



Red Hat 3scale 2-saas

Deployment Options

For Use with Red Hat 3scale 2-saas

Red Hat 3scale 2-saas Deployment Options

For Use with Red Hat 3scale 2-saas

Legal Notice

Copyright © 2018 Red Hat, Inc.

The text of and illustrations in this document are licensed by Red Hat under a Creative Commons Attribution–Share Alike 3.0 Unported license ("CC-BY-SA"). An explanation of CC-BY-SA is available at

<http://creativecommons.org/licenses/by-sa/3.0/>

. In accordance with CC-BY-SA, if you distribute this document or an adaptation of it, you must provide the URL for the original version.

Red Hat, as the licensor of this document, waives the right to enforce, and agrees not to assert, Section 4d of CC-BY-SA to the fullest extent permitted by applicable law.

Red Hat, Red Hat Enterprise Linux, the Shadowman logo, JBoss, OpenShift, Fedora, the Infinity logo, and RHCE are trademarks of Red Hat, Inc., registered in the United States and other countries.

Linux ® is the registered trademark of Linus Torvalds in the United States and other countries.

Java ® is a registered trademark of Oracle and/or its affiliates.

XFS ® is a trademark of Silicon Graphics International Corp. or its subsidiaries in the United States and/or other countries.

MySQL ® is a registered trademark of MySQL AB in the United States, the European Union and other countries.

Node.js ® is an official trademark of Joyent. Red Hat Software Collections is not formally related to or endorsed by the official Joyent Node.js open source or commercial project.

The OpenStack ® Word Mark and OpenStack logo are either registered trademarks/service marks or trademarks/service marks of the OpenStack Foundation, in the United States and other countries and are used with the OpenStack Foundation's permission. We are not affiliated with, endorsed or sponsored by the OpenStack Foundation, or the OpenStack community.

All other trademarks are the property of their respective owners.

Abstract

This guide documents deployment options for Red Hat 3scale 2-saas.

Table of Contents

CHAPTER 1. APICAST OVERVIEW	5
1.1. PREREQUISITES	5
1.2. DEPLOYMENT OPTIONS	5
1.3. ENVIRONMENTS	5
1.4. CONFIGURE THE INTEGRATION SETTINGS	6
1.5. CONFIGURE YOUR SERVICE	6
1.6. MAPPING RULES	7
1.7. MAPPING RULES WORKFLOW	8
1.8. HOST HEADER	9
1.9. PRODUCTION DEPLOYMENT	9
1.10. PUBLIC BASE URL	9
1.11. PROTECTING YOUR API BACKEND	10
1.12. USING APICAST WITH PRIVATE APIS	10
CHAPTER 2. APICAST HOSTED	11
2.1. PREREQUISITES	11
2.2. STEP 1: DEPLOY YOUR API WITH APICAST HOSTED IN A STAGING ENVIRONMENT	11
2.3. STEP 2: DEPLOY YOUR API WITH THE APICAST HOSTED INTO PRODUCTION	12
2.3.1. Bear in mind	12
CHAPTER 3. APICAST ON THE DOCKER CONTAINERIZED ENVIRONMENT	13
3.1. PREREQUISITES	13
3.2. STEP 1: INSTALL THE DOCKER CONTAINERIZED ENVIRONMENT	13
3.3. STEP 2: RUN THE DOCKER CONTAINERIZED ENVIRONMENT GATEWAY	13
3.3.1. The Docker command options	14
3.4. STEP 3: TESTING APICAST	14
3.5. STEP 4: TROUBLESHOOTING APICAST ON THE DOCKER CONTAINERIZED ENVIRONMENT	14
3.5.1. Cannot connect to the Docker daemon error	14
3.5.2. Basic Docker command-line interface commands	15
CHAPTER 4. RUNNING APICAST ON RED HAT OPENSIFT	16
4.1. PREREQUISITES	16
4.2. STEP 1: SET UP OPENSIFT	16
4.2.1. Install the Docker containerized environment	16
4.2.2. Start OpenShift cluster	17
4.2.2.1. Setting up OpenShift cluster on a remote server	17
4.3. STEP 2: DEPLOY APICAST USING THE OPENSIFT TEMPLATE	18
4.4. STEP 3: CREATE ROUTES IN OPENSIFT CONSOLE	19
CHAPTER 5. ADVANCED APICAST CONFIGURATION	22
5.1. DEFINE A SECRET TOKEN	22
5.2. CREDENTIALS	22
5.3. ERROR MESSAGES	23
5.4. CONFIGURATION HISTORY	24
5.5. DEBUGGING	25
5.6. PATH ROUTING	25
CHAPTER 6. APICAST POLICIES	27
6.1. APICAST STANDARD POLICIES	27
6.1.1. Anonymous Access Policy	27
6.1.2. APICast CORS Request Handling Policy	28
6.1.3. Authentication Caching Policy	30
6.1.4. Echo Policy	31

6.1.5. Header Modification Policy	31
6.1.6. Liquid Context Debug Policy	33
6.1.7. Referrer Policy	33
6.1.8. RH-SSO/Keycloak Role Check Policy	33
6.1.9. SOAP Policy	35
6.1.10. Upstream Policy	36
6.1.11. URL Rewriting Policy	37
6.1.11.1. Commands for rewriting the path	37
6.1.11.2. Commands for rewriting the query string	38
6.1.12. URL Rewriting with Captures Policy	39
6.2. ENABLING A STANDARD POLICY	40
6.3. CREATING A POLICY CHAIN IN THE AMP	42
6.4. USING VARIABLES AND FILTERS IN POLICIES	43
CHAPTER 7. APICAST ENVIRONMENT VARIABLES	45
APICAST_BACKEND_CACHE_HANDLER	45
APICAST_CONFIGURATION_CACHE	45
APICAST_CONFIGURATION_LOADER	45
APICAST_CUSTOM_CONFIG	45
APICAST_ENVIRONMENT	45
APICAST_LOG_FILE	46
APICAST_LOG_LEVEL	46
APICAST_ACCESS_LOG_FILE	46
APICAST_OIDC_LOG_LEVEL	46
APICAST_MANAGEMENT_API	46
APICAST_MODULE	46
APICAST_OAUTH_TOKENS_TTL	46
APICAST_PATH_ROUTING	47
APICAST_POLICY_LOAD_PATH	47
APICAST_PROXY_HTTPS_CERTIFICATE_KEY	47
APICAST_PROXY_HTTPS_CERTIFICATE	47
APICAST_PROXY_HTTPS_PASSWORD_FILE	47
APICAST_PROXY_HTTPS_SESSION_REUSE	47
APICAST_REPORTING_THREADS	48
APICAST_RESPONSE_CODES	48
APICAST_SERVICES_LIST	48
APICAST_SERVICE_\${ID}_CONFIGURATION_VERSION	48
APICAST_WORKERS	48
BACKEND_ENDPOINT_OVERRIDE	48
OPENSSL_VERIFY	48
REDIS_HOST	49
REDIS_PORT	49
REDIS_URL	49
RESOLVER	49
THREESCALE_CONFIG_FILE	49
THREESCALE_DEPLOYMENT_ENV	49
THREESCALE_PORTAL_ENDPOINT	50
OPENTRACING_TRACER	50
OPENTRACING_CONFIG	50
OPENTRACING_HEADER_FORWARD	50
APICAST_HTTPS_PORT	50
APICAST_HTTPS_CERTIFICATE	50
APICAST_HTTPS_CERTIFICATE_KEY	51

all_proxy, ALL_PROXY	51
http_proxy, HTTP_PROXY	51
https_proxy, HTTPS_PROXY	51
no_proxy, NO_PROXY	51

CHAPTER 1. APICAST OVERVIEW

APIcast is an NGINX based API gateway used to integrate your internal and external API services with 3scale API Management Platform.

See the articles [Red Hat 3scale API Management Supported Configurations](#) and [Red Hat 3scale API Management - Component Details](#) to get information about the latest released and supported version of APIcast. For the updates on APIcast Hosted version please refer to [Red Hat 3scale API Management Platform SaaS Release Notes](#).

In this guide you'll learn more about deployment options, environments provided, and how to get started.

1.1. PREREQUISITES

APIcast is not a standalone API gateway, it needs connection to 3scale API Manager. In case you don't yet have a 3scale account please follow these steps:

- [Sign up](#) for a new account at [3scale.net](#)
- Activate your 3scale account
- Log in to your 3scale Admin Portal

The Admin Portal URL should look like <https://<DOMAIN>-admin.3scale.net>, where **<DOMAIN>** is the domain you specified on sign up.

1.2. DEPLOYMENT OPTIONS

You can use APIcast hosted or self-managed, in both cases, it needs connection to the rest of the 3scale API management platform:

- **APIcast hosted:** 3scale hosts APIcast in the cloud. In this case, APIcast is already deployed for you and it's limited to 50,000 calls per day.
- **APIcast self-managed:** You can deploy APIcast wherever you want. The self-managed mode is the intended mode of operation for production environments. Here are a few recommended options to deploy APIcast:

```
// DISCONTINUED - xref:apicast-self-managed[Native deployment]:
Install OpenResty and other dependencies on your own server and run
APIcast using the code and configuration provided by 3scale.
```

- [the Docker containerized environment](#): Download a ready to use Docker-formatted container image, which includes all of the dependencies to run APIcast in a Docker-formatted container.
- [OpenShift](#): Run APIcast on a [supported version](#) of OpenShift. You can connect self-managed APIcasts both to a 3scale AMP installation or to a 3scale online account.

1.3. ENVIRONMENTS

By default, when you create a 3scale account or create a new API service, you get an APIcast **hosted** in two different environments:

- **Staging:** Intended to be used only while configuring and testing your API integration. When you have confirmed that your setup is working as expected, then you can choose to deploy it to the production environment.
- **Production:** Limited to 50,000 calls per day and supports the following out-of-the-box authentication options: API key, and App ID and App key pair, OpenID Connect.

When you use Self-managed deployment, you still have the same two environments, and you need to deploy an APIcast instance for each. You can specify which configuration (Staging or Production) the APIcast instance will use by setting the environment variable **THREESCALE_DEPLOYMENT_ENV**, which can take values **staging** or **production**.

1.4. CONFIGURE THE INTEGRATION SETTINGS

Go to **[your_API_name] > Integration > Configuration**.

On top of the Integration page you will see your integration options. By default, the deployment option is APIcast hosted, and the authentication mode is API key. You can change these settings by clicking on **edit integration settings** in the top right corner. Note that OAuth 2.0 authentication is only available for the Self-managed deployment.

1.5. CONFIGURE YOUR SERVICE

You will need to declare your API backend in the Private Base URL field, which is the endpoint host of your API backend. APIcast will redirect all traffic to your API backend after all authentication, authorization, rate limits and statistics have been processed.

Typically, the Private Base URL of your API will be something like <https://api-backend.yourdomain.com:443>, on the domain that you manage (**yourdomain.com**). For instance, if you were integrating with the Twitter API the Private Base URL would be <https://api.twitter.com/>. In this example will use the Echo API hosted by 3scale – a simple API that accepts any path and returns information about the request (path, request parameters, headers, etc.). Its Private Base URL is <https://echo-api.3scale.net:443>. Private Base URL

Staging: configure & test your integration [documentation](#)

[deployed](#) | [deployment history](#)



API



Private Base URL*

Private address of your API that will be called by the API gateway.

Test your private (unmanaged) API is working. For example, for the Echo API we can make the following call with **curl** command:

```
curl "https://echo-api.3scale.net:443"
```

We'll get the following response:

```
{
  "method": "GET",
  "path": "/",
  "args": "",
  "body": "",
```

```

"headers": {
  "HTTP_VERSION": "HTTP/1.1",
  "HTTP_HOST": "echo-api.3scale.net",
  "HTTP_ACCEPT": "*/*",
  "HTTP_USER_AGENT": "curl/7.51.0",
  "HTTP_X_FORWARDED_FOR": "2.139.235.79, 10.0.103.58",
  "HTTP_X_FORWARDED_HOST": "echo-api.3scale.net",
  "HTTP_X_FORWARDED_PORT": "443",
  "HTTP_X_FORWARDED_PROTO": "https",
  "HTTP_FORWARDED": "for=10.0.103.58;host=echo-api.3scale.net;proto=https"
},
"uuid": "ee626b70-e928-4cb1-a1a4-348b8e361733"
}

```

Once you've confirmed that your API is working, you will need to configure the test call for the hosted staging environment. Enter a path existing in your API in the *API test GET request field* (for example, `/v1/word/good.json`).

Save the settings by clicking on the **Update & Test Staging Configuration** button in the bottom right part of the page. This will deploy the APIcast configuration to the 3scale hosted staging environment. If everything is configured correctly, the vertical line on the left should turn green.

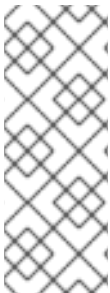
If you are using one of the Self-managed deployment options, save the configuration from the GUI and make sure it is pointing to your deployed API gateway by adding the correct host in the staging or production Public base URL field. Before making any calls to your production gateway, don't forget to click on the **Promote v.x to Production** button.

Find the sample **curl** at the bottom of the staging section and run it from the console:

```

curl "https://XXX.staging.apicast.io:443/v1/word/good.json?
user_key=YOUR_USER_KEY"

```



NOTE

You should get the same response as above, however, this time the request will go through the 3scale hosted APIcast instance. Note: You should make sure you have an application with valid credentials for the service. If you are using the default API service created on sign up to 3scale, you should already have an application. Otherwise, if you see **USER_KEY** or **APP_ID** and **APP_KEY** values in the test curl, you need to create an application for this service first.

And that's it! You have your API integrated with 3scale.

3scale hosted APIcast gateway does the validation of the credentials and applies the rate limits that you defined for the application plan of the application. If you try to make a call without credentials, or with invalid credentials, you will see an error message. The code and the text of the message can be configured, check out the [Advanced APIcast configuration](#) article for more information.

1.6. MAPPING RULES

By default we start with a very simple mapping rule,

▼ MAPPING RULES



Verb	Pattern		Metric or Method (Define)
GET	/	1	hits

[Add Mapping Rule](#)

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

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

▼ MAPPING RULES



Verb	Pattern		Metric or Method (Define)
GET	/hello	1	gethello
GET	/goodbye	1	getgoodbye

[Add Mapping Rule](#)

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

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

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

Wildcards can appear between slashes or between slash and dot.

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

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

Parameters can also have named wildcards.

Note that all mapping rules are evaluated. There is no precedence (order does not matter). If we added a rule **/v1** to the example on the figure above, it would always be matched for the requests whose path starts with **/v1** regardless if it is **/v1/word** or **/v1/sentence**. Keep in mind that if two different rules increment the same metric by one, and the two rules are matched, the metric will be incremented by two.

1.7. MAPPING RULES WORKFLOW

The intended workflow to define mapping rules is as follows:

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

For more advanced configuration options, you can check the [APIcast advanced configuration](#) tutorial.

1.8. HOST HEADER

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

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

▼ AUTHENTICATION SETTINGS

Host Header

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

1.9. PRODUCTION DEPLOYMENT

Once you have configured your API integration and verified it is working in the Staging environment, you can go ahead with one of the APIcast production deployments. See the [Deployment options](#) in the beginning of this article.

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

1.10. PUBLIC BASE URL

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

In [APIcast hosted](#), the Public Base URL is set by 3scale and can't be changed.

If you are using one of the Self-managed deployment options, you can choose your own Public Base URL for each one of the environments provided (staging and production), on a domain name you are managing. Note that this URL should be different from the one of your API backend, and could be

something like <https://api.yourdomain.com:443>, where **yourdomain.com** is the domain that belongs to you. After setting the Public Base URL make sure you save the changes and, if necessary, promote the changes in staging to production.

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

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

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

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

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

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

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

1.11. PROTECTING YOUR API BACKEND

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

1.12. USING APICAST WITH PRIVATE APIS

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

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

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

CHAPTER 2. APICAST HOSTED

Once you complete this tutorial, you'll have your API fully protected by a secure gateway in the cloud.

APIcast hosted is the best deployment option if you want to launch your API as fast as possible, or if you want to make the minimum infrastructure changes on your side.


2.1. PREREQUISITES

- You have reviewed the [deployment alternatives](#) and decided to use APIcast hosted to integrate your API with 3scale.
- Your API backend service is accessible over the public Internet (a secure communication will be established to prevent users from bypassing the access control gateway).
- You do not expect demand for your API to exceed the limit of 50,000 hits/day (beyond this, we recommend upgrading to the self-managed gateway).

2.2. STEP 1: DEPLOY YOUR API WITH APICAST HOSTED IN A STAGING ENVIRONMENT


The first step is to configure your API and test it in your staging environment. Define the private base URL and its endpoints, choose the placement of credentials and other configuration details that you can read about [here](#). Once you're done entering your configuration, go ahead and click on Update & Test Staging Environment button to run a test call that will go through the APIcast staging instance to your API.

Configuration: [configure & test immediately in the staging environment](#) [documentation](#)


API

Private Base URL*

Private address of your API that will be called by the API gateway.


API GATEWAY

Public Base URL*


Public address of your API gateway in the staging environment.

Production Public Base URL*

Public address of your API gateway in the production environment.

MAPPING RULES

AUTHENTICATION SETTINGS


CLIENT

API test GET request

Optional GET request to a API gateway endpoint. We will use this call to validate your API gateway setup using credentials of the first live application. You can try it yourself by copying the following command into your shell:

```
curl "https://api-2445581460490.staging.apicast.io:443/?user_key=063a01e356790b831f749b0b8b726e38"
```

Hit the test button to check the connections between client, gateway & API.

Update & Test in Staging Environment

[← Back to Integration & Configuration](#)

If everything was configured correctly, you should see a green confirmation message.

Before moving on to the next step, make sure that you have configured a secret token to be validated by your backend service. You can define the value for the secret token under **Authentication Settings**. This will ensure that nobody can bypass APIcast's access control.

2.3. STEP 2: DEPLOY YOUR API WITH THE APICAST HOSTED INTO PRODUCTION

At this point, you're ready to take your API configuration to a production environment. To deploy your 3scale-hosted APIcast instance, go back to the 'Integration and Configuration' page and click on the **'Promote to v.x to Production'** button. Repeat this step to promote further changes in your staging environment to your production environment.

[edit APIcast configuration](#)

APIcast Configuration

Private Base URL: `https://echo-api.3scale.net:443`

Mapping rules: `/ => hits`

Credential Location: `query`

Secret Token: `Shared_secret_sent_from_proxy_to_API_backend`

Environments

Staging Environment
<https://api-244581460490.staging.apicast.io:443>

Promote v. 1 to Production

[Configuration history](#)

Production Environment
no configuration has been saved for the production environment yet

It will take between 5 and 7 minutes for your configuration to deploy and propagate to all the cloud APIcast instances. During redeployment, your API will not experience any downtime. However, API calls may return different responses depending on which instance serves the call. You'll know it has been deployed once the box around your production environment has turned green.

Both the staging and production APIcast instances have base URLs on the apicast.io domain. You can easily tell them apart because the staging environment URLs have a staging subdomain. For example:

- staging: <https://api-2445581448324.staging.apicast.io:443>
- production: <https://api-2445581448324.apicast.io:443>

2.3.1. Bear in mind

- 50,000 hits/day is the maximum allowed for your API through the APIcast production cloud instance. You can check your API usage in the Analytics section of your Admin Portal.
- There is a hard throttle limit of 20 hits/second on any spike in API traffic.
- Above the throttle limit, APIcast returns a response code of **403**. This is the same as the default for an application over rate limits. If you want to differentiate the errors, please check the response body.

CHAPTER 3. APICAST ON THE DOCKER CONTAINERIZED ENVIRONMENT

This is a step-by-step guide to deploy APIcast inside a Docker-formatted container that is ready to be used as a 3scale API gateway.

3.1. PREREQUISITES

You must configure APIcast in your 3scale Admin Portal as per the [APIcast Overview](#).

3.2. STEP 1: INSTALL THE DOCKER CONTAINERIZED ENVIRONMENT

This guide covers the steps to set up the Docker containerized environment on Red Hat Enterprise Linux (RHEL) 7.

Docker-formatted containers provided by Red Hat are released as part of the Extras channel in RHEL. To enable additional repositories, you can use either the [Subscription Manager](#) or the yum config manager. For details, see the [RHEL product documentation](#).

To deploy RHEL 7 on an AWS EC2 instance, take the following steps:

1. List all repositories: **sudo yum repolist all**.
2. Find the ***-extras** repository.
3. Enable the **extras** repository: **sudo yum-config-manager --enable rhui-REGION-rhel-server-extras**.
4. Install the Docker containerized environment package: **sudo yum install docker**.

For other operating systems, refer to the following Docker documentation:

- [Installing the Docker containerized environment on Linux distributions](#)
- [Installing the Docker containerized environment on Mac](#)
- [Installing the Docker containerized environment on Windows](#)

3.3. STEP 2: RUN THE DOCKER CONTAINERIZED ENVIRONMENT GATEWAY

1. Start the Docker daemon: **sudo systemctl start docker.service**.
2. Check if the Docker daemon is running: **sudo systemctl status docker.service**. You can download a ready to use Docker-formatted container image from the Red Hat registry: **sudo docker pull registry.access.redhat.com/3scale-amp23/apicast-gateway**.
3. Run APIcast in a Docker-formatted container: **sudo docker run --name apicast --rm -p 8080:8080 -e THREESCALE_PORTAL_ENDPOINT=https://<access_token>@<domain>-admin.3scale.net registry.access.redhat.com/3scale-amp23/apicast-gateway**.

Here, `<access_token>` is the [Access Token](#) for the 3scale Account Management API. You can use the [Provider Key](#) instead of the access token. `<domain>-admin.3scale.net` is the URL of your 3scale admin portal.

This command runs a Docker-formatted container called `"apicast"` on port **8080** and fetches the JSON configuration file from your 3scale portal. For other configuration options, see the [APIcast Overview](#) guide.

3.3.1. The Docker command options

You can use the following options with the **docker run** command:

- **--rm**: Automatically removes the container when it exits.
- **-d** or **--detach**: Runs the container in the background and prints the container ID. When it is not specified, the container runs in the foreground mode and you can stop it using **CTRL + C**. When started in the detached mode, you can reattach to the container with the **docker attach** command, for example, **docker attach apicast**.
- **-p** or **--publish**: Publishes a container's port to the host. The value should have the format `<host port>:<container port>`, so **-p 80:8080** will bind port **8080** of the container to port **80** of the host machine. For example, the [Management API](#) uses port **8090**, so you may want to publish this port by adding **-p 8090:8090** to the **docker run** command.
- **-e** or **--env**: Sets environment variables.
- **-v** or **--volume**: Mounts a volume. The value is typically represented as `<host path>:<container path>[:<options>]`. `<options>` is an optional attribute; you can set it to **ro** to specify that the volume will be read only (by default, it is mounted in read-write mode). Example: **-v /host/path:/container/path:ro**.

For more information on available options, see [Docker run reference](#).

3.4. STEP 3: TESTING APICAST

The preceding steps ensure that your Docker-formatted container is running with your own configuration file and the Docker-formatted image from the 3scale registry. You can test calls through APIcast on port **8080** and provide the correct authentication credentials, which you can get from your 3scale account.

Test calls will not only verify that APIcast is running correctly but also that authentication and reporting is being handled successfully.



NOTE

Ensure that the host you use for the calls is the same as the one configured in the **Public Base URL** field on the **Integration** page.

3.5. STEP 4: TROUBLESHOOTING APICAST ON THE DOCKER CONTAINERIZED ENVIRONMENT

3.5.1. 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](#)).

3.5.2. 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 4. RUNNING APICAST ON RED HAT OPENSIFT

This tutorial describes how to deploy the APIcast API Gateway on Red Hat OpenShift.

4.1. PREREQUISITES

To follow the tutorial steps below, you will first need to configure APIcast in your 3scale Admin Portal as per the [APIcast Overview](#). Make sure *Self-managed Gateway* is selected as the deployment option in the integration settings. You should have both Staging and Production environment configured to proceed.

4.2. STEP 1: SET UP OPENSIFT

If you already have a running OpenShift cluster, you can skip this step. Otherwise, continue reading.

For production deployments you can follow the [instructions for OpenShift installation](#). In order to get started quickly in development environments, there are a couple of ways you can install OpenShift:

- Using **oc cluster up** command – https://github.com/openshift/origin/blob/master/docs/cluster_up_down.md (used in this tutorial, with detailed instructions for Mac and Windows in addition to Linux which we cover here)
- All-In-One Virtual Machine using Vagrant – <https://www.openshift.org/vm>

In this tutorial the OpenShift cluster will be installed using:

- Red Hat Enterprise Linux (RHEL) 7
- Docker containerized environment v1.10.3
- OpenShift Origin command line interface (CLI) - v1.3.1

4.2.1. Install the Docker containerized environment

Docker-formatted container images provided by Red Hat are released as part of the Extras channel in RHEL. To enable additional repositories, you can use either the [Subscription Manager](#), or yum config manager. See the [RHEL product documentation](#) for details.

For a RHEL 7 deployed on a AWS EC2 instance we'll use the following the instructions:

1. List all repositories:

```
sudo yum repolist all
```

Find the ***-extras** repository.

1. Enable *extras* repository:

```
sudo yum-config-manager --enable rhui-REGION-rhel-server-extras
```

2. Install Docker-formatted container images:

```
sudo yum install docker docker-registry
```

3. Add an insecure registry of **172.30.0.0/16** by adding or uncommenting the following line in **/etc/sysconfig/docker** file:

```
INSECURE_REGISTRY='--insecure-registry 172.30.0.0/16'
```

4. Start the Docker containerized environment:

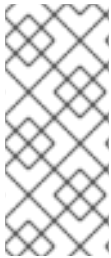
```
sudo systemctl start docker
```

You can verify that the Docker containerized environment is running with the command:

```
sudo systemctl status docker
```

4.2.2. Start OpenShift cluster

Download the latest stable release of the client tools (**openshift-origin-client-tools-
VERSION-linux-64bit.tar.gz**) from [OpenShift releases page](#), and place the Linux **oc** binary extracted from the archive in your **PATH**.



NOTE

- Please be aware that the **oc cluster** set of commands are only available in the 1.3+ or newer releases.
- the docker command runs as the **root** user, so you will need to run any **oc** or docker commands with root privileges.

Open a terminal with a user that has permission to run docker commands and run:

```
oc cluster up
```

At the bottom of the output you will find information about the deployed cluster:

```
-- Server Information ...
OpenShift server started.
The server is accessible via web console at:
https://172.30.0.112:8443

You are logged in as:
  User:      developer
  Password:  developer

To login as administrator:
  oc login -u system:admin
```

Note the IP address that is assigned to your OpenShift server, we will refer to it in the tutorial as **OPENSIFT-SERVER-IP**.

4.2.2.1. Setting up OpenShift cluster on a remote server

In case you are deploying the OpenShift cluster on a remote server, you will need to explicitly specify a public hostname and a routing suffix on starting the cluster, in order to be able to access the OpenShift web console remotely.

For example, if you are deploying on an AWS EC2 instance, you should specify the following options:

```
oc cluster up --public-hostname=ec2-54-321-67-89.compute-1.amazonaws.com -  
-routing-suffix=54.321.67.89.xip.io
```

where **ec2-54-321-67-89.compute-1.amazonaws.com** is the Public Domain, and **54.321.67.89** is the IP of the instance. You will then be able to access the OpenShift web console at <https://ec2-54-321-67-89.compute-1.amazonaws.com:8443>.

4.3. STEP 2: DEPLOY APICAST USING THE OPENSIFT TEMPLATE

1. By default you are logged in as *developer* and can proceed to the next step.
Otherwise login into OpenShift using the **oc login** command from the OpenShift Client tools you downloaded and installed in the previous step. The default login credentials are *username = "developer"* and *password = "developer"*.

```
oc login https://OPENSIFT-SERVER-IP:8443
```

You should see **Login successful.** in the output.

2. Create your project. This example sets the display name as *gateway*

```
oc new-project "3scalegateway" --display-name="gateway" --  
description="3scale gateway demo"
```

The response should look like this:

```
Now using project "3scalegateway" on server  
"https://172.30.0.112:8443".
```

Ignore the suggested next steps in the text output at the command prompt and proceed to the next step below.

3. Create a new Secret to reference your project by replacing **<access_token>** and **<domain>** with yours.

```
oc secret new-basicauth apicast-configuration-url-secret --  
password=https://<access_token>@<domain>-admin.3scale.net
```

Here **<access_token>** is an [Access Token](#) (not a Service Token) for the 3scale Account Management API, and **<domain>-admin.3scale.net** is the URL of your 3scale Admin Portal.

The response should look like this:

```
secret/apicast-configuration-url-secret
```

4. Create an application for your APIcast Gateway from the template, and start the deployment:

```
oc new-app -f https://raw.githubusercontent.com/3scale/3scale-amp-openshift-templates/2.3.0.GA/apicast-gateway/apicast.yml
```

You should see the following messages at the bottom of the output:

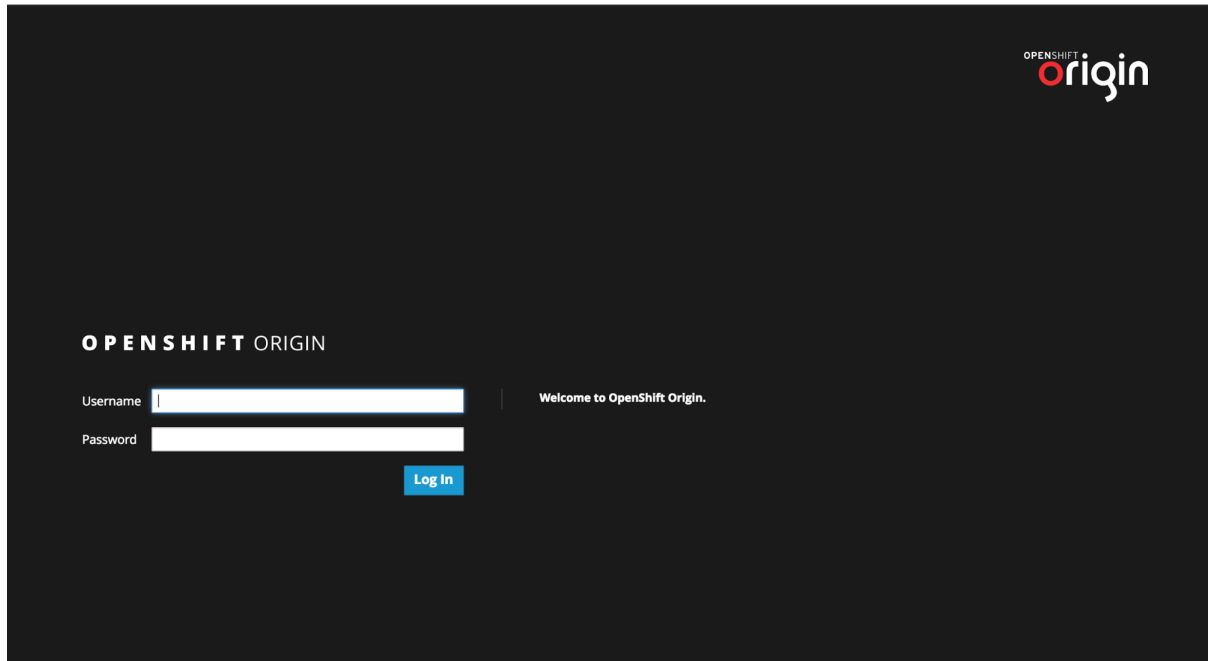
```
--> Creating resources with label app=3scale-gateway ...
      deploymentconfig "apicast" created
      service "apicast" created
--> Success
      Run 'oc status' to view your app.
```

4.4. STEP 3: CREATE ROUTES IN OPENSIFT CONSOLE

1. Open the web console for your OpenShift cluster in your browser: <https://OPENSIFT-SERVER-IP:8443/console/>

Use the value specified in `--public-hostname` instead of **OPENSIFT-SERVER-IP** if you started OpenShift cluster on a remote server.

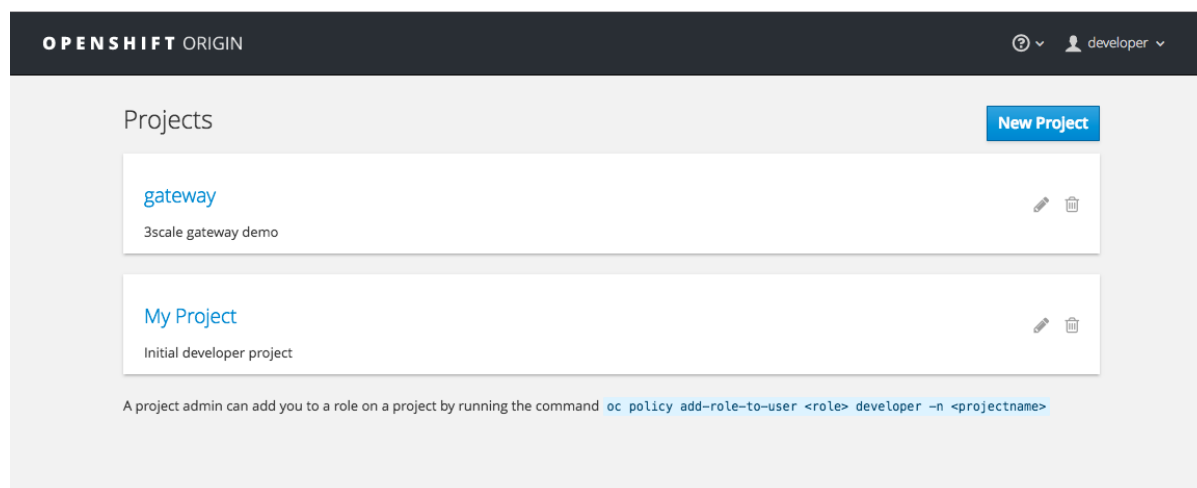
You should see the login screen:



NOTE

You may receive a warning about an untrusted web-site. This is expected, as we are trying to access the web console through secure protocol, without having configured a valid certificate. While you should avoid this in production environment, for this test setup you can go ahead and create an exception for this address.

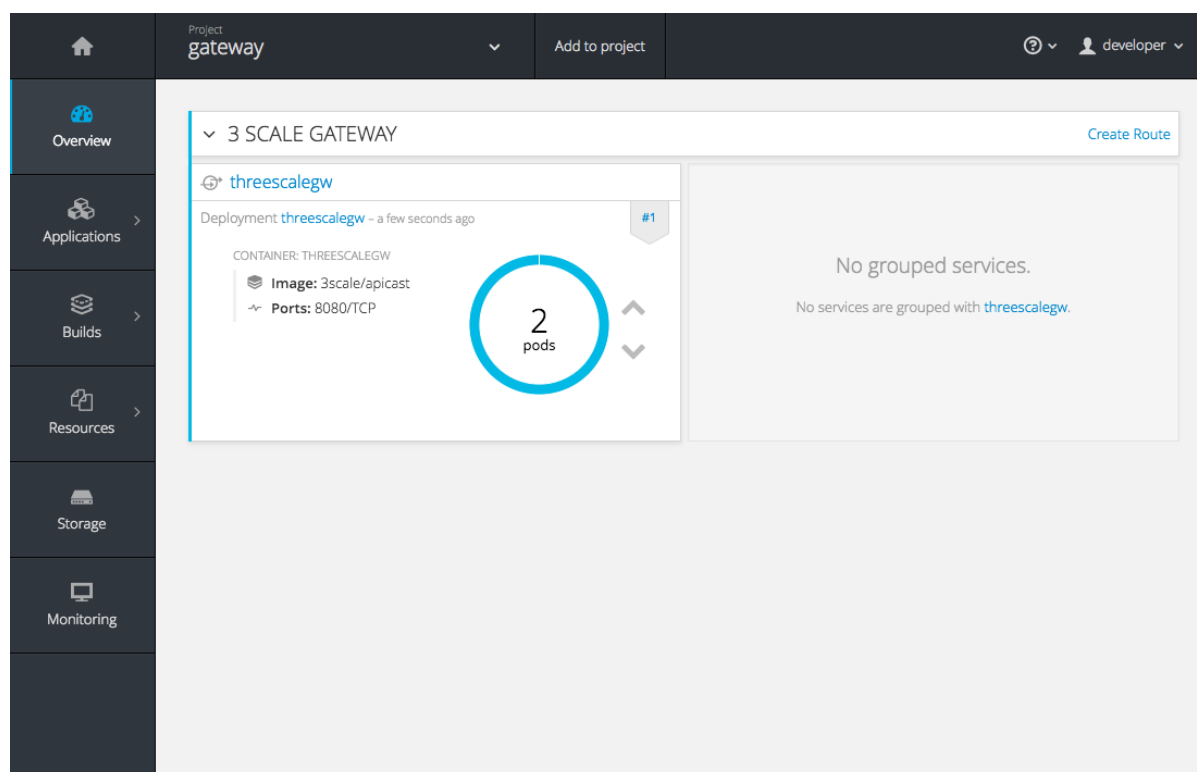
2. Log in using the *developer* credentials created or obtained in the *Setup OpenShift* section above.
You will see a list of projects, including the *"gateway"* project you created from the command line above.



If you do not see your gateway project, you probably created it with a different user and will need to assign the policy role to this user.

- Click on "gateway" and you will see the *Overview* tab. OpenShift downloaded the code for APIcast and started the deployment. You may see the message *Deployment #1 running* when the deployment is in progress.

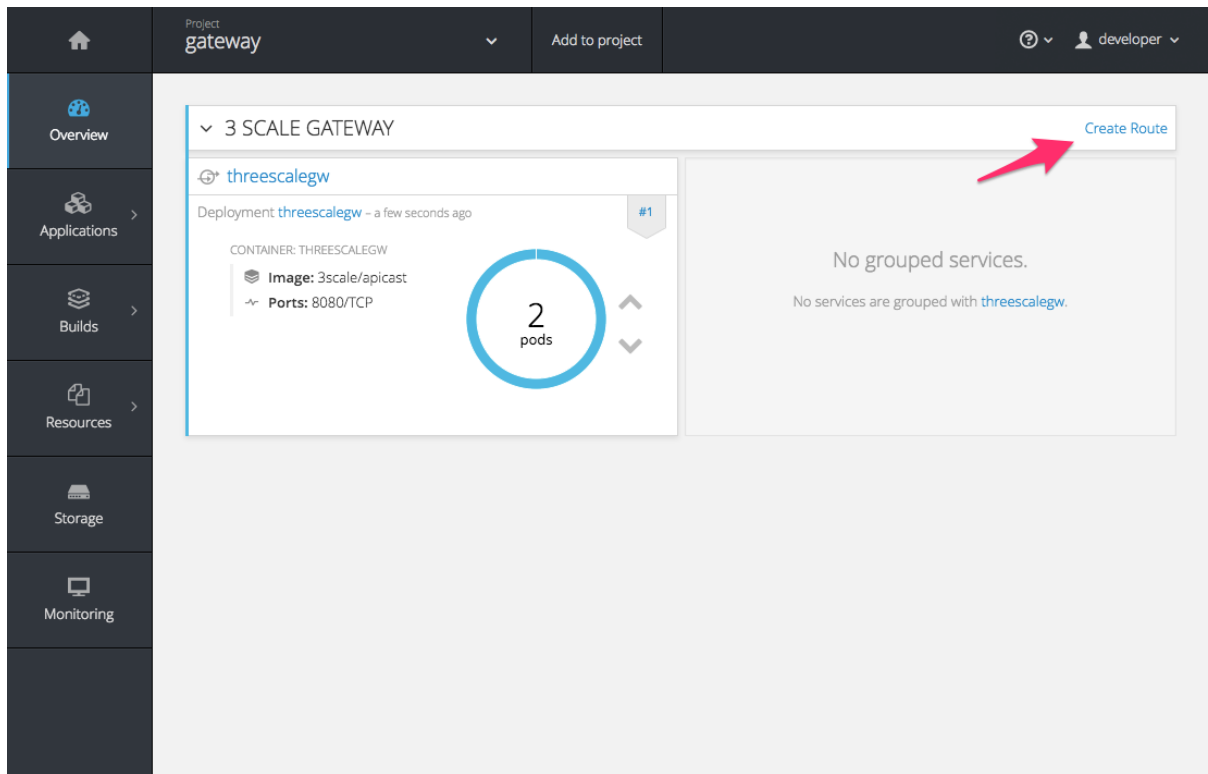
When the build completes, the UI will refresh and show two instances of APIcast (*2 pods*) that have been started by OpenShift, as defined in the template.



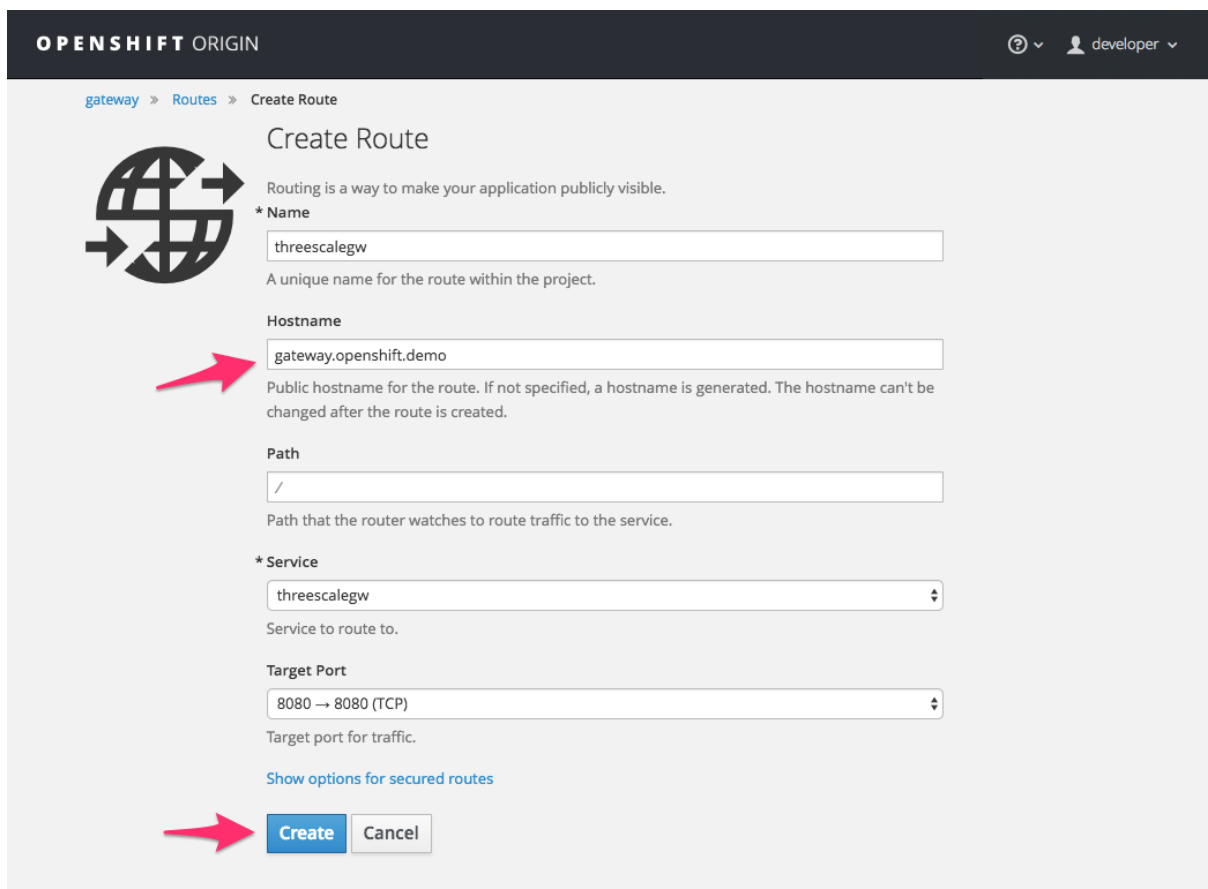
Each APIcast instance, upon starting, downloads the required configuration from 3scale using the settings you provided on the **Integration** page of your 3scale Admin Portal.

OpenShift will maintain two APIcast instances and monitor the health of both; any unhealthy APIcast instance will automatically be replaced with a new one.

- In order to allow your APIcast instances to receive traffic, you'll need to create a route. Start by clicking on **Create Route**.



Enter the same host you set in 3scale above in the section **Public Base URL** (without the `http://` and without the port) , e.g. **gateway.openshift.demo**, then click the **Create** button.



Create a new route for every 3scale service you define. Alternatively, you could avoid having to create a new route for every 3scale service you define by deploying a wildcard router.

CHAPTER 5. ADVANCED APICAST CONFIGURATION

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

5.1. DEFINE A SECRET TOKEN

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

▼ AUTHENTICATION SETTINGS

Host Header

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

Secret Token

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

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

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

5.2. CREDENTIALS

The API credentials within 3scale are always **user_key** or **app_id/app_key** depending on the authentication mode that you are using (OAuth is not available for the API gateway in the staging environment). However, you might want to use different credential names in your API. In this case, you will need to set custom names for the **user_key** if you are using the API key mode:

Auth user key

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

App ID parameter	<input type="text" value="app_id"/>
	Name of the parameter that acts of behalf of app id
App Key parameter	<input type="text" value="app_key"/>
	Name of the parameter that acts of behalf of app key

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

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

CREDENTIALS
LOCATION*

- ☐ As HTTP Headers
- ☒ As query parameters (GET) or body parameters (POST/PUT/DELETE)

CREDENTIALS
LOCATION*

- ☐ As HTTP Headers
- ☒ As query parameters (GET) or body parameters (POST/PUT/DELETE)

5.3. ERROR MESSAGES

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

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

AUTHENTICATION FAILED ERROR

Response Code*	403
Content-type	text/plain; charset=us-ascii
Response Body	Authentication failed

AUTHENTICATION MISSING ERROR

Response Code*	403
Content-type	text/plain; charset=us-ascii
Response Body	Authentication parameters missing

NO MATCH ERROR

Response Code*	404
Content-type	text/plain; charset=us-ascii
Response Body	No Mapping Rule matched

Following are some of the errors:

- Authentication missing: This error is generated whenever an API request does not contain any credentials. It occurs when users do not add their credentials to an API request.
- Authentication failed: This error is generated whenever an API request does not contain the valid credentials. It can be because the credentials are fake or because the application has been temporarily suspended.
- No match: This error means that the request did not match any mapping rule and therefore no metric is updated. This is not necessarily an error, but it means that either the user is trying random paths or that your mapping rules do not cover legitimate cases.

5.4. CONFIGURATION HISTORY

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

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

5.5. 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.

5.6. 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 6. 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 [standard policies](#) provided by Red Hat 3scale.

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

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

6.1. APICAST STANDARD POLICIES

Red Hat 3scale provides the following standard policies:

- [Section 6.1.1, “Anonymous Access Policy”](#)
- [Section 6.1.2, “APIcast CORS Request Handling Policy”](#)
- [Section 6.1.3, “Authentication Caching Policy”](#)
- [Section 6.1.4, “Echo Policy”](#)
- [Section 6.1.5, “Header Modification Policy”](#)
- [Section 6.1.6, “Liquid Context Debug Policy”](#)
- [Section 6.1.7, “Referrer Policy”](#)
- [Section 6.1.8, “RH-SSO/Keycloak Role Check Policy”](#)
- [Section 6.1.9, “SOAP Policy”](#)
- [Section 6.1.10, “Upstream Policy”](#)
- [Section 6.1.11, “URL Rewriting Policy”](#)
- [Section 6.1.12, “URL Rewriting with Captures Policy”](#)

You can [enable and configure](#) standard policies in the 3scale API Management.

6.1.1. Anonymous Access Policy

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

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

Following are the required configuration properties for the policy:

- **auth_type**: You can select from the following options and the property must correspond 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 6.1. Anonymous Access Policy

Anonymous access

builtin – Provides default credentials for unauthenticated requests

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

☒ **Enabled**

auth_type*

app_id_and_app_key

app_key*

myappid

app_id*

secret-app-key-123

6.1.2. APIcast CORS Request Handling Policy

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

- Allowed headers

- Allowed methods
- Allow credentials
- Allowed origin headers

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

Configuration properties

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

Policy object example

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

```
    "allow_origin": "https://example.com"
  }
}
```

For information on how to configure policies, refer to the [Creating a policy chain](#) section of the documentation.

6.1.3. Authentication Caching Policy

The authentication caching policy caches authentication calls made to APIcast. You can select an operating mode to configure the cache operations.

Authentication 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.

6.1.4. Echo Policy

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

Configuration properties

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

Policy object example

```

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

```

For information on how to configure policies, refer to the [Creating a policy chain](#) section of the documentation.

6.1.5. Header Modification Policy

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

The Header modification policy supports the following configuration parameters:

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

Each operation consists of the following parameters:

- **op**: Specifies the operation to be applied. The **add** operation adds a value to an existing header. The **set** operation creates a header and value, and will overwrite an existing header's value if one already exists. The **push** operation creates a header and value, but will not overwrite an existing header's value if one already exists. Instead, **push** will add the value to the existing header.
- **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 6.4, "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 }}`

Policy object example

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

For information on how to configure policies, see the [Creating a policy chain](#) section of the documentation.

6.1.6. Liquid Context Debug Policy



NOTE

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

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

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

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

6.1.7. Referrer Policy

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

6.1.8. RH-SSO/Keycloak Role Check Policy

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

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

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

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

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

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

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

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

The real roles must be specified in the policy.

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

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

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

```
"resource_access": {
```

```

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

```

Specify the client roles in the policy.

```

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

```

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

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

6.1.9. SOAP Policy

The SOAP policy matches SOAP action URLs 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

property	description	values	required?
metric_system_name	The metric_system_name property allows you to specify the 3scale backend metric with which your matched pattern will register a hit.	data type: string, must be a valid metric	yes

Policy object example

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

For information on how to configure policies, refer to the [Creating a policy chain](#) section of the documentation.

6.1.10. Upstream Policy

The Upstream policy allows you to parse a host request header using regular expressions and replace the request header URL with a new 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

property	description	values	required?
url	Using the url property, you can specify the replacement URL in the event of a match. Note that the upstream policy does not check whether or not this URL is valid.	data type: string, ensure this is a valid URL	yes

Policy object example

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

For information on how to configure policies, refer to the [Creating a policy chain](#) section of the documentation.

6.1.11. URL Rewriting Policy

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

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

The policy supports the following two sets of operations:

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

6.1.11.1. Commands for rewriting the path

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

- **op**: Operation to be applied. The options available are: **sub** and **gsub**. The **sub** operation replaces only the first occurrence of a match with your specified regular expression. The **gsub** operation replaces all occurrences of a match with your specified regular expression. See the documentation for the [sub](#) and [gsub](#) operations.

- **regex**: Perl-compatible regular expression to be matched.
- **replace**: Replacement string that is used in the event of a match.
- **options** (optional): Options that define how the regex matching is performed. For information on available options, see the [ngx.re.match](https://openresty.org/en/ngx-re-match.html) section of the OpenResty Lua module project documentation.
- **break** (optional): When set to true (checkbox enabled), if the command rewrote the URL, it will be the last one applied (all posterior commands in the list will be discarded).

6.1.11.2. Commands for rewriting the query string

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

- **op**: Operation to be applied to the query arguments. The following options are available:
 - **add**: Add a value to an existing argument.
 - **set**: Create the arg when not set and replace its value when set.
 - **push**: Create the arg when not set and add the value when set.
 - **delete**: Delete an arg.
- **arg**: The query argument name that the operation is applied on.
- **value**: Specifies the value that is used for the query argument. For value type "liquid" the value should be in the format `{{ variable_from_context }}`. For the **delete** operation the value is not taken into account.
- **value_type** (optional): Defines how the query argument value is evaluated and can either be **plain** for plain text or **liquid** for evaluation as a Liquid template. For more information, see [Section 6.4, "Using variables and filters in policies"](#). If not specified, the type "plain" is used by default.

Example

The URL Rewriting policy is configured as follows:

```
{
  "name": "url_rewriting",
  "version": "builtin",
  "configuration": {
    "query_args_commands": [
      {
        "op": "add",
        "arg": "addarg",
        "value_type": "plain",
        "value": "addvalue"
      },
      {
        "op": "delete",
        "arg": "user_key",
        "value_type": "plain",
```

```

        "value": "any"
      },
      {
        "op": "push",
        "arg": "pusharg",
        "value_type": "plain",
        "value": "pushvalue"
      },
      {
        "op": "set",
        "arg": "setarg",
        "value_type": "plain",
        "value": "setvalue"
      }
    ],
    "commands": [
      {
        "op": "sub",
        "regex": "^/api/v\\d+/",
        "replace": "/internal/",
        "options": "i"
      }
    ]
  }
}

```

The original request URI that is sent to the APIcast:

```

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

```

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

```

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

```

The following transformations are applied:

1. The substring **/api/v1/** matches the only path rewriting command and it is replaced by **/internal/**.
2. **user_key** query argument is deleted.
3. The value **pushvalue** is added as an additional value to the **pusharg** query argument.
4. The value **original** of the query argument **setarg** is replaced with the configured value **setvalue**.
5. The command **add** was not applied because the query argument **addarg** is not present in the original URL.

For information on how to configure policies, see the [Creating a policy chain](#) section of the documentation.

6.1.12. URL Rewriting with Captures Policy

The URL Rewriting with Captures policy is an alternative to the [Section 6.1.11, “URL Rewriting Policy”](#) policy and allows rewriting the URL of the API request before passing it to the API backend.

The URL Rewriting with Captures policy captures 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 transformation object consists of two properties:

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

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

Example

The URL Rewriting with Captures is configured as follows:

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

The original request URI that is sent to the APIcast:

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

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

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

6.2. ENABLING A STANDARD POLICY

Perform the following procedure to enable policies in the admin portal UI:

1. Log in to your AMP
2. Navigate to the **API service**.

The screenshot shows the 'Overview' page for the 'Echo API' in the Red Hat 3Scale API Management console. The left sidebar contains navigation links: Overview (highlighted), Analytics, Applications, Subscriptions, ActiveDocs, and Integration. The main content area is titled 'Overview' and includes a table with the following details:

Name	Echo API
System Name	api

Below the table are sections for 'Latest Apps' (listing apps from lajkagroup, Developer, and Metro), 'Analytics' (showing a line graph for hits), and 'Configuration, Methods and Settings' (indicating integration through the Ruby Plugin and authentication via API key).

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

The screenshot shows the 'Configuration' page for the 'Echo API'. The left sidebar has 'Integration' > 'Configuration' highlighted. The main content area is titled 'Configuration' and includes 'Integration settings' (Deployment Option: APIcast, Authentication: API Key (user_key)) and 'APIcast Configuration' (Private Base URL: https://echo-api.3scale.net:443, Mapping rules: / => hits, Credential Location: query). A link 'edit APIcast configuration' is highlighted in the top right corner of the APIcast Configuration section.

1. Under the **POLICIES** section, click **add policy**

The screenshot shows the 'POLICIES' section in the Red Hat 3Scale API Management console. It features a 'Policy Chain' table with a '+ Add Policy' button. The table lists the following policies:

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

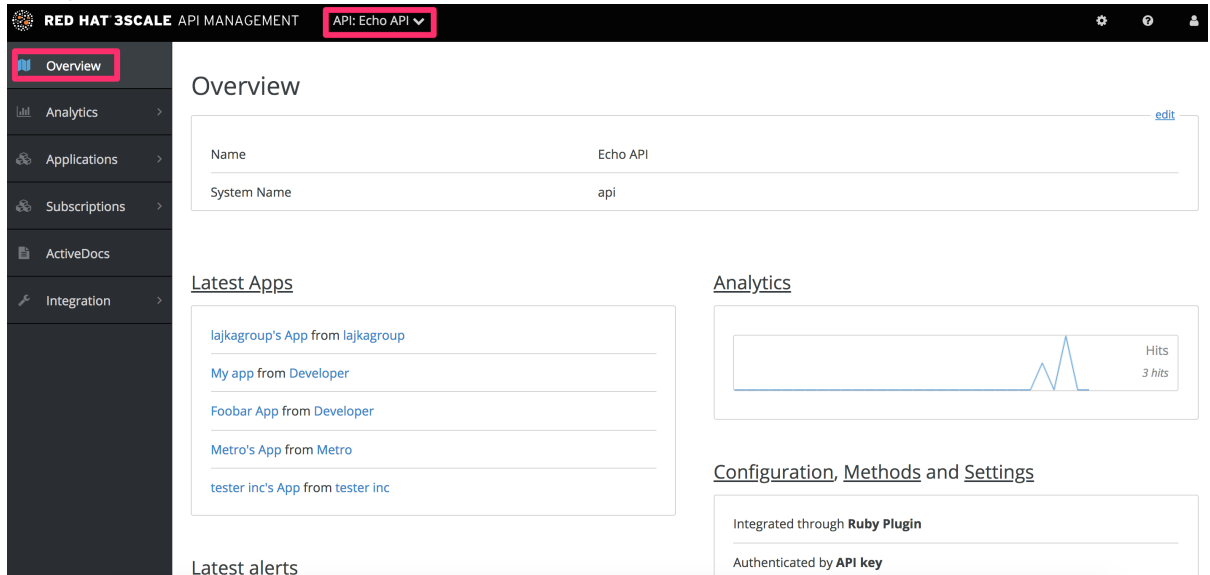
1. Select the policy you want to add and fill out the required fields

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

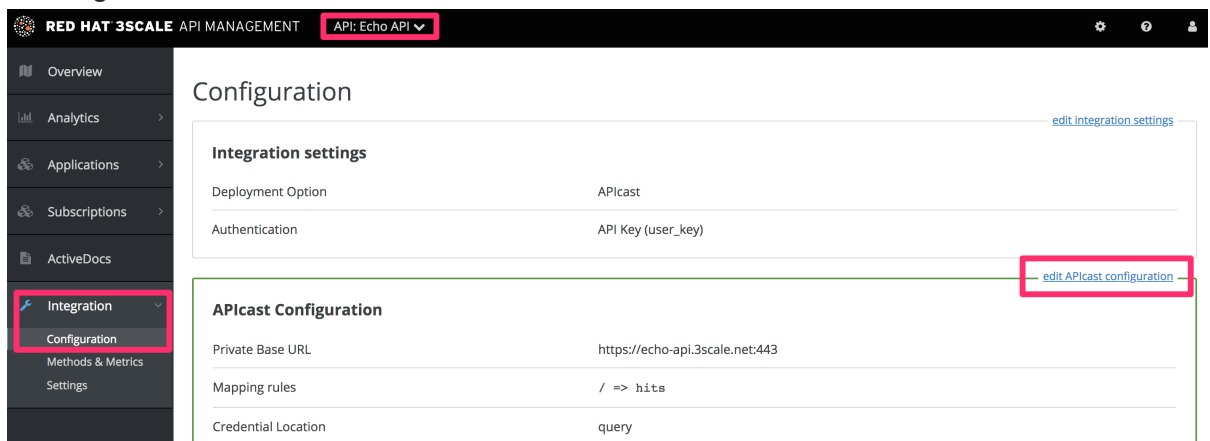
6.3. CREATING A POLICY CHAIN IN THE AMP

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

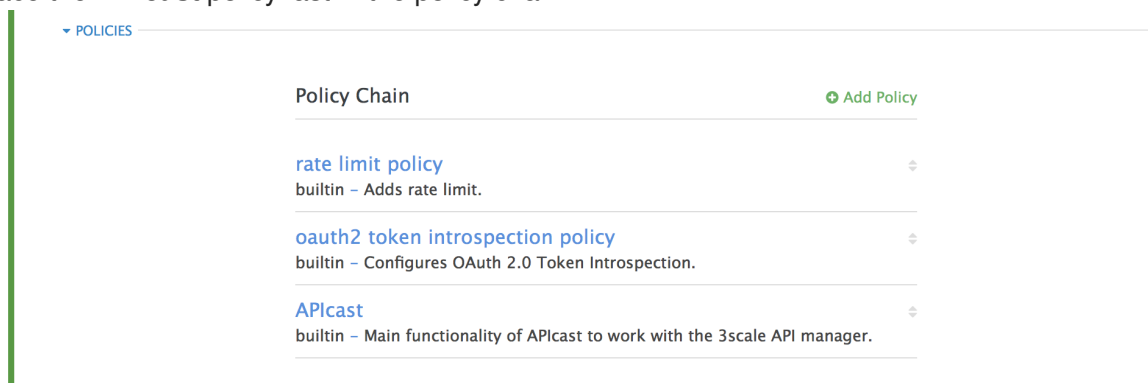
- Log in to your AMP
- Navigate to the API service



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



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



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

6.4. USING VARIABLES AND FILTERS IN POLICIES

Some [Section 6.1, “APIcast Standard Policies”](#) support Liquid templating that allows using not only plain string values, but also variables that are present in the context of the request.

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

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

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

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

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

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

For a full list of objects and values available in the context, see the [Section 6.1.6, “Liquid Context Debug Policy”](#).

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

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

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

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

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

Following is the list of the available functions:

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

CHAPTER 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

APICAST_BACKEND_CACHE_HANDLER

Values: strict | resilient

Default: strict

Deprecated: Use the [Caching](#) policy instead.

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

APICAST_CONFIGURATION_CACHE

Values: a number

Default: 0

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

APICAST_CONFIGURATION_LOADER

Values: boot | lazy

Default: lazy

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

APICAST_CUSTOM_CONFIG

Deprecated: Use [policies](#) instead.

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

APICAST_ENVIRONMENT

Default:

Value: string[:]

Example: production:cloud-hosted

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

APICAST_LOG_FILE

Default: stderr

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

APICAST_LOG_LEVEL

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

Default: warn

Specifies the log level for the OpenResty logs.

APICAST_ACCESS_LOG_FILE

Default: stdout

Defines the file that will store the access logs.

APICAST_OIDC_LOG_LEVEL

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

Default: err

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

APICAST_MANAGEMENT_API

Values:

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

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

APICAST_MODULE

Default: apicast

Deprecated: Use [policies](#) instead.

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

APICAST_OAUTH_TOKENS_TTL

Values: a number

Default: 604800

When configured to authenticate using OAuth, this param specifies the TTL (in seconds) of the tokens created.

APICAST_PATH_ROUTING

Values:

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

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

APICAST_POLICY_LOAD_PATH

Default: **APICAST_DIR/policies**

Value: string[:]

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

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

APICAST_PROXY_HTTPS_CERTIFICATE_KEY

Default:

Value: string

Example: **/home/apicast/my_certificate.key**

The path to the key of the client SSL certificate.

APICAST_PROXY_HTTPS_CERTIFICATE

Default:

Value: string

Example: **/home/apicast/my_certificate.crt**

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

APICAST_PROXY_HTTPS_PASSWORD_FILE

Default:

Value: string

Example: **/home/apicast/passwords.txt**

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

APICAST_PROXY_HTTPS_SESSION_REUSE

Default: on

Values:

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

APICAST_REPORTING_THREADS**Default:** 0**Value:** integer ≥ 0 **Experimental:** Under extreme load might have unpredictable performance and lose reports.

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

APICAST_RESPONSE_CODES**Values:**

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

Default: <empty> (*false*)

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

APICAST_SERVICES_LIST**Value:** a comma-separated list of service IDs

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

APICAST_SERVICE_\${ID}_CONFIGURATION_VERSION

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

APICAST_WORKERS**Default:** auto**Values:** *number* | auto

This is the value that will be used in the nginx **worker_processes** [directive](#). By default, APICast uses **auto**, except for the development environment where **1** is used.

BACKEND_ENDPOINT_OVERRIDE

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

OPENSSL_VERIFY

Values:

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

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

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

REDIS_HOST

Default: 127.0.0.1

APIcast requires a running Redis instance for OAuth 2.0 Authorization code flow. **REDIS_HOST** parameter is used to set the hostname of the IP of the Redis instance.

REDIS_PORT

Default: 6379

APIcast requires a running Redis instance for OAuth 2.0 Authorization code flow. **REDIS_PORT** parameter can be used to set the port of the Redis instance.

REDIS_URL

Default: no value

APIcast requires a running Redis instance for OAuth 2.0 Authorization code flow. **REDIS_URL** parameter can be used to set the full URI as DSN format like: **redis://PASSWORD@HOST:PORT/DB**. Takes precedence over **REDIS_PORT** and **REDIS_HOST**.

RESOLVER

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

THREESCALE_CONFIG_FILE

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

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

You can find sample configuration files in [examples](#) folder.

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

THREESCALE_DEPLOYMENT_ENV

Values: staging | production

Default: production

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

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

THREESCALE_PORTAL_ENDPOINT

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

<schema>://<password>@<admin-portal-domain>. The **<password>** can be either the [provider key](#) or an [access token](#) for the 3scale Account Management API. **<admin-portal-domain>** is the URL used to log into the admin portal.

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

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

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

OPENTRACING_TRACER

Example: `jaeger`

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

If empty, opentracing support will be disabled.

OPENTRACING_CONFIG

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

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

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

Example: `/tmp/jaeger/jaeger.json`

OPENTRACING_HEADER_FORWARD

Default: `uber-trace-id`

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

APICAST_HTTPS_PORT

Default: no value

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

APICAST_HTTPS_CERTIFICATE

Default: no value

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

APICAST_HTTPS_CERTIFICATE_KEY

Default: no value

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

all_proxy, ALL_PROXY

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

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

http_proxy, HTTP_PROXY

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

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

https_proxy, HTTPS_PROXY

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

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

no_proxy, NO_PROXY

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

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