SCG for Kubernetes v1.1 Documentation

VMware Spring Cloud Gateway for Kubernetes 1.1

You can find the most up-to-date technical documentation on the VMware website at:

https://docs.vmware.com/

VMware, Inc.
3401 Hillview Ave.
Palo Alto, CA 94304
www.vmware.com

Copyright © 2023 VMware, Inc. All rights reserved. Copyright and trademark information.

Contents

Spring Cloud Gateway for Kubernetes 9
Key Features 9
For Operators 9
For Developers 9
Product Snapshot 10
Release Notes for Spring Cloud Gateway for Kubernetes 11
v1.1.15 11
Included in This Release 11
v1.1.14 11
Included in This Release 11
v1.1.13 11
Included in This Release 11
v1.1.12 11
Included in This Release 11
v1.1.10 11
Included in This Release 12
v1.1.9 12
Included in This Release 12
v1.1.8 12
Included in This Release 12
v1.1.7 12
Included in This Release 12
v1.1.6 12
Included in This Release 12
v1.1.5 12
Included in This Release 12
v1.1.4 12
Included in This Release 12
v1.1.3 13
Included in This Release 13
v1.1.2 13
Included in This Release 13
v1.1.1 13

Included in This Release 13
v1.1.0 13
Included in This Release 13

Operator Guide 15

Installation 15

Installing Spring Cloud Gateway for Kubernetes using the Tanzu CLI 15
Prerequisites 15
Creating the target namespace 15
Adding the image pull secret 15
Install the Spring Cloud Gateway for Kubernetes package repository 16
Install the Spring Cloud Gateway for Kubernetes package 16
Security Considerations 17
Installing the operator with multiple replicas 17
Update Spring Cloud Gateway for Kubernetes 17
Uninstall Steps 18

Installing Spring Cloud Gateway for Kubernetes using Helm 18
Prerequisites 18
Install or Upgrade Steps 18
Download and Extract Installation Artifacts 18
Relocate Images 19
Container Registry Secret 19
Complete the Installation 20
Installing the operator with multiple replicas 21
Security Considerations 21
Uninstall Steps 21
Installation in development environment 21

Installing Spring Cloud Gateway for Kubernetes in Tanzu Application Platform 22
Considerations 22

Troubleshooting Spring Cloud Gateway for Kubernetes 23
Known Issues 23
Check the status of Gateway 23
Get scg-operator and Gateway events 23
Configure Gateway's logging levels 24
Check scg-operator or Gateway logs 25

Resolve unresponsive scg-operator 25
Restart Gateway 25
Manually delete Custom Resource Definitions 25
Failing to pull images 26
Gateway failing to start with Vault integration enabled 26

Developer Guide 27

Getting Started with Spring Cloud Gateway for Kubernetes 27
Create Gateway Instance 27
Deploy Client App 28
Add API Routes to Gateway 29
Delete Gateway Instance 30

Service Instances 31

Configure Spring Cloud Gateway Instances 31
Configure Gateway Instances 31
Configure External Access 34
Using an Ingress Resource 34
TLS Passthrough 36
Gateway Actuator Management Port 36
Configure for High Availability 36
Configure TLS termination 37
Configure Environment Variables 38
Disable SecureHeaders Global Filter 38
Configure Cross-Origin Resource Sharing (CORS) 39
Configure Java Environment Options 41
Configure session expiration 42
Configuring Hardware Resources 42
Configuring Probes 42
Configure Observability 43
Exposing Metrics to Wavefront 43
Using the Spring Cloud Gateway for Kubernetes Dashboard for Wavefront 44
Exposing Metrics to Prometheus 45
Using the Spring Cloud Gateway for Kubernetes Dashboard for Grafana 46
Exposing Tracing to Wavefront 46
Applying custom labels to the Gateway Pods 47
Customizing the service type 47

Using Single Sign-On 48

- Configure Single Sign-On (SSO) 48
- Update Single Sign-On credentials 49
- OpenAPI security schemes (SSO) 50
- Logout 50

Configuring Single Sign-On for Sample Application 50

Configuring Okta OIDC provider 51
- Create authorization server for Animal Rescue 51
- Create users and groups 54
- Create new application 55
- Configuration summary 55

Configure Animal Rescue app 55
- Configure SSO params 55
- Configure routes security 56
- Deploy the app 56
- Test 56

OpenAPI Generated Documentation 57
- Accessing Generated OpenAPI v3 Documentation 57
- Configure OpenAPI Metadata 57
- PUT/POST/PATCH Request Body Schema 58
- Custom HTTP Responses 60

Configure Spring Cloud Gateway Instances in Tanzu Application Platform 60
- Adding Spring Cloud Gateway to a Component 61
- Adding Spring Cloud Gateway as a new Component 61

Client Apps 62

Configuring Gateway Routes 62
- What are API routes 63
- Define Route Config 63
- Default Configuration 64
- Define Service Level Config 64
  - Service Filters 64
  - Service Predicates 65
  - Service SSO Config 65
- Map Routes to Gateway 65
- Available Predicates 66

Available Filters 67
OpenApi Schema References 68
Commercial Route Filters 69
Filters Included In Spring Cloud Gateway OSS 69
Filters Added In Spring Cloud Gateway for Kubernetes 70
AddRequestHeadersIfNotPresent: Request headers modification filter 70
AllowedRequestCookieCount: Allowed request cookie count filter 70
AllowedRequestHeadersCount: Allowed request headers count filter 71
AllowedRequestQueryParamsCount: Allowed request query params count filter 71
BasicAuth: Basic authorization filter 71
BlockAccess: Global Filter to block access 73
CircuitBreaker: Reroute traffic on error response filter 74
Circuit breaker status 75
ClaimHeader: Passing JWT claims header filter 75
ClientCertificateHeader: Validate client certificate filter 76
FallbackHeaders: Allows adding CircuitBreaker exception details in the headers before forwarding 77
Cors: Configuring per-route Cross-Origin Resource Sharing (CORS) behavior 78
JwtKey: Multiple client JWT validation filter 78
ApiKey: API key validation filter 81
RateLimit: Limiting user requests filter 82
Limiting by IP Address 83
RemoveJsonAttributesResponseBody: Response body modification filter 84
RewriteAllResponseHeaders Response headers modification filter 85
RewriteResponseBody: Response body modification filter 85
RewriteJsonAttributesResponseBody: Response body JSON modification filter 86
Roles: Role-based access control filter 87
Scopes: Scope-based access control filter 87
StoreIpAddress: Store IP address filter 88
StoreHeader Store headers filter 88
SsoAutoAuthorize: SSO auto-authorized credentials filter 89
TokenRelay: Passing user identity filter 90
Commercial Route Predicates 90
Predicates Included In Spring Cloud Gateway OSS 91
Predicates Added In Spring Cloud Gateway for Kubernetes 91
Match on JWT claim value: JWTClaim Predicate 91
Custom Extensions 91

Developing Extensions 91
Prerequisites 91
Project setup 92
Gradle 92
Maven 93
Custom Extension Example 95
Custom Filter example code 95
Testing 96

Configuring Extensions 98
Prerequisites 98
Extension Deployment 98
Extensions from ConfigMaps 98
Extensions from OCI Image 100
Extensions from Persistent Volumes 100
Gateway Configuration 102
Validation and Troubleshooting 103
High-Availability deployments 103

OpenAPI Route Conversion 104
Conversion endpoint 104
Conversion request 105
Referencing an OpenAPI endpoint in your cluster 106
Providing Service level filters 106
Providing Route level filters 107
JSON schema to validate requests 107

Spring Cloud Gateway for Kubernetes

This topic provides an overview of VMware Spring Cloud® Gateway for Kubernetes v1.1.

Key Features

Spring Cloud Gateway for Kubernetes includes the following key features:

- Polyglot supported routability for application services written in any language that wish expose HTTP endpoints on Gateway instances
- Includes Kubernetes operator for handling API gateway custom resources applied to cluster and Kubernetes "native" experience
- Commercial container images to manage, create and dynamically update API routes on instances
- Dynamic application route configuration, enabling API route updates for continuous integration (CI) and continuous delivery (CD) pipelines
- Gateway-defined Single Sign-On (SSO) configuration combined with commercial SSO route filters
- Simplified OpenID Connect (OIDC) Single Sign-On (SSO) configuration for each API gateway instance
- Commercial API route filters for SSO authentication, role-based access control, scopes authorization, authorized token relay, client certificate authorization, rate limiting and circuit breaker
- High availability configuration for setting count, memory, and vCPU of API gateway instances
- Access to configure JVM performance optimizations for API gateway instance specific use cases
- Local development and testing enabled to validate API route configurations before promoting to environments on way to production

For Operators

For information about installing and managing Spring Cloud Gateway for Kubernetes, see the Operator Guide.

For Developers

For information about creating and managing Gateway instances and connecting them to client apps, see the Developer Guide.

Product Snapshot

The following table provides version and version-support information about Spring Cloud Gateway for Kubernetes.

Element	Details
1.1.15 Release Date	November 14, 2022
Spring Cloud OSS Version	2021.0.3
Spring Boot OSS Version	2.7.5
Supported IaaS	Kubernetes 1.19 - 1.23

Release Notes for Spring Cloud Gateway for Kubernetes

These are release notes for Spring Cloud Gateway for Kubernetes.

v1.1.15

Release Date: November 14, 2022

Included in This Release

- Resolved security vulnerability

v1.1.14

Release Date: November 2, 2022

Included in This Release

- Resolved security vulnerability
- Fix user flow when calling logout endpoint

v1.1.13

Release Date: October 25, 2022

Included in This Release

- Resolved following security vulnerabilities: CVE-2022-42003, CVE-2022-42004 and CVE-2022-31684

v1.1.12

Release Date: September 26, 2022

Included in This Release

- Resolved following security vulnerabilities: CVE-2022-2526, CVE-2022-25857 and CVE-2022-27664

v1.1.10

Release Date: September 12, 2022

Included in This Release

- Resolved following security vulnerabilities: CVE-2022-2526 and CVE-2022-25857

v1.1.9

Release Date: August 25, 2022

Included in This Release

- Resolved following security vulnerabilities: CVE-2022-37434

v1.1.8

Release Date: August 10, 2022

Included in This Release

- Resolved following security vulnerabilities: CVE-2021-4209 and CVE-2022-2509

v1.1.7

Release Date: July 13, 2022

Included in This Release

- Resolved following security vulnerabilities: CVE-2022-34903

v1.1.6

Release Date: July 1, 2022

Included in This Release

- Resolved following security vulnerabilities: CVE-2022-2068

v1.1.5

Release Date: June 8, 2022

Included in This Release

- Resolved following security vulnerabilities: https://nvd.nist.gov/vuln/detail/CVE-2022-1304

v1.1.4

Release Date: June 6, 2022

Included in This Release

- Resolved following security vulnerabilities: USN-5446-1
- Improved OpenAPI conversion service error information to help with troubleshooting
- Resolved issue with OpenAPI conversion of date time format in some circumstances

v1.1.3

Release Date: May 25, 2022

Included in This Release

- Resolved the following CVE: CVE-2022-22970
- Fixed issues with OpenAPI auto-generation when particular attributes are present in specification provided

v1.1.2

Release Date: May 23, 2022

Included in This Release

- Improved notification events and logging when API route is not registered due to invalid filter configuration
- Fixed issue with image pull secret not getting updated during installation when changed
- Resolved the following CVE with base image patch: CVE-2019-20838, CVE-2020-14155

v1.1.1

Release Date: May 16, 2022

Included in This Release

- Resolved the following CVE with base image patch: CVE-2022-1292, CVE-2022-1343, CVE-2022-1434, CVE-2022-1473

v1.1.0

Release Date: April 27, 2022

Included in This Release

- Added OpenAPI conversion service which can generate `SpringCloudGatewayRouteConfig` custom resources from OpenAPI specifications
- Added Carvel support with packaging and installation repository
- Added ability to configure custom API gateway instance values for installation
- Added ability to install via `tanzu` CLI including into a specified namespace

- Added custom annotation support for API gateway instance pods
- Added support for using LoadBalancer and NodePort as additional ingress options
- Added option to load custom extensions from init container in addition to ConfigMap and Persistent Volume
- Added ApiKey global filter to validate API usage by client request using `X-API-Key` header against HashiCorp Vault stored API keys
- Added BlockAccess global filter that can be configured to block API traffic based on IP address or JWT claim
- Added request filter for adding header if not present
- Added request filters for constraining cookie, header and request header counts
- Upgraded to Java 17 in container images
- Using Spring Cloud OSS version `2021.0.1`

Operator Guide

These topics describe how to install and troubleshoot Spring Cloud Gateway for Kubernetes, as well as how to configure single sign-on.

Installation

These topics describe how to install Spring Cloud Gateway for Kubernetes.

Installing Spring Cloud Gateway for Kubernetes using the Tanzu CLI

This page will give an overview of the installation process for Spring Cloud Gateway for Kubernetes using the Tanzu cli.

Prerequisites

Before beginning the installation or upgrade process, ensure that you have installed the following tools on your local machine:

- The `tanzu` command-line interface (CLI) tool. For information about installing this tool, see the Tanzu Kubernetes Grid documentation.

You will also need sufficiently recent versions of the Carvel controllers running on your Kubernetes cluster:

- `kapp-controller` version >= `0.24.0`
- `secretgen-controller` version >= `0.5.0`

Creating the target namespace

First, create the destination namespace for the Spring Cloud Gateway for Kubernetes installation. For package repository the namespace is `tap-install`, image pull secret and actual SCG can be customized via CLI.

```
kubectl create namespace tap-install
```

Adding the image pull secret

For the `tanzu` cli to install the Spring Cloud Gateway images, it requires the credentials for the Tanzu image registry, which is hosted on the VMware Tanzu Network. To create this secret, run:

```
tanzu secret registry add tap-registry \
    --namespace tap-install \
    --server "registry.tanzu.vmware.com" \
    --username "{registry username}" \
    --password "{registry password}" \
    --export-to-all-namespaces
```

Replace `{registry username}` and `{registry password}` with your Tanzu Network credentials.

The `--export-to-all-namespaces` option instructs the `secretgen-controller` to make this image pull secret available to managed packages in any namespace. This is to support pulling the image when Spring Cloud Gateway instances are created in arbitrary namespaces.

You can check that this step has been successful with the following command:

```
tanzu secret registry list --namespace tap-install
```

You should see output similar to the following:

```
NAME          REGISTRY                      EXPORTED           AGE
tap-registry  dev.registry.tanzu.vmware.com  to all namespaces  6s
```

The `EXPORTED` column should show `to all namespaces`.

Install the Spring Cloud Gateway for Kubernetes package repository

Next, install the Spring Cloud Gateway for Kubernetes package repository:

```
tanzu package repository add scg-package-repository \
    --namespace tap-install \
    --url registry.tanzu.vmware.com/spring-cloud-gateway-for-kubernetes/scg-package-re
pository:{version}
```

where `{version}` is the version you wish to install, e.g. `1.1.0`.

Once the repository is successfully installed, the `tanzu` CLI should respond with:

```
Added package repository 'scg-package-repository' in namespace 'tap-install'
```

You can then check the packages available for installation with:

```
tanzu package available list --namespace tap-install
```

The list of available packages should now contain Spring Cloud Gateway:

```
NAME                                   DISPLAY-NAME          SHORT-DESCRIPTION     LA
TEST-VERSION
...
spring-cloud-gateway.tanzu.vmware.com  Spring Cloud Gateway  Spring Cloud Gateway  {v
ersion}
```

Install the Spring Cloud Gateway for Kubernetes package

You are now ready to install Spring Cloud Gateway for Kubernetes.

```
tanzu package install spring-cloud-gateway \
    --namespace tap-install \
    --package-name spring-cloud-gateway.tanzu.vmware.com \
    --version {version}
```

Once successful, the `tanzu` CLI will report `Added installed package 'spring-cloud-gateway'`.

Security Considerations

As described above in the image pull secret installation step, the `--export-to-all-namespaces` option to the `tanzu` CLI instructs the `secretgen-controller` to make the image pull secret available to packages in any namespace.

Additionally, a `ClusterRole` named `scg-operator-resources-role` is created with permissions to manage specific Spring Cloud Gateway resources deployed in any namespace in the cluster. To see the specific resources and permissions managed by the cluster role, run:

```
kubectl describe ClusterRole scg-operator-resources-role
```

Installing the operator with multiple replicas

The Spring Cloud Gateway Operator defaults to a single replica. This should be suitable for most environments as the operator is resilient to downtime as it's data is stored in the Kubernetes clusters `Etcd` data store. Customers can opt to configure multiple replicas of the operator using the flag `--replica_count` with the installation script. Increasing the number of replicas will enable leadership election between the operator Pods. The leadership election mechanism is built into Kubernetes and is described in this blog post from the Kubernetes team

To enable multiple operator replicas with leadership election, install the product as follows;

Create a file containing the configuration for multiple replicas

```
scgOperator:
  replicaCount: 2
```

Then install the product using the `tanzu` CLI

```
tanzu package install spring-cloud-gateway \
    --namespace tap-install \
    --package-name spring-cloud-gateway.tanzu.vmware.com \
    --values-file config-with-multiple-replicas.yaml \
    --version {version}
```

Update Spring Cloud Gateway for Kubernetes

To update Spring Cloud Gateway, you just need to first update the package repository with the new version:

```
tanzu package repository update scg-package-repository \
```

```
    --url registry.tanzu.vmware.com/spring-cloud-gateway-for-kubernetes/scg-package-re
pository:{version} \
    --namespace tap-install
```

And once this is done you can update the Spring Cloud Gateway for Kubernetes installed package using:

```
tanzu package installed update spring-cloud-gateway --namespace tap-install --version
{version}
```

Uninstall Steps

To uninstall Spring Cloud Gateway, run:

```
tanzu package installed delete spring-cloud-gateway --namespace tap-install
```

Once this is done you can remove the Spring Cloud Gateway package repository using:

```
tanzu package repository delete scg-package-repository --namespace tap-install
```

Installing Spring Cloud Gateway for Kubernetes using Helm

This page will give an overview of the installation process for Spring Cloud Gateway for Kubernetes management components using a Helm chart.

Prerequisites

Before beginning the installation or upgrade process, ensure that you have installed the following tools on your local machine:

- The Docker command-line interface (CLI) tool, `docker`. For information about installing the `docker` CLI tool, see the Docker documentation.
- The Helm command-line interface (CLI) tool, `helm`. For information about installing the `helm` CLI tool, see the Helm documentation.

Install or Upgrade Steps

There are two options to install or upgrade Spring Cloud Gateway for Kubernetes.

- The simplest, using the provided scripts to relocate the SCG images and then install the components.
- The advanced installation, manually setting the image paths and other options. This is useful when images are already deployed in a trusted container registry and allow skipping the relocate step.

Download and Extract Installation Artifacts

Spring Cloud Gateway for Kubernetes is provided as a compressed archive file containing a series of utility scripts, manifests, and required images.

To download the components:

1. Visit VMware Tanzu Network and log in.
2. Navigate to the Spring Cloud Gateway for Kubernetes product listing.
3. In the **Releases** list, select the version that you wish to install or upgrade to.
4. Download "Spring Cloud Gateway for Kubernetes Installer".
5. Extract the contents of the archive file:

   ```
   $ tar zxf spring-cloud-gateway-k8s-[VERSION].tgz
   ```

   The extracted directory contains the following directory layout:

   ```
   $ ls spring-cloud-gateway-k8s-[VERSION]

   dashboards/      helm/      images/    scripts/
   ```

Relocate Images

Next, relocate the Spring Cloud Gateway for Kubernetes images to your private image registry. The images must be loaded into the local Docker daemon and pushed into the registry.

To relocate the images:

1. Use the `docker` CLI tool or your cloud provider CLI to authenticate to your image registry.
2. Run the image relocation script, located in the `scripts` directory.

   ```
   $ ./scripts/relocate-images.sh <REGISTRY_URL>
   ```

   In this example command, replace the `<REGISTRY_URL>` placeholder with the URL for your image registry. For example:

   ```
   $ ./scripts/relocate-images.sh myregistry.example.com/spring-cloud-gateway
   ```

   The script will load the two Spring Cloud Gateway for Kubernetes images and push them into the image registry. This script will also generate a file named `helm/scg-image-values.yaml`. The contents of this file will resemble the following:

   ```
   scg-operator:
     image: "myregistry.example.com/spring-cloud-gateway/scg-operator:v[VERSION]"
   gateway:
     image: "myregistry.example.com/spring-cloud-gateway/gateway:v[VERSION]"
   ```

Container Registry Secret

If your cluster needs authentication to access the relocated images, then an image pull secret name (with default name `spring-cloud-gateway-image-pull-secret`) must be provided in the operator namespace before running the installation:

```
$ kubectl create secret docker-registry spring-cloud-gateway-image-pull-secret -n ${in
stallation_namespace} \
```

```
--docker-server=${registry} \
--docker-username=${username} \
--docker-password=${password}

secret/spring-cloud-gateway-image-pull-secret created
```

If it fails to create the secret because the namespace was not found, create the namespace first. For example:

```
error: failed to create secret namespaces "spring-cloud-gateway" not found

$ kubectl create ns spring-cloud-gateway
namespace/spring-cloud-gateway created
```

If your secret name is different than `spring-cloud-gateway-image-pull-secret`, ensure to edit `helm/scg-image-values.yaml` with your secret name as follows:

```
scg-operator:
  image: "myregistry.example.com/spring-cloud-gateway/scg-operator:v[VERSION]"
  registryCredentialsSecret: my-image-pull-secret
gateway:
  image: "myregistry.example.com/spring-cloud-gateway/gateway:v[VERSION]"
```

Complete the Installation

You are now ready to install Spring Cloud Gateway for Kubernetes.

If you used the `relocate-images.sh` script from the previous section, you can simply use the script `./scripts/install-spring-cloud-gateway.sh`. By default, the Spring Cloud Gateway for Kubernetes operator and backing applications will be deployed in the `spring-cloud-gateway` namespace.

If you already have images in a known registry, or need to customize other aspects, you can change the installation defaults using the additional options. For example, you can install in another namespace

```
$ ./scripts/install-spring-cloud-gateway.sh --namespace my_namespace_name
```

Set an image pull secret

```
$ ./scripts/install-spring-cloud-gateway.sh --registry_credentials_secret my_image_sec
ret
```

Or, skip the relocation script and define the images paths directly

```
$ ./scripts/install-spring-cloud-gateway.sh --operator_image myregistry.org/scg-operat
or:1.0.1 --gateway_image myregistry.org/gateway:1.0.0
```

Use `--help` to display the details for all available options.

Regardless of the installation method, after running the script, you should see a new deployment named `scg-operator` in your chosen namespace.

```
$ kubectl get all -n ${installation_namespace}
```

```
NAME                                  READY   STATUS    RESTARTS   AGE
pod/scg-operator-7c6b749b9-6llt8      1/1     Running   0          72s

NAME                    TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)   AGE
service/scg-operator    ClusterIP   10.96.38.53    <none>        80/TCP    72s

NAME                              READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/scg-operator      1/1     1            1           72s

NAME                                        DESIRED   CURRENT   READY   AGE
replicaset.apps/scg-operator-7c6b749b9      1         1         1       72s
```

Installing the operator with multiple replicas

The Spring Cloud Gateway Operator defaults to a single replica. This should be suitable for most environments as the operator is resilient to downtime as its data is stored in the Kubernetes clusters `etcd` data store. Customers can opt to configure multiple replicas of the operator using the flag `--replica_count` with the installation script. Increasing the number of replicas will enable leadership election between the operator Pods. The leadership election mechanism is built into Kubernetes and is described in this blog post from the Kubernetes team.

To enable multiple operator replicas with leadership election, install the product as follows;

```
$ ./scripts/install-spring-cloud-gateway.sh --replica_count 2
```

Security Considerations

In order to allow users to create Spring Cloud Gateways in different namespaces, `scg-operator` does the following. If your cluster uses a secret used to authenticate to your registry and pull the Gateway image from it, `spring-cloud-gateway-image-pull-secret` is copied to every new namespace where a Gateway is created. Additionally, a `ClusterRole` named `scg-operator-resources-role` is created with permissions to manage specific Spring Cloud Gateway resources deployed in any namespace in the cluster. To see the specific resources and permissions managed by the cluster role, run

```
$ kubectl describe ClusterRole scg-operator-resources-role
```

Uninstall Steps

To uninstall Spring Cloud Gateway and all its managed components, run

```
$ helm uninstall spring-cloud-gateway -n ${installation_namespace}
$ kubectl delete namespace ${installation_namespace}
```

Installation in development environment

Spring Cloud Gateway for Kubernetes can be installed in a development cluster such as KinD. For that, create a file called `kind-config.yaml`, with the following YAML definition:

```
kind: Cluster
```

```
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
  image: kindest/node:v1.21.10@sha256:84709f09756ba4f863769bdcabe5edafc2ada72d3c8c44d6
515fc581b66b029c
  kubeadmConfigPatches:
    - |
      kind: InitConfiguration
      nodeRegistration:
        kubeletExtraArgs:
          node-labels: "ingress-ready=true"
  extraPortMappings:
    - containerPort: 80
      hostPort: 80
      protocol: TCP
    - containerPort: 443
      hostPort: 443
      protocol: TCP
```

Then create the KinD cluster with the following command:

```
$ kind create cluster --config kind-config.yaml
```

And you should see an output similar to:

```
Creating cluster "kind" ...
 ✓ Ensuring node image (kindest/node:v1.21.10)
 ✓ Preparing nodes
 ✓ Writing configuration
 ✓ Starting control-plane 
 ✓ Installing CNI
 ✓ Installing StorageClass
Set kubectl context to "kind-kind"
You can now use your cluster with:

kubectl cluster-info --context kind-kind

Thanks for using kind!
```

Note that you still need to use an external registry to relocate the images. If you prefer to load the images to KinD directly, replace the line from `relocate-images.sh`

```
docker push "$destination_image"
```

by

```
kind load docker-image "$destination_image"
```

Installing Spring Cloud Gateway for Kubernetes in Tanzu Application Platform

Spring Cloud Gateway for Kubernetes can be installed in Tanzu Application Platform using the Tanzu CLI. For more details, refer to Installing using the Tanzu CLI.

Considerations

When beginning the installation the destination namespace for the Spring Cloud Gateway for Kubernetes installation should be `tap-install`.

Troubleshooting Spring Cloud Gateway for Kubernetes

This topic describes how to troubleshoot Spring Cloud Gateway for Kubernetes operator and instances.

Known Issues

- In `SpringCloudGatewayMapping` object, the `gatewayRef` field cannot be modified once created. In order to move routes from an old Gateway to a new Gateway, delete the old mapping object, change the `gatewayRef` in yaml to the new Gateway, and apply the new yaml.
- In Google Kubernetes Engine (GKE), due to missing Kubernetes Event API, the `scg-operator` will throw `ApiException` and won't log any events

Check the status of Gateway

You can check the current status of your gateway by running

```
$ kubectl get scg my-gateway
NAME                 READY   REASON
my-gateway           True    Created
```

Get `scg-operator` and Gateway events

In case of errors events are published for the Operator and the Gateway components (mappings and routes as well), you can display them using the `describe` option.

```
$ kubectl describe scg my-gateway
$ kubectl describe scgm my-gateway-mapping
$ kubectl describe scgrc my-gateway-route-config
```

For example, in case some routes are not present, using `kubectl describe scgm` may show the referenced gateway is not present. Creating such Gateway instance would fix the issue.

```
Events:
  Type     Reason    Age        From                           Message
  ----     ------    ----       ----                           -------
  Warning  NotFound  <unknown>  SpringCloudGatewayController  Specified SpringCloudGat
eway resource "demo-gateway" is not found / not ready
```

Another useful event to look for when troubleshooting is `RouteUpdateException`. This event is triggered when a Route is not valid. For example, when the filter name is wrong:

```
apiVersion: "tanzu.vmware.com/v1"
```

```
kind: SpringCloudGatewayRouteConfig
metadata:
  name: test-gateway-routes
spec:
  routes:
    - uri: https://example.com
      predicates:
        - Path=/test/**
      filters:
        - InvalidFilter=1
      title: "my-test-route"
```

If we get the events in the namespace, we can see how everything succeeded but we have now an extra `Warning` with the `RouteUpdateException` explaining that the route `my-test-route` is wrong.

```
$ kubectl get events --watch
LAST SEEN   TYPE      REASON                    OBJECT                               MESSAG
E
11s         Normal    SuccessfulCreate          statefulset/my-gateway               create
 Pod my-gateway-0 in StatefulSet my-gateway successful
0s          Normal    RoutesUpToDate            pod/my-gateway-0                     Pod "m
y-gateway-0-1/my-gateway-0" is RoutesUpToDate with all routes
0s          Normal    Created                   springcloudgateway/my-gateway        Spring
CloudGateway resource my-gateway is Created
0s          Warning   RouteUpdateException      /my-gateway-0                        Failed
 to update route with title 'my-test-route' and uri 'https://example.com' due to: 'Pod
 update failed, request to http://10.244.1.4:8090/actuator/gateway/routes/my-gateway-0
-1-mapping-0 failed. Response code 400, message Bad Request'
0s          Normal    Created                   springcloudgatewaymapping/mapping    Routes
 specified in SpringCloudGatewayRouteConfig "initial-route-config" is Created on pod "
my-gateway-0-1/my-gateway-0"
```

Configure Gateway's logging levels

The following loggers may contain valuable troubleshooting information at the `DEBUG` and `TRACE` levels:

```
io.pivotal.spring.cloud.gateway              # filters and predicates including custom
 extensions
org.springframework.cloud.gateway            # API gateway
org.springframework.http.server.reactive.    # HTTP server interactions
org.springframework.web.reactive             # API gateway reactive flows
org.springframework.boot.autoconfigure.web.  # API gateway autoconfiguration
org.springframework.security.web             # Authentication & Authorization informat
ion
reactor.netty                                # Reactor Netty
```

You can set a specific logger's logging level for a gateway statefulset by running the following command, which will automatically update the underlying pod.

```
$ kubectl set env statefulset.apps/my-gateway logging_level_org_springframework_cloud_
gateway=TRACE
```

You can also configure a gateway instance with specific logging levels using the `spec.env` property:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
  name: gateway-demo
spec:
  ...
  env:
    - name: logging.level.org.springframework.web
      value: debug
```

Check `scg-operator` or Gateway logs

You can access the `scg-operator` or your Gateway logs by running

```
$ kubectl logs deployment.apps/scg-operator -n spring-cloud-gateway
$ kubectl logs statefulset.apps/my-gateway
```

Resolve unresponsive `scg-operator`

If you find that `scg-operator` won't fully start or respond, and none of the above techniques help point out the problem, you could try increasing the requested CPU resource by 100m by running

```
$ kubectl edit deployment.apps/scg-operator
```

Alternatively, you can restart the deployment by running

```
$ kubectl rollout restart deployment.apps/scg-operator
```

Restart Gateway

In case of some errors, a restart might help solve the issue. You can restart your Gateway by running

```
$ kubectl rollout restart statefulset.apps/my-gateway
```

Manually delete Custom Resource Definitions

If there are problems while uninstalling, sometimes the Custom Resource Definitions don't get deleted. After uninstalling, you can check if there are any of the Spring Cloud Gateway Custom Resource Definitions by running

```
$ kubectl get crds
NAME                                                CREATED AT
springcloudgatewaymappings.tanzu.vmware.com         2021-02-17T11:52:09Z
springcloudgatewayrouteconfigs.tanzu.vmware.com     2021-02-17T11:28:12Z
springcloudgateways.tanzu.vmware.com                2021-02-17T11:28:12Z
```

If any of these three appear, you can manually delete them by running

```
$ kubectl delete crd springcloudgatewaymappings.tanzu.vmware.com
$ kubectl delete crd springcloudgatewayrouteconfigs.tanzu.vmware.com
```

```
$ kubectl delete crd springcloudgateways.tanzu.vmware.com
```

Failing to pull images

When running the installation script `./scripts/install-spring-cloud-gateway.sh` and you see errors pulling an image:

```
Events from from installation namespace:
LAST SEEN   TYPE      REASON              OBJECT                                 MESSAGE
2m          Normal    Scheduled           pod/scg-operator-7c6b749b9-hbrkx    Successf
ully assigned spring-cloud-gateway/scg-operator-7c6b749b9-hbrkx to kind-control-plane
36s         Normal    Pulling             pod/scg-operator-7c6b749b9-hbrkx    Pulling
image "my.registry/scg-operator:1.0.1"
2m          Normal    SuccessfulCreate    replicaset/scg-operator-7c6b749b9   Created
pod: scg-operator-7c6b749b9-hbrkx
2m          Normal    ScalingReplicaSet   deployment/scg-operator             Scaled u
p replica set scg-operator-7c6b749b9 to 1
36s         Warning   Failed              pod/scg-operator-7c6b749b9-hbrkx    Failed t
o pull image "my.registry/scg-operator:1.0.1": rpc error: code = Unknown desc = failed
 to pull and unpack image "my.registry/scg-operator:1.0.1": failed to resolve referenc
e "my.registry/scg-operator:1.0.1": unexpected status code [manifests 1.0.1]: 401 Unau
thorized
36s         Warning   Failed              pod/scg-operator-7c6b749b9-hbrkx    Error: E
rrImagePull
12s         Normal    BackOff             pod/scg-operator-7c6b749b9-hbrkx    Back-off
 pulling image "my.registry/scg-operator:1.0.1"
12s         Warning   Failed              pod/scg-operator-7c6b749b9-hbrkx    Error: I
magePullBackOff
Error installing Spring Cloud Gateway operator
```

Check to make sure you created an image pull secret (with default name `spring-cloud-gateway-image-pull-secret`) to your registry. See the Installation page for a step-by-step guide.

Gateway failing to start with Vault integration enabled

In case of a gateway pod that is blocked in `Init` state after enabling the API Key filter or the JWT Key filter in the Spring Cloud Gateway configuration, please check that the Service Account name you used when setting up Vault role matches the one you specified in the SCG configuration, and that it's running in the same namespace as the Gateway.

Developer Guide

These topics describe how to use Spring Cloud Gateway for Kubernetes.

Getting Started with Spring Cloud Gateway for Kubernetes

This topic describes how to quickly get started using Spring Cloud Gateway for Kubernetes to provide an API gateway for a microservice architecture.

Tip: This topic uses sample apps from the spring-cloud-services-samples / animal-rescue repository on GitHub. To follow along, clone the repository and check instructions in `README.md`.

This will give an overview of managing route configurations for applications providing Application Programming Interfaces (API) via a Gateway instance. This overview assumes that Spring Cloud Gateway for Kubernetes management components have already been installed.

The components are

- Gateway Instances - Represent each one of the Spring Cloud Gateways deployed
- Route Configurations - Is a set of routes that can be applied to one or many gateways
- Mappings - A mapping defines which route configurations go with which gateways

Create Gateway Instance

To create a Spring Cloud Gateway for Kubernetes instance, create a file called `gateway-config.yaml`, with the following YAML definition:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
  name: my-gateway
```

Next, apply this definition to your Kubernetes cluster:

```
$ kubectl apply -f gateway-config.yaml
```

This configuration will create a new Gateway instance. By default, the instance will be created alongside a ClusterIP service in the current namespace. To check the status of it, you can use the Kubernetes `get` command.

```
$ kubectl get scg my-gateway

NAME                READY   REASON
my-gateway          True    Created
```

To add routes and to map the routes to the gateway, we need to create a `SpringCloudGatewayRouteConfig` object describing the routes and a `SpringCloudGatewayMapping` object that maps the route config to the gateway.

Create a file called `route-config.yaml` with the following YAML definition:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: my-gateway-routes
spec:
  routes:
  - uri: https://github.com
    predicates:
      - Path=/github/**
    filters:
      - StripPrefix=1
```

Create a file called `mapping.yaml` with the following YAML definition:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayMapping
metadata:
  name: test-gateway-mapping
spec:
  gatewayRef:
    name: my-gateway
  routeConfigRef:
    name: my-gateway-routes
```

Apply both the definitions to your Kubernetes cluster.

The instance will include one route (`test-route`) that uses a `Path` predicate to define the path within the gateway, and the `StripPrefix` filter to remove the path before redirecting.

To validate that the gateway is functioning locally you can port-forward the ClusterIP service.

```
$ kubectl -n=spring-cloud-gateway port-forward service/my-gateway 8080:80
```

You should now be able to access the Gateway from localhost:8080/github.

For information about enabling external access to your Gateway instance, see Configure External Access.

Deploy Client App

In this section we will describe a sample scenario using the Animal Rescue backend API sample application. The following YAML describes the backend application deployment as a service on Kubernetes. For the sake of example we will assume that the target namespace is `animal-rescue` on the Kubernetes cluster.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: animal-rescue-backend
```

```
spec:
  selector:
    matchLabels:
      app: animal-rescue-backend
  template:
    metadata:
      labels:
        app: animal-rescue-backend
    spec:
      containers:
        - name: animal-rescue-backend
          image: springcloudservices/animal-rescue-backend
          env:
            - name: spring.profiles.active
              value: k8s
          resources:
            requests:
              memory: "256Mi"
              cpu: "100m"
            limits:
              memory: "512Mi"
              cpu: "500m"
---
apiVersion: v1
kind: Service
metadata:
  name: animal-rescue-backend
spec:
  ports:
    - port: 80
      targetPort: 8080
  selector:
    app: animal-rescue-backend
```

This assumes there is an image available in your container image repository named `springcloudservices/animal-rescue-backend`. To deploy the application, save the YAML into a file named `animal-rescue-backend.yaml` and run the following command.

```
$ kubectl apply -f animal-rescue-backend.yaml --namespace animal-rescue
```

Add API Routes to Gateway

Now that the Animal Rescue backend application is running as a service named `animal-rescue-backend` you can describe the route configuration to be applied to `my-gateway`.

Create a file called `animal-rescue-backend-route-config.yaml` with the following definition:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: animal-rescue-backend-route-config
spec:
  service:
    name: animal-rescue-backend
  routes:
    - predicates:
```

```
        - Path=/api/**
      filters:
        - StripPrefix=1
```

Create another file called `animal-rescue-backend-mapping.yaml` with the following definition:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayMapping
metadata:
  name: animal-rescue-backend-mapping
spec:
  gatewayRef:
    name: my-gateway
  routeConfigRef:
    name: animal-rescue-backend-route-config
```

The `SpringCloudGatewayMapping` and `SpringCloudGatewayRouteConfig` object kinds are processed by the Spring Cloud Gateway for Kubernetes management components to update the desired Gateway instance provided in the `spec.gatewayRef` property value. The application to route traffic for the configured `routes` is supplied in the `spec.service` property value.

Apply both definitions to your Kubernetes cluster.

```
$ kubectl apply -f animal-rescue-backend-route-config.yaml
$ kubectl apply -f animal-rescue-backend-mapping.yaml
```

Assuming that `my-gateway` had an ingress applied already for FQDN of `my-gateway.my-example-domain.com`, the Animal Rescue backend API will be available under the path `my-gateway.my-example-domain.com/api/...`. One of the endpoints available in the sample application is `GET /api/animals` which lists all of the animals available for adoption requests. This endpoint should now be accessible using the following command.

```
# Using https://httpie.io/
$ http my-gateway.my-example-domain.com/api/animals

# Using curl
$ curl my-gateway.my-example-domain.com/api/animals
```

If you are not using an ingress, you can port forward the gateway:

```
$ kubectl port-forward service/my-gateway 8080:80
```

And with another terminal window, call the `/api/animals` endpoint:

```
# Using https://httpie.io/
$ http localhost:8080/api/animals

# Using curl
$ curl localhost:8080/api/animals
```

For more information about adding API routes for an app to a Gateway instance, see Add Routes to Gateway.

Delete Gateway Instance

Gateway instances can be easily deleted using the Kubernetes cli `delete` command.

```
$ kubectl delete scg my-gateway
```

After that, if you list the existing Gateways with `kubectl get scg` you'll notice it's no longer running.

Note: Deleting a Gateway does not delete related Route Configuration or Mappings. For that, you can use `kubectl delete scgrc <routeconfig-name>` or `kubectl delete scgm <mapping-name>`.

Service Instances

These topics describe how to create and manage Spring Cloud Gateway for Kubernetes service instances.

Configure Spring Cloud Gateway Instances

This topic describes how to configure and update a Spring Cloud Gateway for Kubernetes instance.

Configure Gateway Instances

To create a Gateway instance, you must create a resource of type `SpringCloudGateway`. The definition for `SpringCloudGateway` specifies:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
  name:             # Name given to this Gateway instance (required)
  labels:
    my-custom-label: hello # Labels defined in the Gateway resource will also be appli
ed to the Gateway Pods for simplified management
  annotations:
    my-custom-annotation: my-value # Annotations defined on the Gateway resource will
also be applied to the Gateway Pods for simplified management
spec:
  count:            # Number of container instances (pods) to scale Gateway for high av
ailability (HA) configuration
  tls:              # Set a list of TLS-enabled hosts
    - hosts:        # Array of hostnames for which to perform TLS termination using the
 specified certificate
      secretName:   # Name of TLS secret to load certificate and key from
  sso:
    secret:         # Secret name to be used for SSO configuration
    roles-attribute-name:
                    # Roles attribute name used to extract user roles for Roles filter
(default: 'roles')
    inactive-session-expiration-in-minutes:
                   # Time to life of inactive sessions in minutes, 0 means sessions won
't expire.
  observability:
```

```
    metrics:
      wavefront:
        enabled:  # If wavefront metrics should be pushed
      prometheus:
        enabled:  # If a prometheus endpoint should be exposed
        annotations:
          enabled: # If scrapping annotations should be included in the Pod
    tracing:
      wavefront:
        enabled: # If wavefront traces should be pushed
    wavefront:
      secret:        # Secret name to be used for wavefront configuration
      source:        # The wavefront source (default: Gateway Pod name, `gateway-0`).
      application:   # The wavefront application (default: Gateway Namespace `namespac
e`).
      service:       # The wavefront service (default: Gateway name `my-gateway`).
  api:
    groupId:       # Unique identifier for the group of APIs available on the Gateway
instance (default: normalized title of the Gateway instance)
    title:         # Title describing the context of the APIs available on the Gateway
 instance (default: name of the Gateway instance)
    description:   # Detailed description of the APIs available on the Gateway instanc
e (default: `Generated OpenAPI 3 document that describes the API routes configured for
 '[Gateway instance name]' Spring Cloud Gateway instance deployed under '[namespace]'
namespace.`)
    documentation: # Location of additional documentation for the APIs available on th
e Gateway instance
    version:       # Version of APIs available on this Gateway instance (default: `uns
pecified`)
    serverUrl:     # Base URL that API consumers will use to access APIs on the Gatewa
y instance
    cors:
      allowedOrigins:          # Allowed origins to make cross-site requests, applied
globally
      allowedOriginPatterns:   # Allowed origin patterns to make cross-site requests,
applied globally
      allowedMethods:          # Allowed HTTP methods on cross-site requests, applied
globally
      allowedHeaders:          # Allowed headers in cross-site request, applied global
ly
      maxAge:                  # How long, in seconds, the response from a pre-flight
request can be cached by clients, applied globally
      allowCredentials:        # Whether user credentials are supported on cross-site
requests, applied globally
      exposedHeaders:          # HTTP response headers to expose for cross-site reques
ts, applied globally
      perRoute:                # A map of URL Patterns to Spring Framework CorsConfigu
ration, to configure CORS per route.

  java-opts:       # JRE parameters for Gateway instance to enhance performance

  env:             # Set a list of [configuration](https://cloud.spring.io/spring-clou
d-gateway/reference/html/appendix.html#common-application-properties) environment vari
ables to configure this Gateway instance
    - name:          # Name of the environment variable
      value:         # Value of environment variable

  extensions:                 # Additional configurations for global features (e.g. cu
stom filters, Api Key,...)
```

```
    custom:                    # Array of custom extensions to load (name must match th
e ConfigMap name).
    secretsProviders:          # Array of secret providers. These are identified by a n
ame and follow conventions similar to `volumes`. Currently only supports Vault.
    filters:
      apiKey:                  # API Key specific configurations
        enabled:
        secretsProviderName:
      jwtKey:                  # JWT Key specific configurations
        enabled:
        secretsProviderName:

  resources:
    requests:      # Requested amount of compute resources for the Gateway instance
      cpu:
      memory:
    limits:        # Maximum amount of compute resources allowed for the Gateway insta
nce
      cpu:
      memory:

  livenessProbe:
    initialDelaySeconds: # Number of seconds after the container has started before pr
obes are initiated
    failureThreshold:    # When a probe fails, Kubernetes will try failureThreshold ti
mes before giving up
    periodSeconds:       # How often (in seconds) to perform the probe
    timeoutSeconds:      # Number of seconds after which the probe times out
    successThreshold:    # Minimum consecutive successes for the probe to be considere
d successful after having failed
  readinessProbe:
    initialDelaySeconds:
    failureThreshold:
    periodSeconds:
    timeoutSeconds:
    successThreshold:
  startupProbe:
    initialDelaySeconds:
    failureThreshold:
    periodSeconds:
    timeoutSeconds:
    successThreshold:

  securityContext: # SecurityContext applied to the Gateway pod(s).
    fsGroup:       # Set to 1000 by default
    runAsGroup:
    runAsUser:

  serviceAccount:  # Name of the ServiceAccount associated to the Gateway instance
    name:

  service:         # Configuration of the Kubernetes service for the gateway
    type:          # Determines how the Service is exposed. Either ClusterIP, NodePort
, or LoadBalancer. Defaults to ClusterIP.
    nodePort:      # The port on which this service is exposed when type=NodePort or L
oadBalancer.
```

Following is an example Gateway instance configuration file:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
  name: my-gateway
spec:
  count: 3
  api:
    title: My Exciting APIs
    description: Lots of new exciting APIs that you can use for examples!
    version: 0.1.0
    serverUrl: https://gateway.example.com
  env:
    - name: spring.cloud.gateway.httpclient.connect-timeout
      value: "90s"
```

Configure External Access

Each Gateway instance has an associated service of type `ClusterIP`. You can expose this service via common Kubernetes approaches such as ingress routing or port forwarding. Consult your cloud provider's documentation for Ingress options available to you.

Using an Ingress Resource

Before adding an Ingress, ensure that you have an ingress controller running in your Kubernetes cluster according to your cloud provider documentation.

To use an Ingress resource for exposing a Gateway instance:

1. In the namespace where the Gateway instance was created, locate the `ClusterIP` service associated with the Gateway instance. You can either use this service as an Ingress backend or change it to a different Service type.

2. Create a file called `ingress-config.yaml`, with the following YAML definition:

   ```
   apiVersion: networking.k8s.io/v1
   kind: Ingress
   metadata:
     name: my-gateway-ingress
     namespace: my-namespace
     annotations:
       kubernetes.io/ingress.class: contour
   spec:
     rules:
       - host: my-gateway.my-example-domain.com
         http:
           paths:
             - path: /
               pathType: Prefix
               backend:
                 service:
                   name: my-gateway
                   port:
                     number: 80
   ```

   For the `host` and `serviceName` values, substitute your desired hostname and service name.

   This example Ingress resource configuration uses the Project Contour Ingress Controller.

You can adapt the example configuration if you wish to use another Ingress implementation.

3. Apply the Ingress definition file. The Ingress resource will be created in the same namespace as the Gateway instance.

4. Examine the newly created Ingress resource:

```
$ kubectl -n my-namespace get ingress my-gateway-ingress

NAME                  CLASS    HOSTS                                ADDRES
S        PORTS   AGE
my-gateway-ingress    <none>   my-gateway.my-example-domain.com     34.69.
53.79    80      2m51s
```

```
$ kubectl -n my-namespace describe ingress my-gateway-ingress

Name:             my-gateway-ingress
Namespace:        my-namespace
Address:          34.69.53.79
Default backend:  default-http-backend:80 (<error: endpoints "default-http-back
end" not found>)
Rules:
  Host                                      Path  Backends
  ----                                      ----  --------
  my-gateway.my-example-domain.com
                                            /   my-gateway:80 ()
```

As the example output shows, the `my-gateway.my-example-domain.com` virtual host in the Ingress definition is mapped to the `my-gateway` service on the backend.

5. Ensure that you can resolve the Ingress definition hostname (in this example, `my-gateway.my-example-domain.com`) to the IP address of the Ingress resource.

The IP address is shown in the `Address` field of the output from the `kubectl describe` command.

For local testing, use the command below to open your /etc/hosts file.

```
sudo vim /etc/hosts
```

Resolve the hostname by adding a line to the hosts file.

```
34.69.53.79     my-gateway.my-example-domain.com
```

For extended evaluation, you might create a wildcard DNS A record that maps any virtual host on the domain name (for example, `*.my-example-domain.com`) to the Ingress resource.

6. You should now be able to connect to your Gateway instance via the Ingress resource, using a web browser or an HTTP client such as HTTPie or cURL.

```
$ http my-gateway.my-example-domain.com/github
$ http my-gateway.my-example-domain.com/github/<YOUR_GITHUB_USERNAME>
```

These requests should receive responses from the GitHub homepage (`https://github.com`) or from the requested path on the GitHub website.

7. Test the SSO configuration, for example using an HTTP client such as HTTPie:

```
$ http my-gateway.my-example-domain.com/github
```

   This request should result in a `302` HTTP status code response, redirecting to the SSO login page. If you use a web browser to access the route `my-gateway.my-example-domain.com/github`, you will be redirected to the SSO login page. After authenticating, you will be redirected to the GitHub home page.

TLS Passthrough

If you would like to enable TLS termination on your Gateway instance, you will need to route requests to port 443, rather than port 80, of the gateway service.

You will also need to configure your Ingress to allow TLS passthrough - this configuration is Ingress implementation dependent.

As an example, to do this using Contour, instead of using the Ingress API you will need to create an HTTPProxy instance, using the TLS passthrough option:

```
apiVersion: projectcontour.io/v1
kind: HTTPProxy
metadata:
  name: my-gateway-httpproxy
spec:
  virtualhost:
    fqdn: my-gateway.my-example-domain.com
    tls:
      passthrough: true
  tcpproxy:
    services:
      - name: my-gateway
        port: 443
```

Gateway Actuator Management Port

Spring Cloud Gateway for Kubernetes instances are created with a Spring Boot actuator management port. The management port is 8090 on each Gateway instance pod based on the HA configuration. This management port can be used for monitoring using the following endpoints:

- `/actuator/info` - display version and other Gateway instance information
- `/actuator/health` - displays Gateway instance health indicator as status value `UP` or `DOWN`
- `/actuator/gateway/routes` - retrieve list of all API routes currently available on Gateway instance
- `/actuator/gateway/globalfilters` - retrieve list of global filters enabled on Gateway instance
- `/actuator/gateway/routefilters` - retrieve list of route filters available on Gateway instance

Configure for High Availability

You can configure Spring Cloud Gateway for Kubernetes to run multiple instances in High Availability as you would do with a normal Kubernetes resource.

While a Gateway is running you can use `kubectl scale` to modify the number of replicas. For example, given a Gateway that has 1 replica, the following will increase the number of replicas to 4.

```
$ kubectl scale scg my-gateway --replicas=4
```

And to decrease the number back to the original value.

```
$ kubectl scale scg my-gateway --replicas=1
```

In initial configuration, you can specify the number of replicas using the `spec.count` parameter. The following example configures a replica count of 3.

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
  name: my-gateway
spec:
  count: 3
```

So long as no other changes are introduced in the descriptor, you can safely modify `spec.count` and re-apply to increase or decrease the number of replicas.

To verify your changes use `kubectl get pods` to check that the pods match the count number.

Configure TLS termination

You can configure gateway instances to perform TLS termination, using different certificates for different routes.

Certificates and their associated private keys are loaded from Kubernetes TLS secrets. Create a TLS type secret for each certificate you would like the Gateway to serve. The easiest way to do this is with `kubectl` and PEM encoded certificate and key files:

```
kubectl create secret tls my-tls-secret-name --cert=path/to/tls.crt --key=path/to/tls.
key
```

The `tls.crt` file can contain multiple CA certificates concatenated together with the server certificate to represent a complete chain of trust.

The `tls.key` file should contain the private key for the server certificate in PKCS#8 or PKCS#1 format.

Next, create a Gateway resource which references your TLS certificates. Each entry in the `spec.tls` array contains a `secretName` which references the TLS secret containing the certificate(s)/key pair you want to serve, and a list of `hosts`. When a request arrives at the gateway referencing one of these hosts, the gateway will serve the certificate from the matching secret.

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
  name: test-gateway-tls
```

```
spec:
  count: 1
  tls:
    - hosts:
        - host-a.my-tls-gateway.my-example-domain.com
        - host-b.my-tls-gateway.my-example-domain.com
      secretName: tls-secret-1
    - hosts:
        - host-c.my-tls-gateway.my-example-domain.com
        - host-d.my-tls-gateway.my-example-domain.com
      secretName: tls-secret-2
```

The client with which you make requests to your gateway must support Server Name Indication, in order to pass the requested host to the gateway as part of the TLS handshake.

To verify that everything is working as expected, you can use `openssl` to check the certificates that are returned for each of the configured hosts. For example:

```
openssl s_client -showcerts -servername host-a.my-tls-gateway.my-example-domain.com -c
onnect <your ingress ip>:443 | openssl x509 -text
```

where `<your ingress ip>` should be replaced with the external IP of your TLS passthrough enabled ingress.

Configure Environment Variables

You can define a map of environment variables to configure the API gateway using the `spec.env` property. The following example configure the connection timeout from API gateway to application services and the Spring Framework web package logging level.

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
  name: gateway-demo
spec:
  ...
  env:
    - name: spring.cloud.gateway.httpclient.connect-timeout
      value: "90s"
    - name: logging.level.org.springframework.web
      value: debug
```

Disable SecureHeaders Global Filter

The backing app for a Spring Cloud Gateway service instance has a custom SecureHeaders filter globally enabled by default. This filter adds the following headers to the response:

Enabled Secure Header	Default Value
`Cache-Control`	`no-cache, no-store, max-age=0, must-revalidate`
`Pragma`	`no-cache`
`Expires`	`0`

Enabled Secure Header	Default Value
`X-Content-Type-Options`	`nosniff`
`Strict-Transport-Security`	`max-age=631138519`
`X-Frame-Options`	`DENY`
`X-XSS-Protection`	`1; mode=block`

If you do not want any secure headers being added to the response, you can disable the global filter for the entire gateway instance by setting `disable-secure-headers` to `true`:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
  name: my-gateway
spec:
  env:
    - name: spring.cloud.gateway.secure-headers.disabled
      value: "true"
```

To disable a specific header for a given route, you could use `RemoveResponseHeader` filter for the route. For example, to remove `X-Frame-Options` header for a route, you might run:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: my-gateway-routes
spec:
  routes:
    - uri: https://httpbin.org
      predicates:
        - Path=/remove-cache-control/**
      filters:
        - StripPrefix=1
        - RemoveResponseHeader=X-Frame-Options
```

To disable a specific header globally for all routes, you could set an environment variable on the gateway according to the `SecureHeaders` filter doc:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
  name: my-gateway
spec:
  env:
    - name: spring.cloud.gateway.filter.secure-headers.disable
      value: "x-frame-options"
```

Configure Cross-Origin Resource Sharing (CORS)

You can define a global CORS behavior that will be applied to all route configurations mapped to it.

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
```

```
metadata:
  name: my-gateway
spec:
  api:
    cors:
      allowedOrigins:
        - "https://foo.example.com"
      allowedMethods:
        - "GET"
        - "PUT"
        - "POST"
      allowedHeaders:
        - '*'
```

The following parameters can be configured in the `spec.api.cors` block:

Parameter	Function	Example
`allowedOrigins`	Allowed origins to make cross-site requests. The special value "*" allows all domains. These values will be combined with the values from allowedOriginPatterns.	`allowedOrigins: https://example.com`
`allowedOriginPatterns`	Alternative to allowedOrigins that supports more flexible origins patterns with "*" anywhere in the host name in addition to port lists. These values will be combined with the values from allowedOrigins.	`allowedOriginPatterns: - https://*.test.com:8080`
`allowedMethods`	Allowed HTTP methods on cross-site requests. The special value "*" allows all methods. If not set, "GET" and "HEAD" are allowed by default.	`allowedMethods: - GET - PUT - POST`
`allowedHeaders`	Allowed headers in cross-site requests. The special value "*" allows actual requests to send any header.	`allowedHeaders: - X-Custom-Header`
`maxAge`	How long, in seconds, the response from a pre-flight request can be cached by clients.	`maxAge: 300`
`allowCredentials`	Whether user credentials are supported on cross-site requests. Valid values: `true`, `false`.	`allowCredentials: true`
`exposedHeaders`	HTTP response headers to expose for cross-site requests.	`exposedHeaders: - X-Custom-Header`

You can also configure CORS behavior per route. However, the global CORS configuration must not be set. Each route defined on the gateway should have a matching path predicate on the route config.

Note that you can also define per-route cors behavior through the Cors filter.

The example below configures CORS behavior for the `/get/**` and `/example/**` routes:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
  name: my-gateway
spec:
  api:
    cors:
      perRoute:
        '[/get/**]':
```

```
          allowedOrigins:
            - "https://foo.example.com"
          allowedMethods:
            - "GET"
            - "PUT"
            - "POST"
          allowedHeaders:
            - '*'
        '[/example/**]':
          allowedOrigins:
            - "https://bar.example.com"
          allowedMethods:
            - "GET"
            - "POST"
          allowedHeaders:
            - '*'
```

Each route can be configured with the same parameters as in the table above.

Here is a matching route config:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: my-gateway-routes
spec:
  routes:
    - uri: https://httpbin.org
      predicates:
        - Path=/get/**
      filters:
        - StripPrefix=1
    - uri: https://httpbin.org
      predicates:
        - Path=/example/**
      filters:
        - StripPrefix=1
```

Note: To avoid browser calls failing due to duplicated headers (for example, receiving multiple 'Access-Control-Allow-Origin' or multiple 'Access-Control-Allow-Credentials') because a downstream service is also doing CORS processing, duplicates in these two headers are automatically removed and the one configured in the gateway will always predominate.

Configure Java Environment Options

For JVM tuning it is possible to define Java Environment Options (`JAVA_OPTS`) in the Spring Cloud Gateway for K8s configuration.

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
  name: my-gateway
spec:
  count: 2
```

```
  java-opts: -XX:+PrintFlagsFinal -Xmx512m
```

This will restart the pods and apply the options to the underlying gateway instances.

Configure session expiration

If you need to be able to discard inactive sessions after a certain time (e.g 10 minutes), just add the `inactive-session-expiration-in-minutes` configuration.

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
  name: my-gateway
spec:
  sso:
    secret: my-sso-credentials
    inactive-session-expiration-in-minutes: 10
```

This does not modify any authorization server token expiration (or ttl) configuration. It only affects the session information managed inside the gateway.

Configuring Hardware Resources

Similarly to other Kubernetes resources, you can optionally define the required memory (RAM) and CPU for a Gateway under `spec.resources`.

By default each instance is initialized with:

Resource	Requested	Limit
Memory	256Mi	512Mi
CPU	500m	2

But you can change it as seen in the example below. Note that less than the required may cause issues and is not recommended.

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
  name: my-gateway
spec:
  resources:
    requests:
      memory: "512Mi"
      cpu: "1"
    limits:
      memory: "1Gi"
      cpu: "2"
```

Configuring Probes

Similarly to other Kubernetes resources, you can optionally configure the `livenessProbe`, `readinessProbe`, and `startupProbe`, for a Gateway.

By default each instance is initialized with:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
  name: my-gateway
spec:
  livenessProbe:
    initialDelaySeconds: 5
    failureThreshold: 10
    periodSeconds: 3
    timeoutSeconds: 1
    successThreshold: 1
  readinessProbe:
    initialDelaySeconds: 5
    failureThreshold: 10
    periodSeconds: 3
    timeoutSeconds: 1
    successThreshold: 1
  startupProbe:
    initialDelaySeconds: 10
    failureThreshold: 30
    periodSeconds: 3
    timeoutSeconds: 1
    successThreshold: 1
```

But you can change them in order to better match your requirements.

Configure Observability

Spring Cloud Gateway for Kubernetes can be configured expose tracing and to generate a set of metrics and tracings based on different monitoring signals to help with understanding behaviour in aggregate.

Note: Metrics and Tracing are independent from each other.

Exposing Metrics to Wavefront

To expose metrics to Wavefront we need to create a `Secret` with the following data: `wavefront.api-token` and `wavefront.uri`, representing Wavefront's API token and Wavefront's URI endpoint respectively. For example:

```
apiVersion: v1
kind: Secret
metadata:
  name: metrics-wavefront-secret
data:
  wavefront.api-token: "NWU3ZCFmNjYtODlkNi00N2Y5LWE0YTMtM2U3OTVmM2Y3MTZk"
  wavefront.uri: "aHR0cHM6Ly92bAdhcmUud2F2ZWZyb250LmNvbQ=="
```

Then, in the `SpringCloudGateway` kind, reference the secret created in the step before under the `metrics` section. For example:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
  name: test-gateway-metrics
spec:
  observability:
    metrics:
      wavefront:
        enabled: true
    wavefront:
      secret: metrics-wavefront-secret
      source: my-source
      application: my-shopping-application
      service: gateway-service
```

After applying the configuration, Wavefront will start receiving the metrics provided by default by Spring Cloud Gateway.

Note: If you are also using wavefront for tracing, ensure you specify the same secret and source in both specs.

Using the Spring Cloud Gateway for Kubernetes Dashboard for Wavefront

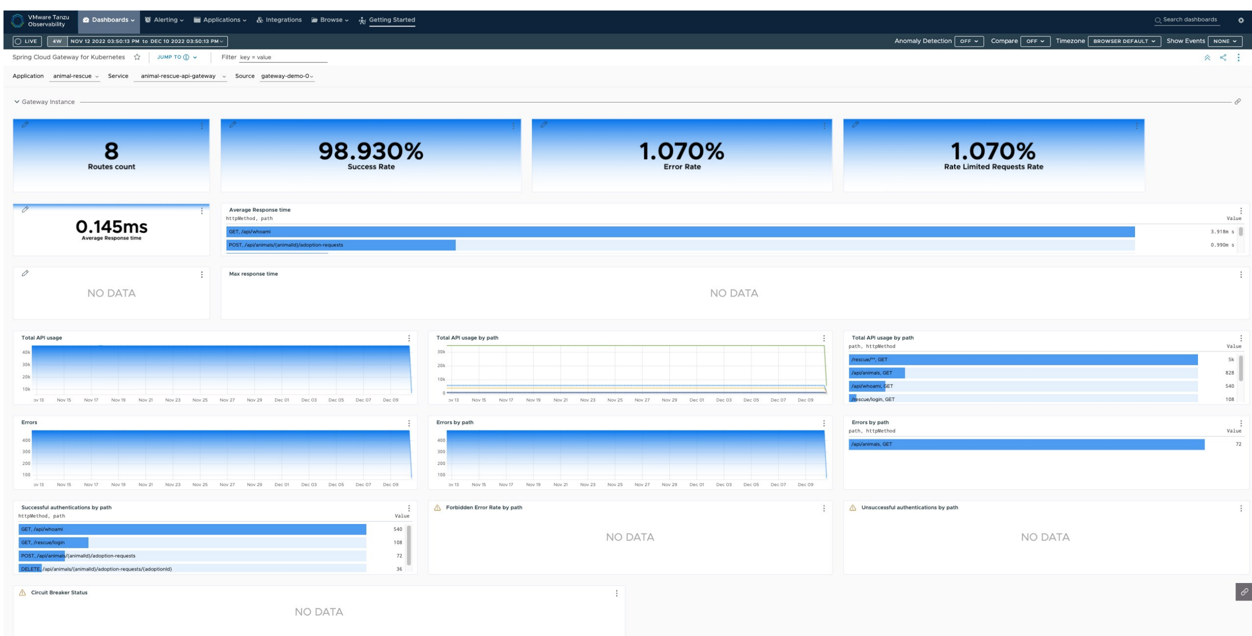
Spring Cloud Gateway for Kubernetes has a pre-built dashboard you can use in Wavefront.

If you are using VMware's Wavefront, then you can clone and customize the already created Spring Cloud Gateway for Kubernetes Dashboard.

Alternatively, you can find a `dashboards` folder inside `Spring Cloud Gateway for Kubernetes` release artifacts which contains a Wavefront template.

To import it, we need to create an API Token and execute the following command:

```
curl -XPOST 'https://vmware.wavefront.com/api/v2/dashboard' --header "Authorization: B
earer ${WAVEFRONT_API_TOKEN}" --header "Content-Type: application/json" -d "@wavefront
-spring-cloud-gateway-for-kubernetes.json"
```



Exposing Metrics to Prometheus

To expose metrics to Prometheus we need to add a prometheus section in the `SpringCloudGateway` kind and if we want scrapping annotations to be added into the gateway pods, for example:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
  name: test-gateway-metrics
spec:
  observability:
    metrics:
      prometheus:
        enabled: true
```

After applying the configuration, the Prometheus actuator endpoint will be available.

If, in addition to this, we want the scrapping annotations to be added to all the Spring Cloud Gateway Pods, we should create our Prometheus connfigurations with `annotations` set to `true`, for example:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
  name: test-gateway-metrics-with-annotations
spec:
  observability:
    metrics:
      prometheus:
        enabled: true
        annotations:
          enabled: true
```

This will add the following annotations to every Spring Cloud Gateway Pod:

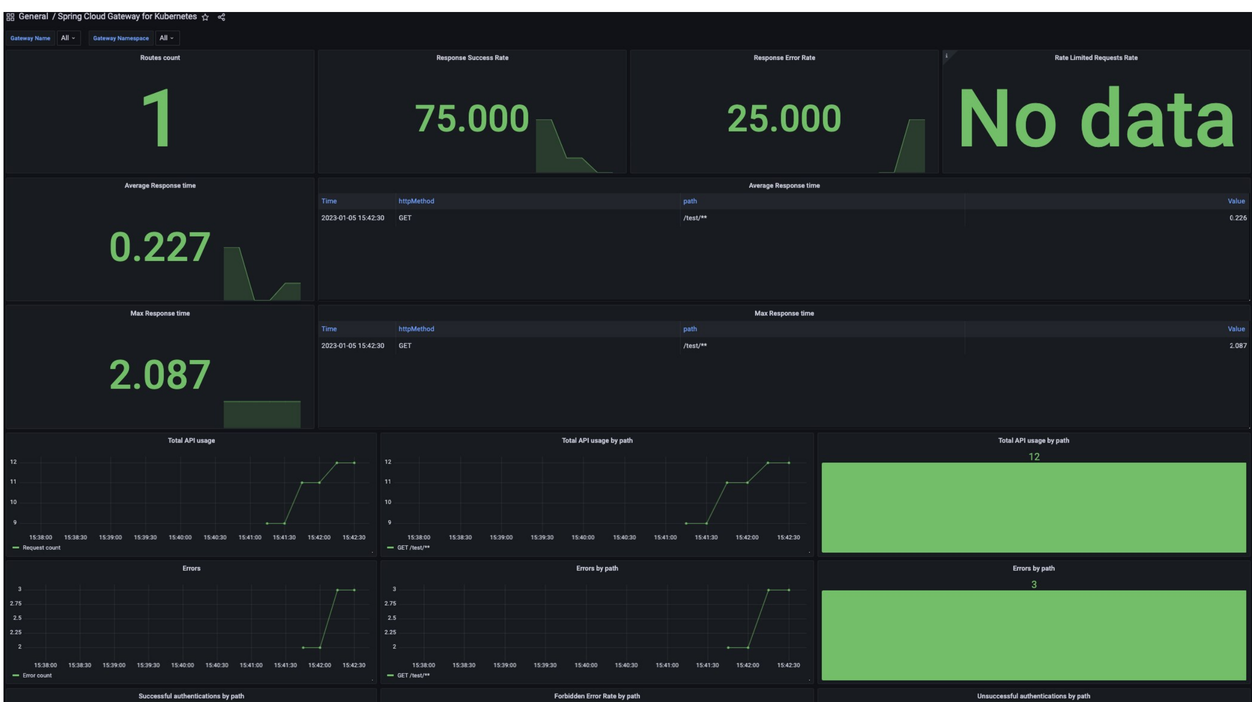
```
   annotations:
     prometheus.io/scrape: "true"
```

```
    prometheus.io/path: "/actuator/prometheus"
    prometheus.io/port: "8090"
```

Using the Spring Cloud Gateway for Kubernetes Dashboard for Grafana

You can find a `dashboards` folder inside `Spring Cloud Gateway for Kubernetes` release artifacts which contains a Grafana template.

To import it you can follow the how to import guide.



Exposing Tracing to Wavefront

To expose tracing to Wavefront we need to create a `Secret` with the following data: `wavefront.api-token` and `wavefront.uri`, representing Wavefront's API token and Wavefront's URI endpoint respectively. For example:

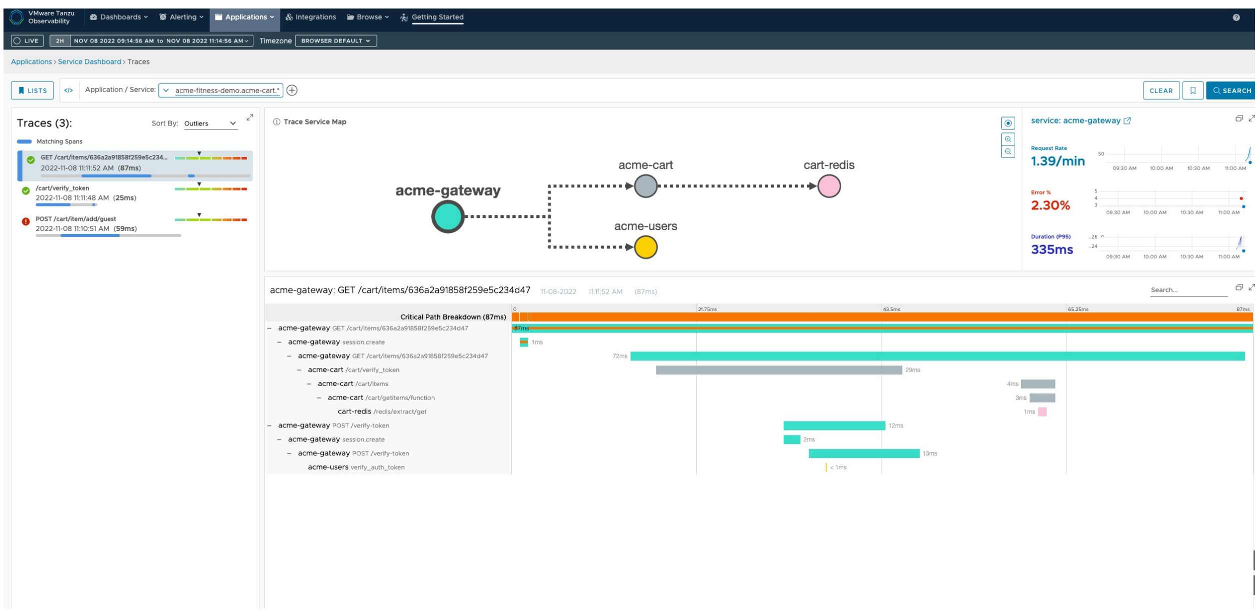
```
apiVersion: v1
kind: Secret
metadata:
  name: tracing-wavefront-secret
data:
  wavefront.api-token: "NWU3ZCFmNjYtODlkNi00N2Y5LWE0YTMtM2U3OTVmM2Y3MTZk"
  wavefront.uri: "aHR0cHM6Ly92bAdhcmUud2F2ZWZyb250LmNvbQ=="
```

Then, in the `SpringCloudGateway` kind, reference the secret created in the step before under the `tracing` section. For example:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
  name: test-gateway-tracing
spec:
  observability:
    tracing:
```

```
      wavefront:
        enabled: true
    wavefront:
      secret: tracing-wavefront-secret
      source: my-source
      application: my-shopping-application
      service: gateway-service
```

After applying the configuration, Wavefront will start receiving the traces



Note: If you are also using wavefront for metrics, ensure you specify the same secret and source in both specs.

Applying custom labels to the Gateway Pods

Custom labels can be added to the Gateway configuration. These labels will be propagated to the Pods created by the gateway operator For example:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
  name: test-gateway-tracing
  labels:
    test-label: test
spec:
  count: 2
```

Then the Pods can be listed by specifying the label:

```
kubectl get pods -l test-label=test
```

Customizing the service type

By default, the gateway is exposed with a ClusterIP service. You can change the type to a NodePort or a LoadBalancer by specifying the `spec.service.type`. You can also configure the exposed port by

specifying `spec.service.port`. If not specified, the port will automatically be assigned.

For example:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
  name: my-gateway
spec:
  service:
    type: NodePort
    nodePort: 32222
```

Note, for local development, your cluster needs to be configured to expose your chosen `nodePort` before you can send traffic to the nodes from the host.

Using Single Sign-On

Spring Cloud Gateway for Kubernetes supports authentication and authorization using Single Sign-On (SSO) with an OpenID identity provider which supports OpenID Connect Discovery protocol.

On top of using a SSO authentication workflow, you can also set up filters to support:

- Scope-based Access Control
- Role-based Access Control

Configure Single Sign-On (SSO)

You can configure Spring Cloud Gateway for Kubernetes to authenticate requests via Single Sign-On (SSO), using an OpenID identity provider.

To configure a Gateway instance to use SSO:

1. Create a file called `sso-credentials.txt`, including the following properties:

   ```
   scope=openid,profile,email
   client-id={your_client_id}
   client-secret={your_client_secret}
   issuer-uri={your-issuer-uri}
   ```

   For the `client-id`, `client-secret`, and `issuer-uri` values, use values from your OpenID identity provider. For the `scope` value, use a list of scopes to include in JWT identity tokens. This list should be based on the scopes allowed by your identity provider.

   `issuer-uri` configuration should follow Spring Boot convention, as described in the official Spring Boot documentation:

   The provider needs to be configured with an issuer-uri which is the URI that the it asserts as its Issuer Identifier. For example, if the issuer-uri provided is "https://example.com", then an OpenID Provider Configuration Request will be made to "https://example.com/.well-known/openid-configuration". The result is expected to be an OpenID Provider Configuration Response.

   Note that only authorization servers supporting OpenID Connect Discovery protocol can be

used.

2. Configure external authorization server to allow redirects back to the gateway. Please refer to your authorization server's documentation and add `https://<gateway-external-url-or-ip-address>/login/oauth2/code/sso` to the list of allowed redirect URIs.

3. In the Spring Cloud Gateway for Kubernetes namespace, create a Kubernetes secret using the `sso-credentials.txt` file created in the previous step:

```
$ kubectl create secret generic my-sso-credentials --from-env-file=./sso-credentials.txt
```

4. Examine the secret using the `kubectl describe` command. Verify that the `Data` column of the secret contains all of the required properties listed above.

5. Add the SSO secret in the `SpringCloudGateway` definition in the `spec.sso.secret` field. In the `routes` list of the `SpringCloudGatewayRouteConfig` object, add the setting `ssoEnabled: true` to each route that must have authenticated access. See the following updated `gateway-config.yaml` and `route-config.yaml` files:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
  name: my-gateway
spec:
  api:
    serverUrl: https://my-gateway.my-example-domain.com
    title: Animal Rescue APIs
    description: Make and track adoption requests for animals that need to be rescued.
    version: "1.0"
  sso:
    secret: my-sso-credentials
```

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: my-gateway-routes
spec:
  routes:
  - uri: https://github.com
    ssoEnabled: true
    predicates:
      - Path=/github/**
    filters:
      - StripPrefix=1
```

   With `ssoEnabled` set to `true`, the Gateway instance will use SSO for all API routes that are configured to allow authenticated access only.

6. Apply the updated Gateway and RouteConfig definition file:

```
$ kubectl apply -f gateway-config.yaml
$ kubectl apply -f route-config.yaml
```

Update Single Sign-On credentials

To update the SSO credentials for the gateway:

1. Update the value in secret (e.g. `my-sso-credentials`) by deleting the old secret then recreate it again:

   ```
   $ kubectl delete secret my-sso-credentials
   $ kubectl create secret generic my-sso-credentials --from-env-file=./sso-creden
   tials-updated.txt
   ```

   Alternatively, edit existing secret with new base64 encoded values:

   ```
   $ echo $NEW_CLIENT_SECRET | base64 | pbcopy
   $ kubectl edit secret my-sso-credentials
   ```

2. Rollout restart the gateway statefulset to enforce secret update:

   ```
   kubectl rollout restart statefulset my-gateway
   ```

Refer to SSO Setup Guide for Animal Rescue demo app with Okta Identity Provider for more details.

OpenAPI security schemes (SSO)

When `SSOEnabled` is set to `true` on any route, two `securityScheme` (See https://swagger.io/docs/specification/authentication) are registered as a component in the OpenAPI spec generated:

- AuthBearer to enable a dialog for providing a Bearer Authorization header
- OpenId to enable a dialog for getting a token from an OIDC configuration and adding it as a header

And, the schemes are bound to any of those routes. Other routes will not be affected and the scheme will not be applied on them.

Logout

Spring Cloud Gateway for Kubernetes instances provide a default API endpoint to logout of the current SSO session: `GET /scg-logout`.

If the OIDC provider supports RP-Initiated Logout, the `/scg-logout` call will also log the user out of the OIDC provider session.

You can redirect the user to another endpoint or url by adding a `redirect` query parameter to the logout call. For example, a `GET` call to `/scg-logout?redirect=/home` will redirect the user to the `/home` page.

Configuring Single Sign-On for Sample Application

In this guide, you'll learn how to configure Okta identity provider to use with the sample application Animal Rescue.

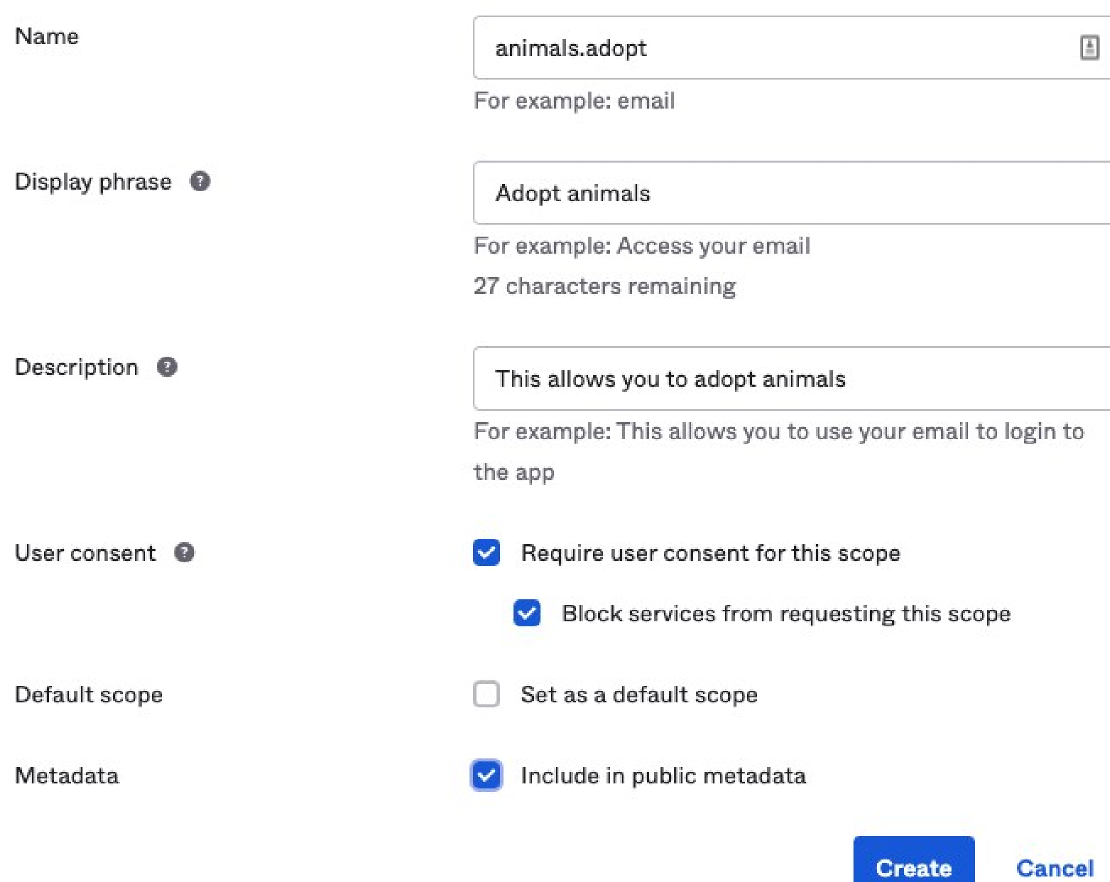
Configuring Okta OIDC provider

Login to Okta admin dashboard. You can use a free developer account or configure your existing account.

Create authorization server for Animal Rescue

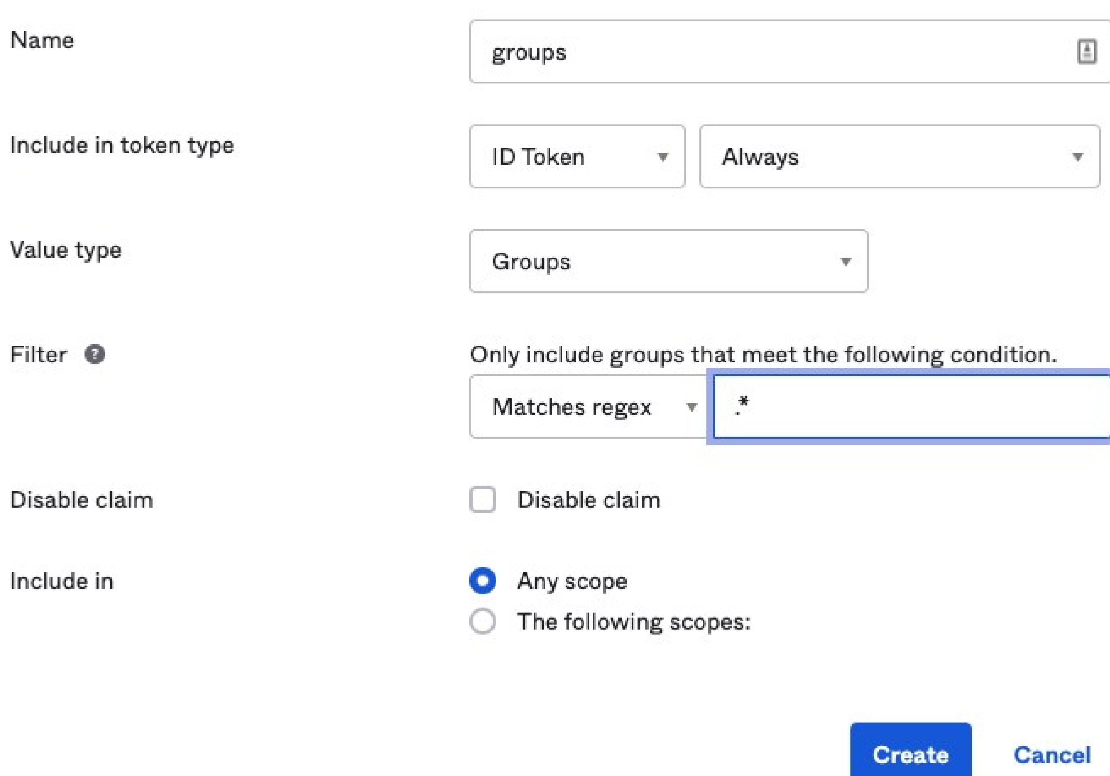
A new authorization server is required because Animal Rescue will need it's own set of scopes and claims.

1. Go to Security → API
2. Under the Authorization Servers tab, click "Add Authorization Server".
3. Use "Animal Rescue" as the name and set the audience to `api://animal-rescue`.
4. Now go to new created settings page, copy the value in "Issuer" field. This should be used as `issuer-uri` during Gateway setup.
5. Switch to "Scopes" tab and add a new scope: `animals.adopt` (with any display name and description). Check the box for "User Consent" and "Metadata"

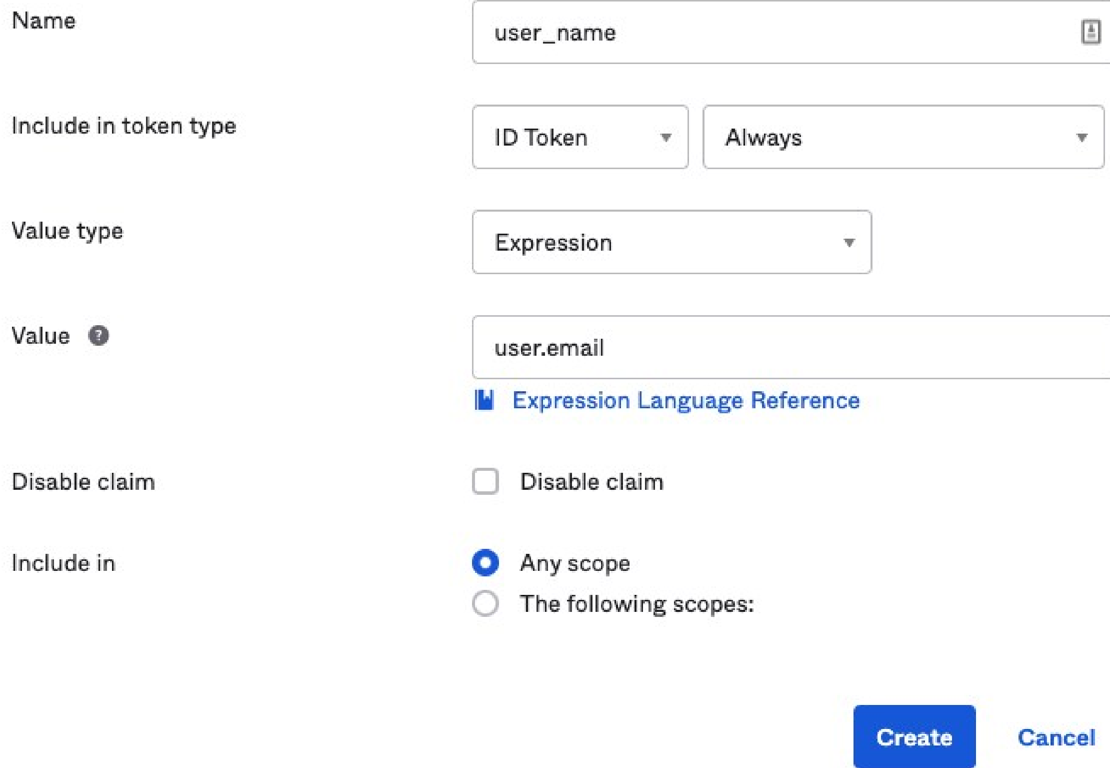


6. Switch to "Claims" tab and add a new claim: `groups`, set "Include in token type" to always include to ID Token, value type to "Groups" with filter matching regex ".*" (so all groups are included). Optionally, configure "Include in" to `groups` scope (you need to create the scope

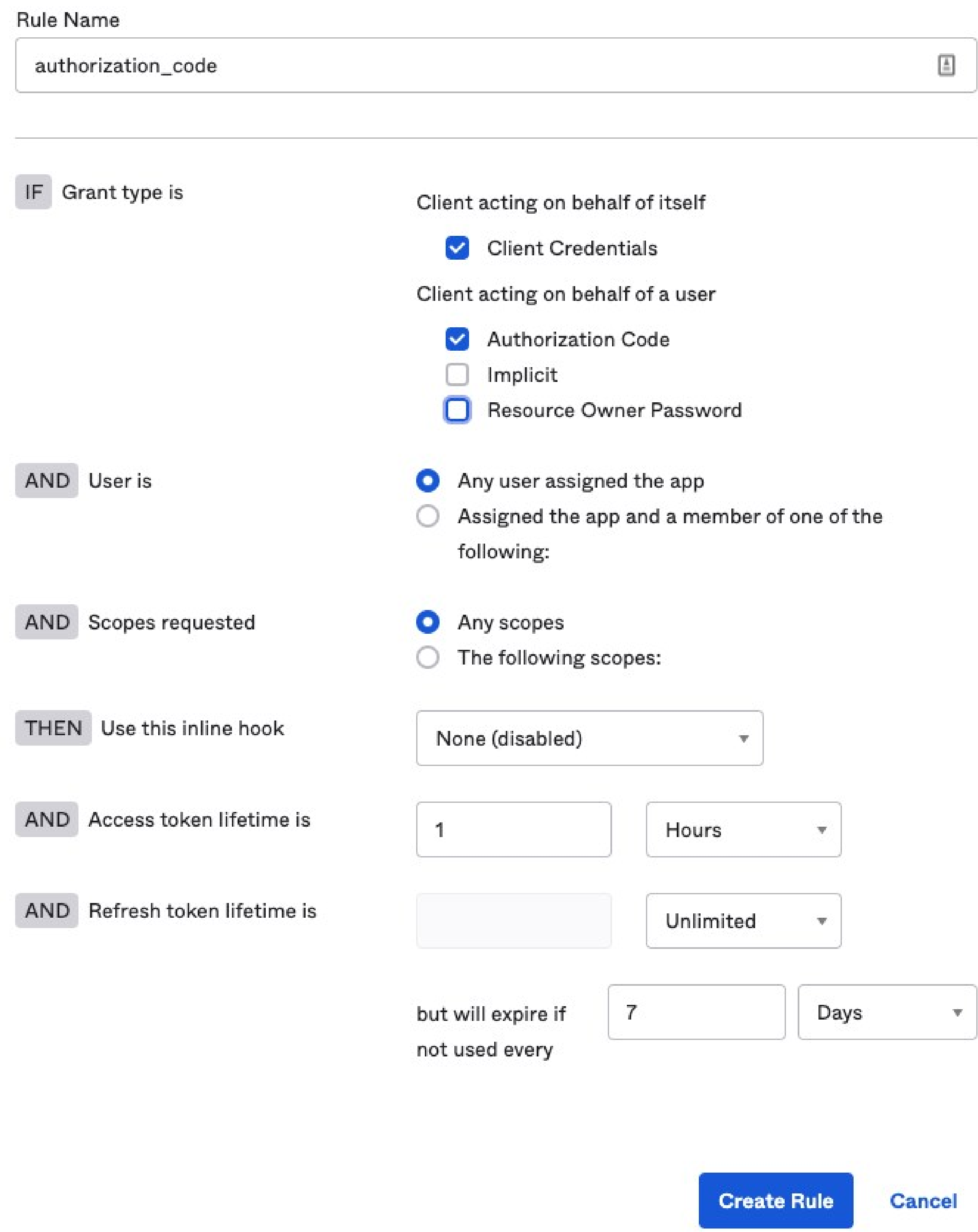
first) if you'd like to include groups information only when a certain scope is requested and approved.



7. Add a new claim `user_name` and set it to be always included into ID token, configure value to be `user.email`. The claim value can be configured using Okta Expression Language.



8. Switch to "Access Policies" tab and create a "Default" access policy, assigned to all clients.
9. Add a new rule to allow `authorization_code` grant, for any user, any scope.



Create users and groups

Navigate to "Directory → People" from the main menu

1. Click "Add Person" and configure all required fields.

Navigate to "Directory → Groups" from the main menu

1. Click "Add Group" and create "Adopter" group.

2. Click "Manage People" in "Adopter" group and add the accounts you created above.

Create new application

Navigate to "Applications → Applications" in the main menu.

1. Click "Create App Integration".
2. Select "OIDC - OpenID Connect" as the Sign-on method and select "Web Application" as the application type.
3. In "Sign-in redirect URIs" add `<gateway url>/login/oauth2/code/sso`. If your gateway has not been deployed yet, you can skip this step for now and add the redirect URI later.
4. Enable "Authorization Code" grant type for the app.
5. In "Assignments" section, select `Limit access to selected groups` and add the "Adopter" group.
6. Copy "Client ID" and "Client Secret".

Configuration summary

After you completed the steps above, you should have the following values:

- Issuer URI. That should be the value from the authorization server you created, not your account Okta domain.
- Client ID.
- Client secret.
- One or two test users ideally with different groups for testing.

Make sure you have them before proceeding to the next step.

Configure Animal Rescue app

Clone the repo first.

Configure SSO params

In the `animal-rescue` repo,

1. Create `backend/secrets/sso-credentials.txt` with the following:

```
jwk-set-uri=<issuer uri>/v1/keys
```

2. Create `gateway/sso-secret-for-gateway/secrets/test-sso-credentials.txt` with the following:

```
scope=openid,profile,email,groups,animals.adopt
client-id=<client id>
client-secret=<client id>
issuer-uri=<issuer uri>
```

If you decided to use `groups` scope to get groups information, make sure it is listed in `scope` parameter.

The issuer URI must *exactly* match the value from the server configuration, including trailing slashes! You can always check expected value by navigating to `<issuer-uri>/.well-known/openid-configuration` URL.

3. Edit `gateway/gateway-demo.yaml` and add `roles-attribute-name` into `sso` section:

```
sso:
  secret: animal-rescue-sso
  roles-attribute-name: "groups"
```

The default value is "roles". Alternatively you can configure Okta to return the "roles" claim instead of "groups".

Configure routes security

Edit `backend/k8s/animal-rescue-backend-route-config.yaml` file. Add `Scopes=animals.adopt` filter to `/api/animals/*/adoption-requests/**` route if you'd like to use scopes to authorize access to Adoption Request API, or `Roles=Adopter` if you'd like to use roles. You can keep both filters as well.

```
    - ssoEnabled: true
      tokenRelay: true
      predicates:
        - Path=/api/animals/*/adoption-requests/**
        - Method=POST,PUT,DELETE
      tags:
        - "pet adoption"
      filters:
        - Scopes=animals.adopt
```

Deploy the app

Run `kustomize build . | kubectl apply -f -` or refer to Animal Rescue README for most up to date deployment instructions.

Test

Port-forward the gateway demo-demo service:

```
kubectl port-forward service/gateway-demo 8080:80
```

Navigate to your gateway URL, http://localhost:8080/rescue.

Note: If you are using dynamic IP address you may need to go back to Okta and configure this IP address in the list of allowed Redirect URIs.

Try logging in with different test users, within or without "Adopter" groups and add, edit or delete adoption request. You should see a successful response or "Request failed with status code 403" error message depending on your groups list and approved scopes.

OpenAPI Generated Documentation

See below for how to provide API Gateway metadata and how API route configurations are used to auto-generate OpenAPI v3 documentation.

Accessing Generated OpenAPI v3 Documentation

The Spring Cloud Gateway for Kubernetes operator manages all API Gateway instances on Kubernetes cluster. When you apply any `SpringCloudGateway`, `SpringCloudGatewayRouteConfig` or `SpringCloudGatewayMapping` custom resources onto the Kubernetes cluster, the operator will act to reconcile the environment with those request resource changes. In addition to handling custom resource reconciliation, the operator also has an OpenAPI v3 compliant auto-generated documentation endpoint. You can access this endpoint by exposing the `scg-operator` service with an ingress and then access its `/openapi` endpoint. An example ingress applied to the `scg-operator` service in the `spring-cloud-gateway` namespace is shown below:

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: scg-openapi
  namespace: spring-cloud-gateway
  annotations:
    kubernetes.io/ingress.class: contour
spec:
  rules:
  - host: scg-openapi.mydomain.com
    http:
      paths:
      - path: /
        pathType: Prefix
        backend:
          service:
            name: scg-operator
            port:
              number: 80
```

Now you can access the auto-generated OpenAPI v3 endpoint by going to `https://scg-openapi.mydomain.com/openapi`. Application developers can provide their API route configuration to be exposed on an API Gateway instance and those API routes will then be input for generated documentation. This leads to consistent APIs based on API route configuration predicates, filters and metadata across all service instances and the APIs they expose.

It is important to note that a separate OpenAPI v3 document will be generated for each API Gateway instance and the `/openapi` endpoint provides an array of these documents for all of the instances on this Kubernetes cluster.

Configure OpenAPI Metadata

The following descriptive metadata can be defined when configuring an API Gateway instance:

- `serverUrl`: Publicly accessible user-facing URL of this Gateway instance. It is important to

note that this configuration does not create a new route mapping for this URL, this is only for metadata purposes to display in the OpenAPI generated documentation.

- `title`: Title describing the context of the APIs available on the Gateway instance (default: `Spring Cloud Gateway for K8S`)
- `description`: Detailed description of the APIs available on the Gateway instance (default: `Generated OpenAPI 3 document that describes the API routes configured for '[Gateway instance name]' Spring Cloud Gateway instance deployed under '[namespace]' namespace.`)
- `version`: Version of APIs available on this Gateway instance (default: `unspecified`)
- `documentation`: Location of additional documentation for the APIs available on the Gateway instance

Here is an example of an API Gateway configuration using this descriptive metadata:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
  name: my-gateway
spec:
  api:
    serverUrl: https://gateway.example.org
    title: My Exciting APIs
    description: Lots of new exciting APIs that you can use for examples!
    version: 0.1.0
    documentation: https://docs.example.org
```

This will be displayed in the `/openapi` endpoint of the operator as:

```
"info": {
  "title": "My Exciting APIs",
  "description": "Lots of new exciting APIs that you can use for examples!",
  "version": "0.1.0"
},
"externalDocs": {
  "url": "https://docs.example.org"
},
"servers": [
  {
    "url": "https://gateway.example.org"
  }
],
```

PUT/POST/PATCH Request Body Schema

For PUT, POST and PATCH operations, you may add the OpenAPI Schema of Request Body objects.

As in the example below, add `model.requestBody` property to a route with the proper information.

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
```

```
  name: myapp-route-config
spec:
  service:
    name: myapp-service
  routes:
    - id: example-route-id
      predicates:
        - Path=/users/**
      model:
        requestBody:
          description: User to add
          content:
            'application/json':
              schema:
                type: object
                description: User schema
                properties:
                  name:
                    type: string
                  age:
                    type: integer
                    format: int32
                required:
                  - name
```

The model, alongside with the available HTTP methods and headers will be published under `paths`.

```
"paths": {
    "/users/**": {
        "summary": "example-route-id",
        "get": {
            "responses": {
                "200": {
                    "description": "Ok"
                }
            }
        },
        "post": {
            "requestBody": {
                "description": "User to add",
                "content": {
                    "application/json": {
                        "schema": {
                            "required": [
                                "name"
                            ],
                            "type": "object",
                            "properties": {
                                "name": {
                                    "type": "string"
                                },
                                "age": {
                                    "type": "integer",
                                    "format": "int32"
                                }
                            },
                            "description": "User schema"
                        }
                    }
```

```
            }
        },
        "responses": {
            "200": {
                "description": "Ok"
            }
        }
    }
```

Custom HTTP Responses

In order to add custom HTTP responses for your paths, you may add the OpenAPI Schema of Responses objects.

As in the example below, add `model.responses` property to a route with the proper information.

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: myapp-route-config
spec:
  service:
    name: myapp-service
  routes:
    - id: example-route-id
      predicates:
        - Path=/users/**
      model:
        responses:
          200:
            description: "Obtain a list of users"
            content:
              application/json:
                schema:
                  type: object
                  description: User schema
                  properties:
                    name:
                      type: string
                    age:
                      type: integer
                      format: int32
          3XX:
            description: "Redirection applied"
            headers:
              X-Redirected-From:
                schema:
                  type: string
                  description: URL from which the request was redirected.
          default:
            description: "Unexpected error"
```

If you don't provide any HTTP responses, the operator will generate by default a `200 Ok` response for every path's operation. Some filters may add custom responses as well to document their inner functionality. You can overwrite these responses too by including them in this section.

Configure Spring Cloud Gateway Instances in Tanzu Application Platform

This topic describes how to add a Spring Cloud Gateway for Kubernetes as a Component of your Organization Catalog.

There are two basic scenarios.

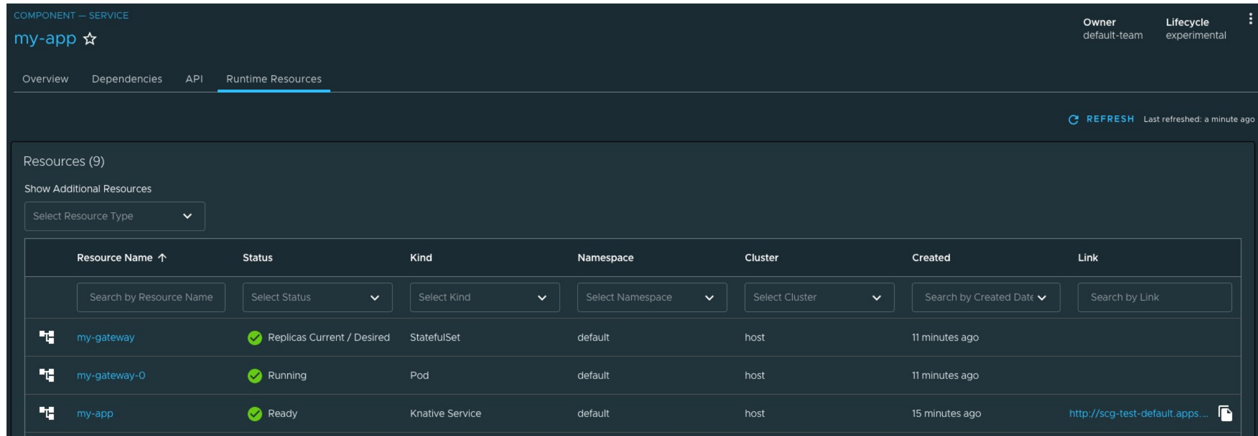
Adding Spring Cloud Gateway to a Component

In this scenario we want to add the Gateway instance to a running Component. For that we need to add the labels as in the example below.

Take care to match the `PART_OF` to the name of the label `app.kubernetes.io/part-of` in your Component.

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
  name: animal-rescue
  namespace: animal-rescue
  labels:
    tanzu.app.live.view: "true"
    tanzu.app.live.view.application.actuator.port: "8090"
    tanzu.app.live.view.application.flavours: spring-boot_spring-cloud-gateway
 app.kubernetes.io/part-of={PART_OF}
```

Once applied, the gateway pods will be visible in the list of resources.



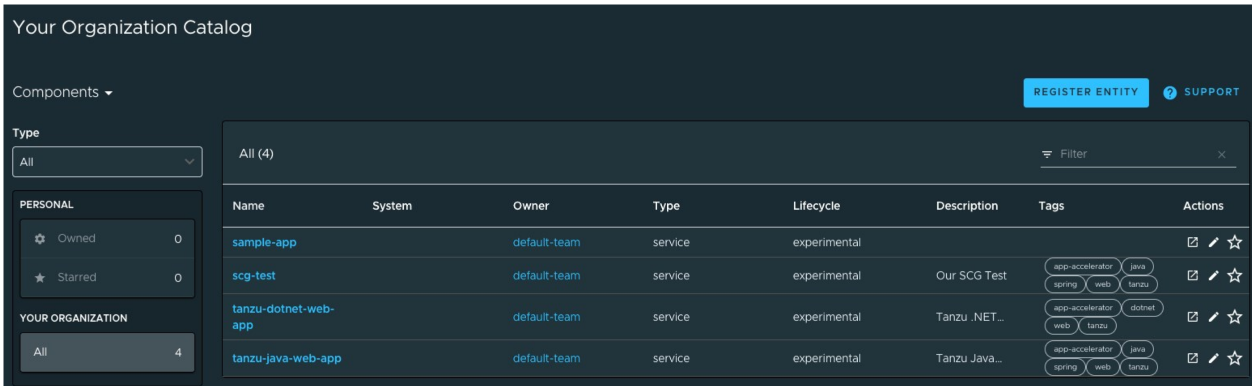
Adding Spring Cloud Gateway as a new Component

If you want to present the Gateway as an independent Component, you still need to add the labels seen before. But matching `part-of` value with the one used in your Component, as seen below.

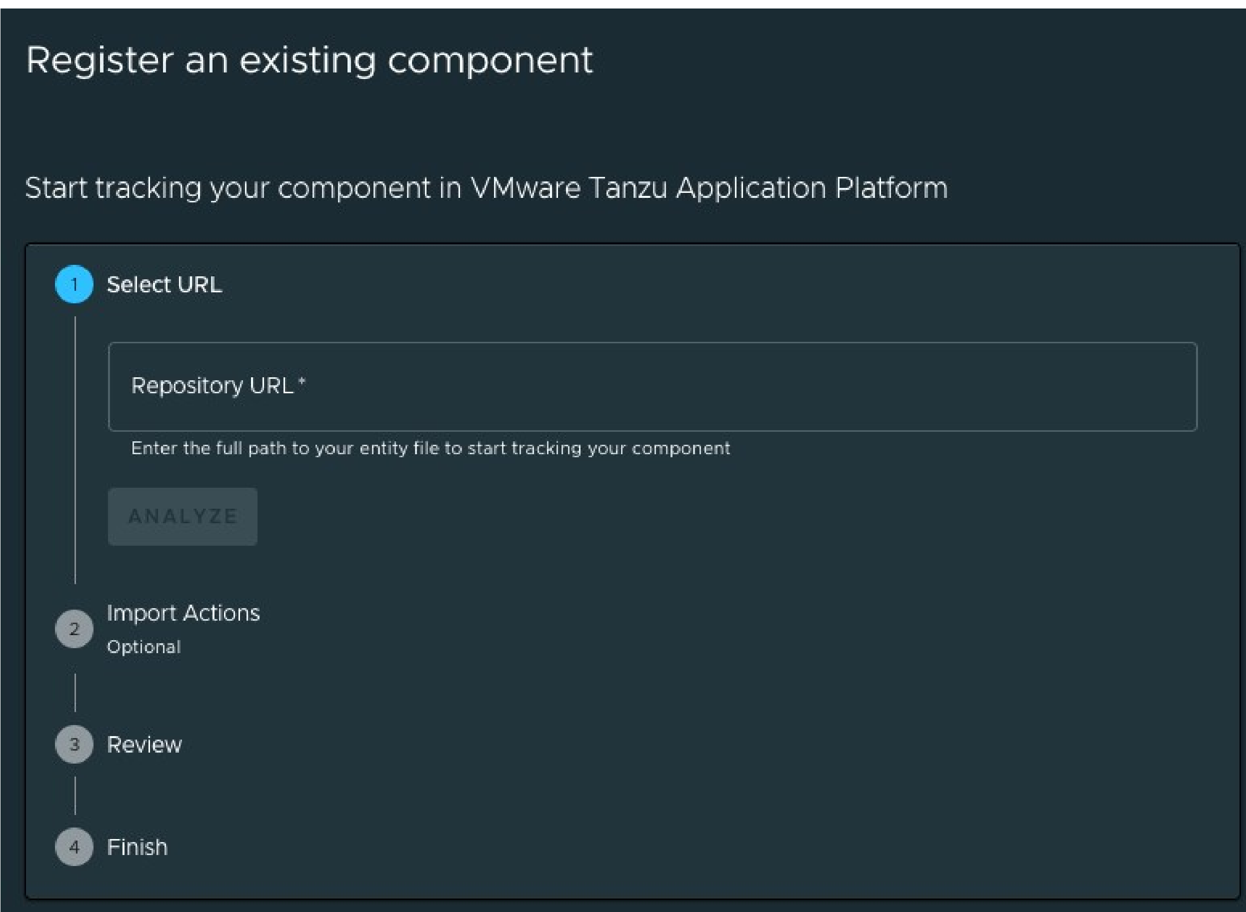
```
apiVersion: backstage.io/v1alpha1
kind: Component
metadata:
  name: my-gateway
  description: My application gateway
  annotations:
    'backstage.io/kubernetes-label-selector': 'app.kubernetes.io/part-of={PART_OF}'
```

```
spec:
  type: service
  lifecycle: experimental
  owner: default-team
```

Then, use the "Registry Entity" button at the top of the Catalog view.



And provide an url pointing to the Component descriptor file.



This will make the new component visible in the Catalog view.

Client Apps

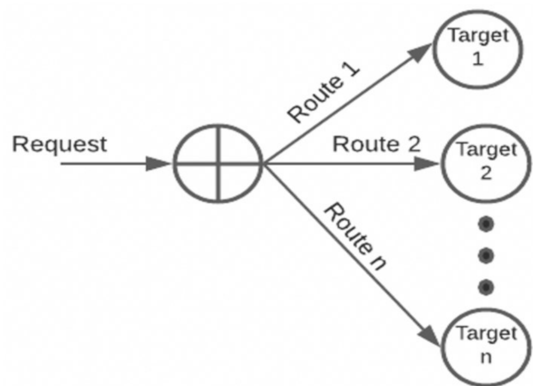
These topics describe how to use Spring Cloud Gateway for Kubernetes with client applications.

Configuring Gateway Routes

This topic describes how to add, update, and manage API routes for apps that use a Spring Cloud Gateway for Kubernetes instance.

What are API routes

Spring Cloud Gateway instances match requests to target endpoints using configured API routes. A route is assigned to each request by evaluating a number of conditions, called *predicates*. Each predicate may be evaluated against request headers and parameter values. All of the predicates associated with a route must evaluate to true for the route to be matched to the request. The route may also include a chain of *filters*, to modify the request before sending it to the target endpoint, or the received response.



Define Route Config

To define the API routes that your service intends to expose for consumers, you must create a `SpringCloudGatewayRouteConfig` resource. The definition for `SpringCloudGatewayRouteConfig` specifies:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name:            # Name given to this route configuration (required)
spec:
  service:         # Kubernetes Service to route traffic to specified `spec.routes`.
    name:          # Name of the service, required unless route defines `uri`.
    namespace:     # (Optional) If not set will use the RouteConfig's namespace.
    port:          # (Optional) If not set will use one of the available service ports
.
    predicates:    # (Optional) Predicates to be prepended to all routes. See Availabl
e Predicates.
    filters:       # (Optional) Filters to be prepended to all routes. See Available F
ilters.
    ssoEnabled:    # (Optional) Define SSO validation for all routes. See "Using Singl
e Sign-On".
  routes:          # Array of API routes.
    - title:       # (Optional) A title, will be applied to methods in the generated O
penAPI documentation
      description: # (Optional) A description, will be applied to methods in the gener
ated OpenAPI documentation
```

```
      uri:         # (Optional) Full uri, will override `service.name`
      ssoEnabled:  # Enable SSO validation. See "Using Single Sign-On"
      tokenRelay:  # Pass currently-authenticated user's identity token to application
 service
      predicates:  # See Available Predicates below
      filters:     # See Available Filters below
      order:       # Route processing order, same as Spring Cloud Gateway
      tags:        # Classification tags, will be applied to methods in the generated
OpenAPI documentation
  basicAuth:
    secret:        # The secret name containing basic auth credentials.
  openapi:
    components:
      schemas:       # Reusable schema objects.
      requestBodies: # Reusable request body objects.
```

Note: `service.name` is the recommended method for traffic configuration. Use `routes.uri` only when accessing external resources.

As example, create a file called `myapp-route-config.yaml`, with the following YAML definition:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: myapp-route-config
spec:
  service:
    name: myapp
  routes:
  - predicates:
      - Path=/api/public/**
    filters:
      - StripPrefix=2
```

Default Configuration

To simplify configuration, `StripPrefix=1` is applied by default when no value for StripPrefix is set.

Define Service Level Config

In order to avoid repetition across all or most API routes in their route configuration, the following properties can be defined at service level: `predicates`, `filters`, and `ssoEnabled`.

Service Filters

To have certain filters prepended to all routes, you can use the `service.filters` property. For example, that's how you can add rate limiting to all routes:

```
spec:
  service:
    name: myapp
    filters:
      - RateLimit=2,10s
```

Service Predicates

To have certain predicates prepended to all routes, you can use the `service.predicates` property. Example of all routes configured with a mandatory header:

```
spec:
  service:
    name: myapp
    predicates:
      - Header=X-Request-Id
```

Service SSO Config

To define SSO validation for all routes, you can use the `service.ssoEnabled` property. Example of all routes configured with SSO:

```
spec:
  service:
    name: myapp
    ssoEnabled: true
```

Each route can then override it, as below:

```
spec:
  service:
    name: myapp
    ssoEnabled: true
  routes:
    - predicates:
      - Path=/api/users
      ssoEnabled: false
```

Map Routes to Gateway

To add API routes to a Spring Cloud Gateway for Kubernetes instance, you must create a resource of type `SpringCloudGatewayMapping` that references both a `SpringCloudGateway` and a `SpringCloudGatewayRouteConfig` resource. The definition for `SpringCloudGatewayMapping` specifies:

```

apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayMapping
metadata:
  name:             # Name given to this route mapping (required)
spec:
  gatewayRef:       # Gateway instance which will serve traffic to the provided route c
onfig
    name:           # Name of the Gateway instance
    namespace:      # (Optional) If not set will use the Mapping's namespace
  routeConfigRef:   # Route configuration with the routes to apply to the gateway insta
nce
    name:           # Name of the route configuration resource
    namespace:      # (Optional) If not set will use the Mapping's namespace
```

Continuing the previous example, create a file called `myapp-mapping.yaml`, with the following YAML definition:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayMapping
metadata:
  name: myapp-mapping
spec:
  gatewayRef:
    name: my-gateway
    namespace: my-gateway-ns
  routeConfigRef:
    name: myapp-route-config
```

Apply the two definitions to your Kubernetes cluster:

```
$ kubectl apply -f myapp-route-config.yaml
$ kubectl apply -f myapp-mapping.yaml
```

Spring Cloud Gateway for Kubernetes processes the two objects and updates the Gateway instance named in the `spec.gateway` property value (in this example, the `my-gateway` instance). For the routes configured in the `spec.routes` section, the Gateway instance will route traffic to the app named in the `spec.service` property value (in this example, the `myapp` app).

After creating the mapping and route config resources, you should be able to access the `myapp` app at the fully qualified domain name (FQDN) used by the Gateway instance and the path `/api/*`. For example, if your Gateway instance is exposed by an Ingress resource at the domain `gateway.example.com`, you can access the `myapp` app at the following URL:

```
https://gateway.example.com/api/my-path
```

Note: Application services must respect `X-Forwarded-*` as the API Gateway is acting as a reverse proxy on behalf of the client. For Spring Boot applications, this can be configured by setting `server.forward-headers-strategy=NATIVE`. Please utilize the appropriate approach for your application's programming language and framework.

Available Predicates

For more detailed documentation on how to use the OSS Spring Cloud Gateway predicates, see the Spring Cloud Gateway OSS predicates documentation.

Predicate	Description
After	matches requests made after a certain datetime
Before	matches requests made before a certain datetime
Between	matches requests made between two certain datetimes
Cookie	matches requests with a certain cookie

Predicate	Description
Header	matches requests with a certain header
Host	matches requests with a certain host pattern
Method	matches requests to HTTP method (GET/POST)
Path	matches requests with path of certain pattern(s)
Query	matches requests with certain query parameter (optional with value pattern)
RemoteAddr	matches requests of a certain remote IP address
Weight	Split requests between a set of targets in a group
JWTClaim	Match on JWT claim value

Available Filters

For more detailed documentation on how to use the OSS Spring Cloud Gateway filters, see the Spring Cloud Gateway OSS filters documentation. The detailed documentation on additional filters provided by Spring Cloud Gateway for Kubernetes commercial product are listed on the Commercial Route Filters page.

Filter	Description
AddRequestHeader	Adds a header to a request
AddRequestHeadersIfNotPresent	Adds headers if not present in the original request
AddRequestParameter	Adds a request parameter to a request query string
AddResponseHeader	Adds a header to a matching response
AllowedRequestCookieCount	Determines if a matching request is allowed to proceed base on number of cookies
AllowedRequestHeadersCount	Determines if a matching request is allowed to proceed based on headers
AllowedRequestQueryParamsCount	Determines if a matching request is allowed to proceed base on query params
ApiKey	validate API keys from `X-API-Key` header against those stored in Hashicorp Vault
BasicAuth	Adds BasicAuth credentials as header
CircuitBreaker	Wraps routes in a circuit breaker
ClaimHeader	Copies data from a JWT claim into an HTTP Header
ClientCertificateHeader	Validate X-Fowarded-Client-Cert header certificate (optional fingerprint)
Cors	Configuring per-route Cross-Origin Resource Sharing (CORS)
DeDupeResponseHeader	Removes duplicates of certain headers
FallbackHeaders	Adds circuit breaker exception to a header
JwtKey	Adds multiple client JWT token validation

Filter	Description
MapRequestHeader	Maps a header from another one
PrefixPath	Adds a prefix to a matching request path
PreserveHostHeader	Preserves original host header when sending a request
RateLimit	Determines if a matching request is allowed to proceed base on volume
RedirectTo	Redirects a matching request with certain HTTP code to a certain URL
RemoveJsonAttributesResponseBody	Removes JSON attributes and its value from a JSON content
RemoveRequestHeader	Removes a header from a matching request
RemoveResponseHeader	Removes a header from a response
RemoveRequestParameter	Removes a query parameter from a matching request
RewriteAllResponseHeaders	Removes a query parameter from a matching request
RewritePath	Similar to RewriteResponseHeader, but applies transformation to all headers
RewriteLocationResponseHeader	Modifies the value of the location response header
RewriteResponseHeader	Rewrite the response header value
RewriteResponseBody	Rewrite the response body from a matching request
RewriteJsonAttributesResponseBody	Rewrite JSON attributes using JSON Path notations
Roles	List authorized roles needed to access route
Scopes	List scopes needed to access route
SecureHeaders	Adds some headers to a response per a security recommendation
SetPath	Manipulates a matching request path
SetResponseHeader	Replaces a certain response header
SetStatus	Sets HTTP status of a response
SSO Login	Redirects to authenticate if no valid Authorization token
StoreIpAddress	Store IP address value in the context of the application
StoreHeader	Store a header value in the context of the application
StripPrefix	Strips parts from a path of a matching request (default: 1)
Retry	Retries a matching request
RequestSize	Constrains a matching request with a certain request size
SetRequestHostHeader	Overrides host header value of a matching request
SsoAutoAuthorize	Adds a fake SSO authorization for development purposes
TokenRelay	Forwards OAuth2 access token to downstream resources

OpenApi Schema References

OpenApi references can be used by multiple API routes so that they don't have to duplicate definitions in route configuration. It works via the `'$ref'` property, which targets an object in the `openapi` section. Currently, this feature is only supported for requests and responses.

In the following example, we're referencing `UserRequest` and `UserResponse` objects, which in turn point to `schemas.User`:

```
routes:
- predicates:
    - Path=/api/users
    - Method=POST
  model:
    requestBody:
      content:
        'application/json':
          schema:
            '$ref': "/components/requestBodies/UserRequest"
    responses:
      '200':
        content:
          'application/json':
            schema:
              '$ref': "/components/schemas/UserResponse"
openapi:
  components:
    schemas:
      User:
        type: object
        properties:
          id:
            type: string
          name:
            type: string
          email:
            type: string
            format: email
      UserResponse:
        '$ref': "/components/schemas/User"
    requestBodies:
      UserRequest:
        required: ["name", "email"]
        '$ref': "/components/schemas/User"
```

Commercial Route Filters

The open-source Spring Cloud Gateway project includes a number of built-in filters for use in Gateway routes. Spring Cloud Gateway provides a number of custom filters in addition to those included in the OSS project.

Filters Included In Spring Cloud Gateway OSS

Filters in Spring Cloud Gateway OSS can be used in Spring Cloud Gateway for Kubernetes. Spring Cloud Gateway OSS includes a number of `GatewayFilter` factories used to create filters for routes.

For a complete list of these factories, see the Spring Cloud Gateway OSS documentation.

Filters Added In Spring Cloud Gateway for Kubernetes

Following sections offers information about the custom filters added in VMware Spring Cloud Gateway and how you can use them.

`AddRequestHeadersIfNotPresent`: Request headers modification filter

This filter adds certain request headers if those are not present in the original request. It accepts a list of key value pairs.

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: my-gateway-routes
spec:
  service:
    name: myapp
  routes:
  - predicates:
      - Path=/foo
    filters:
      - AddRequestHeadersIfNotPresent=Content-Type:application/json,Connection:keep-al
ive
```

In the example, a raw request to `/foo` will have the headers `Content-Type: application/json` and `Connection: keep-alive` included into the original request.

In case the request comes with:

- `Content-Type`: only `Connection: keep-alive` will be added.
- `Connection`: only `Content-Type: application/json` will be added.
- both `Content-Type` and `Connection`: the original request will be left untouched.

`AllowedRequestCookieCount`: Allowed request cookie count filter

This filter provides a convenient method to set maximum number of allowed cookies on a request. It accepts up to the maximum number of cookies integer value and will respond with a 431 Request Header Fields Too Large error if exceeded.

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: my-gateway-routes
spec:
  service:
    name: myapp
  routes:
    - ssoEnabled: true
      predicates:
        - Path=/api/**
      filters:
```

```
        - AllowedRequestCookieCount=2
```

In the example, request will proceed if it has 2 cookies or less.

`AllowedRequestHeadersCount`: Allowed request headers count filter

This filter provides a convenient method to set the maximum allowed headers in the request coming from our target service through the gateway. It accepts a integer value for the maximum number of headers and if it is exceeded it will respond with a 431 Request header fields too large.

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: my-gateway-routes
spec:
  service:
    name: myapp
  routes:
  - ssoEnabled: true
    predicates:
      - Path=/api/**
    filters:
      - AllowedRequestHeadersCount=4
```

In the example, request will proceed if it has 4 or fewer headers, including cookies.

`AllowedRequestQueryParamsCount`: Allowed request query params count filter

This filter provides a convenient method to set a maximum allowed query parameters of the request coming from target service through the gateway. It accepts a number of maximum query parameters and it's exceeded, it will respond with a 414 URL Too Large HTTP error.

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: my-gateway-routes
spec:
  service:
    name: myapp
  routes:
    - ssoEnabled: true
      predicates:
        - Path=/api/**
      filters:
        - AllowedRequestQueryParamsCount=3
```

In the example, request will proceed if it has 3 query parameters or less.

`BasicAuth`: Basic authorization filter

The `BasicAuth` filter ***relays*** Basic Authorization credentials to a route. It will **not** authenticate requests. It will **not** return a `HTTP 401 Unauthorized` status line with a `WWW-Authenticate` header for

unauthenticated requests.

To use it, you must first store the basic auth username and password in a Kubernetes secret, with their respective keys, `username` and `password`.

This can be done via:

```
kubectl create secret generic basic-auth-secret --from-literal=username=***** --from-l
iteral=password=*****
```

The secret must be in the same namespace as the `SpringCloudGatewayRouteConfig`.

Next, in your `SpringCloudGatewayRouteConfig`, put the name of the secret you created at `spec.basicAuth.secret`.

Finally, add the `BasicAuth` filter to the route.

An example is shown below:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: test-gateway-routes
spec:
  service:
    name: myapp
  basicAuth:
    secret: basic-auth-secret
  routes:
    - predicates:
        - Path=/api/**
      filters:
        - StripPrefix=0
        - BasicAuth
```

If you have multiple routes, the Basic Auth credentials will only be relayed to the routes that include the `BasicAuth` filter.

If the secret cannot be found, the RouteConfig will not be created. A Kubernetes event will be emitted in that namespace, like so:

```
$ kubectl get event
LAST SEEN   TYPE      REASON                         OBJECT
                 MESSAGE
117s        Warning   RoutesDefinitionException   springcloudgatewaymapping/test-gatew
ay-mapping       Failed to retrieve routes from route config in mapping test-gateway-m
apping: Failed to find secret 'basic-auth-secret' in the 'user-namespace' namespace.
```

This will also be logged in the `scg-operator` pod, which is in the `spring-cloud-gateway` namespace by default:

```
$ kubectl logs deployment.apps/scg-operator
2021-06-16 19:38:01.459 ERROR 1 --- [ingController-2] c.v.t.s.route.RoutesDefinitionRe
solver   : Failed to find secret 'basic-auth-secret' in the 'user-namespace' namespace
.
```

Note: The `BasicAuth` filter will not work together with the `TokenRelay` filter as both filters use the `Authorization` header.

`BlockAccess`: Global Filter to block access

This is a Global Filter that provides the ability to block access by ip/domain or JWT claims that apply to all existing routes and requests. As it works globally, it must be activated and composed using configuration properties.

- `spring.cloud.gateway.k8s.block.access.enabled` must be set to `true` to enable this filter

There are three configuration properties to setup the possible blocks:

- By IP/domain:
  - `spring.cloud.gateway.k8s.block.access.domains`: it accepts a list of IPs or domains separated by commas and will block any request coming from the configured values.
- By JWT claims:
  - `spring.cloud.gateway.k8s.block.access.claimValues`: it accepts a list of claim values separated by commas, it will search for the specified values in the JWT Claims and will block any authenticated request with any of the configured claim values.
  - `spring.cloud.gateway.k8s.block.access.claimNames`: is a complementary property to the previous one, it accepts a list of claim names separated by commas and it will search for the specified values in the `claimValues` property in the specified claim names in this property. It will block any authenticated request with any of the configured claim values.

Example using only `claimValues` property:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
  name: my-gateway
spec:
  env:
    - name: spring.cloud.gateway.k8s.block.access.enabled
      value: "true"
    - name: spring.cloud.gateway.k8s.block.access.domains
      value: "192.168.0.1,test.com"
    - name: spring.cloud.gateway.k8s.block.access.claimValues
      value: "client.write,cc_testuser"
```

Will block access if the request comes from `test.com` or the IP `192.168.0.1`, it also will block access if any of the JWT claims contains `client.write` or `cc_testuser` values.

Example using `claimValues` and `claimNames` properties:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
  name: my-gateway
spec:
```

```
  env:
    - name: spring.cloud.gateway.k8s.block.access.enabled
      value: "true"
    - name: spring.cloud.gateway.k8s.block.access.domains
      value: "test.com"
    - name: spring.cloud.gateway.k8s.block.access.claimNames
      value: "sub"
    - name: spring.cloud.gateway.k8s.block.access.claimValues
      value: "write,cc_testuser"
```

Will block access if the request comes from `test.com` and it also will block access if the JWT claims `sub` contains `write` or `cc_testuser` values.

Note: The JWT Claim Block Access global filter only supports the block on API calls with the authentication header, it doesn't support blocking by cookie session.

`CircuitBreaker`: Reroute traffic on error response filter

The `CircuitBreaker` filter provides the ability to reroute a request when an API route is responding with an error response code.

When defining a RouteConfiguration, you can add the `CircuitBreaker` filter by including it in the list of `filters` for the route. For example, you can add a route with a fallback route to forward on error response:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: myapp-route-config
spec:
  service:
    name: myapp
  routes:
  - predicates:
      - Path=/api/**
    filters:
      - CircuitBreaker=myCircuitBreaker,forward:/inCaseOfFailureUseThis
```

You can also add several options for fine tuning:

- A list of status codes that will trigger the fallback behaviour, this can be expressed in number and text format separated by a colon.
- The failure rate threshold above which the circuit breaker will be opened (default 50%, expressed as float value).
- Duration of wait time before closing again (default 60s).

```
      - CircuitBreaker=myCircuitBreaker,forward:/inCaseOfFailureUseThis,401:NOT_FOUND:
500
```

```
      - CircuitBreaker=myCircuitBreaker,forward:/inCaseOfFailureUseThis,401:NOT_FOUND:
500,10,30s
```

Circuit breaker status

By querying for the circuit breaker metrics, you can monitor the status of the circuit breaker:

```
actuator/metrics/resilience4j.circuitbreaker.state?tag=state:{circuit-breaker-state}&t
ag=name:{circuit-breaker-name}
```

- where `{circuit-breaker-state}` is one of `closed`, `disabled`, `half_open`, `forced_open`, `open`, `metrics_only`
- where `{circuit-breaker-name}` is the name of your circuit breaker, e.g. `myCircuitBreaker`

The metrics endpoint will return a value of `1` in the `$.measurements[].value` JSON path if the circuit breaker is in this state.

For more more information and other metrics, see Resilience4j CircuitBreaker Metrics.

`ClaimHeader`: Passing JWT claims header filter

The `ClaimHeader` filter allows passing a JWT claim value as an HTTP Header. It works both with and without SSO enabled, with the consideration that when SSO is not enabled the JWT token is expected in `Authorization` Header and won't be validated.

This filter is useful in scenarios where the target service does not handle JWT authorization, but still needs some piece of information from the JWT token.

The `ClaimHeader` filter configuration requires 2 parameters:

- Claim name: case sensitive name of the claim to pass.
- Header name: name of the HTTP

The following configurations shows how to extract the claim Subject and pass in an HTTP Header called `X-Claim-Sub`.

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: myapp-route-config
spec:
  service:
    name: myapp
  routes:
  - predicates:
      - Path=/api/**
    filters:
      - ClaimHeader=sub,X-Claim-Sub
```

If you need to pass more than one claim, simply apply the filter repeatedly.

```
    filters:
      - ClaimHeader=sub,X-Claim-Sub
      - ClaimHeader=iss,X-Claim-Iss
      - ClaimHeader=iat,X-Claim-Iat
```

Note: In case the header is already present, the value(s) from the claim will be added to it. That is, previous values sent in the SCG request will be preserved.

`ClientCertificateHeader`: Validate client certificate filter

The `ClientCertificateHeader` filter validates the client SSL certificate used to make a request to an app through the Gateway. You can also use this filter to validate the client certifcate's fingerprint.

Note: This filter relies on Kubernetes container's ability to recognize a client certificate's Certificate Authority (CA).

When adding a route to a Gateway service instance, you can add the `ClientCertificateHeader` filter by including it in the list of `filters` for the applicable route.

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: myapp-route-config
spec:
  service:
    name: myapp
  routes:
  - predicates:
      - Path=/api/**
    filters:
      - ClientCertificateHeader=*.example.com
```

To validate the client SSL certificate's fingerprint, add the name of the hash used for the fingerprint, and the fingerprint value, after the CN, using the following format:

```
[CN],[HASH]:[FINGERPRINT]
```

where:

- `[CN]` is the Common Name
- `[HASH]` is the hash used for the fingerprint, either `sha-1` or `sha-256`
- `[FINGERPRINT]` is the fingerprint value

The following example uses the `ClientCertificateHeader` filter to ensure that a client certificate uses a CN of `*.example.com` and a SHA-1 fingerprint of `aa:bb:00:99`:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: myapp-route-config
spec:
  service:
    name: myapp
  routes:
  - predicates:
      - Path=/api/**
```

```
    filters:
      - ClientCertificateHeader=*.example.com,sha-1:aa:bb:00:99
```

The fingerprint value is not case-sensitive, and the colon character `:` is not required to separate hexadecimal digits in a fingerprint. The following example works too:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: myapp-route-config
spec:
  service:
    name: myapp
  routes:
  - predicates:
      - Path=/api/**
    filters:
      - ClientCertificateHeader=*.example.com,sha-1:AABB0099
```

`FallbackHeaders`: Allows adding `CircuitBreaker` exception details in the headers before forwarding

The `FallbackHeaders` filter provides the ability to add `CircuitBreaker` execution exception details in the headers of a request forwarded to a fallback route in an external application

When defining a RouteConfiguration, you can add the `FallbackHeaders` filter by including it in the list of `filters` for the fallback route. For example, you can add the fallback route to add `X-Exception`:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: myapp-route-config
spec:
  service:
    name: myapp
  routes:
  - predicates:
      - Path=/api/**
    filters:
      - CircuitBreaker="myCircuitBreaker,forward:/inCaseOfFailureUseThis"
  - uri: http://localhost:9994
    predicates:
      - Path=/fallback
    filters:
      - FallbackHeaders
```

You can optionally configure just the executionExceptionTypeHeaderName by editing the filter above like:

```
    filters:
     - FallbackHeaders= My-Execution-Exception-Type
```

Or change all executionExceptionTypeHeaderName, executionExceptionMessageHeaderName, rootCauseExceptionTypeHeaderName using the following modification

```
    filters:
     - FallbackHeaders= My-Execution-Exception-Type, My-Execution-Exception-Message, M
y-Root-Cause-Exception-Type
```

`Cors`: Configuring per-route Cross-Origin Resource Sharing (CORS) behavior

You can define CORS behavior on a route with the `Cors` filter, instead of configuring it on the gateway.

In this example, the `allowedOrigins` is set to `https://example.com`, and the `allowedMethods` are GET, POST, DELETE.

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: my-gateway-routes
spec:
  routes:
    - uri: https://httpbin.org
      predicates:
        - Path=/get/**
      filters:
        - Cors=[allowedOrigins:https://example.com,allowedMethods:GET;POST;DELETE]
```

The following table describes the parameters you can configure on the filter:

Parameter	Function	Example
`allowedOrigins`	Allowed origins to make cross-site requests. The special value "*" allows all domains. These values will be combined with the values from allowedOriginPatterns.	`Cors=[allowedOrigins:https://example.com]`
`allowedOriginPatterns`	Alternative to allowedOrigins that supports more flexible origins patterns with "*" anywhere in the host name in addition to port lists. These values will be combined with the values from allowedOrigins.	`Cors=[allowedOriginPatterns:https://*.test.com:8080]`
`allowedMethods`	Allowed HTTP methods on cross-site requests. The special value "*" allows all methods. If not set, "GET" and "HEAD" are allowed by default.	`Cors=[allowedMethods:GET;PUT;POST]`
`allowedHeaders`	Allowed headers in cross-site requests. The special value "*" allows actual requests to send any header.	`Cors=[allowedHeaders:X-Custom-Header]`
`maxAge`	How long, in seconds, the response from a pre-flight request can be cached by clients.	`Cors=[maxAge:300]`
`allowCredentials`	Whether user credentials are supported on cross-site requests. Valid values: `true`, `false`.	`Cors=[allowCredentials:true]`
`exposedHeaders`	HTTP response headers to expose for cross-site requests.	`Cors=[exposedHeaders:X-Custom-Header]`

`JwtKey`: Multiple client JWT validation filter

The `JwtKey` filter allows validating JSON Web Tokens (JWT) generated by different providers with

different signature keys. It is expected that every request has a key id that allows identifying which key validates the token signature.

SpringCloudGateway integrates with Vault on Kubernetes and assumes a Vault Agent Injector has been deployed to the cluster. This filter requires additional Vault integration parameters defined in the custom resource to be enabled in SpringCloudGateway. The required parameters are `serviceAccount.name`, `extensions.secretsProviders`, and `jwtKey.enabled` alongside `jwtKey.secretsProviderName` where:

- `serviceAccount.name` is the name of the ServiceAccount used by the gateway instances
- `extensions.secretsProviders` is the element from which keys will be obtained
  - `name` is an arbitrary name to be referenced later on `jwtKey` configuration
  - `vault.roleName` is the Vault role with read access to the secrets (according to Vault policies configured)
  - `vault.path` (optional) is the secret's full path in Vault. For example, for keys `my-secrets/scg/keys/123...` and `my-secrets/scg/keys/456...`, path must be `my-secrets/scg/keys`.
  - `vault.authPath` (optional) is the authentication path for Vault's Kubernetes auth method. For example, `/auth/cluster-1-auth`, `/auth/cluster-2-auth`. If not set, secrets are expected to be under `jwt-keys-for-vmware-tanzu/{namespace}-{gateway_name}`
- `jwtKey.enabled` is the flag indicating that the Vault integration is enabled
- `jwtKey.secretsProviderName` is the vault secrets provider name defined previously

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
  name: mygateway
spec:
  serviceAccount:
    name: scg-service-account

  extensions:
    secretsProviders:
      - name: vault-jwt-keys
        vault:
          roleName: scg-role
    filters:
      jwtKey:
        enabled: true
        secretsProviderName: vault-jwt-keys
```

Secrets within Vault must follow this structure:

- `secret full name` is the path where the secrets are held. Unless `path` property is set in the `secretProvider` it must be composed of `jwt-keys-for-vmware-tanzu/{namespace}-{gateway_name}/{kid}`
- `kid` is the key id to uniquely identify the public key (`RSA`) or the private key (`HMAC`). This kid should match the value obtained in the *key id location*

- `alg` is the algorithm used to encrypt the public key (currently supporting `RSA` only) or the private key (`HS256`, `HS384`, or `HS512`)
- `key` is the actual public key, as a PEM format (supporting both `CERTIFICATE` and `PUBLIC KEY` formats), or private key with at least 32 bytes in length

RSA:

```
vault kv put jwt-keys-for-vmware-tanzu/customnamespace-mygateway/client_0 \
              kid="client_0" \
              alg="RSA" \
              key="-----BEGIN CERTIFICATE-----\
              MIIBIyEpEBgkqhkiG9w ..."
```

HMAC:

```
vault kv put jwt-keys-for-vmware-tanzu/customnamespace-mygateway/client_0 \
              kid="client_0" \
              alg="HS256" \
              key="Key-Must-Be-at-least-32-bytes-in-length!"
```

Note: If you need to add, remove or just update a key in Vault, you can use any of Vault supported methods (HTTP API, CLI...) Every interaction will update the keys in the gateway after no more than 5 minutes.

When defining a RouteConfiguration, you can add the `JwtKey` filter by including it in the list of `filters` for the route.

The configuration provided to the JwtKey filter indicates the *key id location*. This *key id location* describes whether the key id is found in an HTTP header or in a JWT claim or header value, with the following syntax:

- `- JwtKey={header:X-JWT-KEYID}` the key id location is expected to be in an **HTTP header** named `X-JWT-KEYID`
- `- JwtKey={claim:kid}` the key id location is expected to be in a **JWT claim** or **header** named `kid`

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: myapp-route-config
spec:
  service:
    name: myapp
  routes:
  - predicates:
      - Path=/api/**
    filters:
      - JwtKey={header:X-JWT-KEYID}
```

Once the gateway is up and running you can see the loaded keys in the info endpoint. Each key is shown with its id and the time when it was last modified.

```
"jwtkeys": [
  {
    "id": "client_0"
    "lastRefreshTime": "2021-09-07T07:57:01+0000",
  }
]
```

`ApiKey`: API key validation filter

The `ApiKey` filter allows validating API keys generated by API portal for VMware Tanzu 1.1.0. It is expected that every request has a `X-API-Key` header specified that allows the filter to validate against the hashed value stored in Hashicorp Vault.

SpringCloudGateway integrates with Vault on Kubernetes and assumes a Vault Agent Injector has been deployed to the cluster. This filter requires additional Vault integration parameters defined in the custom resource to be enabled in SpringCloudGateway. The required parameters are `serviceAccount.name`, `extensions.secretsProviders`, and `apiKey.enabled` alongside `apiKey.secretsProviderName` where:

- `serviceAccount.name` is the name of the ServiceAccount used by the gateway instances
- `extensions.secretsProviders` is the element from which keys will be obtained
  - `name` is an arbitrary name to be referenced later on `apiKey` configuration
  - `vault.roleName` is the Vault role with read access to the secrets (according to Vault policies configured)
  - `vault.path` (optional) is the same vault path you configured when setting up API key management in API portal. If not set, the value will be `api-portal-for-vmware-tanzu` by default
- `apiKey.enabled` is the flag indicating that the API key validation on all requests is enabled
- `apiKey.secretsProviderName` is the vault secrets provider name defined previously

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
  name: mygateway
spec:
  api:
    groupId: my-group-id

  serviceAccount:
    name: scg-service-account

  extensions:
    secretsProviders:
      - name: vault-api-keys
        vault:
          roleName: api-key-role
          path: my-api-portal
    filters:
      apiKey:
        enabled: true
        secretsProviderName: vault-api-keys
```

For the example configuration above, to ensure access to the vault path, you need to configure your Hashicorp Vault instance:

1. Create a Vault access policy to API portal path for the Gateway, including your gateway `groupId` (see the configuring instances section for more details)

   ```
   $ vault policy write scg-policy - <<EOF
     path "my-api-portal/data/my-group-id" {
       capabilities = ["read"]
     }
     path "my-api-portal/metadata/my-group-id" {
       capabilities = ["list"]
     }
     EOF
   ```

   The sample command above uses `scg-policy` as the name. You may use a different name for the policy, just make sure you use the same policy name in next step.

2. Create a role that binds a namespaced service account to that policy, following Kubernetes Auth Method:

   ```
   $ vault write auth/kubernetes/role/api-key-role \
     bound_service_account_names=scg-service-account \
     bound_service_account_namespaces=scg-namespace \
     policies=scg-policy \
     ttl=24h
   ```

   The `bound_service_account_namespaces` above needs to be the namespace where you install your Spring Cloud Gateway instance, and the `bound_service_account_names` needs to refer to a service account in the same namespace.

After applying the configuration, all routes defined in the `SpringCloudGatewayRouteConfig` will require the `X-API-Key` header to be accessed.

For example using an HTTP client such as HTTP or cURL:

```
  $ http GET my-gateway.my-example-domain.com/github X-API-Key:{my-api-key}
  $ curl -X GET my-gateway.my-example-domain.com/github --header "X-API-key:{my-api-ke
y}"
```

If you want to double-check that API key management is enabled and that keys have been loaded, you can visit `actuator/info` endpoint which should display:

```
apikey:
    enabled: true
    loaded: true
```

`RateLimit`: Limiting user requests filter

The `RateLimit` filter limits the number of requests allowed per route during a time window.

When defining a RouteConfiguration, you can add the `RateLimit` filter by including it in the list of `filters` for the route. The filter accepts 4 options:

- Number of requests accepted during the window.
- Duration of the window: by default milliseconds, but you can use `s`, `m` or `h` suffix to specify it in seconds, minutes or hours.
- (Optional) User partition key: it's also possible to apply rate limiting per user, that is, different users can have its own throughput allowed based on an identifier found in the request. Set whether the key is in a JWT claim or HTTP header with '' or '' syntax.
- (Optional) It is possible to rate limit by IP addresses. Note, this cannot be combined with the rate limiting per user.

For example, you can add a route to limit all users to one request every 10 seconds:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: myapp-route-config
spec:
  service:
    name: myapp
  routes:
  - predicates:
      - Path=/api/**
    filters:
      - RateLimit=1,10s
```

Provided you are within the allowed limits, the response will succeed and report the number of accepted request you can still do in the `X-Remaining` HTTP header. When the limit is exceeded, response will fail with *429 Too Many Requests* status, and inform the remaining time until a request will be accepted in `X-Retry-In` HTTP header (in milliseconds)

If you want to expose a route for different sets of users, each one identified by its own `client_id` HTTP header, use:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: myapp-route-config
spec:
  service:
    name: myapp
  routes:
  - predicates:
      - Path=/api/**
    filters:
      - RateLimit=1,10s,{header:client_id}
```

The rate limit `1,10s` will be applied individually for each set of users. When the header (or claim) is not present access will be rejects with a simple *429 Too Many Requests* response (without additional headers).

Limiting by IP Address

Rate limiting by IP address can accept a multiple IP addresses, seperated by a semi-colon.

When rate limiting by IP address, the filter checks the `X-Forwarded-For` header, if present, for the IP. As there can be multiple IPs added to this header, you can optionally set the max trusted index to read from, by setting this as the first value.

The default value of `1` will read the last IP from the header, while a value of `2` will read the second last IP, and so on. The index must be greater than zero.

Here is an example to rate limit by IP address:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: myapp-route-config
spec:
  service:
    name: myapp
  routes:
  - predicates:
      - Path=/api/**
    filters:
      - RateLimit=2,10s,{IPs:2;127.0.0.1;192.168.0.1}
```

In the example above, the max trusted index is set to `2`. If the `X-Forwarded-For` header had a value of `4.4.4.4, 8.8.8.8, 127.0.0.1`, the gateway would return 403 forbidden because the second-last IP, `8.8.8.8`, is not in the allowed IPs. However, if the header was set to `4.4.4.4, 127.0.0.1, 8.8.8.8`, the gateway will return successfully.

Note: If you are using an ingress, ensure it is configured to pass the incoming `X-Forwarded-For` header upstream to the gateway.

`RemoveJsonAttributesResponseBody`: Response body modification filter

This filter provides a convenient method to apply a transformation to JSON body content from target service through the gateway. It accepts a list of attribute names to search for and an optional last parameter from the list can be a boolean to remove the attributes just at root level (default value if not present at the end of the parameter configuration, `false`) or recursively (`true`).

Note: Applying the recursive deletion mode on a large JSON data will affect on service latency.

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: my-gateway-routes
spec:
  service:
    name: myapp
  routes:
  - ssoEnabled: true
    predicates:
```

```
      - Path=/api/**
    filters:
      - RemoveJsonAttributesResponseBody=origin,foo,true
```

In the example, the attributes `origin` and `foo` will be deleted from the JSON content body at any level.

`RewriteAllResponseHeaders` Response headers modification filter

This filter provides a convenient method to apply a transformation to all headers coming from target service through the gateway. It accepts a regular expression to search for in header values and text to replace the matching expression with.

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: my-gateway-routes
spec:
  service:
    name: myapp
  routes:
  - ssoEnabled: true
    predicates:
      - Path=/api/**
    filters:
      - RewriteAllResponseHeaders=\d,0
```

In the example, any header value containing a number (`\d` matches any number from 0 to 9) will be replaced by 0.

`RewriteResponseBody`: Response body modification filter

This filter provides a convenient method to apply a transformation to any body content from target service through the gateway, it won't apply any transformation to response headers. It accepts a list of regular expressions (separated by commas) to search for in value and text to replace the matching expression with (separated by colon).

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: my-gateway-routes
spec:
  service:
    name: myapp
  routes:
  - ssoEnabled: true
    predicates:
      - Path=/api/**
    filters:
      - RewriteResponseBody=foo:bar,/path-one/:/path-two/
```

In the example, in a body content of any type:

- `foo` will be replaced by `bar`

- `/path-one/` will be replaced by `/path-two/`

`RewriteJsonAttributesResponseBody`: Response body JSON modification filter

This filter provides a convenient method to apply a transformation to JSON content from target service through the gateway using JSON Path notations. It accepts a list of elements (separated by commas) where the first parameter is the selector of the JSON node and the second one is the value to set into that JSON node, those two parameters must be separated by colon.

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: my-gateway-routes
spec:
  service:
    name: myapp
  routes:
  - ssoEnabled: true
    predicates:
      - Path=/api/**
    filters:
      -  RewriteJsonAttributesResponseBody=slides[1].title:Welcome,date:11-11-2022
```

Given the following JSON in a body content:

```
{
   "date":"01-01-2022 11:00",
   "slides":[
      {
         "title":"Presentation",
         "type":"all"
      },
      {
         "title":"Overview",
         "type":"image"
      }
   ],
   "title":"Sample Title"
}
```

Applying the example:

- `date` at root level will be replaced by `11-11-2022`
- `title` at second element of the `slides` array will be replaced by `Welcome`

```
{
   "date":"11-11-2022",
   "slides":[
      {
         "title":"Presentation",
         "type":"all"
      },
      {
         "title":"Welcome",
```

```
            "type":"image"
         }
      ],
      "title":"Sample Title"
}
```

`Roles`: Role-based access control filter

Similarly to scope-based access control, it's possible to use custom Claim properties to apply role-base access control with the `Roles` filter. Furthermore, if you are not using SSO feature, you can still use role-based control provided you apply JwtKey filter.

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: my-gateway-routes
spec:
  service:
    name: myapp
  routes:
  - ssoEnabled: true
    predicates:
      - Path=/api/**
    filters:
      - Roles=role_01,role_02
```

By default, SpringCloudGateway will check the role values under the `roles` claim, but you can change it using `spec.sso.roles-attribute-name` property in the Gateway. SCG expects the roles claim to be an array (`roles = ["user-role-1", "user-role-2"]`), but a single string is also accepted when role only contains one value (`roles = "user-role"`).

Additionally, `spec.sso.roles-attribute-name` also supports nested JSON values like `custom-data.user.roles`.

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
  name: mygateway
spec:
  sso:
    roles-attribute-name: my-roles
```

`Scopes`: Scope-based access control filter

When SSO is enabled, you can add fine-tune access control based on OIDC scopes by adding the `Scopes` filter.

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: my-gateway-routes
spec:
  service:
```

```
    name: myapp
  routes:
  - ssoEnabled: true
    predicates:
      - Path=/api/**
    filters:
      - Scopes=api.read,api.write,user
```

`StoreIpAddress`: Store IP address filter

This filter provides a convenient method to store the IP address of the request coming from target service through the gateway, it can be useful for tracing purposes. It accepts a parameter name under which to store the IP.

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: my-gateway-routes
spec:
  service:
    name: myapp
    filters:
      - StoreIpAddress=ipAddress
```

In the example, will store the IP address in the context of the application under `ipAddress` attribute. Attributes can be pulled implementing a custom extension:

```
((exchange, chain) -> {
    String attribute = exchange.getAttributeOrDefault("ipAddress", "Attribute not foun
d");
    ...
    return chain.filter(exchange);
});
```

`StoreHeader` Store headers filter

This filter provides a convenient method to populate a header value into the context of the application coming from target service through the gateway, it can be useful for tracing purposes. It accepts a list of header names to search for and a last parameter with the attribute name under which want to store the header value. It's important to highlight that the list of header names must be in order of priority, once it finds one header, it stops looking for the rest and includes it in the context of the application under the last parameter received.

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: my-gateway-routes
spec:
  service:
    name: myapp
    filters:
      - StoreHeader=x-tracing-header,custom-id,x-custom-id,tracingParam
```

In the example, will search for `x-tracing-header`, `custom-id` and `x-custom-id` and once it finds one, it will store its value on the application context under `tracingParam` attribute. Attributes can be pulled implementing a custom extension:

```
((exchange, chain) -> {
    List<String> attributes = exchange.getAttributeOrDefault("tracingParam", Collectio
ns.emptyList());
    ...
    return chain.filter(exchange);
});
```

`SsoAutoAuthorize`: SSO auto-authorized credentials filter

This filter must be applied **only for development purposes**, it accepts a list of roles or scopes (separated by commas) to inject a fake SSO authorization with those authorities associated.

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: my-gateway-routes
spec:
  service:
    name: myapp
    filters:
      - SsoAutoAuthorize=SCOPE_test,ROLE_test
```

Additional configuration is required to reduce the change that this local development utility is not deployed to production environments:

- System property (`JAVA_OPTS` property) `com.vmware.tanzu.springcloudgateway.dev.mode.enabled` must be set to `true`.
- Configuration property `com.vmware.tanzu.springcloudgateway.sso.auto.authorize.enabled` must be set to `true`.

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
  name: my-gateway
spec:
  env:
    - name: com.vmware.tanzu.springcloudgateway.sso.auto.authorize.enabled
      value: "true"
  java-opts: "-Dcom.vmware.tanzu.springcloudgateway.dev.mode.enabled=true"
```

Note: If no SSO configuration is present, you will need to create a dummy configuration activating an SSO profile and setting a valid issuer uri, for example the Google Issuer URL (`https://accounts.google.com`).

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
```

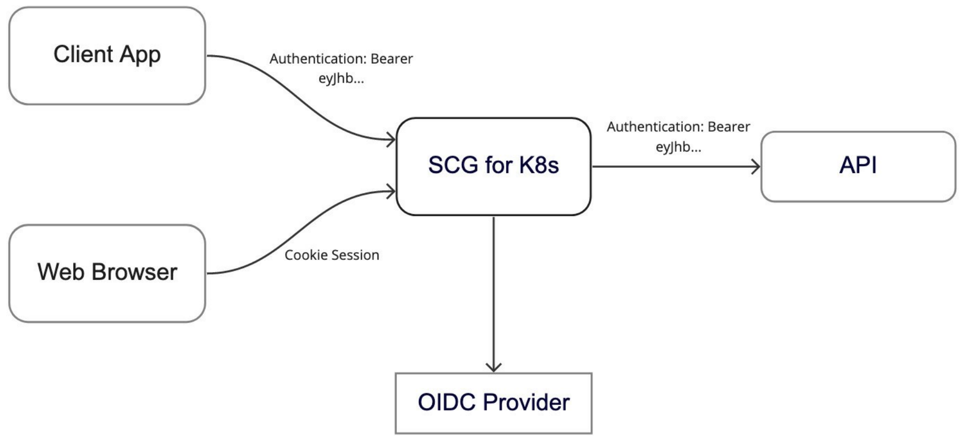
```
  name: my-gateway
spec:
  env:
    - name: spring.profiles.include
      value: "sso"
    - name: spring.security.oauth2.client.provider.sso.issuer-uri
      value: "https://accounts.google.com"
    - name: com.vmware.tanzu.springcloudgateway.sso.auto.authorize.enabled
      value: "true"
  java-opts: "-Dcom.vmware.tanzu.springcloudgateway.dev.mode.enabled=true"
```

`TokenRelay`: Passing user identity filter

A Token Relay is where an OAuth2 or OIDC consumer acts as a Client and forwards the incoming token to outgoing resource requests. In this case, the consumer can be any service accessible from any of the configured routes with `ssoEnabled: true`.

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: my-gateway-routes
spec:
  service:
    name: myapp
  routes:
  - ssoEnabled: true
    tokenRelay: true
    predicates:
      - Path=/api/**
```

When enabling `TokenRelay`, Spring Cloud Gateway for Kubernetes will pass the currently-authenticated user's identity token to the app when the user accesses the app's route.



Note: The `TokenRelay` filter will not work together with the `BasicAuth` filter as both filters use the `Authorization` header.

Commercial Route Predicates

The open-source Spring Cloud Gateway project includes a number of built-in predicates for use in Gateway routes. Spring Cloud Gateway provides a number of custom predicates in addition to those included in the OSS project.

Predicates Included In Spring Cloud Gateway OSS

Predicates in Spring Cloud Gateway OSS can be used in Spring Cloud Gateway for Kubernetes. Spring Cloud Gateway OSS includes a number of `RoutePredicate` factories used to create predicates for routes. For a complete list of these factories, see the Spring Cloud Gateway OSS documentation.

Predicates Added In Spring Cloud Gateway for Kubernetes

Following sections offers information about the custom predicates added in VMware Spring Cloud Gateway and how you can use them.

Match on JWT claim value: `JWTClaim` Predicate

When JWT token is in an HTTP header it is possible to match a route against a claim's value.

The predicate reads the token directly without any manipulation or validation at the beginning of the request processing. This means you don't require to enable SSO on the gateway but the token signature is not validated unless SSO is enabled or specific filter is applied (for example, JwtKey). While it's not mandatory to send the header in `Authorization` header it must comply with `Bearer {token}` format.

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: my-gateway-routes
spec:
  service:
    name: myapp
  routes:
  - ssoEnabled: true
    predicates:
      - Path=/api/**
      - JWTClaim=Authorization,sub,.+@my.org$
```

In the example, a request must match '/api/**' as well as contain a token whose subject ends with '@my.org' in order to be routed.

Custom Extensions

These topics describe how to develop and configure custom extensions for Spring Cloud Gateway for Kubernetes.

Developing Extensions

This page will explain how to develop a custom Extension for Spring Cloud Gateway for Kubernetes.

Prerequisites

A Gateway Extension is a JAVA JAR package with classes that enhance SCG for Kubernetes features by adding custom Spring Cloud Gateway Filter and Predicate factories, as well as Global Filters.

The requirements to build one are:

- Java 11 to 17 compatible.
- Spring Configuration classes must be under package `com.vmware.scg.extensions`.

Project setup

You can use any IDE and build system provided you have the appropriate dependencies and packaging setup.

Gradle

1. Initialize the Gradle project for a Java library with a Groovy build script. Make sure you set the source package to `com.vmware.scg.extensions`.

```
$ gradle init
```

2. Update the `build.gradle` file for your extension library

```
plugins {
  id 'java-library'
}

group = 'com.vmware.scg.extensions'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '11'

repositories {
  mavenCentral()
}

ext {
  set('springCloudVersion', "2021.0.3")
  set('springBootVersion', "2.7.5")
}

dependencies {
  implementation platform("org.springframework.boot:spring-boot-dependenc
ies:${springBootVersion}")
  implementation platform("org.springframework.cloud:spring-cloud-depende
ncies:${springCloudVersion}")
  implementation 'org.springframework.cloud:spring-cloud-starter-gateway'
  /* Not required for the sample app but will be useful for more complex
extensions
  implementation 'org.springframework.boot:spring-boot-starter-oauth2-cli
ent'
  implementation 'org.springframework.boot:spring-boot-starter-security'
      * /
```

```
  testImplementation 'org.springframework.boot:spring-boot-starter-test'
  // Not required for the sample app but will be useful for more complex
extensions
  // testImplementation 'org.springframework.security:spring-security-tes
t'
  testImplementation 'com.github.tomakehurst:wiremock:2.27.2'
 }

 test {
  useJUnitPlatform()

  testLogging {
   exceptionFormat = 'full'
       }
 }
```

3. Delete any `.java` files created by the generator.

Note: While other versions may work for development, only Spring Boot version 2.5.x and Spring Cloud 2020.0.4 are fully supported for runtime.
It's safe to add other dependencies, provided they don't cause classpath issues with the current ones. However, it's not recommended to overload the extensions given the possible impact in resources and performance.

Maven

1. Generate a Maven library archetype. Make sure you set the `groupId` to `com.vmware.scg.extensions`.

```
$ mvn archetype:generate -DgroupId=com.vmware.scg.extensions -DarchetypeArtifa
ctId=maven-archetype-quickstart
```

2. Update the `pom.xml` file for your extension library

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.or
g/2001/XMLSchema-instance"
   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://m
aven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
   <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
     <version>2.7.5</version>
    <relativePath/> <!-- lookup parent from repository -->
   </parent>
  <groupId>com.vmware.scg.extensions</groupId>
  <artifactId>mycustomfilter</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>mycustomfilter</name>
  <description>SCG for K8s extension</description>
   <properties>
    <java.version>11</java.version>
    <spring-cloud.version>2021.0.3</spring-cloud.version>
   </properties>
```

```
  <dependencies>
    <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
    </dependency>
   <!-- Not required for the sample app but will be useful for mor
e complex extensions
    <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-client</artifact
Id>
    </dependency>
    <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
        -->

    <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
      <scope>test</scope>
    </dependency>
   <!-- Not required for the sample app but will be useful for mor
e complex extensions
    <dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-test</artifactId>
      <scope>test</scope>
    </dependency>
        -->
    <dependency>
    <groupId>com.github.tomakehurst</groupId>
     <artifactId>wiremock</artifactId>
      <version>2.27.2</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <dependencyManagement>
    <dependencies>
        <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-dependencies</artifact
Id>
      <version>${spring-cloud.version}</version>
         <type>pom</type>
        <scope>import</scope>
       </dependency>
    </dependencies>
  </dependencyManagement>
 </project>
```

3. Delete any `.java` files created by the generator.

Note: While other versions may work for development, only Spring Boot version 2.7.5 and Spring Cloud 2021.0.3 are fully supported for runtime.
It's safe to add other dependencies, provided they don't cause classpath issues with the current ones. However, it's not recommended to overload the extensions given

the possible impact in resources and performance.

Custom Extension Example

The following is a simple example of a custom extensions that adds an HTTP header to the request sent to the target service. This will cover the basic development concepts as well as testing to get you started.

For more in-depth information (for example, implementing custom predicates or custom configurations), refer to Spring Cloud Gateway Developer Guide.

Custom Filter example code

You can start creating a custom filter like this.

```
package com.vmware.scg.extensions.filter;

import java.util.List;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

import org.springframework.cloud.gateway.filter.GatewayFilter;
import org.springframework.cloud.gateway.filter.factory.AbstractGatewayFilterFactory;
import org.springframework.stereotype.Component;
import org.springframework.web.server.ServerWebExchange;

@Component
public class AddMyCustomHeaderGatewayFilterFactory
  extends AbstractGatewayFilterFactory<Object> {

    private static final Logger LOGGER = LoggerFactory.getLogger(AddMyCustomHeaderGate
wayFilterFactory.class);

 private static final String MY_HEADER_KEY = "X-My-Header";

  @Override
 public GatewayFilter apply(Object config) {
   return (exchange, chain) ->
                {
     ServerWebExchange updatedExchange
             = exchange.mutate()
                        .request(request -> {
                              request.headers(head
ers -> {
                                         headers.put(
MY_HEADER_KEY, List.of("my-header-value"));
                                         LOGGER.info(
"Processed request, added" + MY_HEADER_KEY + " header");
                                              });
                                           })
                               .build();
     return chain.filter(updatedExchange);
          };
        }
}
```

In the code, you can see that:

- We named the filter `AddMyCustomHeaderGatewayFilterFactory` this will make it available as `AddMyCustomHeader` under the route configurations. Ensure your extension name does not collide with any of the existing predicates or filters.
- The filter will be automatically detected using `@Component` annotation, but for complex configurations you can use normal Spring `@Configuration` classes.
- Since we do not require any special configuration, extending `AbstractGatewayFilterFactory` with `Object` is enough.
- Inside the `apply` method we only need to add our header. In this simple example we are adding it always, but you could be more creative. For example, changing the response status with `exchange.getResponse().getStatusCode()` and adapting the exchange response.
- We add a normal `org.slf4j.Logger` to provide traces, these have no special requirements and will appear in the pod logs.

Testing

To test the extension we can use Spring Boot conventional tools without needing much heavy lifting or Kubernetes.

First, add the test dependency `com.github.tomakehurst:wiremock:2.27.2` or higher to you project. We will use `WireMockServer` to simulate a service that responds to routed traffic, and also to assert what the service receives.

Next, create a test class like this one:

```
package com.vmware.scg.extensions;

import com.github.tomakehurst.wiremock.WireMockServer;
import com.github.tomakehurst.wiremock.matching.EqualToPattern;
import org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.TestInstance;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.test.autoconfigure.web.reactive.AutoConfigureWebTestCl
ient;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.web.reactive.server.WebTestClient;

import static com.github.tomakehurst.wiremock.client.WireMock.get;
import static com.github.tomakehurst.wiremock.client.WireMock.getRequestedFor;
import static com.github.tomakehurst.wiremock.client.WireMock.ok;
import static com.github.tomakehurst.wiremock.client.WireMock.urlPathEqualTo;

@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT)
@AutoConfigureWebTestClient
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
class AddMyCustomHeaderTest {
```

```
    final WireMockServer wireMock = new WireMockServer(9090);

    @Autowired
 WebTestClient webTestClient;

 @BeforeAll
 void setUp() {
  wireMock.stubFor(get("/add-header").willReturn(ok()));
   wireMock.start();
       }

 @AfterAll
 void tearDown() {
   wireMock.stop();
       }

  @Test
 void should_apply_extension_filter() {
       webTestClient
               .get()
        .uri("/add-header")
          .exchange()
         .expectStatus()
            .isOk();

  wireMock.verify(getRequestedFor(urlPathEqualTo("/add-header"))
     .withHeader("X-My-Header", new EqualToPattern("my-head
er-value")));
       }

 @SpringBootApplication
 public static class GatewayApplication {

  public static void main(String[] args) {
   SpringApplication.run(GatewayApplication.class, args);
              }
       }
}
```

Finally, add this configuration to your `application.yaml` under **test** resources.

Note: External configuration files under 'src/main/resources' are not supported yet and may cause issues.

```
spring:
  cloud:
    gateway:
      routes:
        - uri: http://localhost:9090
          predicates:
            - Path=/add-header/**
          filters:
            - StripPrefix=0
            - AddMyCustomHeader
```

In the code above, you can see that:

- Since we are building a library, we need to create a fake Spring Boot app `GatewayApplication` to initialize a basic context.
- The test configuration `application.yaml` creates a basic routing configuration to apply our extension `AddMyCustomHeader`.
- We are initializing `WebTestClient` with `@AutoConfigureWebTestClient` for both REST calls and assertions.

After building the plugin jar file with either `./gradle clean build` or `mvn clean package`, head to Configuring Extensions to fully deploy the extension in a SCG for K8s instance.

Configuring Extensions

This page will explain how to configure and deploy an extension in Spring Cloud Gateway for Kubernetes. If you have doubts about the development process, refer to Extensions Development.

Extensions can be added with two simple steps to any Gateway Instance, including those already running. In short, you just need to create a Kubernetes ConfigMap and enable it in the desired Gateway Instance.

Prerequisites

The requirements for these steps are:

- SCG for K8s packaged extension (in JAR)
- Docker command line tool if packaging extension as a OCI image
- Kubernetes cluster

Extension Deployment

Extensions can be stored in OCI Images, Kubernetes ConfigMaps or Volumes.

OCI Images and ConfigMaps provide a simple and easy to use approach for deploying custom extensions. Using a ConfigMap has an advantage that SCG for K8s can detect when they change and update automatically. Using an OCI image can be easier to automate, as it only requires building an image from a Dockerfile and pushing it to a registry. ConfigMaps have a 1MB size limit restriction, for bigger extensions you can use OCI images or Persistent Volumes. Volumes avoid size limitations but have several restrictions and depend heavily on K8s and storage implementations provided in your environment.

Extensions from ConfigMaps

Provided you have the JAR package of less than 1MB, simply create a ConfigMap as follows:

```
$ kubectl create configmap extension-name --from-file=extension.jar -n gateway_names
pace
```

The config map name will identify the extension later.

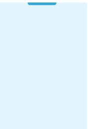
You can confirm that the ConfigMap was successfully created with the contents of the jar:

```
$  kubectl get configmap extension-name -o yaml
```

You will see something that looks like this:


```
apiVersion: v1
binaryData:
  mycustomfilter-0.0.1-SNAPSHOT.jar: UEsDBAoAAAgIABV491IAAAAAAgAAAAAAAAAJAAAATUVUQS1JT
kYvAwBQSwMECgAACAgAFXj3UrJ/Au4bAAAAGQAAABQAAABNRVRBLUlORi9NQU5JRkVTVC5NRvNNzMtMSy0u0Q1
LLSrOzM+zUjDUM+Dl4uUCAFBLAwQKAAAICAAVePdSAAAAAAIAAAAAAAAABAAAAGNvbS8DAFBLAwQKAAAICAAVe
PdSAAAAAAIAAAAAAAACwAAAGNvbS92bXdhcmUvAwBQSwMECgAACAgAFXj3UgAAAAACAAAAAAAAAA8AAABjb20
vdm13YXJlL3NjZy8DAFBLAwQKAAAICAAVePdSAAAAAAIAAAAAAAAGgAAAGNvbS92bXdhcmUvc2NnL2V4dGVuc
2lvbnMvAwBQSwMECgAACAgAFXj3UgAAAAACAAAAAAAACkAAABjb20vdm13YXJlL3NjZy9leHRlbnNpb25zL21
5Y3VzdG9tZmlsdGVyLwMAUEsDBAoAAAgIABV491IVhwu7iwUAAPgOAABUAAAAY29tL3Ztd2FyZS9zY2cvZXh0Z
W5zaW9ucy9teWN1c3RvbWZpbHRlci9BZGRNeUN1c3RvbUhlYWRlckdhdGV3YXlGaWx0ZXJGYWN0b3J5LmNsYXN
zrVdrWxNHFH4nCAvJAjGtUK0i2mgBxfXWagm1hRjBmngBC9Kb3exOkoXNbtwLmN7v9/pbavs8rfChP6BPf1Ofn
tldIVzkweCH3ZmdOed9zzlzZs7sP/+t/A3gAh7E8RxeT1F/PIEsLqfAMJnAVbwlXtcSyKOQQkyMncRNCbfaMdW
O6QRu4+04ZjDbgTjutGMugXfwbhzv4X0JHzC05W9MTOSmGFJ52ykrrlk6P6/k7XKZOxmGzsLc3cnc2OXc1N1ru
TkhNK8uqoqpWmVl2nMMqyyEsrbleqrlzaimzwly1LAM7xJDy8DgDMOerK3TaHfesPh1v1rkzm21aPKAUVPNGdU
xxHc0uMerGC7DzbxmV5XF6pLqcMXVygq/73HLNYhJqdY13/XsaskwPe4oY7peqGeDkUmu6tyZUD2+pNavBNNXV
M2znTqZ2arWamadIT/Q4MSN4jzXvMxg6HxNeFRy1Cpfsp0FRTNtX1fKIZwS0a1DJ9g2zbZKRnl9bCJYhi5TrRZ
1NR1wp88yPBjYkmqJFxWXO4tEMB00s7yYu69VCIxnmjQuW1ENi1xzeBADRbMpljW/aBpuhWQLtmWThe08omE4+
/SmUVg1QcPw+q6spPzwazqN67lVc9bH7oyEDxmkcd8waY0Z5KuWRdqm6rqcEmZi67hWPK/22PogDsYij9yYpKk
pfs/nrpeOUDMiXyUnHGS4/CwgN/pxmmFwG1sFRJjGbmhNJfxgOL4zJVrRUc2MdmDHtFG2VM93KJzazheoFG4aZ
azoeg71t9pRo5vT/RKRx6dt39E4iRLl0I725ikBxLB/yrc8o8pnDNego2DMsmxP9cSOJ6AtbXcJg9tevcaVrF2
t2Ra3vIyM/TjAkBy3bU9YXytwr2Lrbncb1CSOUVOUoUGXwGWUUE6iQmNGEoqMeSxIMGWcRlWGBVtCTcjeIwEni
XMyzoOMuf5sjyaGxJ3hQn04lKCNUK0Ph4s+vCgOVAmeDB+LEpZk3EddxvPYRyG56dgap+TX+6OEPdmv6jrXG8D
6QxwJH8n4GJ9I+FTGZ/icofBMU0HCFwzju98rcXyJr+JUp76mYzXkZxgZ2MXhfPppDzQirvqUdtQZfQLxduoNu
37fVmVmRnj3DZGomsZrdMQcJpJAzPcMUyn5liYyXhEV1a8GOBebtYFhbmBb7F2511oUXYbzTUSJ1DO7KBYMt5o
ootvXQbEu39IZ2UzEnr4grHHkDVFoYnaJIb3lvWS9KCmnd1IF6O5V8z2xOpsvbBsht+BIbrwK0r3MsEr2hqyOA
EWZ6t2osXq4dZS59xjkWKN2ULofh3T9tXOk+TBL+E7G9/hJRhrHZJwQx/pZnGM4usZsWIv2AlfyQVmm8qCWVk/
i6tqXhF/E9di2F/waw7+NpkcAYWWZVC3d5G46lMw8KeCbFW9T4WpuMqTcXndw82xWNc1pw+MZCb/SYuzIH4a+7
eVwBOLXBPTvcUBUJuodQA966f/khWA0gTbqU1Gm94s0olDLqG0d+gvsYSBykN5twWAXDtFbDgXQh8PUMvQTSag
8Qm1MSK+I5rcN2r2Bdk8oEWmL3lG8BJaMi5SIcEySaaG2b2gZLQwr2AMsozVGrzYqjydOLkOKNeD3kDUgJxi5k
SCEJFnUQ7iCrz9EwnG8HPD1YQCDJNmDdgzRWIyYTwRuC+YMIQpbO4dWaJ74OmKYXQtDPDD+IIWgvyEUnXQlOQM
QEOVyBHSVaAOnh1KJlPwInX+ia/Z3dKeSy9gbwxpkV2DfIYLqI5sON8D20oXmFWrb8erq8hwJZoCOVPwRUn+g+
2GwBGtB3kuhuBCAX8RrQTtCToWzo9SeCjKghX5D72K4rYPszAbtD/gZP1IIL0Uhe4OeNzFGj+jlcAUTUe86bvw
PUEsDBAoAAAgIABV491KTBtcyAwAAAAEAAAAWAAAAYXBwbGljYXRpb24ucHJvcGVydGllc+MCAFBLAQIUAwoAA
AgIABV491IAAAAAAgAAAAAAAAAJAAAAAAAAAAAAAEADtQQAAAABNRVRBLUlORi9QSwECFAMKAAAICAAVePdSsn8
C7hsAAAAZAAAAFAAAAAAAAAAAAAAAApIEpAAAATUVUQS1JTkYvTUFOSUZFU1QuTUZQSwECFAMKAAAICAAVePdSA
AAAAAIAAAAAAAAABAAAAAAAAAAAABAA7UF2AAAAY29tL1BLAQIUAwoAAAgIABV491IAAAAAAgAAAAAAAAAALAAA
AAAAAAAAAEADtQZoAAABjb20vdm13YXJlL1BLAQIUAwoAAAgIABV491IAAAAAAgAAAAAAAAAPAAAAAAAAAAAAAE
ADtQcUAAABjb20vdm13YXJlL3NjZy9QSwECFAMKAAAICAAVePdSAAAAAAIAAAAAAAAGgAAAAAAAAAAAABAA7UH
0AAAAY29tL3Ztd2FyZS9zY2cvZXh0ZW5zaW9ucy9QSwECFAMKAAAICAAVePdSAAAAAAIAAAAAAAAKQAAAAAAA
AAAABAA7UEuAQAAY29tL3Ztd2FyZS9zY2cvZXh0ZW5zaW9ucy9teWN1c3RvbWZpbHRlci9QSwECFAMKAAAICAA
VePdSFYcLu4sFAAD4DgAAVAAAAAAAAAAAAAAAApIF3AQAAY29tL3Ztd2FyZS9zY2cvZXh0ZW5zaW9ucy9teWN1c
3RvbWZpbHRlci9BZGRNeUN1c3RvbUhlYWRlckdhdGV3YXlGaWx0ZXJGYWN0b3J5LmNsYXNzUEsBAhQDCgAACAg
AFXj3UpMG1zIDAAAAAQAAABYAAAAAAAAAAAAAAKSBdAcAAGFwcGxpY2F0aW9uLnByb3BlcnRpZXNQSwUGAAAAA
AkACQCGAgAAqwcAAAAA
kind: ConfigMap
metadata:
  creationTimestamp: "2021-07-23T21:02:47Z"
  name: my-custom-header
  namespace: testing
  resourceVersion: "10535421"
  selfLink: /api/v1/namespaces/testing/configmaps/my-custom-header
  uid: 405e72ce-b025-4552-bee3-5d795c9013ce
```

Note: It's possible to create a config map with multiple jars, for example if you need a



third-party library. Ensure that these do not cause classpath conflicts with project template. In case of doubt check the "Runtime description for Extensions developers" for full list of included libraries.

Extensions from OCI Image

Create a Dockerfile in the root of your project that contains the custom extension

```
FROM gradle:7-jdk11-alpine AS build
COPY --chown=gradle:gradle . /home/gradle/src
WORKDIR /home/gradle/src
RUN gradle build --no-daemon -PspringCloudVersion=2022.0.1

FROM alpine

RUN mkdir -p /app/extensions

COPY --from=build /home/gradle/src/build/libs/*.jar /app/extensions/gateway-extension.
jar
```

Now build the image using the docker command. You should specify the image registry you will push the extension to, this should be the same registry as the gateway and operator is using (you check this by running the command `k get deployment -o jsonpath="{..image}" -n spring-cloud-gateway`).

```
docker build . -t myregistry.example.com/scg-test-extensions:dev
```

Then push the image to the registry

```
docker push myregistry.example.com/scg-test-extensions:dev
```

Now you can use the custom extension by creating a gateway like this:

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
name: test-gateway
spec:
extensions:
  custom:
    - myregistry.example.com/scg-test-extensions:dev
```

This will create a gateway that uses a Kubernetes Init Container to load the custom extensions.

Extensions from Persistent Volumes

For bigger than 1MB deployments, you can load extensions from Kubernetes Volumes. This sections shows how to setup a Persistent Volume using exclusively Kubernetes tools, this requires an extra pod (`upload-extensions-pod`) to access the storage backend which is not ideal. Please refer to your storage implementations in your Cloud/K8s provider, ideal solutions would provide direct access to the storage backend in order to update the contents of the volumes without additional Kubernetes components.

Persistent Volumes comes with some restrictions:

- The gateway expects a persistent volume claim.
- The gateway expects the extensions to be located in `/{mount_name}/extensions`.
- The contents of the directory should all be readable from the pod. For example, some storage providers may add a `/{mount_name}/lost+found/` directory that is not readable. This potential issue is mitigated by expecting the extensions in the `/{mount_name}/extensions/` subdirectory.
- If the PersistentVolume has an access mode of `ReadWriteOnce`, the gateway pods must be scheduled on the same node to concurrently access the volume. For High-Availability scenarios (gateways across multiple cluster nodes), a StorageClass with support for `ReadOnlyMany` is required. Consult specific storage implementations for your Cloud/K8s provider.
- If the PersistentVolume has an access mode of `ReadWriteMany` or `ReadOnlyMany`, the gateway pods can be scheduled under different nodes. However, a cloud provider may decide to have the PersistentVolume only be accessible within one Availability Zone (AZ), so the gateway pods need to be scheduled to the same AZ. Consult specific storage implementations for your Cloud/K8s provider.

To test this feature, create a PersistentVolumeClaim, along with a pod for uploading files. For details on how to create a matching compatible Volume, we suggest starting checking Kubernetes docs on Persistent Volumes.

```
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: extensions-pvc
spec:
  storageClassName: standard
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
---
apiVersion: v1
kind: Pod
metadata:
  name: upload-extensions-pod
spec:
  containers:
    - name: task-pv-container
      image: nginx
      volumeMounts:
        - mountPath: /mount
          name: extensions
  initContainers:
    - name: init-extensions-dir
      image: nginx
      command: ['sh', '-c', 'mkdir -p /mount/extensions']
      volumeMounts:
        - mountPath: /mount
```

```
          name: extensions
  volumes:
    - name: extensions
      persistentVolumeClaim:
        claimName: extensions-pvc
```

Next, copy your extension to `/mount/extensions`

```
  $ kubectl cp add-my-custom-header.jar upload-extensions-pod:/mount/extensions/add-my
-custom-header.jar -c task-pv-container
```

Finally, specify the PersistentVolumeClaim in the custom extensions array of the gateway and the custom route filter in the route config, as shown in the next section.

Gateway Configuration

With the extension deployed in the cluster, update or create a new Gateway with the `extensions` option.

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
  name: my-gateway
spec:
  extensions:
    custom:
      - extension-name
```

Tip: `extensions` is an array element, allowing to enable multiple extensions at the same time.

This will automatically restart the Gateway with the new extension(s) available.

Once it is running, you can update the extension configmap and automatically restart the gateway with:

```
$ kubectl create configmap extension-name --from-file=extension.jar -o yaml --dry-run=
client | kubectl apply -f -
```

If you are using extensions from persistence, add the name of the PersistentVolumeClaim in the `spec.extensions.custom` array as well.

Now that the extension is available, it can be used in the respective Route Configuration.

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayRouteConfig
metadata:
  name: my-gateway-routes
spec:
  routes:
  - uri: https://httpbin.org
    predicates:
      - Path=/add-header/**
    filters:
      - AddMyCustomHeader
```

If there's no mapping already, add it to complete the configuration.

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGatewayMapping
metadata:
  name: test-gateway-mapping
spec:
  gatewayRef:
    name: my-gateway
  routeConfigRef:
    name: my-gateway-routes
```

Validation and Troubleshooting

With the deployment completed, enable traffic to the gateway `kubectl port-forward service/my-gateway 8080:80` and open http://localhost:8080/add-header/get in your web browser.

You will be greeted with a response similar to the one below, for simplicity some data has been removed. There you should see the custom `X-My-Header` header.

```
{
  "args": {},
  "headers": {
    "Forwarded": "proto=http;host=\"localhost:8080\";for=\"127.0.0.1:58598\"",
    "Host": "httpbin.org",
    "X-Forwarded-Host": "localhost:8080",
    "X-Forwarded-Prefix": "/add-header",
    "X-My-Header": "my-header-value"
  },
  "url": "https://localhost:8080/get"
}
```

If you cannot see the extension working:

- Obtain the output of a gateway instance (`kubectl logs statefulset.apps/my-gateway`) to validate if your log traces appear.
- See the Gateway events with `kubectl describe scg my-gateway` for diagnostics messages. If the extension could not be loaded you will see a message like `ConfigMap '{extension_name}' not found. Skipping configuration.`
- Check the ConfigMap or PersistentVolumeClaim is available in the same namespace as the gateway.
- Ensure the ConfigMap or PersistentVolumeClaim name matches the extension configuration in the Gateway.

High-Availability deployments

Previous `kubectl cp` approach can cause issues with providers that don't support `ReadWriteMany`. For example Google Kubernetes Engine only supports `ReadWriteOnce` and `ReadOnlyMany`, so upload pod and multiple gateway instances cannot run simultaneously in different nodes.

In those scenarios, to provide 100% availability you can use the automatic update features of SCG for K8s by switching to a different PersistentVolumeClaim. Provided you have:

- 1 Volume and VolumeClaim with old extension version
- 1 running Spring Cloud Gateway for Kubernetes with 2 or more instances using old extension
- Jar file(s) of the new extensions version

You should:

1. Create new Volume and VolumeClaim
2. Upload new extension version as described in previous step
3. Update Gateway `extensions.custom` value with new VolumeClaim name

```
apiVersion: "tanzu.vmware.com/v1"
kind: SpringCloudGateway
metadata:
  name: my-gateway
spec:
  extensions:
    custom:
      - extension-name-new
```

This will initiate a controlled update of the Gateway instances one by one ensuring no downtime.

OpenAPI Route Conversion

Spring Cloud Gateway includes an OpenAPI Route Conversion tool to help generate a RouteConfig for a given OpenAPI spec. This feature is bundled with the Spring Cloud Gateway operator and is exposed as an API. It can accept both OpenAPI 2.0 and OpenAPI 3.0 specs.

Conversion endpoint

An OpenAPI Conversion service can be found in an SCG operator instance. If you have the service exposed in your kubernetes cluster via `port-forward` or `Ingress`, you only need to send a POST request with path `/api/convert/openapi` to the reachable SCG-operator instance.

The next attributes are supported by the OpenAPI Conversion service

Field	Description
`service`	Kubernetes Service to route traffic to `spec.routes` spec which doesn't contain any `service` configuration. `.namespace`: (Optional) If not set will use the RouteConfig's namespace. `.name`: Name of a service to route to. Takes lower precedence than `uri`. Either `name` or `uri` are required unless all routes define their own uri. `.port`: (Optional) If not set will use one of the available service ports. `.filters`: (Optional) Predicates to be prepended to all routes.
`openapi`	`.location`: URL of the OpenApi Spec to use.
`routes`	`.filters`: Route filters to allow the modification of the incoming HTTP request or outgoing HTTP response in some manner. `.predicates`: Predicates to match on different attributes of the HTTP request.

For more details about the JSON schema, please, check the section JSON schema to validate requests.

Conversion request

You can generate a RouteConfig for your service calling the operator endpoint. For example, given you have the operator exposed at `http://operator.scg` and your Kubernetes service `my-service`. For your OpenAPI specification "https://petstore3.swagger.io/api/v3/openapi.json", you only need to make a call to the endpoint at `api/convert/openapi`:

```
curl --request POST 'http://operator.scg/api/convert/openapi' \
--header 'Content-Type: application/json' \
--data-raw '{
    "service": {
        "name": "my-service"
    },
    "openapi": {
        "location": "https://petstore3.swagger.io/api/v3/openapi.json"
    }
}'
```

This endpoint will return a JSON response, e.g:

```
{
  "apiVersion": "tanzu.vmware.com/v1",
  "kind": "SpringCloudGatewayRouteConfig",
  "metadata": {
    "name": "my-service"
  },
  "spec": {
    "openapi": {
      "components": {
        "schemas": {...},
        "requestBodies": {...},
        "securitySchemes": {...}
      },
      "ref": "https://petstore3.swagger.io/api/v3/openapi.json"
    },
    "routes": [
      {
        "description": "Update an existing pet by Id",
        "model": {
          "requestBody": {...},
          "responses": {...}
        },
        "predicates": [
          "Path=/pet",
          "Method=PUT"
        ],
        "tags": [
          "pet"
        ],
        "title": "Update an existing pet"
      },
      ...
    ],
```

```
    "service": {
      "name": "my-service"
    }
  }
}
```

Referencing an OpenAPI endpoint in your cluster

For example, the request body below will pull the OpenAPI spec exposed via the `openapi` service in the `development` namespace, effectively making a call the uri `openapi.development.svc.cluster.local:8080/openapi`:

```
{
    "service": {
        "name": "openapi",
        "namespace": "development",
        "port": "8080"
    },
    "openapi": {
        "location": "/openapi"
    }
}
```

Providing Service level filters

To provide service level filters, you can specify a `filters` array inside `service` of the request body:

```
{
    "openapi": {
        "location": "https://petstore3.swagger.io/api/v3/openapi.json"
    },
    "service": {
        "name": "my-service",
        "filters": ["StripPrefix=1"]
    }
}
```

Example result JSON:

```
{
  "apiVersion": "tanzu.vmware.com/v1",
  "kind": "SpringCloudGatewayRouteConfig",
  "metadata": {
    "name": "my-service"
  },
  "spec": {
    "openapi": {...},
    "routes": [...],
    "service": {
      "filters": [
        "StripPrefix=1"
      ],
      "name": "my-service"
    }
  }
```

```
}
```

Providing Route level filters

Route level filters can be applied with a wildcard, and be overridden with an exact match. In addition, you can specify multiple methods to match with.

For example, given a request body of:

```
{
  "service": {
    "name": "test-service"
  },
  "openapi": {
    "location": "https://petstore3.swagger.io/api/v3/openapi.json"
  },
  "routes": [
    {
      "predicates": ["Method=GET", "Path=/pet/findByStatus"],
      "filters": ["RateLimit=2,10s", "StripPrefix=1"]
    },
    {
      "predicates": ["Method=GET", "Path=/pet**"],
      "filters": ["RateLimit=3,5s", "StripPrefix=1"]
    },
    {
      "predicates": ["Method=PUT,DELETE", "Path=/user**"],
      "filters": ["RateLimit=4,15s", "StripPrefix=2"]
    }
  ]
}
```

- The path `GET /pet/findByStatus` will have the filters `RateLimit=2,10s` and `StripPrefix=1` applied
- The paths `GET /pet/findByTags` and `GET /pet/{petId}` will have the filters `RateLimit=2,10s` and `StripPrefix=1` applied
- The paths `PUT /user/{username}` and `DELETE /user/{username}` will have the filters `RateLimit=4,15s` and `StripPrefix=2` applied

See available filters here.

JSON schema to validate requests

You can fetch the JSON schema to validate requests by calling `/json/schema` on the Spring Cloud Gateway operator.

For example, given you have the Spring Cloud Gateway operator exposed at `http://operator.scg`, you can run:

```
curl --request GET 'http://operator.scg/json/schema' --header 'Accept: application/jso
n'
```