a1ee8bf10a4e0d1f13853a76f7c8d5f4
```

In the default setting, developers can create up to five keys per application. This allows a developer to create a new key, add it to their code, redeploy their application, and then disable old keys. This does not cause any application downtime the way an API Key Regeneration would.

Statistics and rate limits are always kept at the application ID level of granularity and not per API Key. If a developer wants to track two sets of statistics, they should create two applications rather than two keys.

It is also possible to change the mode in the system and allow applications to be created in the absence of application keys. In this case the 3scale system will authenticate access based on the App ID only (and no key checks are made). This mode is useful for widget type scenarios or where rate limits are applied to users rather than applications. In most cases you will want your API to enforce the presence of at least one application key per application present. This setting is available in **[your_API_name] > Integration > Settings**.

11.3.3. OpenID Connect

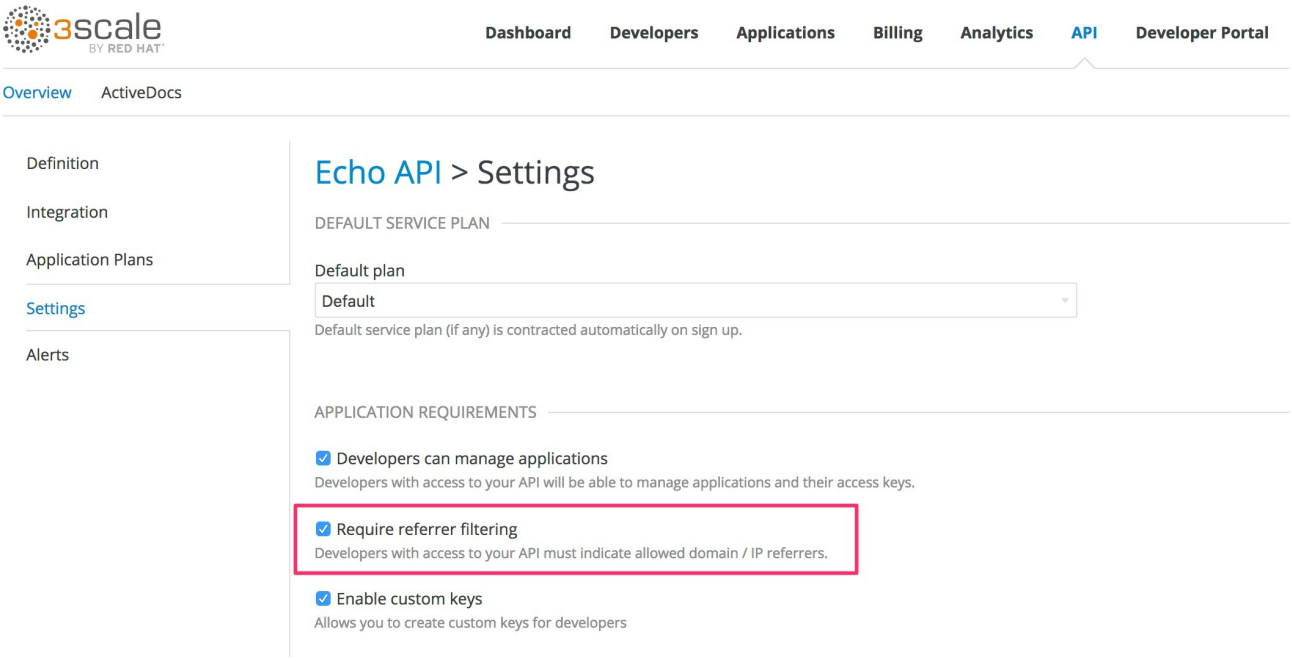
For information on OpenID Connect authentication, see OpenID Connect integration chapter.

11.4. REFERRER FILTERING

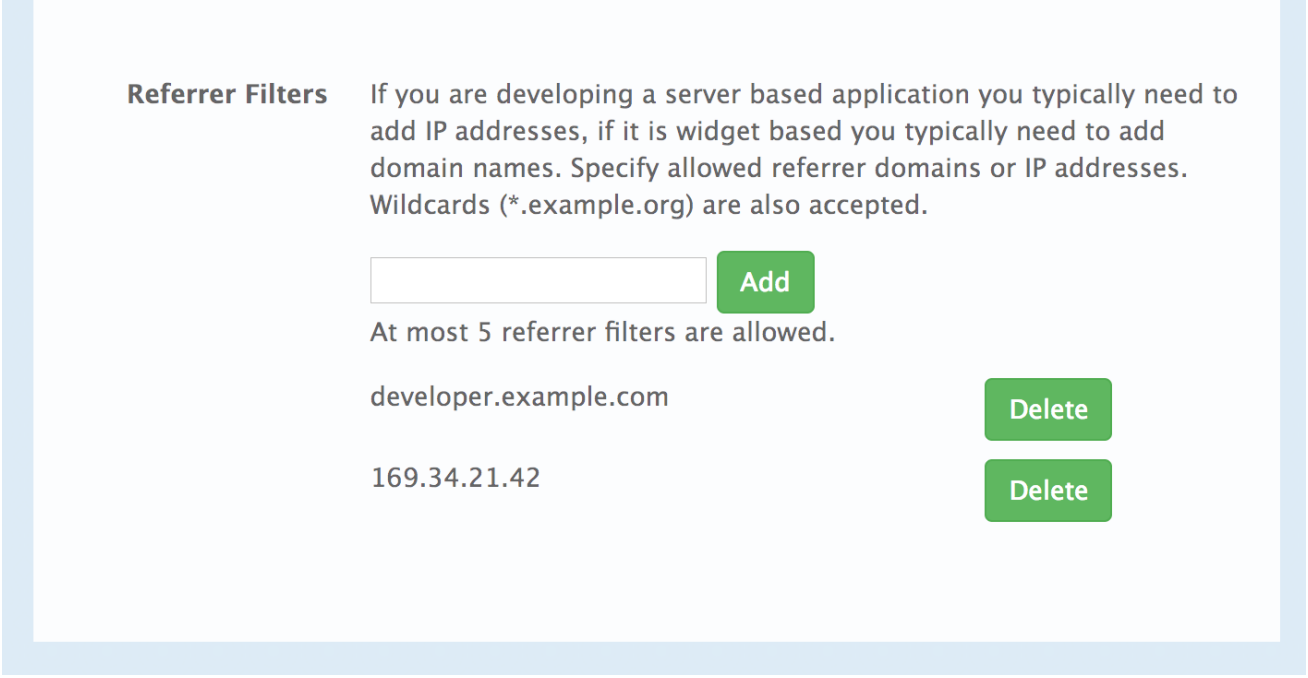
3scale supports the Referrer Filtering feature that can be used to whitelist IP addresses or domain names from where an application can access the API. The API clients specify the referrer value in the **Referrer** header. The purpose and the usage of the Referrer header are described in the RFC 7231, section 5.5.2: Referer.

For the Referrer Filtering feature to work, you must enable the APIcast Referrer policy in the service policy chain.

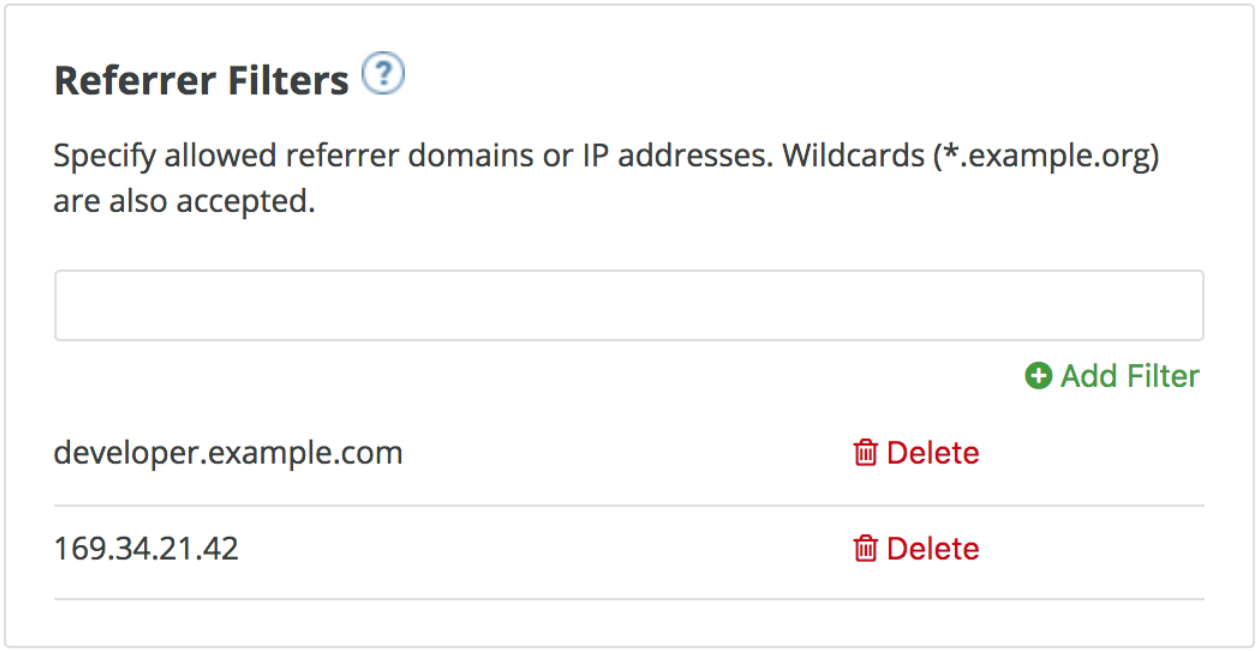
To enable the Referrer Filtering feature, go to **[your_API_name] > Applications > Settings > Usage Rules**. Select **Require referrer filtering** and click **Update Product**.



The developers with access to your API must configure allowed domain/IP referrers from the developer portal.



In the Admin Portal on the application details page for all applications that belong to this service a new **Referrer Filters** section displays. Here, the admin can also configure a whitelist of the allowed Referrer header values for this application.



You can set a maximum of five referrer values per application.

The value can only consist of Latin letters, numbers, and special characters *, ., and -. * can be used for wildcard values. If the value is set to *, any referrer value will be allowed, so the referrer check will be bypassed.

When the **Require referrer filtering** feature and the **3scale Referrer** policy are enabled, the authorization works as follows:

1. The applications that do not have Referrer Filters specified are authorized normally only using the provided credentials.

2. For the applications that have Referrer Filters values set, APIcast extracts the referrer value from the **Referer** header of the request and sends it as **referrer** param in the AuthRep (authorize and report) request to the Service Management API. The following table shows the AuthRep responses for different combination of the referrer filtering parameters.

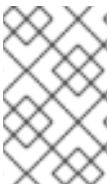
referrer parameter passed?	Referrer Filters configured for the app?	Referrer parameter value	HTTP Response	Response body
Yes	Yes	matches referrer filter	200 OK	**<status> <authorized>true</authorized> </status>**
Yes	No	matches referrer filter	200 OK	**<status> <authorized>true</authorized> </status>**

referrer parameter passed?	Referrer Filters configured for the app?	Referrer parameter value	HTTP Response	Response body
Yes	Yes	does not match referrer filter	409 Conflict	**<status><authorized>false</authorized><reason>referrer "test.example.com" is not allowed</reason>** (**test.example.com** is an example)
Yes	No	does not match referrer filter	200 OK	**<status><authorized>true</authorized></status>**
Yes	Yes	*	200 OK	**<status><authorized>true</authorized></status>**
Yes	No	*	200 OK	**<status><authorized>true</authorized></status>**
No	Yes	–	409 Conflict	**<status><authorized>false</authorized><reason>referrer is missing</reason>**
No	No	–	200 OK	**<status><authorized>true</authorized></status>**

The calls that are not authorized by AuthRep are rejected by APIcast with an "Authorization Failed" error. You can configure the exact status code and the error message on the service Integration page.

CHAPTER 12. INTEGRATING 3SCALE WITH AN OPENID CONNECT IDENTITY PROVIDER

To authenticate API requests, 3scale can integrate with an identity provider that complies with the OpenID Connect specification. For full compatibility with 3scale, the identity provider can be Red Hat Single Sign-On (RH-SSO) or a third-party identity provider that implements default Keycloak client registration. For compatibility with the 3scale API gateway (APIcast), any identity provider that implements OpenID Connect can be used.



NOTE

3scale does not use the RFC 7591 Dynamic Client Registration Mechanism . For full compatibility between 3scale and an OpenID Connect identity provider, there is a dependency on default Keycloak client registration.

The foundation for OpenID Connect is the OAuth 2.0 Authorization Framework (RFC 6749). OpenID Connect uses a JSON Web Token (JWT) (RFC 7519) in an API request to authenticate that request. When you integrate 3scale with an OpenID Connect identity provider, the process has two main parts:

- APIcast parses and verifies the JWT in the request. If successful, APIcast authenticates the identity of the API consumer client application.
- The 3scale Zync component synchronizes 3scale application details with the OpenID Connect identity provider.

3scale supports both of these integration points when RH-SSO is the OpenID Connect identity provider. See the supported version of RH-SSO on the Supported Configurations page. However, RH-SSO is not a requirement. You can use any identity provider that supports the OpenID Connect specification and the default Keycloak client registration. APIcast integration is tested with RH-SSO and ForgeRock.

The following sections provide information and instructions for configuring 3scale to use an OpenID Connect identity provider:

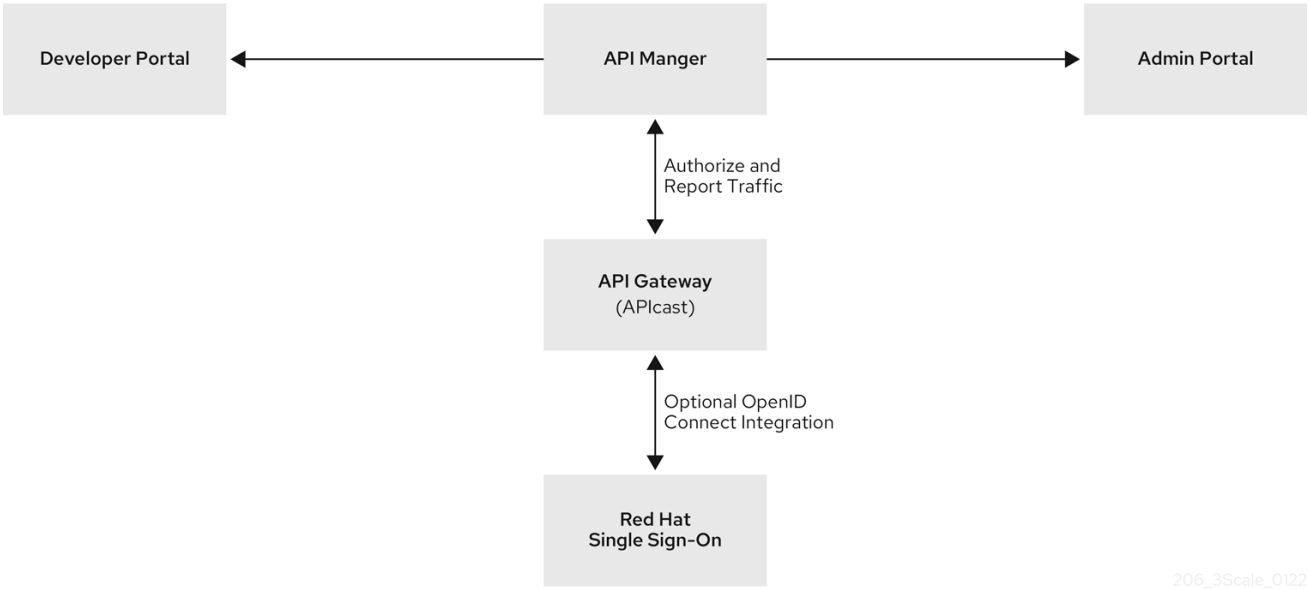
- Overview of integrating 3scale and an OpenID Connect identity provider
- How APIcast processes JSON Web Tokens
- How 3scale Zync synchronizes application details with OpenID Connect identity providers
- Integrating 3scale with Red Hat Single Sign-On as the OpenID Connect identity provider
- Integrating 3scale with third-party OpenID Connect identity providers
- Testing 3scale with OpenID Connect identity providers
- Example of a 3scale integration with and OpenID Connect identity provider

12.1. OVERVIEW OF INTEGRATING 3SCALE AND AN OPENID CONNECT IDENTITY PROVIDER

Each main 3scale component participates in authentication as follows:

- APIcast verifies the authenticity of the authentication tokens presented by API consumer applications. In a default 3scale deployment, APIcast can do this because it implements auto discovery of an API product's OpenID Connect configuration.
- API providers use the Admin Portal to set up authentication flows.
- If a 3scale-managed API does not authenticate requests with standard API keys or with application identifier and key pairs, then API providers must integrate 3scale with an OpenID Connect identity provider. In the figure below, the OpenID Connect identity provider is Red Hat Single Sign-On (RH-SSO).
- With authentication configured and a live Developer Portal, API consumers use your Developer Portal to subscribe to an application plan that provides access to a particular 3scale API product.
- When OpenID Connect is integrated with 3scale, subscription triggers the configured flow for API consumer applications to obtain JSON Web Tokens (JWTs) from the OpenID Connect identity provider. An API provider specifies this flow when configuring the API product to use OpenID Connect.

Figure 12.1. shows the main 3scale components with an OpenID Connect identity provider



After subscribing to an application plan, an API consumer receives authentication credentials from the integrated OpenID Connect identity provider. These credentials enable authentication of requests that an API consumer application sends to an upstream API, which is the API that is provided by the 3scale API product that the API consumer has access to.

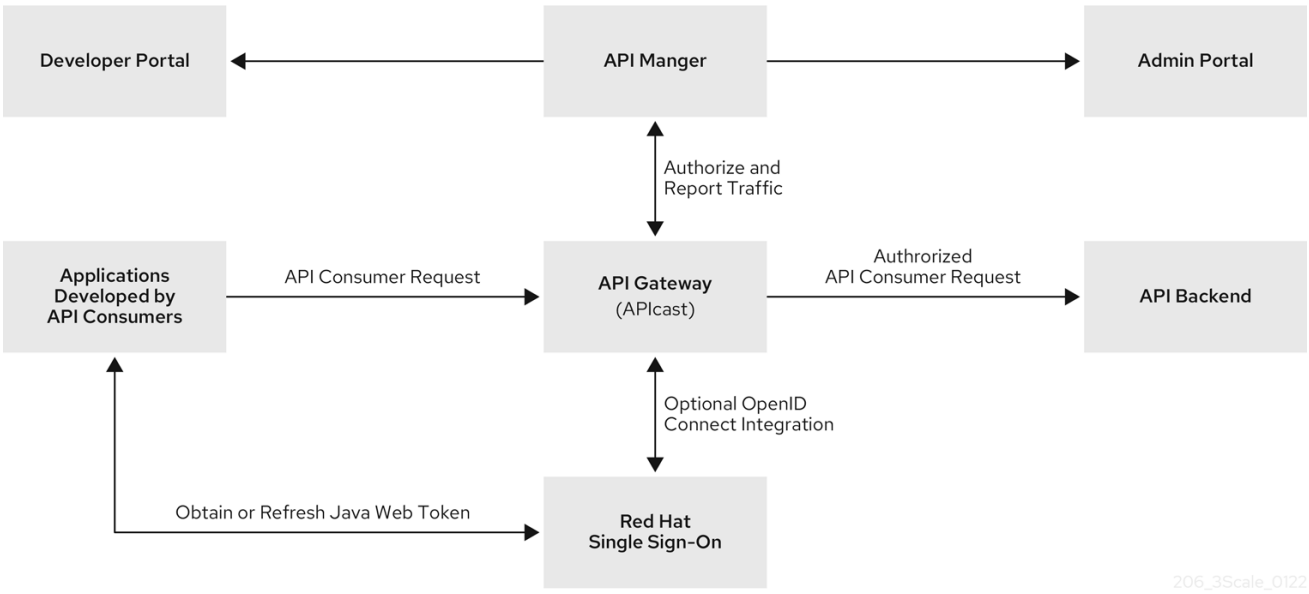
Credentials include a client ID and a client secret. An application created by an API consumer uses these credentials to obtain a JSON Web Token (JWT) from the OpenID Connect identity provider. When you configure your 3scale integration with OpenID Connect, you select the flow for how an API consumer application obtains a JWT. In an API consumer application that uses the default **Authorization Code** flow with RH-SSO, the application must do the following:

1. Initiate an OAuth authorization flow with the OpenID Connect identity provider before the first request to the upstream API backend. The authorization code flow redirects the end-user to RH-SSO. The end-user logs in to obtain an authorization code.
2. Exchange the authorization code for a JWT.

3. Receive a JWT from RH-SSO upon authentication.
4. Send an API request that contains the JWT to the upstream API backend.
5. Send subsequent API requests with the same JWT until it expires.
6. Refresh the JWT or send a new request to the OpenID Connect identity provider to obtain a new JWT. Which action is required depends on the OpenID Connect identity provider.

APIcast receives requests from API consumers and checks the JWT in the request. If APIcast verifies the JWT, APIcast sends the request, including the JWT, to the upstream API backend.

Figure 12.2. shows that the OpenID Connect identity provider is RH-SSO but configuration with other OpenID Connect identity providers is possible.



12.2. HOW APICAST PROCESSES JSON WEB TOKENS

APIcast processes each request by checking the JSON Web Token (JWT) that the OpenID Connect identity provider returned when it authenticated an API consumer application. The request contains the JWT in the format that was issued by the integrated OpenID Connect identity provider. The JWT must be in the **Authorization** header and it must use the **Bearer** schema. For example, the header should look like this:

```
Authorization: Bearer
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9.eyJpc3MiOiJodHRwczovL2lkcC5leGFtcGxlLmNvbSIsInN1YiI
6ImFiYzEyMyIsIm5iZiI6MTUzNzg5MjQ5NCwiZXhwIjoxNTM3ODk2MDk0LCJpYXQiOjE1Mzc4OTI0OT
QsImp0aSI6ImlkMTIzNDU2IiwidHlwIjoiQmVhcmVyIn0.LM2PSmQ0k8mR7eDS_Z8iRdGta-Ea-
pJRrf4C6bAiKz-Nzhxpm7fF7oV3BOipFmimwkQ_-mw3kN--
oOc3vU1RE4FTCQGbzO1SAWHOZqG5ZUx5ugaASY-
hUHIohy6PC7dQl0e2NlAeqqg4MuZtEwrpESJW-
VnGdljrAS0HsXzd6nENM0Z_ofo4ZdTKvIKsk2KrdyVBOcjgVjYongtppR0cw30FwnpqfeCkuATeINN5OK
HXOibRA24pQyIF1s81nnmxLnjnVbu24SFE34aMGRXYzs4icMI8sK65eKxbvwV3PIG3mM0C4ilZPO26d
oP0YrLfVwFcqEirmENUAcHXz7NuvA
```

There are a number of open source tools for securely decoding a JWT. Be careful that you do not decode a JWT in a public web tool. In a decoded JWT, you can see that the token has three parts:

- The header provides information about how the token was formed and what algorithm was used to sign the token.
- The payload identifies the API consumer that sent the request. The details can include the read and write actions that this API consumer can perform, an email address for the API consumer, and other information about the API consumer.
- The signature is a cryptographic signature that indicates that the token has not been altered.

APIcast checks the JWT for the following characteristics:

- **Integrity**: Is the JWT being altered by a malicious user? Is the signature valid?
  The JWT contains a signature that the token's receiver can verify to ensure that a known issuer signed the token. This verification also guarantees that its content remains as created. 3scale supports RSA signatures based on public/private key pairs. The issuer signs the JWT by using a private key. APIcast verifies the token by using a public key. APIcast uses OpenID Connect Discovery for getting the JSON Web Keys (JWK) that can be used to verify the JWT signature.
- **Timing**: Is the current time later than the time when the token becomes acceptable for processing? Has the JWT expired? In other words, APIcast checks the JWT **nbf** (not before time) and **exp** (expiration time) claims.
- **Issuer**: Was the JWT issued by an OpenID Connect identity provider that is known to APIcast? In other words, APIcast verifies that the issuer specified in the JWT is the same issuer that an API provider configured in the **OpenID Connect Issuer** field. Specification of the issuer is part of the procedure for integrating 3scale and an OpenID Connect identity provider. This is the JWT **iss** claim.
- **Client ID**: Does the token contain a 3scale client application ID that is known to APIcast? This client ID must match a **ClientID Token Claim** that the API provider specified in the procedure for integrating 3scale with the OpenID Connect identity provider. This is the JWT **azp** (authorized party that the JWT was issued to) claim and the **aud** (audience) claim.

If any JWT validation or authorization checks fail, APIcast returns an **Authentication failed** error. Otherwise, APIcast sends the request to the 3scale upstream API backend. The **Authorization** header remains in the request, so the API backend can also use the JWT to check the user and client identity.

12.3. HOW 3SCALE ZYNC SYNCHRONIZES APPLICATION DETAILS WITH OPENID CONNECT IDENTITY PROVIDERS

Zync is a 3scale component that reliably pushes data about 3scale applications to an OpenID Connect identity provider. In this interaction, a 3scale application corresponds to an OpenID Connect identity provider client. In other words, Zync communicates with the OpenID Connect identity provider to create, update, and delete OpenID Connect clients.

Zync implements Keycloak default client registration . The use of this API means that the client representation is specific to Keycloak and RH-SSO. The identity provider returns a client ID and a client secret, which are the authentication credentials for the 3scale application.

Each time 3scale creates, updates, or deletes an application, Zync communicates with the OpenID Connect identity provider to update the corresponding client accordingly. Successful synchronization requires the following settings for a given 3scale product:

- The authentication mechanism is OpenID Connect.
- The **OpenID Connect Issuer Type** is either:

    - **RH-SSO** when Red Hat Single Sign-On is the OpenID Connect identity provider. With this issuer type, Zync sends client registration requests to the Keycloak/RH-SSO default client registration API.

    - **REST API** for other OpenID Connect identity providers. With this issuer type, Zync sends client registration requests as shown in the Zync REST API example .

- A URL such as the following is the **OpenID Connect Issuer**: **http://id:secret@example.com/api_endpoint**.

When deployed to an OpenShift cluster, there are two Zync processes:

- **zync** is a REST API that receives notifications from **system-sidekiq** and enqueues background jobs to **zync-que**. There are notifications for new, updated, and deleted 3scale applications.

- **zync-que** processes these background jobs, which communicate with **system-app** and with the OpenID Connect identity provider. For example, when RH-SSO is the configured OpenID Connect identity provider, Zync creates, updates and deletes clients in the RH-SSO realm.

12.4. INTEGRATING 3SCALE WITH RED HAT SINGLE SIGN-ON AS THE OPENID CONNECT IDENTITY PROVIDER

As an API provider, you can integrate 3scale with Red Hat Single Sign-On (RH-SSO) as the OpenID Connect identity provider. The procedures documented here are for a 3scale API product that requires OpenID Connect for authenticating API requests.

Part of this procedure is to establish an SSL connection between 3scale Zync and RH-SSO, because Zync communicates with RH-SSO to exchange tokens. If you do not configure the SSL connection between Zync and RH-SSO, the tokens would be open for anyone listening.

3scale 2.2 and later versions support custom CA certificates for RH-SSO with the SSL_CERT_FILE environment variable. This variable points to the local path of the certificates bundle. Integrating 3scale with RH-SSO as the OpenID Connect identity provider consists of configuring the following elements in the following order:

- If RH-SSO does *not* use a certificate issued by a trusted Certificate Authority (CA), you must configure 3scale Zync to use custom CA certificates. This is not required if RH-SSO uses a certificate issued by a trusted CA.

- Configure RH-SSO to have a 3scale client.

- Configure 3scale to work with RH-SSO.

Prerequisites

- The RH-SSO server must be available over **HTTPS** and it must be reachable by **zync-que**. To test this, you can run **curl https://rhsso-fqdn** from within the **zync-que** pod, for example:

```
oc rsh -n $THREESCALE_PROJECT $(oc get pods -n $THREESCALE_PROJECT --field-selector=status.phase==Running -o name | grep zync-que) /bin/bash -c "curl -v https://<rhsso-fqdn>/auth/realms/master"
```

- OpenShift cluster administrator permissions.

- A 3scale API product for which you want to configure OpenID Connect integration with RH-SSO.

See the following sections for details:

- Configuring 3scale Zync to use custom Certificate Authority certificates
- Configuring RH-SSO to have a 3scale client
- Configuring 3scale to work with RH-SSO

12.4.1. Configuring 3scale Zync to use custom Certificate Authority certificates

This is not required when RH-SSO uses a certificate issued by a trusted Certificate Authority (CA). However, if RH-SSO does *not* use a certificate issued by a trusted CA you must configure 3scale Zync before you configure RH-SSO to have a 3scale client and before you configure 3scale to work with RH-SSO.

Procedure

1. Obtain a CA certificate chain in **.pem** format and save each certificate as a separate file, for example: **customCA1.pem**, **customCA2.pem**, and so on.
2. Test each certificate file to confirm that it is a valid CA. For example:

```
openssl x509 -in customCA1.pem -noout -text | grep "CA:"
```

   This outputs either **CA:TRUE** or **CA:FALSE**. You want the output to be **CA:TRUE** for each certificate file. If the output is **CA:FALSE**, the certificate is not a valid CA.

3. Validate each certificate file with the **cURL** command below. For example:

```
curl -v https://<secure-sso-host>/auth/realms/master --cacert customCA1.pem
```

   Replace **<secure-sso-host>** with the fully qualified domain name of the RH-SSO host.

   The expected response is a JSON configuration of the RH-SSO realm. If validation fails your certificate might not be correct.

4. Gather the existing content of the **/etc/pki/tls/cert.pem** file on the **zync-que** pod:

```
oc exec <zync-que-pod-id> -- cat /etc/pki/tls/cert.pem > zync.pem
```

5. Append the content of each custom CA certificate file to **zync.pem**, for example:

```
cat customCA1.pem customCA2.pem ... >> zync.pem
```

6. Attach the new file to the **zync-que** pod as a configmap object:

```
oc create configmap zync-ca-bundle --from-file=./zync.pem
oc set volume dc/zync-que --add --name=zync-ca-bundle --mount-path
/etc/pki/tls/zync/zync.pem --sub-path zync.pem --source='{"configMap":{"name":"zync-ca-
bundle","items":[{"key":"zync.pem","path":"zync.pem"}]}}'
```

   This completes the addition of the certificate bundle to the **zync-que** pod.

7. Verify that the certificate is attached and the content is correct:

```
oc exec <zync-que-pod-id> -- cat /etc/pki/tls/zync/zync.pem
```

8. Configure the **SSL_CERT_FILE** environment variable on Zync to point to the new CA certificate bundle:

```
oc set env dc/zync-que SSL_CERT_FILE=/etc/pki/tls/zync/zync.pem
```

12.4.2. Configuring RH-SSO to have a 3scale client

In your OpenShift RH-SSO dashboard, configure RH-SSO to have a 3scale client. This is a specialized, administrative client. Each time an API consumer subscribes to an API in your Developer Portal, 3scale uses the RH-SSO administrative client that you create in this procedure to create a client for the API consumer application.

Procedure

1. In the RH-SSO console, create a realm for a 3scale client or select an existing realm to contain your 3scale client.
2. In the new or selected realm, click **Clients** in the left navigation panel.
3. Click **Create** to create a new client.
4. In the **Client ID** field, specify a name that helps you identify this client as the 3scale client, for example **oidc-issuer-for-3scale**.
5. Set the **Client Protocol** field to **openid-connect**.
6. Save the new client.
7. In the settings for the new client, set the following:
   - **Access Type** to **confidential**.
   - **Standard Flow Enabled** to **OFF**.
   - **Direct Access Grants Enabled** to **OFF**.
   - **Service Accounts Enabled** to **ON**. This setting enables this client to issue service accounts.
     a. Click **Save**.
8. Set the service account roles for the client:
   a. Navigate to the **Service Account Roles** tab of the client.
   b. In the **Client Roles** drop down list, click **realm-management**.
   c. In the **Available Roles** pane, select **manage-clients** and assign the role by clicking **Add selected >>**.
9. Note the client credentials:
   a. Make a note of the client ID (**<client_id>**).

b. Navigate to the **Credentials** tab of the client and make a note of the **Secret** field (**<client_secret>**).

10. To facilitate testing the authorization flow, add a user to the realm:

a. On the left side of the window, expand **Users**.

b. Click **Add user**.

c. Enter a username, set **Email Verified** to **ON**, and click **Save**.

d. On the **Credentials** tab, set the password. Enter the password in both fields, set the **Temporary** switch to **OFF** to avoid the password reset at the next login, and click **Set password**.

e. When the pop-up window displays, click **Set password**.

12.4.3. Configuring 3scale to work with RH-SSO

Configure 3scale to work with RH-SSO by specifying integration settings in the 3scale Admin Portal.

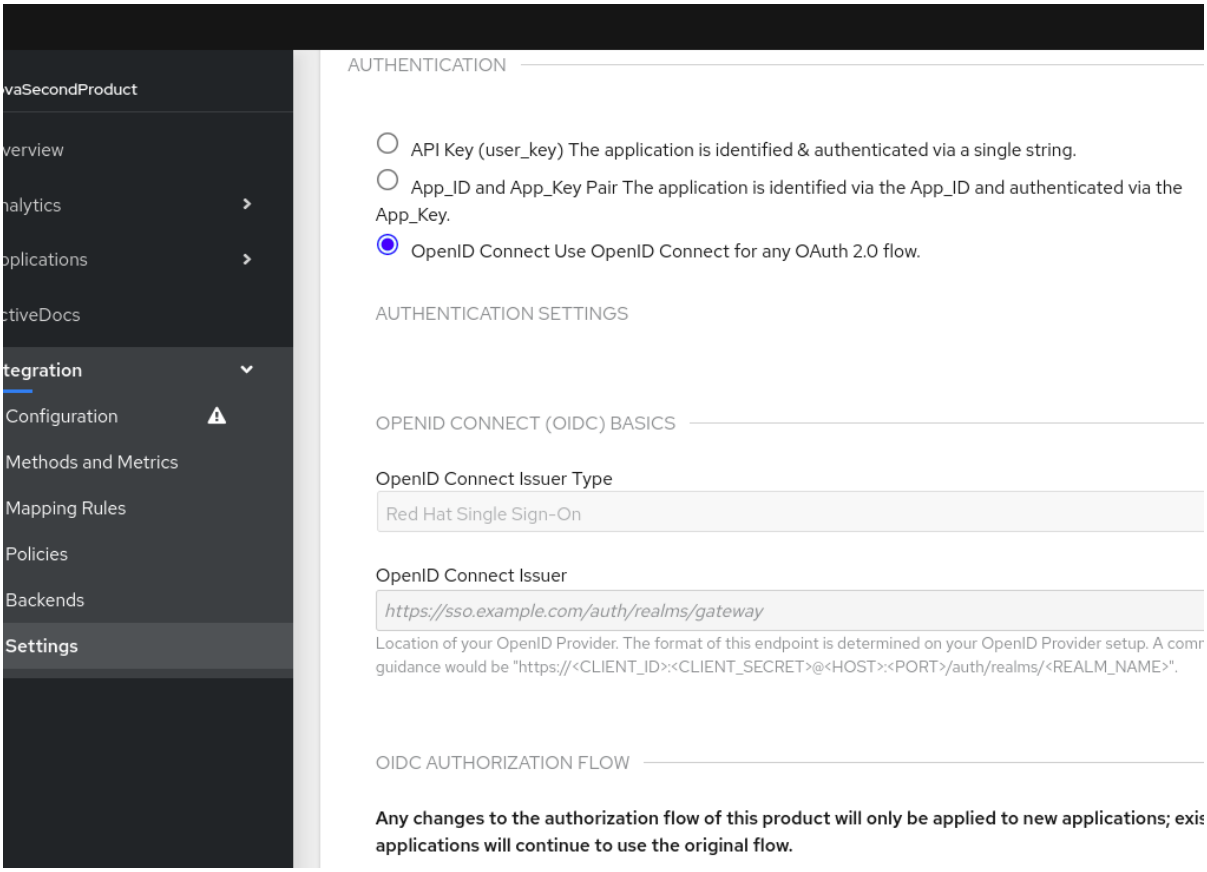
Procedure

1. In the 3scale Admin Portal, in the top level selector, click **Products** and select the 3scale API product for which you are enabling OpenID Connect authentication.

2. Navigate to **[Your_product_name] > Integration > Settings**

3. Under **Authentication** , select **OpenID Connect Use OpenID Connect for any OAuth 2.0 flow**

Figure 12.3. displays the OPENID CONNECT (OIDC) BASICS section.



4. In the **OpenID Connect Issuer Type** field, ensure that the setting is **Red Hat Single Sign-On**.

5. In the **OpenID Connect Issuer** field, enter the URL for the configured OpenID Connect identity provider. The format for this URL looks like this:

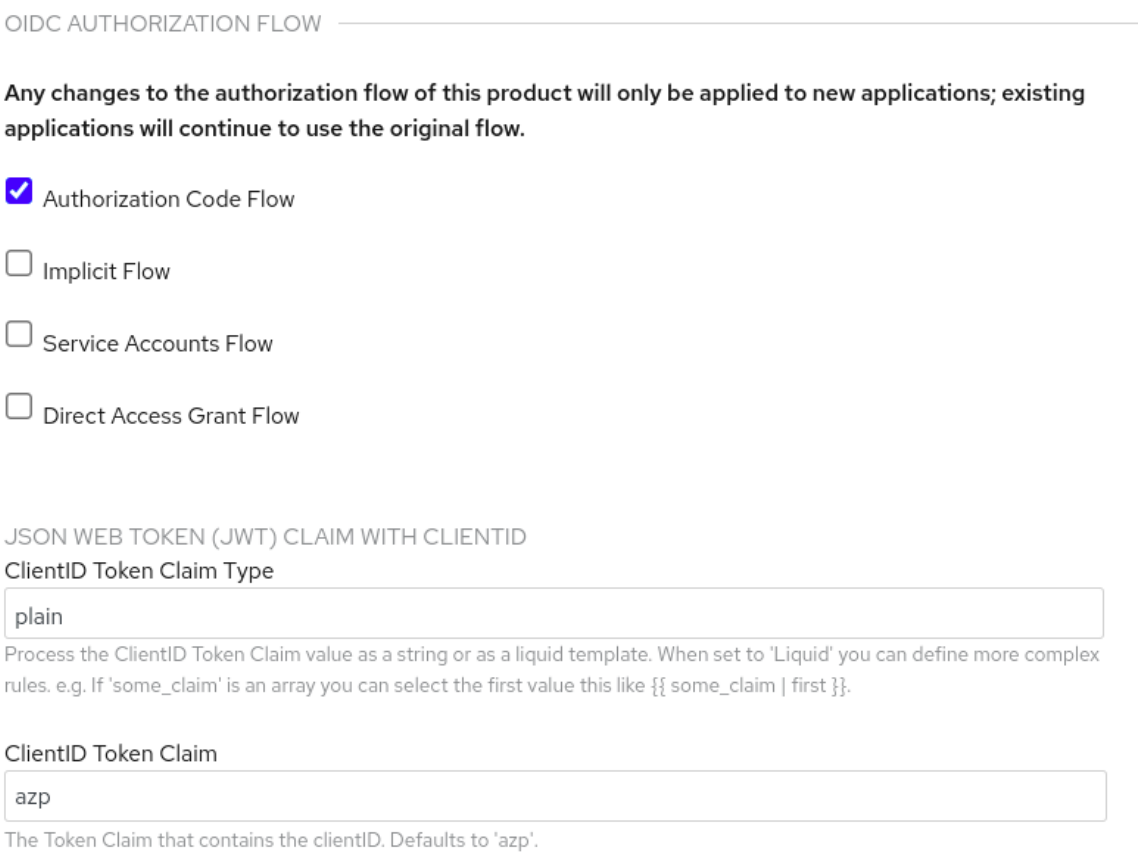
```
https://<client_id>:<client_secret>@<rhsso_host>:<rhsso_port>/auth/realms/<realm_name>
```

6. Replace the placeholders with the previously noted RH-SSO client credentials, the host and port for your RH-SSO server, and the name of the realm that contains the RH-SSO client.

7. Under **OIDC AUTHORIZATION FLOW**, select one or more of the following:

   - **Authorization Code Flow** - In RH-SSO this is the Standard Flow.
   - **Implicit Flow**
   - **Service Accounts Flow** - Also referred to as Client Credentials Flow.
   - **Direct Access Grant Flow** - Also referred to as Resource Owner Password Credentials.

   Figure 12.4. shows where you select the authorization flow:



   This configures how API consumers obtain JSON Web Tokens (JWTs) from the OpenID Connect identity provider. When 3scale integrates RH-SSO as the OpenID Connect identity provider, Zync creates RH-SSO clients that have only the **Authorization Code Flow** enabled. This flow is recommended as the most secure and suitable for most cases. Be sure to select an OAuth 2.0 flow that is supported by your OpenID Connect identity provider.

8. Scroll down and click **Update Product** to save the configuration.

9. In the left navigation panel, click **Integration > Configuration**.

10. Scroll down to click **Promote v.*x* to APIcast Staging**.

Next steps

Test the integration with RH-SSO as the identity provider. When everything is working as it should, return to the **Integration > Configuration** page and scroll down to promote the APIcast staging version to be the production version.

12.5. INTEGRATING 3SCALE WITH THIRD-PARTY OPENID CONNECT IDENTITY PROVIDERS

As an API provider, you can configure an HTTP integration between 3scale and a third-party OpenID Connect identity provider. That is, you can configure an OpenID Connect identity provider other than Red Hat Single Sign-On. 3scale uses this integration to authenticate requests from API consumers and to update the third-party identity provider with the latest 3scale application details.

Most of the work required to integrate 3scale with a third-party OpenID Connect identity provider involves these tasks:

- Meeting the 3scale Zync-related prerequisites.
- Configuring your OpenID Connect identity provider to authorize requests from 3scale applications.

Prerequisites

- 3scale Zync is installed.
- Your chosen third-party OpenID Connect identity provider:
  - Adheres to Zync's OpenAPI specification as provided by 3scale .
  - Allows registration of a client with **<client_id>** and **<client_secret>** declared as a parameter in the request. 3scale is always the source of client identity management in the integration between 3scale and a third-party OpenID Connect identity provider.
  - Is configured for authorizing requests from 3scale applications.
- If your configuration cannot fulfill the previous prerequisite, you must implement a custom adapter that is based on the Zync abstract adapter. Zync uses this adapter to interact with your OpenID Connect identity provider. To create this adapter, you can modify rest_adapter.rb, which is part of the 3scale Zync REST API example .
  You can include the **rest_adapter.rb** module in the **zync-que** pod according to the method that best fits your requirements. For example, you could mount a **configMap** through a volume or you can build a new image for Zync.

Procedure

1. In the 3scale Admin Portal, in the top level selector, click **Products** and select the 3scale API product for which you are enabling OpenID Connect authentication.
2. Navigate to **[Your_product_name] > Integration > Settings**.
3. Under **Authentication**, select **OpenID Connect Use OpenID Connect for any OAuth 2.0 flow**
   This displays the **OPENID CONNECT (OIDC) BASICS** section.

4. In the **OpenID Connect Issuer Type** field, ensure that the setting is **REST API**.
5. In the **OpenID Connect Issuer** field, enter the URL for your OpenID Connect identity provider. The format for this URL looks like this:

   ```
   https://<client_id>:<client_secret>@<oidc_host>:<oidc_port>/<endpoint>
   ```

   For example, in the Zync **rest_adapter.rb** example, the URL endpoint is hard-coded as **{endpoint}/clients**. Your endpoint might be **{endpoint}/register** or something else.
6. Under **OIDC AUTHORIZATION FLOW**, select one or more of the following:
   - **Authorization Code Flow**
   - **Implicit Flow**
   - **Service Accounts Flow**
   - **Direct Access Grant Flow**

   This configures how API consumer applications receive JSON Web Tokens (JWTs) from the OpenID Connect identity provider. The **Authorization Code Flow** is recommended as the most secure and suitable for most cases. Be sure to select an OAuth 2.0 flow that is supported by your OpenID Connect identity provider.
7. Scroll down and click **Update Product** to save the configuration.
8. In the left navigation panel, click **Integration > Configuration**.
9. Scroll down to click **Promote v.*x* to APIcast Staging**.

Next steps

Test the integration with the third-party identity provider. When everything is working as it should, return to the **Integration > Configuration** page and scroll down to promote the APIcast staging version to be the production version.

12.6. TESTING 3SCALE INTEGRATIONS WITH OPENID CONNECT IDENTITY PROVIDERS

After integrating 3scale with an OpenID Connect identity provider, test the integration to confirm that:

- API consumers receive access credentials when they subscribe to a 3scale-managed API.
- APIcast can authenticate requests from API consumers.

Prerequisites

- Integration between 3scale and your OpenID Connect identity provider is in place for a particular 3scale API product. This integration uses the **Authorization Code Flow**.
- An application plan is available for API consumers to subscribe to in your Developer Portal. This application plan provides access to a 3scale-managed API, that is, a 3scale product, for which you configured OpenID Connect authentication.

- An application that sends requests to the upstream API. The upstream API is a backend of the 3scale product that the API consumer application can access as a result of the subscription. Alternatively, you can use Postman to send requests.

Procedure

1. In the Developer Portal, subscribe to an application plan.
   This creates an application in the Developer Portal. The OpenID Connect identity provider should return a client ID and a client secret that you can see in your application's page in the Developer Portal.

2. Note the client ID and the client secret for the application.

3. Verify that your OpenID Connect identity provider now has a client with the same client ID and client secret. For example, when Red Hat Single Sign-On (RH-SSO) is the OpenID Connect identity provider, you will see a new client in the configured RH-SSO realm.

4. In the application page in the Admin Portal, in the **REDIRECT URL** field, enter the URL for the application that sends API requests to the upstream API.

5. Verify that your OpenID Connect identity provider has the correct redirect URL.

6. Discover the URL that receives authentication requests for your OpenID Connect identity provider by using this endpoint:

   ```
   .well-known/openid-configuration
   ```

   For example:

   ```
   https://<rhsso_host>:<rhsso_port>/auth/realms/<realm_name>/.well-known/openid-configuration
   ```

7. For the base URL, use the value that an API provider configured in the OpenID Connect Issuer field.

8. An API consumer who is writing an application that consumes the upstream API, does the following:

   a. Initiates an authorization flow with your OpenConnect identity provider. This request must contain the 3scale application's client ID and client secret. In some cases, the end-user identity is also required.

   b. Receives the identity provider's response, which contains an authorization code.

9. The API consumer's application does the following:

   a. Exchanges the authorization code for a JWT.

   b. Receives a JWT from RH-SSO upon authentication.

   c. Sends an API request that contains the JWT to the upstream API backend.

If APIcast accepts the JWT in the request, your application will receive a response from the API backend.

Alternatively, in place of an API consumer application, use Postman to test that the token flow is correctly implemented.

12.7. EXAMPLE OF A 3SCALE INTEGRATION WITH AN OPENID CONNECT IDENTITY PROVIDER

This example shows the flow when you integrate 3scale with Red Hat Single Sign-On (RH-SSO) as the OpenID Connect identity provider. This example has the following characteristics:

- In the Admin Portal, an API provider defined a 3scale API product and configured that product to use RH-SSO as the OpenID Connect identity provider.
- This product's OpenID Connect configuration includes:
  - Public base URL: **https://api.example.com**
  - Private base URL: **https://internal-api.example.com**
  - OpenID Connect Issuer: **https://zync:41dbb98b-e4e9-4a89-84a3-91d1d19c4207@idp.example.com/auth/realms/myrealm**
  - **Authorization Code Flow**, which is the standard flow, is selected.
- In the 3scale Developer Portal, there is an application with the following characteristics. This application is the result of an API consumer subscribing for access to a 3scale API product provided by a particular application plan in the Developer Portal.
  - **Client ID**: **myclientid**
  - **Client Secret**: **myclientsecret**
  - **Redirect URL**: **https://myapp.example.com**
- In RH-SSO, in the **myrealm** realm, there is a client with these same characteristics:
  - **Client ID**: **myclientid**
  - **Client Secret**: **myclientsecret**
  - **Redirect URL**: **https://myapp.example.com**
- The **myrealm** realm has this user:
  - Username: **myuser**
  - Password: **mypassword**
- 3scale Zync client in **myrealm** has the correct **Service Account** roles

The flow is as follows:

1. An end-user (API consumer) sends an Authorization request to the authentication server, RH-SSO in this example, at the following endpoint:

   ```
   https://idp.example.com/auth/realms/myrealm/protocol/openid-connect/auth
   ```

   In the request, the application provides these parameters:

   - Client ID: **myclientid**

- Redirect URL: **https://myapp.example.com**

2. The application redirects the end-user to the RH-SSO log-in window.
3. The end-user logs in to RH-SSO with these credentials:
   - Username: **myuser**
   - Password: **mypassword**
4. Depending on the configuration, and whether this is the first time that the end-user is authenticating in this specific application, the consent window might display.
5. RH-SSO issues an authorization code to the end-user.
6. The API consumer application uses the following endpoint to send a request to exchange the authorization code for a JWT:

   ```
   https://idp.example.com/auth/realms/myrealm/protocol/openid-connect/token
   ```

   The request contains the authorization code and these parameters:
   - Client ID: **myclientid**
   - Client secret: **myclientsecret**
   - Redirect URL: **https://myapp.example.com**.
7. RH-SSO returns a JSON Web Token (JWT) with an access_token field such as **eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lk…xBArNhqF-A**.
8. The API consumer application sends an API request to **https://api.example.com** with a header such as:

   ```
   Authorization: Bearer eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lk…xBArNhqF-A.
   ```
9. The application should receive a successful response from **https://internal-api.example.com**.