



Red Hat 3Scale 2.0

Deployment Options

For Use with Red Hat 3Scale 2.0

Red Hat 3Scale 2.0 Deployment Options

For Use with Red Hat 3Scale 2.0

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 2.0.

Table of Contents

CHAPTER 1. APICAST OVERVIEW	4
1.1. PREREQUISITES	4
1.2. DEPLOYMENT OPTIONS	4
1.3. ENVIRONMENTS	4
1.4. API CONFIGURATION	5
1.5. CONFIGURE THE INTEGRATION SETTINGS	5
1.6. CONFIGURE YOUR SERVICE	5
1.7. MAPPING RULES	6
1.8. MAPPING RULES WORKFLOW	7
1.9. HOST HEADER	8
1.10. PRODUCTION DEPLOYMENT	8
1.11. PUBLIC BASE URL	8
1.12. PROTECTING YOUR API BACKEND	9
1.13. USING APICAST WITH PRIVATE APIS	9
CHAPTER 2. APICAST HOSTED	10
2.1. PREREQUISITES	10
2.2. STEP 1: DEPLOY YOUR API WITH APICAST HOSTED IN A STAGING ENVIRONMENT	10
2.3. STEP 2: DEPLOY YOUR API WITH THE APICAST HOSTED INTO PRODUCTION	11
2.3.1. Bear in mind	11
CHAPTER 3. APICAST SELF-MANAGED	12
3.1. PREREQUISITES	12
3.2. STEP 1: INSTALL OPENRESTY AND DEPENDENCIES	12
3.3. STEP 2: DEPLOY AND RUN APICAST	13
CHAPTER 4. APICAST ON THE DOCKER CONTAINERIZED ENVIRONMENT	14
4.1. PREREQUISITES	14
4.2. STEP 1: INSTALL THE DOCKER CONTAINERIZED ENVIRONMENT	14
4.3. STEP 2: RUN THE DOCKER CONTAINERIZED ENVIRONMENT GATEWAY	14
4.3.1. The Docker command options	15
4.4. STEP 3: TESTING APICAST	15
4.5. STEP 4: TROUBLESHOOTING APICAST ON THE DOCKER CONTAINERIZED ENVIRONMENT	16
4.5.1. Cannot connect to the Docker daemon error	16
4.5.2. Basic Docker command-line interface commands	16
4.6. STEP 5: CUSTOMISING THE GATEWAY	16
4.6.1. Custom Lua logic	16
4.6.2. Customising the configuration files	17
CHAPTER 5. RUNNING APICAST ON RED HAT OPENSIFT	18
5.1. PREREQUISITES	18
5.2. STEP 1: SET UP OPENSIFT	18
5.2.1. Install the Docker containerized environment	18
5.2.2. Start OpenShift cluster	19
5.2.2.1. Setting up OpenShift cluster on a remote server	19
5.3. STEP 2: DEPLOY APICAST USING THE OPENSIFT TEMPLATE	20
5.4. STEP 3: CREATE ROUTES IN OPENSIFT CONSOLE	21
5.4.1. Configure wildcard domains (technical preview)	23
5.4.1.1. Pre-requisites	24
5.4.1.2. Steps	24
CHAPTER 6. ADVANCED APICAST CONFIGURATION	25
6.1. DEFINE A SECRET TOKEN	25

6.2. CREDENTIALS	25
6.3. ERROR MESSAGES	26
6.4. CONFIGURATION HISTORY	27
6.5. DEBUGGING	28
6.6. EXTENDING THE GATEWAY	29
CHAPTER 7. APICAST AND OAUTH 2.0	30
7.1. PREREQUISITES	30
7.2. APICAST CONFIGURATION IN 3SCALE ADMIN PORTAL	30
7.3. RUNNING APICAST WITH OAUTH	31
7.4. TESTING THE FLOW	31
7.4.1. Requesting an Authorization Code	32
7.4.2. Exchanging the Authorization Code for an Access Token	33
CHAPTER 8. CODE LIBRARIES	34
8.1. PYTHON	34
8.2. RUBY GEM	34
8.3. PERL	34
8.4. PHP	34
8.5. JAVA	34
8.6. .NET	35
8.7. NODE.JS	35
CHAPTER 9. PLUGIN SETUP	36
9.1. HOW PLUGINS WORK	36
9.2. STEP 1: SELECT YOUR LANGUAGE AND DOWNLOAD THE PLUGIN	36
9.3. STEP 2: CHECK THE METRICS YOU'VE SET FOR YOUR API	37
9.4. STEP 3: INSTALL THE PLUGIN CODE TO YOUR APPLICATION	37
9.5. STEP 4: ADD CALLS TO AUTHORIZE AS API TRAFFIC ARRIVES	39
9.5.1. Usage on authrep mode	39
9.6. STEP 5: DEPLOY AND RUN TRAFFIC	40

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.

The latest released and supported version of APIcast is 2.0.

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 built-in:** Two APIcast (staging and production) come by default with the 3scale API Management Platform (AMP) installation. They come pre-configured and ready to use out-of-the-box.
- **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:
 - **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, you get 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.

1.4. API CONFIGURATION

Follow the next steps to configure APIcast in no time.

1.5. CONFIGURE THE INTEGRATION SETTINGS

Go to the **Dashboard** → **API** tab and click on the Integration link. If you have more than one API in your account, you'll need to select the API first.

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.

1.6. 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](#)

The screenshot shows a configuration form for an API. On the left, there is a green puzzle piece icon labeled 'API'. To the right, there is a search bar with a question mark icon. Below the search bar, the 'Private Base URL*' field is filled with the text 'https://echo-api.3scale.net:443'. Below this field, there is a small text label: '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.

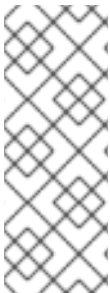
If you are using one of the Self-managed deployment options, save the configuration from the GUI and make sure it is pointing to your deployed API gateway by adding the correct host in the staging or production Public base URL field. Before making any calls to your production gateway, don't forget to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

1.7. 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	getgoodby

[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.

1.8. 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.

1.9. 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.

1.10. 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*.

1.11. 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 v2 will only accept calls to the hostname which is specified in the Public Base URL. For example, if for the Echo API used as an example above, 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.

1.12. 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.

1.13. 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](#)

The screenshot shows a configuration interface for APIcast. It is divided into three main sections: API, API GATEWAY, and CLIENT. Each section has a title, a field for configuration, and a help icon (question mark in a circle).

- API:**
 - Field: **Private Base URL*** with value `https://echo-api.3scale.net:443`.
 - Text: Private address of your API that will be called by the API gateway.
- API GATEWAY:**
 - Field: **Public Base URL*** with value `https://api-2445581460490.staging.apicast.io:443`.
 - Text: Public address of your API gateway in the staging environment.
 - Field: **Production Public Base URL*** with value `https://api-2445581460490.apicast.io:443`.
 - Text: Public address of your API gateway in the production environment.
 - Section: **MAPPING RULES** (indicated by a right-pointing arrow).
 - Section: **AUTHENTICATION SETTINGS** (indicated by a right-pointing arrow).
- CLIENT:**
 - Field: **API test GET request** with value `/`.
 - Text: 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:
 - Code block:

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

At the bottom left, there is a note: "Hit the test button to check the connections between client, gateway & API." At the bottom right, there is a blue button labeled "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.

The screenshot shows the APIcast configuration and environment management interface. At the top, there is a section for 'APIcast Configuration' with a link to 'edit APIcast configuration'. Below this, the configuration details are listed: Private Base URL: https://echo-api.3scale.net:443, Mapping rules: / => h1ta, Credential Location: query, and Secret Token: Shared_secret_sent_from_proxy_to_API_backend. Below the configuration is a section for 'Environments' with a link to 'Configuration history'. Under 'Environments', there are two environment cards. The first is 'Staging Environment' with the URL https://api-244581460490.staging.apicast.io:443 and a blue button labeled 'Promote v. 1 to Production'. The second is 'Production Environment' with the text '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 SELF-MANAGED

This tutorial shows the necessary steps to deploy the latest version of APIcast on your own server to have it ready to be used as a 3scale API gateway.

For details on the previous version of self-managed APIcast (Nginx downloadable configuration files), please see [here](#)

3.1. PREREQUISITES

You will need to configure APIcast in your 3scale Admin Portal as per the [APIcast Overview](#), if you haven't done so already. Make sure *Self-managed Gateway* is selected as the deployment option in the integration settings.

You should also have a server where you'll deploy your API gateway(s). This tutorial covers how to install your self-managed APIcast instance on a server running Red Hat Enterprise Linux – the operating system supported by the Red Hat 3scale API management platform. The server can be located either in the cloud, or on premise.

3.2. STEP 1: INSTALL OPENRESTY AND DEPENDENCIES

APIcast requires some external modules for NGINX. Even though it's possible to compile NGINX with these modules from source, we strongly recommend using OpenResty – an excellent bundle that already includes all the necessary modules.

This guide covers the steps to set up the official pre-built packages that OpenResty provides for Red Hat Enterprise Linux (RHEL) version 7. The latest installation instructions can be found in the [OpenResty documentation](#).

For other operating systems please refer to the [OpenResty installation instructions](#).

First, add the *openresty* repository to your RHEL system by creating the file `/etc/yum.repos.d/OpenResty.repo` with the following content:

```
[openresty]
name=Official OpenResty Repository
baseurl=https://copr-
be.cloud.fedoraproject.org/results/openresty/openresty/epel-7-$basearch/
skip_if_unavailable=True
gpgcheck=1
gpgkey=https://copr-
be.cloud.fedoraproject.org/results/openresty/openresty/pubkey.gpg
enabled=1
enabled_metadata=1
```

Install OpenResty and the **resty** command-line utility with the following command:

```
sudo yum install openresty openresty-resty
```

You can learn more about these and other OpenResty packages in [OpenResty documentation](#).

APIcast uses LuaRocks for managing Lua dependencies. As it's not in the standard Yum repositories, you must first enable the [EPEL](#) (Extra Packages for Enterprise Linux) package repository. Please refer to the [following article](#) on Red Hat Customer Portal for more information on how to enable EPEL on

RHEL.

For RHEL 7 you can run the following command:

```
sudo yum install https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm
```

Install LuaRocks:

```
sudo yum install luarocks
```

If you are using OAuth authentication method, please refer to [APIcast OAuth](#) guide for more details on how to set it up.

3.3. STEP 2: DEPLOY AND RUN APICAST

In the latest version of APIcast the code that implements the gateway logic is separated from the configuration of your API. In order to deploy APIcast to your server, you'll first need to get the APIcast code.

To use the latest stable version of APIcast, check the APIcast [releases page](#).

Go to the **apicast** directory, that you checked out with git or extracted from the downloaded archive.

Run the following command to install all the Lua dependencies:

```
sudo luarocks make apicast/*.rockspec --tree /usr/local/openresty/luajit
```

You can start APIcast using the **bin/apicast** executable included in the package.

```
THREESCALE_PORTAL_ENDPOINT=https://<access_token>@<domain>-admin.3scale.net bin/apicast
```

Here **<access_token>** is an [Access Token](#) for the 3scale Account Management API, and **<domain>-admin.3scale.net** is the URL of your 3scale Admin Portal.

This command will start APIcast and download the latest APIcast configuration from the 3scale Admin Portal.

bin/apicast executable accepts a number of options, you can check them out by running:

```
bin/apicast -h
```

Additional parameters can be specified using environment variables.

```
APICAST_LOG_FILE=logs/error.log bin/apicast -c config.json -d -v -v -v
```

The above command will run APIcast as a daemon (**-d** option,) using the configuration file **config.json**, with the error logs at **debug** level (**-v -v -v**) and written to the **logs/error.log** file inside the **apicast** directory (the **prefix** directory).

And that's it! You should now be running APIcast Self-Managed. If you're looking to install APIcast on [OpenShift](#) or using the [Docker containerized environment](#), we recommend checking out those tutorials.

CHAPTER 4. APICAST ON THE DOCKER CONTAINERIZED ENVIRONMENT

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

4.1. PREREQUISITES

You will need to configure APIcast in your 3scale Admin Portal as per the [APIcast Overview](#), if you haven't done so already.

4.2. 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 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:

- List all repositories:

```
sudo yum repolist all
```

- Find the ***-extras** repository
- Enable *extras* repository:

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

- Install the Docker containerized environment package:

```
sudo yum install docker
```

For other operating systems please refer to the Docker documentation on:

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

4.3. STEP 2: RUN THE DOCKER CONTAINERIZED ENVIRONMENT GATEWAY

Start the Docker daemon:

```
sudo systemctl start docker.service
```

To check if the Docker daemon is running use the following command:

```
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-amp20/apicast-gateway:1.0
```

Run APICast in a Docker-formatted container with the command:

```
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-amp20/apicast-
gateway:1.0
```

Here `<access_token>` is an [Access Token](#) for the 3scale Account Management API, the [Provider Key](#) can also be used instead of the access token, and `<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 grabs the JSON configuration file from your 3scale portal. For other configuration options check out the [APICast Overview](#) guide.

4.3.1. The Docker command options

Here are some useful options that can be used with `docker run` command:

- `--rm` Automatically remove the container when it exits
- `-d` or `--detach` Run container in background and print container ID. When it is not specified, the container runs in foreground mode, and you can stop it by **CTRL + c**. When started in detached mode, you can reattach to the container with the `docker attach` command, for example, `docker attach apicast`.
- `-p` or `--publish` Publish 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 link: <https://support.3scale.net/doc/management-api.md> [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` Set environment variables
- `-v` or `--volume` Mount a volume. The value is typically represented as `<host path="">[:<options>].<container path="">[:<options>]`. `<options>` is an optional attribute, it can be set to `:ro` to specify that the volume will be read only (it is mounted in read-write mode by default). Example: `-v /host/path:/container/path:ro`.

See the [Docker run reference](#) for more information on available options.

4.4. STEP 3: TESTING APICAST

So now your Docker-formatted container is running with your own configuration file and the Docker-formatted image from the 3scale registry. You will want to 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

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

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

4.5.1. *Cannot connect to the Docker daemon error*

If you see the error message:

```
docker: Cannot connect to the Docker daemon. Is the docker daemon running on this host?.
```

then it may be that the Docker service hasn't started. You can check the status of the Docker daemon by running **sudo systemctl status docker.service**.

Make sure you are running as root user, as the Docker containerized environment requires root permissions in RHEL by default (see more information [here](#)).

4.5.2. Basic Docker command-line interface commands

If you started the container in 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 the list of the running containers and their IDs and names with the command **sudo docker ps**.

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

Please refer to [Docker commands reference](#) for more information on available commands.

4.6. STEP 5: CUSTOMISING THE GATEWAY

There are some customizations that cannot be managed through the admin portal and require writing custom logic to APIcast itself.

4.6.1. Custom Lua logic

The easiest way to customise APIcast logic is to rewrite the existing file with your own and attach it as a volume.

```
-v $(pwd)/path_to/file.lua:/opt/app-root/src/src/file.lua:ro
```

The `-v` flag attaches the local file to the stated path in the Docker-formatted container.

4.6.2. Customising the configuration files

The same steps apply to custom configuration files as the Lua scripting. If you just want to add to the existing conf files rather than overwrite then ensure the name of your new file does not clash with pre-existing ones. This will automatically be included in the main *nginx.conf*.

To make edits to the *config.json* you can grab this file from your admin portal with the following URL: <https://<account>-admin.3scale.net/admin/api/nginx/spec.json> and copy & paste the contents locally. You can pass this local JSON file with the following command to start APICast:

```
docker run --name apicast --rm -p 8080:8080 -v  
$(pwd)/path_to/config.json:/opt/app/config.json:ro -e  
THREESCALE_CONFIG_FILE=/opt/app/config.json
```

You should now be able to run APICast on the Docker containerized environment. For other deployment options, check out the related articles.

CHAPTER 5. RUNNING APICAST ON RED HAT OPENSIFT

This tutorial describes how to use APICAST v2 – the dockerized 3scale API Gateway that is packaged for easy installation and operation on Red Hat OpenShift v3.

5.1. 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.

5.2. 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

5.2.1. 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:

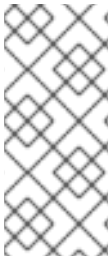
```
sudo systemctl start docker
```

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

```
sudo systemctl status docker
```

5.2.2. 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**.

5.2.2.1. 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>.

5.3. 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/2.0.0.GA-redhat-2/apicast-gateway/apicast.yml
```

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

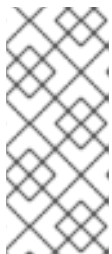
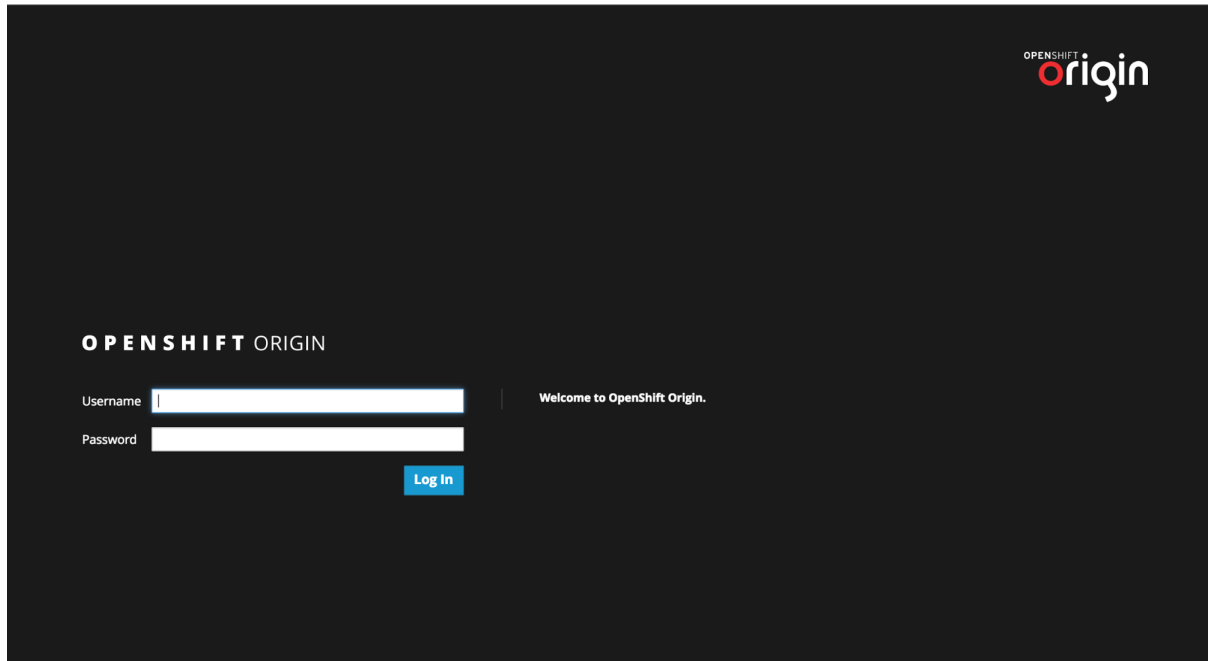
```
--> Creating resources with label app=3scale-gateway ...
      deploymentconfig "apicast" created
      service "apicast" created
--> Success
      Run 'oc status' to view your app.
```

5.4. 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.

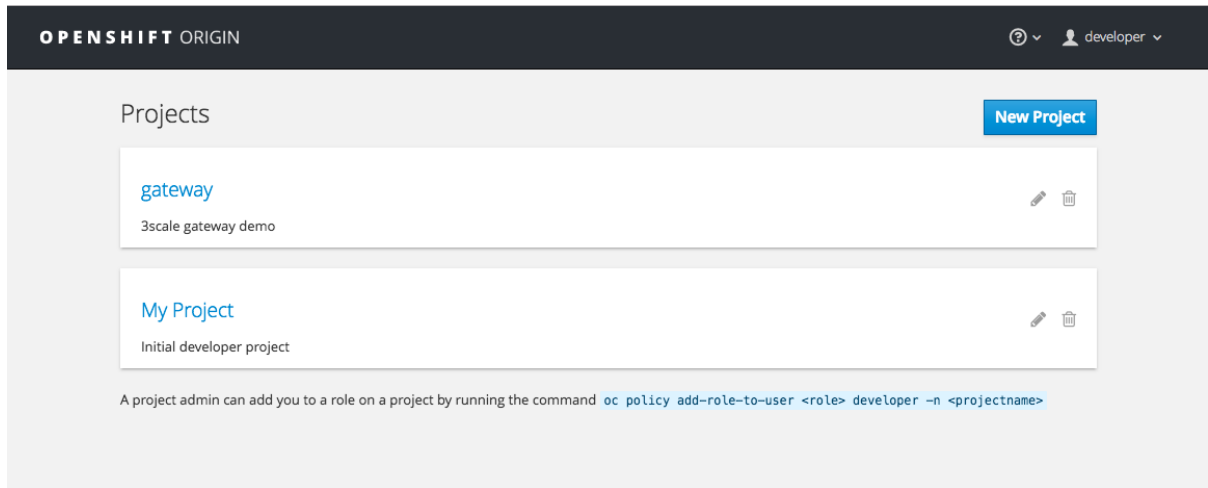
You should see the login screen:



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.

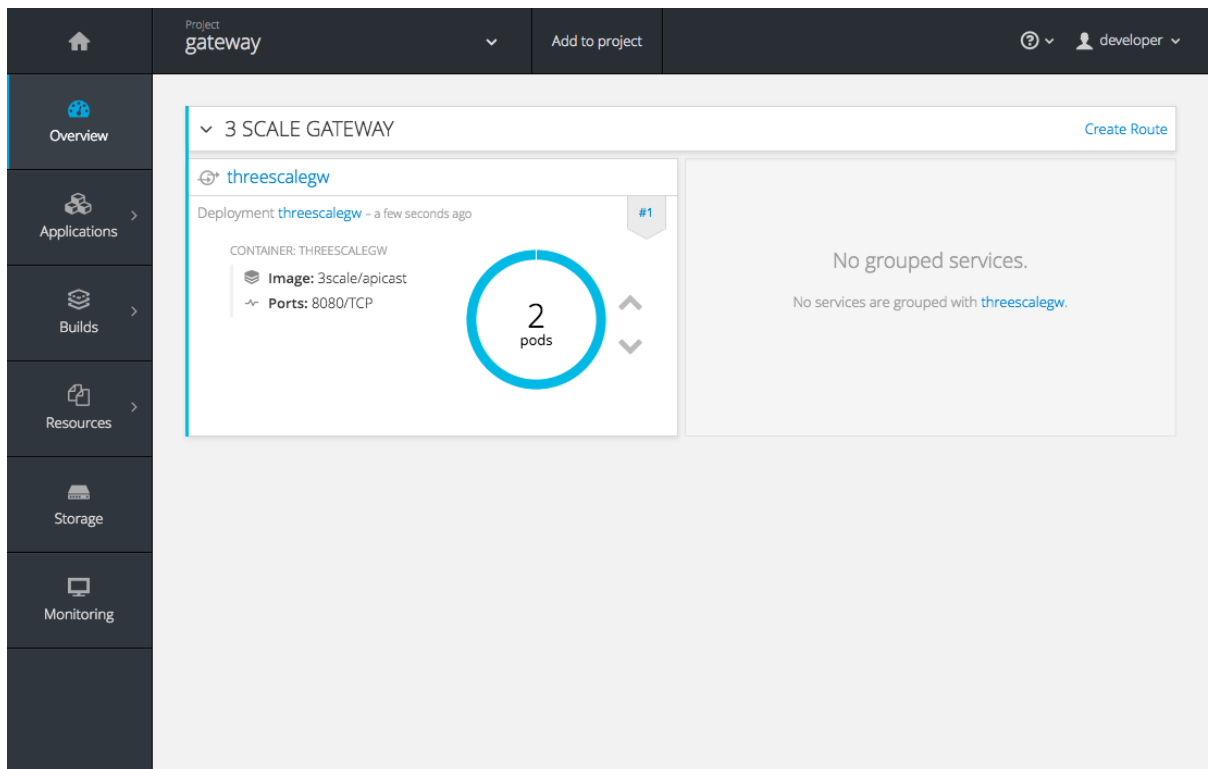
2. Log in using the *developer* credentials created or obtained in the *Setup OpenShift* section above. You will see a list of projects, including the *"gateway"* project you created from the command line above.



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 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.

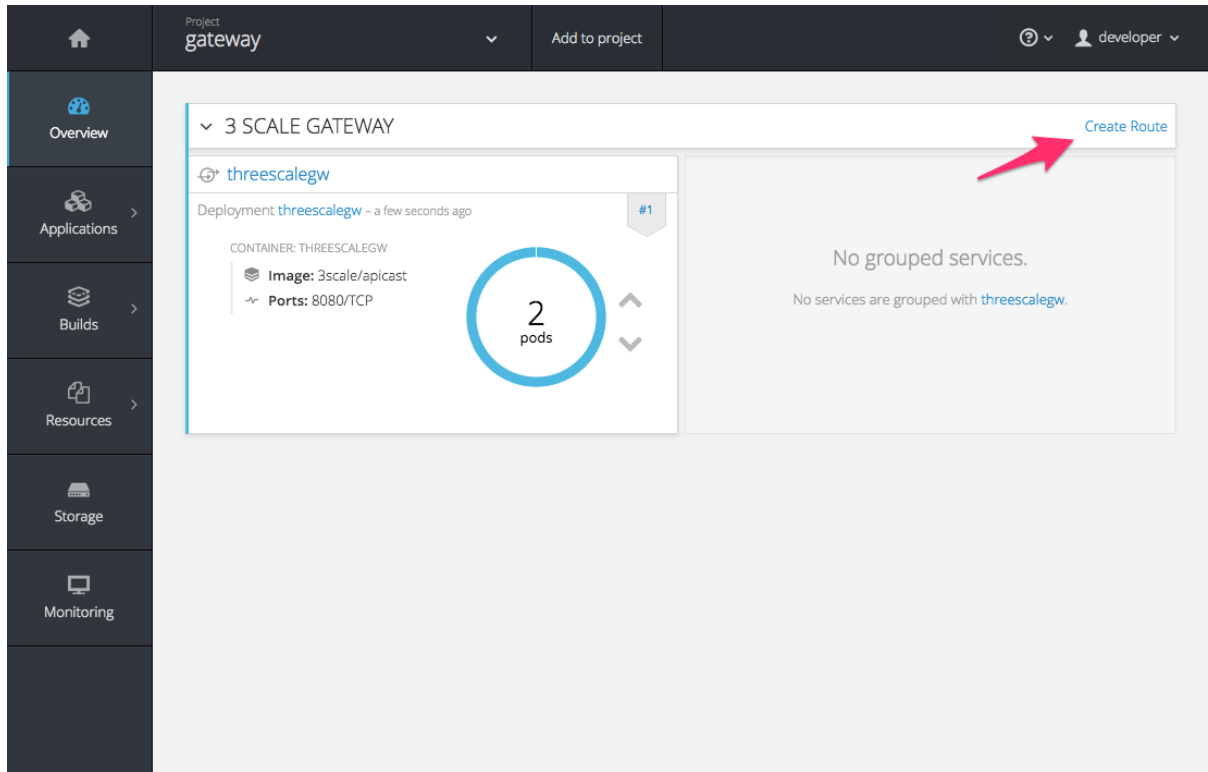
When the build completes, the UI will refresh and show two instances of APIcast (2 pods) that have been started by OpenShift, as defined in the template.



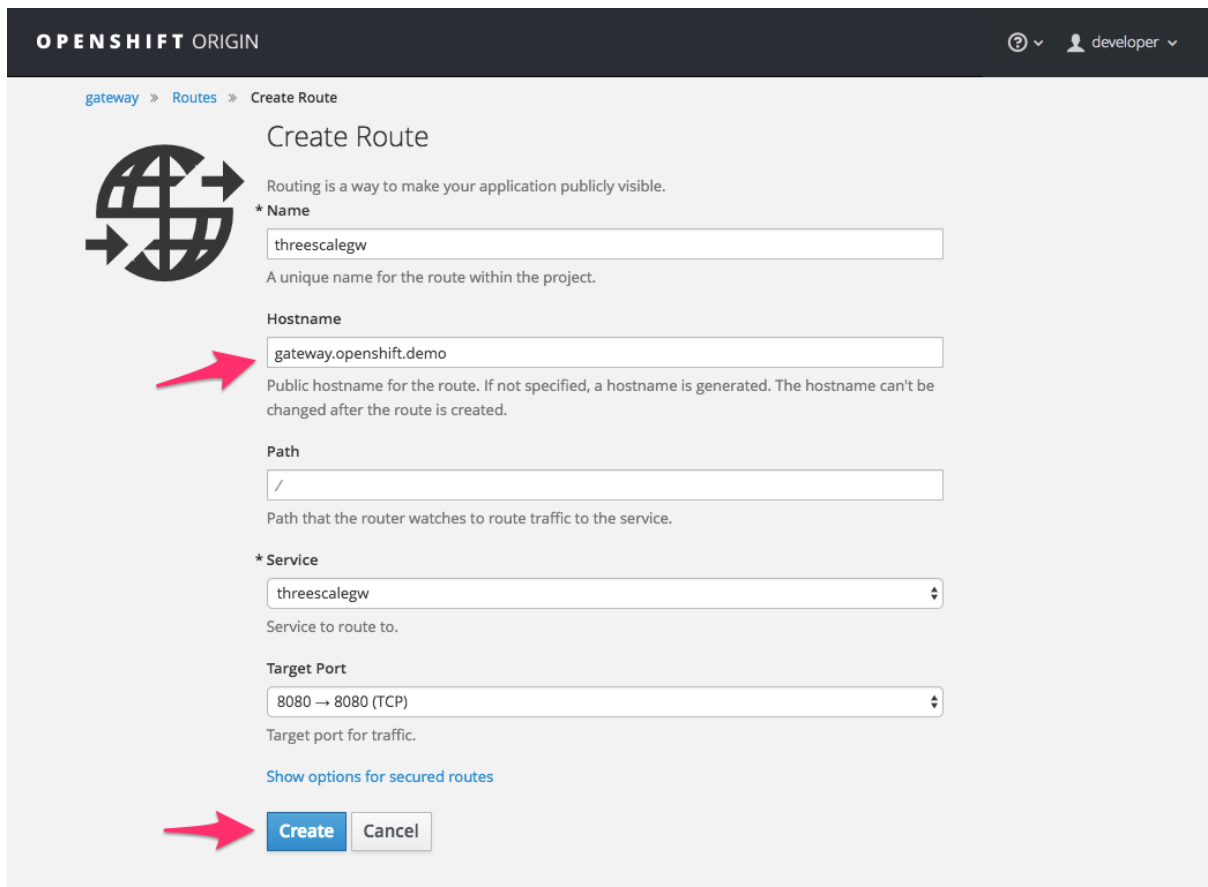
Each APIcast instance, upon starting, downloads the required configuration from 3scale using the settings you provided on the **Integration** page of your 3scale Admin Portal.

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.

5.4.1. Configure wildcard domains (technical preview)

Technical preview: Note that the wildcard router is currently a technical preview feature.

5.4.1.1. Pre-requisites

Use of the wildcard domain feature requires the following:

- OpenShift version 3.4
- a wildcard domain that is not being used for any other routes, or as another project's namespace domain
- your router must be configured to allow wildcard routes

5.4.1.2. Steps

Perform the following steps to configure a wildcard domain:

1. From a terminal session, log in to OpenShift: **oc login**
2. Create a [wildcard router](#) configured with your wildcard domain
3. Download the wildcard.yml template from the latest release in [this](#) GitHub repository.
4. Switch to the project which contains your AMP deployment: **oc project <project_name>**
5. Enter the **oc new-app** command, specifying the following:
 - the **-file** option and the path to the wildcard.yml template
 - the **--param** option and the WILDCARD_DOMAIN of your openshift cluster
 - the **--param** option and the TENANT_NAME from the project which contains your AMP deployment

```
oc new-app -f wildcard.yml --param WILDCARD_DOMAIN= guide.
```

6. Run OpenShift V3 on your dedicated datacenter or on your favorite cloud platform using the advanced installation documentation [listed above](#).
7. Register a custom domain name for your API services, and configure your API integration in 3scale Admin Portal, and in OpenShift by adding new routes.
8. Learn more about OpenShift from the [OpenShift documentation](#)

CHAPTER 6. ADVANCED APICAST CONFIGURATION

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

6.1. DEFINE A SECRET TOKEN

For security reasons, any request from 3scale's gateway to your API backend will contain 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	<input type="text"/>
	Lets you define a custom <code>Host</code> request header. This is needed if your API backend only accepts traffic from a specific host.
Secret Token	<input type="text" value="Shared_secret_sent_from_proxy_to_API_backend"/>
	Enables you to block any direct developer requests to your API backend; each 3scale API gateway call to your API backend contains a request header called <code>x-3scale-proxy-secret-token</code> . 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 will act 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 so wish. This gives an extra layer of security to protect your public endpoint while you're 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 might know your API backend could 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's always better to have a fence available.

6.2. CREDENTIALS

The API credentials within 3scale are always **user_key** or **app_id/app_key** depending on the authentication mode you're 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'll need to set custom names for the **user_key** if you're using API key mode:

Auth user key

or for the **app_id** and **app_key**:

App ID parameter

Name of the parameter that acts of behalf of app id

App Key parameter

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 3scale's backend. Note that the new credential name has to be alphanumeric.

You can decide whether 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)

6.3. ERROR MESSAGES

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

It's important to note that 3scale's API gateway in the staging environment will do a pass through 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 since some requests will be terminated by the gateway.

AUTHENTICATION FAILED ERROR

Response Code*	<input type="text" value="403"/>
Content-type	<input type="text" value="text/plain; charset=us-ascii"/>
Response Body	<input type="text" value="Authentication failed"/>

AUTHENTICATION MISSING ERROR

Response Code*	<input type="text" value="403"/>
Content-type	<input type="text" value="text/plain; charset=us-ascii"/>
Response Body	<input type="text" value="Authentication parameters missing"/>

NO MATCH ERROR

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

These errors are the following:

- Authentication missing: this error will be generated whenever an API request does not contain any credentials. This occurs when users forget to add their credentials to an API request.
- Authentication failed: this error will be generated whenever an API request does not contain valid credentials. This 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, therefore no metric is updated. This is not necessary an error, but it means that either the user is trying random paths or that your mapping rules do not cover legitimate cases.

6.4. CONFIGURATION HISTORY

Every time you click on the **Update & Test Staging Configuration** button, the current configuration will be 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 we provide a history of all your configuration versions with their associated JSON files. These files can be used to check what configuration you had deployed at any moment on time. If you want to, you can recreate any deployments manually.

6.5. DEBUGGING

Setting up the gateway configuration is easy, but still some errors can occur on the way. For those cases the gateway can return some useful debug information that will be helpful to track down what is going on.

To enable the debug mode on 3scale's API gateway in the staging environment you can add the following header with your provider key to a request to your gateway: **X-3scale-debug:**

YOUR_PROVIDER_KEY

When the header is found and the provider key 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[version_1]=1&usage[word]=1
```

Basically, **X-3scale-matched-rules** tells you which mapping rules have been activated by the request. Note that it is a list. The header **X-3scale-credentials** returns the credentials that have been passed to 3scale's backend. Finally **X-3scale-usage** tells you the usage that will be reported to 3scale's backend.


You can check the logic for your mapping rules and usage reporting in the Lua file, in the function **extract_usage_x()** where **x** is your **service_id**.

```
...
local args = get_auth_params(nil, method)
local m = ngx.re.match(path, [=^[^/v1/word/([\w_\. -]+)\.json]=])
if (m and method == "GET") then
  -- rule: /v1/word/{word}.json --
  params["word"] = m[1]
  table.insert(matched_rules, "/v1/word/{word}.json")
  usage_t["word"] = set_or_inc(usage_t, "word", 1)
  found = true
end
...
```

In this example, the comment **-- rule: /v1/word/{word}.json --** shows which particular rule the Lua code refers to. Each rule has a Lua snippet like the one above. Comments are delimited by **--**, **-- [,] --** in Lua, and with **#** in NGINX.

Unfortunately, there is no automatic rollback for Lua files if you make any changes. However, if your current configuration is not working but the previous one was OK, you can download the previous configuration files from the deployment history.

Staging – configure & test your integration [visit documentation](#) deployed [deployment history](#)

 API ?

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

6.6. EXTENDING THE GATEWAY

3scale's hosted APIcast is quite flexible, but there are sometimes things that cannot be done because the console interface doesn't allow it or because of security reasons due to a multi-tenant gateway.

If you need to extend your API gateway, you can always run it locally on your servers ([see the APIcast self-managed section to deploy your own API gateway](#)) and point it to the proper service and configuration from your 3scale account.

When you're running the gateway on your servers, you can customize it to enable any feature you might need – NGINX with Lua is an extremely powerful, open-source piece of technology.

We have written a blog post explaining [how to augment APIs with NGINX and Lua](#). Here are some examples that can be done:

- Basic DoS protection: white-lists, black-lists, rate-limiting at the second level.
- Define arbitrarily complex mapping rules.
- API rewrite rules – for example, you might want API requests starting with `/v1/*` to be rewritten to `/versions/1/*` when they hit your API backend.
- Content filtering: you can add checks on the content of the requests, either for security or to filter out undesired side effects.
- Content rewrites: you can transform the results of your API.
- Many, many more. Combining the power and flexibility of NGINX with Lua scripting is an awesome combination.

CHAPTER 7. APICAST AND OAUTH 2.0

Please note that OAuth authentication mode is not available on APIcast hosted yet.

This document refers to configuring OAuth 2.0 for the **latest version of APIcast**. To run APIcast, follow any of its deployment method instructions: [Self-Managed](#), [the Docker containerized environment](#) or [OpenShift](#).

For details on OAuth support in the previous version of APIcast (Nginx downloadable configuration files), please see [here](#).

7.1. PREREQUISITES

APIcast offers support for the OAuth 2.0 Authorization Code flow out of the box as long as the following pre-requisites are met:

- An Authorization Server as defined in [RFC6749#1.1](#) with the one exception that the access tokens will be issued by APIcast instead. In this case the Authorization Server will only authenticate the resource owner and obtain their authorization.
- A [Redis](#) instance.

7.2. APICAST CONFIGURATION IN 3SCALE ADMIN PORTAL

In order to configure this, we will first need to choose the OAuth Authentication method from the Integration Settings (**API > Integration > edit integration settings**) screen on our 3scale Admin Portal. We will also need to ensure we have selected the *Self-managed Gateway* option, as it is not currently possible to run APIcast with OAuth 2.0 Authorization Code support on the APIcast cloud gateway.

Once that is done, we will see an additional field in the Integration Screen (**API > Integration**) under *Authentication Settings: OAuth Authorization Endpoint*. Here we define where Resource Owners will be redirected to in order to authenticate and confirm they authorize a given client access to their resources (as per [RFC6749#4.1](#)).

All other fields on the Integration Screen should be configured as per the [APIcast Overview](#) document. Since we will be running all components locally for this example, my *Public Base URL* where APIcast is running will be <http://localhost:8080>.

Production: Self-managed Gateway

To deploy an on-premises API gateway, add the Public Base URL of your API, download the Nginx Config files and [follow the documentation](#) to install in your servers.



API

Private Base URL



API GATEWAY

Public Base URL

Public address of your API gateway in the production environment. This is used to customize the `server_name` directive in the NGINX Config file which will otherwise be set to the variable `$hostname`.

In order to show a sample integration with an Authorization Server we will use a very simple Ruby app to act as an Authorization Server. This app will run on localhost port 3000 with the OAuth Authorization Endpoint at <http://localhost:3000/auth/login>.

▼ AUTHENTICATION SETTINGS

OAuth Authorization Endpoint

Since the service has OAuth authentication enabled you have to provide a URL where the consumers login and authenticate. Read more about [API gateway and OAuth setup](#) on our support page.

The sample code for this app can be found in the `apicast/examples` directory under `oauth2/auth-server`. There is also a file named `docker-compose.yml` that allows you to deploy a test environment to test the API Integration and OAuth 2.0 Authorization Code Flow..

7.3. RUNNING APICAST WITH OAUTH

In order to start APIcast in OAuth mode, we will first need to have a Redis instance running and pass in the instance details when running APIcast.

In our case, this is running on `localhost:6379`. Since this is running on the default port, we only need to specify `REDIS_HOST`, if this were running on any other port, we would also have to specify `REDIS_PORT`. We can then start APIcast like so:

```
REDIS_HOST=localhost
THREESCALE_PORTAL_ENDPOINT=https://MY_PROVIDER_KEY@MY_ADMIN_PORTAL.3scale.
net bin/apicast -v
```

Here we have added the `-v` flag to run APIcast in verbose mode, in order to allow us to get more debugging output in case anything goes wrong, but this can be omitted.

Our Authorization Server should also be up and running and ready to receive requests.

7.4. TESTING THE FLOW

In order to test this flow, we will be running all components: APIcast, Redis, Authorization Server and Client, using the [Docker Compose tool](#) as per the `oauth2` example listed in the `examples` directory in the [APIcast GitHub repository](#). This will take care of starting up all of the required components. However, you can also start each of them up individually. Please ensure you do not already have any APIcast, Redis or other component instances already running on the same ports at the same time. Note that you will need to have the [Docker containerized environment](#) and the [Docker Compose tool](#) installed to use these examples.

The first component to come into play is the sample client application, which would be written by a Developer to request an access token from APIcast. This is a simple Ruby app that will act as the client (as per the client role defined in [RFC6749#1.1](#)) in the Authorization Code Flow. The sample code for the client app can be found in the `apicast/examples` folder under `oauth2/client`.

We will be running this application on `localhost:3001`. The application has a `/callback` endpoint defined to receive and process any authorization codes and access tokens from APIcast. As such, this location will need to be set up on our 3scale account, under a test application, as the **Redirect URL**.

3scale's App

 Edit  Delete

Description Default application created on signup.

Service API

State


 Live **suspend**


API Credentials

Client ID 72abc10a

Client Secret



 Regenerate

Redirect URL <http://localhost:3001/callback>  Change

7.4.1. Requesting an Authorization Code

Once all the configuration is in place and we have started all of our components (either using the **docker-compose** template or individually,) we can navigate to <http://localhost:3001>. Here we can enter a **client_id**, **redirect_uri** and **scope**, then click **Authorize** to request an authorization code. We will get the **client_id** and **client_secret** values from the 3scale application above. The **redirect_url** will be defined by the client application itself, and as such will already come pre-filled. The **scope** value defines the type of access requested. In this case it can be any string value and it will be displayed to the resource owner when they provide their consent.

The client application will then redirect the resource owner to the Authorization endpoint (as per [RFC6749#4.1.1](https://tools.ietf.org/html/rfc6749#4.1.1)), in this case the **/authorize** endpoint on our APIcast instance:

localhost:8080/authorize. The resulting client request will be something like:

```
GET /authorize?
response_type=code&client_id=CLIENT_ID&state=STATE_VALUE&redirect_uri=http
s%3A%2F%2Flocalhost%3A3001%2Fcallback HTTP/1.1
Host: localhost:8080
```

APIcast will check the **client_id** and **redirect_uri** values and redirect the resource owner to the Authorization Server endpoint (<http://localhost:3000/auth/login>) for authentication. APIcast will forward on the original request parameters to the Authorization Server.

The resource owner should log in to the Authorization Server (our sample Authorization Server will accept any values for username and password) at which point they will be presented with a consent page to accept or deny the request for access.

The Authorization server will then either redirect back to APIcast (on <http://localhost:8080/callback>) to issue an authorization code on request acceptance or the application's `redirect_uri` directly on request denial. This redirection request should include the original `state` value as well as the client's `redirect_uri`, e.g

```
HTTP/1.1 302 Found
Location: https://localhost:8080/callback?
state=STATE_VALUE&redirect_uri=http://localhost:3001/callback
```

If the request is accepted, an authorization code will be issued (as per [RFC6749#4.1.2](#)) by APIcast for the client.

This Authorization Code is sent to the client's Redirect URL (client callback endpoint on <http://localhost:3001/callback> in this case) and will be displayed at the sample client. The redirection request back to the client will be something like:

```
HTTP/1.1 302 Found
Location: https://localhost:3001/callback?
state=STATE_VALUE&code=AUTHORIZATION_CODE
```

We can then exchange this for an access token, by filling in our application's `client_secret`.

7.4.2. Exchanging the Authorization Code for an Access Token

Once an authorization code is returned back to the sample client, you can exchange that for an access token by once again entering in the `client_id`, additionally providing a `client_secret` and clicking **Get Token** to request an access token. At this point, the client application makes a request to the APIcast access token endpoint, in our case <http://localhost:8080/oauth/token>, sending the client credentials and Redirect URL along with the authorization code. This client request will be something like:

```
POST /oauth/token HTTP/1.1
Host: localhost:8080
Content-Type: application/x-www-form-urlencoded
```

```
grant_type=authorization_code&code=AUTHORIZATION_CODE&client_id=CLIENT_ID&
client_secret=CLIENT_SECRET&redirect_uri=https%3A%2F%2Flocalhost%3A3001%2F
callback
```

Ideally the `client_id` and `client_secret` would be sent in the Authorization header using Basic HTTP Authentication but, for simplicity, the example client we're using sends them as body parameters.

APIcast will then validate these credentials and generate an access token with a fixed TTL of one week. An example token will look something like:

```
{
  "token_type": "bearer",
  "expires_in": 604800,
  "access_token": "b2ca2d108c193ef7c96a1f07b3a7c66d289aba42"
}
```

And that's it! We have now added and tested OAuth 2.0 Authorization Code Flow for our APIcast instance.

CHAPTER 8. CODE LIBRARIES

3scale plugins allow you to connect to the 3scale architecture in a variety of core programming languages. Plugins can be deployed anywhere to act as control agents for your API traffic. Download the latest version of the plugin and gems for the programming language of your choice along with their corresponding documentation below.

You can also find us on [GitHub](#). For other technical details, refer to the [technical overview](#).

8.1. PYTHON

Download the code [here](#).

Standard distutils installation – unpack the file and from the new directory run:

```
sudo python setup.py install
```

Or you can put ThreeScale.py in the same directory as your program. See the README for some usage examples.

8.2. RUBY GEM

You can download from [GitHub](#) or install it directly using:

```
gem install 3scale_client
```

8.3. PERL

Download the code bundle [here](#).

It requires:

- » LWP::UserAgent (installed by default on most systems)
- » XML::Parser (now standard in perl >= 5.6)

Run this to install into the default library location:

```
perl Makefile.PL && make install
```

Run this for further install options:

```
make help
```

8.4. PHP

Download the code [here](#) and drop into your code directory – see the README in the bundle.

8.5. JAVA

Download the JAR bundle [here](#).

Just unpack it into a directory and then run the maven build.

8.6. .NET

Download the code [here](#), and see the bundled README.txt for instructions.

8.7. NODE.JS

Download the code [here](#), or install it through the package manager npm:

```
npm install 3scale
```

CHAPTER 9. PLUGIN SETUP

By the time you complete this tutorial, you'll have configured your API to use the available 3scale code plugins to manage access traffic.

There are several different ways to add 3scale API Management to your API – including using [APIcast \(API gateway\)](#), [Amazon API Gateway with Lambda](#), [3scale Service Management API](#), or code plugins. This tutorial drills down into how to use the code plugin method to get you set up.

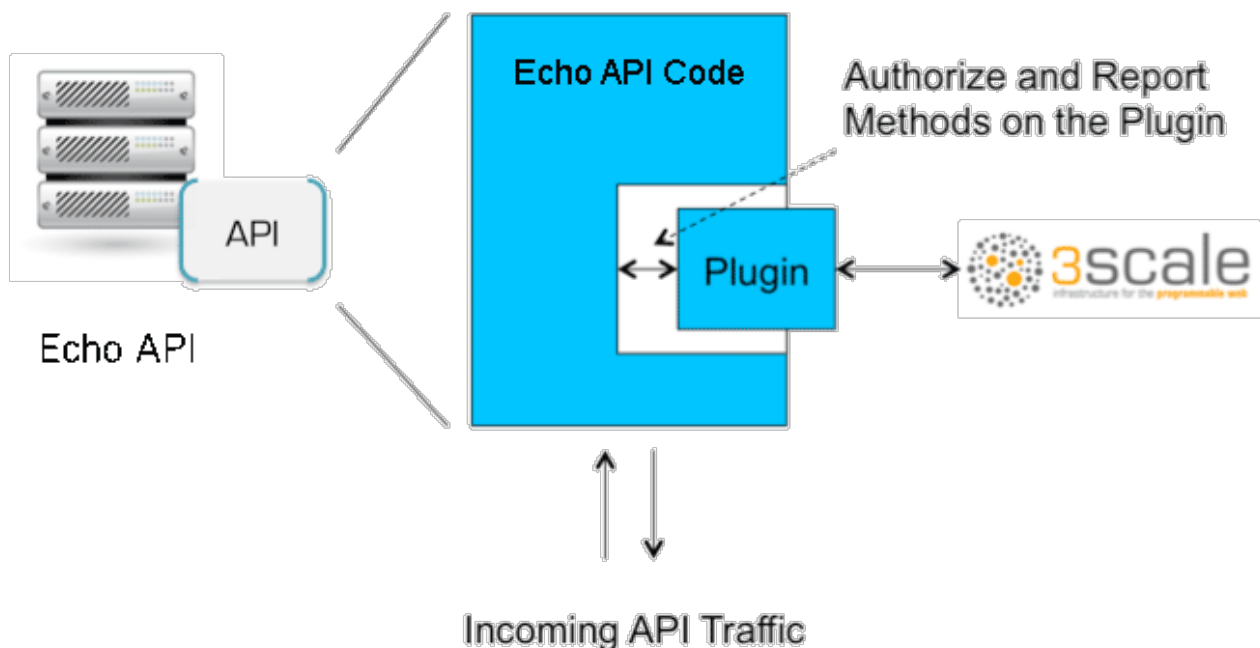
9.1. HOW PLUGINS WORK

3scale API plugins are available for a variety of implementation languages including Java, Ruby, PHP, .NET, and others – the full list can be found in the [code libraries section](#). The plugins provide a wrapper for the 3scale Service Management API. Find the Service Management API in the 3scale ActiveDocs, available in your Admin Portal, under the **Documentation** → **3scale API Docs** section, to enable:

- API access control and security
- API traffic reporting
- API monetization

This wrapper connects back into the 3scale system to set and manage policies, keys, rate limits, and other controls that you can put in place through the interface. Check out the [Getting Started](#) guide to see how to configure these elements.

Plugins are deployed within your API code to insert a traffic filter on all calls as shown in the figure.



9.2. STEP 1: SELECT YOUR LANGUAGE AND DOWNLOAD THE PLUGIN

Once you've created your 3scale account ([signup here](#)), navigate to the [code libraries](#) section on this site and choose a plugin in the language you plan to work with. Click through to the code repository to get the bundle in the form that you need.

If your language is not supported or listed, [let us know](#) and we'll inform you of any ongoing support efforts for your language. Alternatively, you can connect directly to the [3scale Service Management API](#).

9.3. STEP 2: CHECK THE METRICS YOU'VE SET FOR YOUR API

As described in the [API definition tutorial](#), you can configure multiple metrics and methods for your API on the API control panel. Each metric and method has a system name that will be required when configuring your plugin. You can find the metrics in **API > Definition** area of your Admin Portal.

3scale

Dashboard Developers Applications Billing Analytics **API** Developer Portal Settings

Overview ActiveDocs

Definition

Integration

Application Plans

Settings

Alerts

Definition

Name: API
System Name: api

[Create new method](#)

Methods

Add the methods of this API to get data on their individual usage. Method calls trigger the built-in Hits-metric. Usage limits and pricing rules for individual methods are defined from within each [Application Plan](#). A method needs to be mapped to one or more URL patterns in the [Mapping Rules section](#) of the integration page so specific calls to your API up the count of specific methods.

Method	System Name	Unit	Description	Mapped	New method
transactions/create_single	transactions/create_single	hit		✓	
transactions/create_multiple	transactions/create_multiple	hit		✓	
transactions/confirm	transactions/confirm	hit		✓	
transactions/destroy	transactions/destroy	hit		Add a mapping rule	

Metrics

Hits are the built-in top-level metric and the parent metric of the methods. Other top level metrics can be added here if needed. A metric needs to be mapped to one or more URL patterns in the [Mapping Rules section](#) of the integration page so specific calls to your API up the count of specific metrics.

[Create new metric](#)

Metric	System Name	Unit	Description	Mapped	New metric
Hits	hits	hit	Number of API hits	✓	
Number of transactions	transactions	transaction		Add a mapping rule	

For more advanced information on metrics, methods, and rate limits see the specific [tutorial on rate limits](#).

9.4. STEP 3: INSTALL THE PLUGIN CODE TO YOUR APPLICATION

Armed with this information, return to the code and add the downloaded code bundle to your application. This step varies for each type of plugin, and the form that it takes depends on the way each language framework uses libraries. For the purposes of this tutorial, this example will proceed with Ruby plugin instructions. Other plugins integration details are included in the README documentation of each repository.

This library is distributed as a gem, for which Ruby 2.1 or JRuby 9.1.1.0 are minimum requirements:

```
gem install 3scale_client
```

Or alternatively, download the source code from GitHub.

If you are using Bundler, please add this to your Gemfile:

```
gem '3scale_client'
```

and do a bundle install.

If you are using Rails' config.gems, put this into your config/environment.rb

```
config.gem '3scale_client'
```

Otherwise, require the gem in whatever way is natural to your framework of choice.

Then, create an instance of the client:

```
client = ThreeScale::Client.new(service_tokens: true)
```



NOTE

unless you specify `service_tokens: true` you will be expected to specify a `provider_key` parameter, which is deprecated in favor of Service Tokens:

```
client = ThreeScale::Client.new(provider_key: 'your_provider_key')
```

This will communicate with the 3scale platform SaaS default server.

If you want to create a Client with a given host and port when connecting to an on-premise instance of the 3scale platform, you can specify them when creating the instance:

```
client = ThreeScale::Client.new(service_tokens: true, host:
'service_management_api.example.com', port: 80)
```

or

```
client = ThreeScale::Client.new(provider_key: 'your_provider_key', host:
'service_management_api.example.com', port: 80)
```

Because the object is stateless, you can create just one and store it globally. Then you can perform calls in the client:

```
client.authorize(service_token: 'token', service_id: '123', usage:
usage)
client.report(service_token: 'token', service_id: '123', usage: usage)
```

If you had configured a (deprecated) provider key, you would instead use:

```
client.authrep(service_id: '123', usage: usage)
```



NOTE

`service_id` is mandatory since November 2016, both when using service tokens and when using provider keys

**NOTE**

You might use the option `warn_deprecated: false` to avoid deprecation warnings. This is enabled by default.

SSL and Persistence

Starting with version 2.4.0 you can use two more options: **secure** and **persistent** like:

```
client = ThreeScale::Client.new(provider_key: '...', secure: true,
  persistent: true)
```

secure

Enabling `secure` will force all traffic going through HTTPS. Because establishing SSL/TLS for every call is expensive, there is `persistent`.

persistent

Enabling `persistent` will use HTTP Keep-Alive to keep open connection to our servers. This option requires installing gem `net-http-persistent`.

9.5. STEP 4: ADD CALLS TO AUTHORIZE AS API TRAFFIC ARRIVES

The plugin supports the 3 main calls to the 3scale backend:

- **authrep** grants access to your API and reports the traffic on it in one call.
- **authorize** grants access to your API.
- **report** reports traffic on your API.

9.5.1. Usage on authrep mode

You can make request to this backend operation using **service_token** and **service_id**, and an authentication pattern like **user_key**, or **app_id** with an optional key, like this:

```
response = client.authrep(service_token: 'token', service_id:
  'service_id', app_id: 'app_id', app_key: 'app_key')
```

Then call the **success?** method on the returned object to see if the authorization was successful.

```
if response.success?
  # All fine, the usage will be reported automatically. Proceed.
else
  # Something's wrong with this application.
end
```

The example is using the **app_id** authentication pattern, but you can also use other patterns such as **user_key**.

Example:

```

class ApplicationController < ActionController
  # Call the authenticate method on each request to the API
  before_filter :authenticate

  # You only need to instantiate a new Client once and store it as a
  global variable
  # If you used a provider key it is advisable to fetch it from the
  environment, as
  # it is secret.
  def create_client
    @@threescale_client ||= ThreeScale::Client.new(service_tokens: true)
  end

  # To record usage, create a new metric in your application plan. You
  will use the
  # "system name" that you specified on the metric/method to pass in as the
  key to the usage hash.
  # The key needs to be a symbol.
  # A way to pass the metric is to add a parameter that will pass the name
  of the metric/method along
  #
  # Note that you don't always want to retrieve the service token and
  service_id from
  # the parameters - this will depend on your application.
  def authenticate
    response = create_client.authrep(service_token:
params['service_token']
                                     service_id: params['service_id'],
                                     app_id: params['app_id'],
                                     app_key: params['app_key'],
                                     usage: { params['metric'].to_sym =>
1 })
    if response.success?
      return true
      # All fine, the usage will be reported automatically. Proceed.
    else
      # Something's wrong with this application.
      puts "#{response.error_message}"
      # raise error
    end
  end
end
end

```

9.6. STEP 5: DEPLOY AND RUN TRAFFIC

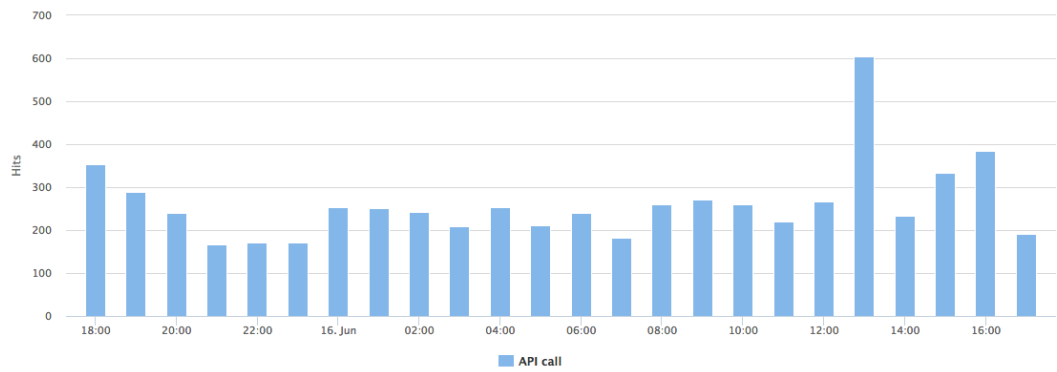
Once the required calls are added to your code, you can deploy (ideally to a staging/testing environment) and make API calls to your endpoints. As the traffic reaches the API, it will be filtered by the plugin and keys checked against issues API credentials. Refer to the [Getting Started guide](#) for how to generate valid keys – a set will have also been created as sample data in your 3scale account.

To see if traffic is flowing, log in to your API Admin Portal and navigate to the Analytics tab – there you will see traffic reported via the plugin.

Usage

Show last [24 hours](#) [7 days](#) [30 days](#) [12 months](#) from 06/15/2016 until 06/16/2016 per hour

6.3K API call ▼



Once the application is making calls to the API, they will become visible on the statistics dashboard and the Statistics > Usage page for more detailed view.

If you're receiving errors in the connection, you can also check the "Errors" menu item under Monitoring.