



SAP Business Technology Platform

Generated on: 2025-11-18 11:56:46 GMT+0000

SAP Business Technology Platform (SAP BTP) | Cloud

Public

Original content: <https://help.sap.com/docs/BTP/65de2977205c403bbc107264b8eccf4b?locale=en-US&state=PRODUCTION&version=Cloud>

Warning

This document has been generated from SAP Help Portal and is an incomplete version of the official SAP product documentation. The information included in custom documentation may not reflect the arrangement of topics in SAP Help Portal, and may be missing important aspects and/or correlations to other topics. For this reason, it is not for production use.

For more information, please visit <https://help.sap.com/docs/disclaimer>.

API Gateway Module

Use the API Gateway module to expose and secure APIs.

Caution

The APIRule CustomResourceDefinition (CRD) in version `v1beta1` has been deprecated and scheduled for deletion. If you use APIRule custom resources (CRs) `v1beta1`, you must migrate to version v2. See [APIRule Migration](#).

What Is API Gateway?

API Gateway is a Kyma module with which you can expose and secure APIs.

To use the API Gateway module, you must also add the Istio module. Moreover, to expose a workload using the APIRule custom resource, the workload must be part of the Istio service mesh.

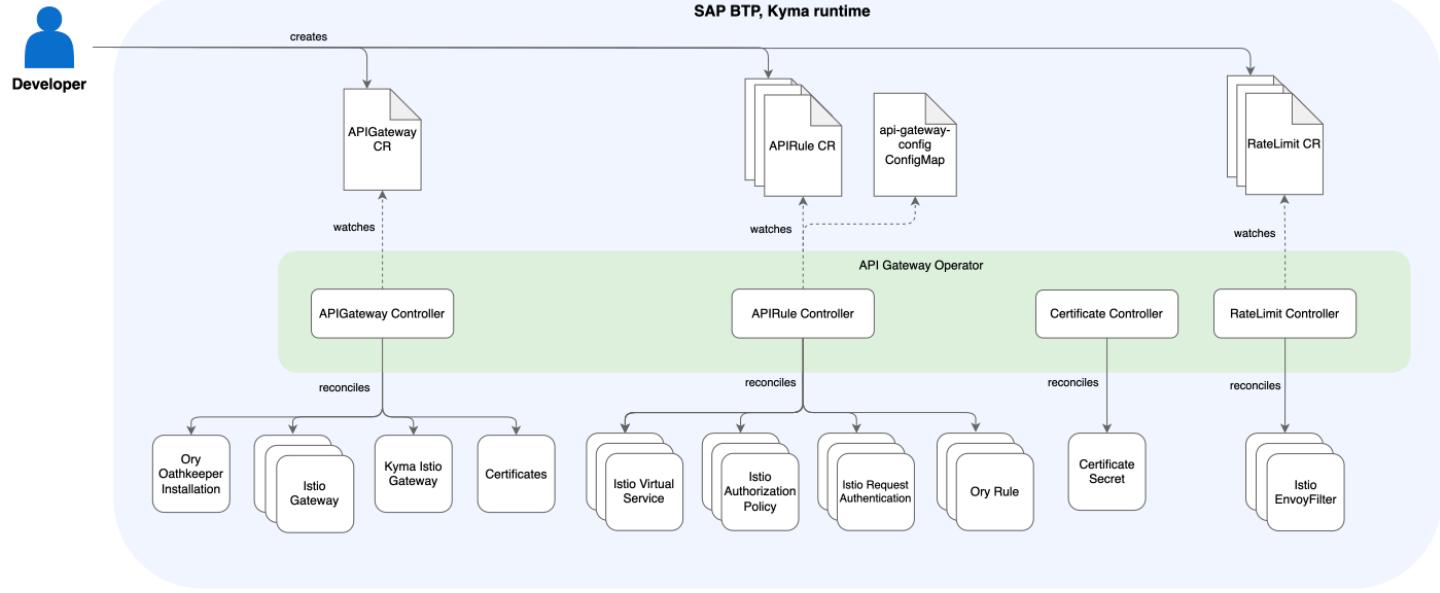
By default, both the API Gateway and Istio modules are added when you create a Kyma runtime instance.

Features

The API Gateway module offers the following features:

- **API Exposure:** The module uses Istio features to help you easily and securely expose your workloads by creating APIRule custom resources. With an APIRule, you can perform the following actions:
 - Group multiple workloads and expose them under a single host.
 - Use a short host name to simplify the migration of resources to a new cluster.
 - Configure the `noAuth` access strategy, which offers a simple configuration to allow access to specific HTTP methods.
 - Secure your workloads by configuring `jwt` or `extAuth` access strategies. The `jwt` access strategy enables you to use Istio's JWT configuration to protect your exposed services and interact with them using JSON Web Tokens. The `extAuth` access strategy allows you to implement custom authentication and authorization logic.
- **Gateway configuration:**
 - The module sets up the default TLS Kyma Gateway, which uses the default domain of your Kyma cluster and a self-signed certificate.
 - The module allows you to configure a custom Gateway, which is recommended for production environments. Additionally, it enables you to expose workloads using a custom domain and DNSEntry.
- **Rate Limiting:** The module simplifies local rate limiting on the Istio service mesh layer. You can configure it using a straightforward RateLimit custom resource.

Architecture



API Gateway Operator

Within the API Gateway module, API Gateway Operator manages the application of API Gateway's configuration and handles resource reconciliation. It contains the following controllers: APIGateway Controller, APIRule Controller, and RateLimit Controller.

APIGateway Controller

APIGateway Controller is responsible for the following:

- Configuring Kyma Gateway
- Managing Certificate and DNSEntry resources

APIRule Controller

APIRule Controller uses [Istio](#) resources to expose and secure APIs.

Certificate Controller

Certificate Controller is responsible for handling the Secret `api-gateway-webhook-certificate` in the `kyma-system` namespace. This Secret contains the Certificate data required for the APIRule conversion webhook.

RateLimit Controller

RateLimit Controller manages the configuration of local rate limiting on the Istio service mesh layer. By creating a RateLimit custom resource (CR), you can limit the number of requests targeting an exposed application in a unit of time, based on specific paths and headers.

API/Custom Resource Definitions

The `apigateways.operator.kyma-project.io` CRD describes the APIGateway CR that APIGateway Controller uses to manage the module and its resources. See [APIGateway Custom Resource](#).

The `apirules.operator.kyma-project.io` CRD describes the APIRule CR that APIRule Controller uses to expose and secure APIs. See [APIRule Custom Resource](#).

The `ratelimits.gateway.kyma-project.io` CRD describes the kind and the format of data that RateLimit Controller uses to configure the request rate limits for applications. See [RateLimit Custom Resource](#).

Resource Consumption

To learn more about the resources used by the API Gateway module, see [Kyma Modules' Sizing](#).

Related Information

[kyma-project.io: API Gateway troubleshooting guides](#)

[kyma-project.io: API Gateway tutorials](#)

[Ory Oathkeeper Introduction](#)

[Istio](#)

Securing Workloads

Choose a security configuration for exposing your workload.

The APIRule custom resource (CR), installed by the API Gateway module, allows you to define the security configuration for an exposed endpoint using the concept of access strategies. The APIRule v2 supports the following access strategies:

- jwt
- extAuth
- noAuth

i Note

To expose a workload using APIRule in version v2, the workload must be a part of the Istio service mesh. See [Enabling Istio Sidecar Proxy Injection](#).

Furthermore, you can expose a workload using Istio VirtualService and restrict access based on the caller's IP, or configure Istio VirtualService and AuthorizationPolicy to enable mutual TLS (mTLS) authentication and validate access based on client certificate.

Read about each method to decide which one to use.

→ Recommendation

For most common scenarios, it is best to configure the `jwt` access strategy within the APIRule CR.

Use the APIRule CR with the `jwt` Access Strategy

The `jwt` access strategy allows you to secure your workload with HTTPS using Istio JWT configuration. It offers a secure and efficient method for protecting your services and interacting with them using JSON Web Tokens (JWTs). This approach is highly recommended if you aim to secure your workloads without the need to implement custom authentication and authorization logic.

See [Exposing and Securing Workloads with a JWT](#).

Use the APIRule CR with the `extAuth` Access Strategy

The `extAuth` access strategy allows you to provide your custom authorization and authentication logic. Use this access strategy when the built-in Istio JWT authentication and authorization mechanisms do not meet your specific requirements, and you want to

offload the logic to a custom external service. It provides flexibility and the ability to tailor security to your application's specific needs.

To use the `extAuth` access strategy, you must first define the authorization provider in the Istio configuration, most commonly in the Istio custom resource (CR). Once you define the provider in the Istio configuration, you can reference it in the APIRule CR, specifying which endpoints and methods should be protected by this provider.

See [Exposing and Securing Workloads with `extAuth`](#).

Use the APIRule CR with the `noAuth` Access Strategy

The `noAuth` access strategy provides a simple configuration for exposing workloads. Use it when you simply need to expose your workloads and allow access to specific HTTP methods without any authentication or authorization checks. This setup is suitable for development and testing environments where security requirements are lower and quick access to services is necessary, or when the data being accessed is not sensitive and does not require strict security measures.

Caution

Exposing a workload to the outside world is a potential security vulnerability, so be careful. In a production environment, always secure the workload you expose.

See [Exposing Workloads with `noAuth`](#).

Restrict IP-Based Access with Istio VirtualService and XFF Header

Applying IP-based access restriction is recommended when you want to limit access to a specific workload to a set of trusted IP addresses, or to prevent unauthorized access from certain IP addresses. By using the XFF header, you can ensure that requests are only coming from trusted sources, and block or allow access based on the originating IP address contained in the XFF header.

Using the XFF header is only supported when you expose your workload with Istio VirtualService instead of the APIRule CR.

See [Using the XFF Header to Configure IP-Based Access to a Workload](#).

Secure a Workload Using a Certificate

After you have set up an mTLS Gateway, you can use a certificate to enforce mutual authentication and ensure only authorized clients can access the workload. This adds an extra layer of security to your application by providing an encrypted communication channel between the client and the server, while also guaranteeing the authenticity of both parties involved in the communication.

To secure your workload with a certificate, you must create an Istio AuthorizationPolicy and VirtualService instead of the APIRule CR.

See [Exposing and Securing Workloads with a Certificate](#).

Exposing and Securing Workloads with a JWT

Learn how to expose a workload and secure it with a JSON Web Token (JWT). To get the token, create a client credentials application using SAP Cloud Identity Services - Identity Authentication.

Prerequisites

- You have the Istio and API Gateway modules added. See [Adding and Deleting a Kyma Module](#).
- You have access to Kyma dashboard. Alternatively, to use CLI instructions, you must install [kubectl](#) and configure it to communicate with your Kyma runtime instance. See [Access a Kyma Instance Using kubectl](#).
- You have installed [curl](#).
- You have a deployed workload.

i Note

To expose a workload using APIRule in version v2, the workload must be a part of the Istio service mesh. See [Enabling Istio Sidecar Proxy Injection](#).

- You have an SAP Cloud Identity Services - Identity Authentication tenant and you have created an OpenID Connect Application in it. See [Create OpenID Connect Application](#) and [Configure OpenID Connect Application for JWT Bearer Flow](#).
- You have configured a Secret for API Authentication and saved the values of Client ID and Client Secret. See [Configure Secrets for API Authentication](#).
- You have [set up your custom domain](#). Alternatively, you can use the default domain of your Kyma cluster and the default Gateway kyma-system/kyma-gateway.

i Note

Because the default Kyma domain is a wildcard domain, which uses a simple TLS Gateway, it is recommended that you set up your custom domain for use in a production environment.

→ Tip

To learn what the default domain of your Kyma cluster is, run the following command:

```
kubectl get gateway -n kyma-system kyma-gateway -o jsonpath='{.spec.servers[0].hosts}'
```

Context

To interact with a workload that is secured using an API Rule of the JWT type, you need an access token. Follow these steps to obtain a JWT using an Identity Authentication tenant. Then, use either kubectl or Kyma dashboard to expose and secure your workload.

Procedure

To obtain a JWT token, you must use curl. To create an APIRule custom resource (CR), you can use either Kyma dashboard or kubectl.

- Use Kyma dashboard.
 1. Export the name of your Identity Authentication instance and your client credentials as environment variables:

```
export IDENTITY_AUTHENTICATION_INSTANCE={YOUR_IDENTITY_AUTHENTICATION_INSTANCE_NAME}
export CLIENT_ID={YOUR_CLIENT_ID}
export CLIENT_SECRET={YOUR_CLIENT_SECRET}
```

2. Encode your client credentials and export them as an environment variable:

```
export ENCODED_CREDENTIALS=$(echo -n "$CLIENT_ID:$CLIENT_SECRET" | base64)
```

- Get values of the `token_endpoint`, `jwks_uri`, and `issuer` parameters:

```
curl -s https://$IDENTITY_AUTHENTICATION_INSTANCE.accounts400.ondemand.com/.well-known/...
```

i Note

You need the values of the `jwks_uri` and `issuer` parameters to secure your workload using an APIRule of the JWT type.

- Export the value of the `token_endpoint` parameter as an environment variable:

```
export TOKEN_ENDPOINT={YOUR_TOKEN_ENDPOINT}
```

- Get the JWT:

```
curl -X POST "$TOKEN_ENDPOINT" -d "grant_type=client_credentials" -d "client_id=$CLIENT_ID" -d "client_secret=$CLIENT_SECRET" -d "scope=api:read:workload"
```

- Go to the namespace in which you want to create an APIRule custom resource.

The namespace must have Istio sidecar proxy injection enabled. See [Enabling Istio Sidecar Proxy Injection](#).

- Go to [Discovery and Network → API Rules](#) and select **Create**.

- Provide all the required configuration details, such as APIRule CR's name, the name and port of the Service you want to expose, the Gateway's name, and the host in the format `{SUBDOMAIN}.{YOUR_DOMAIN}`.

- Add a rule with the following configuration:

Option	Description
Access Strategy	jwt In the JWT section, add an authentication with your issuer and JSON Web Key Set URLs.
Methods	GET
path	/*

- Select **Create**.

- To call the endpoint, send a GET request to the Service:

```
curl -ik -X GET https://{SUBDOMAIN}.{DOMAIN_NAME}/headers
```

You get the error `401 Unauthorized`.

- Now, access the secured workload using the correct JWT.

```
curl -ik -X GET https://{SUBDOMAIN}.{DOMAIN_NAME}/headers --header "Authorization:Bearer $TOKEN"
```

If successful, you get the code `200 OK` in response.

- Use kubectl.

- Export the name of your Identity Authentication instance and your client credentials as environment variables:

```
export IDENTITY_AUTHENTICATION_INSTANCE={YOUR_IDENTITY_AUTHENTICATION_INSTANCE_NAME}
export CLIENT_ID={YOUR_CLIENT_ID}
export CLIENT_SECRET={YOUR_CLIENT_SECRET}
```

- Encode your client credentials and export them as an environment variable:

```
export ENCODED_CREDENTIALS=$(echo -n "$CLIENT_ID:$CLIENT_SECRET" | base64)
```

3. Get values of the `token_endpoint`, `jwks_uri`, and `issuer` parameters:

```
curl -s https://$IDENTITY_AUTHENTICATION_INSTANCE.accounts400.ondemand.com/.well-known/
```

i Note

You need the values of the `jwks_uri` and `issuer` parameters to secure your workload using an APIRule of the JWT type.

4. Export the value of the `token_endpoint` parameter as an environment variable:

```
export TOKEN_ENDPOINT={YOUR_TOKEN_ENDPOINT}
```

5. Get the JWT:

```
curl -X POST "$TOKEN_ENDPOINT" -d "grant_type=client_credentials" -d "client_id=$CLIENT_ID"
```

6. Export the JWT as an environment variable:

```
export ACCESS_TOKEN={YOUR_ACCESS_TOKEN}
```

7. To expose and secure your Service, replace all placeholders and create the following APIRule:

```
cat <<EOF | kubectl apply -f -
apiVersion: gateway.kyma-project.io/v2
kind: APIRule
metadata:
  name: {APIRULE_NAME}
  namespace: {APIRULE_NAMESPACE}
spec:
  hosts:
    - {SUBDOMAIN}.{DOMAIN_NAME}
  service:
    name: {SERVICE_NAME}
    port: {SERVICE_PORT}
  gateway: {GATEWAY_NAME}/{GATEWAY_NAMESPACE}
  rules:
    - jwt:
        authentications:
          - issuer: {ISSUER}
            jwksUri: {JWKS_URI}
      methods:
        - GET
      path: /*
EOF
```

8. To call the endpoint, send a GET request to the Service

```
curl -ik -X GET https://{SUBDOMAIN}.{DOMAIN_NAME}/headers
```

You get the error 401 Unauthorized.

9. Now, access the secured workload using the correct JWT.

```
curl -ik -X GET https://{SUBDOMAIN}.{DOMAIN_NAME}/headers --header "Authorization:Bearer {ACCESS_TOKEN}"
```

If successful, you get the code 200 OK in response.

Exposing and Securing Workloads with extAuth

Expose and secure your workload using the extAuth access strategy.

Prerequisites

- You have the Istio and API Gateway modules added. See [Adding and Deleting a Kyma Module](#).
- You have installed [curl](#) and [kubectl](#). You have configured kubectl to communicate with your Kyma runtime instance. See [Access a Kyma Instance Using kubectl](#).
- You have a deployed workload.

i Note

To expose a workload using APIRule in version v2, the workload must be a part of the Istio service mesh. See [Enabling Istio Sidecar Proxy Injection](#).

- You have [set up your custom domain](#). Alternatively, you can use the default domain of your Kyma cluster and the default Gateway kyma-system/kyma-gateway.

i Note

Because the default Kyma domain is a wildcard domain, which uses a simple TLS Gateway, it is recommended that you set up your custom domain for use in a production environment.

→ Tip

To learn what the default domain of your Kyma cluster is, run the following command:

```
kubectl get gateway -n kyma-system kyma-gateway -o jsonpath='{.spec.servers[0].hosts}'
```

- You have a JSON Web Token (JWT). See [Get a JWT](#).

Context

Learn how to expose and secure services using APIGateway Controller and OAuth2.0 Client Credentials flow. For this purpose, this task uses [oauth2-proxy](#) with an OAuth2.0 compliant authorization server supporting OIDC discovery. APIGateway Controller reacts to an instance of the APIRule custom resource (CR) and creates an Istio [VirtualService](#) and [AuthorizationPolicy](#) with action type CUSTOM.

Procedure

1. Replace the placeholders and define the oauth2-proxy configuration for your authorization server.

→ Tip

To generate COOKIE_SECRET and CLIENT_SECRET, you can use the command `openssl rand -base64 32 | head -c 32 | base64`.

→ Tip

You can adapt this configuration to better suit your needs. See the [additional configuration parameters](#).

```
cat <<EOF > values.yaml
config:
  clientID: {CLIENT_ID}
```

11/18/25, 11:56 AM

```
clientSecret: {CLIENT_SECRET}
cookieName: ""
cookieSecret: {COOKIE_SECRET}

extraArgs:
  auth-logging: true
  cookie-domain: "{DOMAIN_TO_EXPOSE_WORKLOADS}"
  cookie-samesite: lax
  cookie-secure: false
  force-json-errors: true
  login-url: static://401
  oidc-issuer-url: {OIDC_ISSUER_URL}
  pass-access-token: true
  pass-authorization-header: true
  pass-host-header: true
  pass-user-headers: true
  provider: oidc
  request-logging: true
  reverse-proxy: true
  scope: "{TOKEN_SCOPES}"
  set-authorization-header: true
  set-xauthrequest: true
  skip-jwt-bearer-tokens: true
  skip-oidc-discovery: false
  skip-provider-button: true
  standard-logging: true
  upstream: static://200
  whitelist-domain: "*.{DOMAIN_TO_EXPOSE_WORKLOADS}:"*
EOF
```

2. To install oauth2-proxy with your configuration, use [oauth2-proxy helm chart](#) :

```
kubectl create namespace oauth2-proxy
helm repo add oauth2-proxy https://oauth2-proxy.github.io/manifests
helm upgrade --install oauth2-proxy oauth2-proxy -f values.yaml -n oauth2-proxy
```

3. Register oauth2-proxy as an authorization provider in the Istio module.

```
kubectl patch istio -n kyma-system default --type merge --patch '{"spec":{"config":{"authoriz
```

4. To expose and secure the Service, create the following APIRule:

```
cat <<EOF | kubectl apply -f -
apiVersion: gateway.kyma-project.io/v2alpha1
kind: APIRule
metadata:
  name: {APIRULE_NAME}
  namespace: {APIRULE_NAMESPACE}
spec:
  hosts:
    - {SUBDOMAIN}.{DOMAIN_TO_EXPOSE_WORKLOADS}
  service:
    name: {SERVICE_NAME}
    port: {SERVICE_PORT}
  gateway: {GATEWAY_NAME}/{GATEWAY_NAMESPACE}
  rules:
    - extAuth:
        authorizers:
          - oauth2-proxy
      methods:
        - GET
      path: /*
EOF
```

5. To call the endpoint, send a GET request to the exposed Service:

a. To call the endpoint, send a GET request to the exposed Service:

```
curl -ik -X GET https://{SUBDOMAIN}.{DOMAIN_TO_EXPOSE_WORKLOADS}/headers
```

You get the 401 Unauthorized response code.

- a. Now, access the secured workload using the correct JWT.

```
curl -ik -X GET https://{{SUBDOMAIN}}.{{DOMAIN_TO_EXPOSE_WORKLOADS}}/headers --header "Authorization: Bearer {{JWT}}
```

You get the 200 OK response code.

Exposing Workloads with noAuth

Expose your workload using the noAuth access strategy, which allows access to the workload through the specified HTTP methods without requiring authentication.

Prerequisites

- You have the Istio and API Gateway modules added. See [Adding and Deleting a Kyma Module](#).
- You have access to Kyma dashboard. Alternatively, to use CLI instructions, you must install [kubectl](#) and configure it to communicate with your Kyma runtime instance. See [Access a Kyma Instance Using kubectl](#).
- You have installed [curl](#).
- You have a deployed workload.

i Note

To expose a workload using APIRule in version v2, the workload must be a part of the Istio service mesh. See [Enabling Istio Sidecar Proxy Injection](#).

- You have [set up your custom domain](#). Alternatively, you can use the default domain of your Kyma cluster and the default Gateway `kyma-system/kyma-gateway`.

i Note

Because the default Kyma domain is a wildcard domain, which uses a simple TLS Gateway, it is recommended that you set up your custom domain for use in a production environment.

→ Tip

To learn what the default domain of your Kyma cluster is, run the following command:

```
kubectl get gateway -n kyma-system kyma-gateway -o jsonpath='{.spec.servers[0].hosts}'
```

Context

The intended functionality of the noAuth access strategy is to provide a simple configuration for exposing workloads. It only allows access to the specified HTTP methods of the exposed workload.

⚠ Caution

Exposing a workload to the outside world is a potential security vulnerability, so be careful. In a production environment, always secure the workload you expose.

Procedure

You can use either Kyma dashboard or kubectl.

- Use Kyma dashboard.

1. Go to the namespace in which you want to create an APIRule CR.
2. Go to **Discovery and Network > API Rules** and select **Create**.
3. Provide the name of the APIRule CR.
4. Add the name and port of the service you want to expose.
5. Add a Gateway.
6. Add a rule with the following configuration:

Option	Description
Path	/*
Handler	No Auth
Methods	GET

7. Add one more rule with the following configuration:

Option	Description
Path	/post
Handler	No Auth
Methods	POST

8. Send a GET request to the exposed workload:

```
curl -ik -X GET https://{{SUBDOMAIN}}.{{DOMAIN_NAME}}/ip
```

If successful, the call returns the 200 OK response code.

9. Send a POST request to the exposed workload:

```
curl -ik -X POST https://{{SUBDOMAIN}}.{{DOMAIN_NAME}}/post -d "test data"
```

If successful, the call returns the 200 OK response code.

- Use kubectl.

1. To create an APIRule CR with noAuth configuration, replace the placeholders and run the following command. You can adjust the configuration, if needed.

```
cat <<EOF | kubectl apply -f -
apiVersion: gateway.kyma-project.io/v2
kind: APIRule
metadata:
  name: {{APIRULE_NAME}}
  namespace: {{APIRULE_NAMESPACE}}
spec:
  hosts:
    - {{SUBDOMAIN}}.{{DOMAIN_NAME}}
  service:
    name: {{SERVICE_NAME}}
    namespace: {{SERVICE_NAMESPACE}}
    port: {{SERVICE_PORT}}
  gateway: {{NAMESPACE/GATEWAY}}
  rules:
```

```

    - path: /*
      methods: ["GET"]
      noAuth: true
    - path: /post
      methods: ["POST"]
      noAuth: true
EOF

```

- Send a GET request to the exposed workload:

```
curl -ik -X GET https://{{SUBDOMAIN}}.{{DOMAIN_NAME}}/ip
```

If successful, the call returns the 200 OK response code.

- Send a POST request to the exposed workload:

```
curl -ik -X POST https://{{SUBDOMAIN}}.{{DOMAIN_NAME}}/post -d "test data"
```

If successful, the call returns the 200 OK response code.

Exposing and Securing Workloads with a Certificate

Learn how to expose and secure a workload with mutual authentication using a TLS Gateway.

Prerequisites

- You have the Istio and API Gateway modules added. See [Adding and Deleting a Kyma Module](#).
- You have access to Kyma dashboard. Alternatively, to use CLI instructions, you must install [kubectl](#) and configure it to communicate with your Kyma runtime instance. See [Access a Kyma Instance Using kubectl](#).
- You have installed [curl](#).
- You have a deployed workload.

i Note

To expose a workload using APIRule in version v2, the workload must be a part of the Istio service mesh. See [Enabling Istio Sidecar Proxy Injection](#).

- You have [set up your custom domain](#).
- You have set up an mTLS Gateway. See [Setting Up an mTLS Gateway](#).

Context

To expose and secure a workload with mutual authentication using TLS Gateway, you must create an APIRule custom resource (CR) that adds the X-CLIENT-SSL headers to incoming requests. Follow the procedure to learn how to do this.

Procedure

You can use either Kyma dashboard or kubectl.

- Use Kyma dashboard.
 - Go to [Discovery and Network > APIRule](#) and select [Create](#).
 - Add a name for your APIRule CR.

3. Add the name and namespace of the Gateway you want to use.
4. Specify the host.
5. Add a Rule with the following configuration:

Parameter	Option
Path	/*
Methods	GET
Access Strategy	No Auth
Request > Headers	Add the following key-value pairs: <ul style="list-style-type: none"> ■ X-CLIENT-SSL-CN: %DOWNSTREAM_PEER SUBJECT% ■ X-CLIENT-SSL-SAN: %DOWNSTREAM_PEER_URI_SAN% ■ X-CLIENT-SSL-ISSUER: %DOWNSTREAM_PEER_ISSUER%
Service	Add the name and port of the Service you want to expose.

6. Choose **Create**.

- Use kubectl.

1. Replace the placeholders and export the following values as environment variables:

```
export CLIENT_ROOT_CA_CRT_FILE=<CLIENT_ROOT_CA_CRT_FILE>
export CLIENT_CERT_CN=<COMMON_NAME>
export CLIENT_CERT_ORG=<ORGANIZATION>
export CLIENT_CERT_CRT_FILE=<CLIENT_CERT_CRT_FILE>
export CLIENT_CERT_KEY_FILE=<CLIENT_CERT_KEY_FILE>
export NAMESPACE=<YOUR_NAMESPACE>
export DOMAIN_NAME=<DOMAIN_TO_EXPOSE_WORKLOADS>
export GATEWAY=<GATEWAY_NAMESPACE>/<GATEWAY_NAME>
```

2. Create APIRule CR that adds the X-CLIENT-SSL headers to incoming requests:

```
cat <<EOF | kubectl apply -f -
apiVersion: gateway.kyma-project.io/v2
kind: APIRule
metadata:
  name: {APIRULE_NAME}
  namespace: {APIRULE_NAMESPACE}
spec:
  gateway: {GATEWAY_NAMESPACE}/{GATEWAY_NAME}
  hosts:
    - {SUBDOMAIN}.{DOMAIN}
  rules:
    - methods:
        - GET
      noAuth: true
      path: /*
      timeout: 300
      request:
        headers:
          X-CLIENT-SSL-CN: '%DOWNSTREAM_PEER SUBJECT%'
          X-CLIENT-SSL-ISSUER: '%DOWNSTREAM_PEER_ISSUER%'
          X-CLIENT-SSL-SAN: '%DOWNSTREAM_PEER_URI_SAN%'
  service:
    name: {SERVICE_NAME}
```

```
port: {SERVICE_PORT}
EOF
```

Results

To call the secured endpoints of your Service, send a GET with the client certificate that you used to create your Gateway

```
curl --key ${CLIENT_CERT_KEY_FILE} \
--cert ${CLIENT_CERT_CRT_FILE} \
--cacert ${CLIENT_ROOT_CA_CRT_FILE} \
-ik -X GET https://httpbin-vs.$DOMAIN_TO_EXPOSE_WORKLOADS/headers
```

If successful, the call returns the 200 OK response code. If you call the Service without the proper certificates or with invalid ones, you get the error 403 Forbidden.

Using the XFF Header to Configure IP-Based Access to a Workload

Expose your workload and configure IP-based access using the X-Forwarded-For (XFF) header. This helps to enhance security by ensuring that only trusted IPs can interact with your application.

Prerequisites

- You have the Istio and API Gateway modules added. See [Adding and Deleting a Kyma Module](#).
- You have a deployed workload.
- You have access to Kyma dashboard. Alternatively, to use CLI instructions, you must install [kubectl](#) and [curl](#).
- You have [set up your custom domain](#). Alternatively, you can use the default domain of your Kyma cluster and the default Gateway kyma-system/kyma-gateway.

i Note

Because the default Kyma domain is a wildcard domain, which uses a simple TLS Gateway, it is recommended that you set up your custom domain for use in a production environment.

→ Tip

To learn what the default domain of your Kyma cluster is, run the following command:

```
kubectl get gateway -n kyma-system kyma-gateway -o jsonpath='{.spec.servers[0].hosts}'
```

Context

The XFF header is a standard HTTP header that conveys the client IP address and the chain of intermediary proxies that the request traverses to reach the Istio service mesh. This is particularly useful when an application must be provided with the client IP address of an originating request, for example, for access control.

However, there are some technical limitations when using the XFF header. The header might not include all IP addresses if an intermediary proxy does not support modifying the header. Due to [technical limitations of AWS Classic ELBs](#), when using an IPv4 connection, the header does not include the public IP of the load balancer in front of Istio Ingress Gateway. Moreover, Istio

Ingress Gateway's Envoy does not append the private IP address of the load balancer to the XFF header, effectively removing this information from the request. For more information on XFF, see the [IETF's RFC documentation](#) and [Envoy documentation](#).

To use the XFF header, you must configure the corresponding settings in the Istio custom resource (CR). Then, expose your workload using an VirtualService CR and create an AuthorizationPolicy resource with allowed IP addresses specified in the `remoteIpBlocks` field. To learn how to do this, follow the procedure.

Procedure

- Use Kyma dashboard.
 1. Go to [Cluster Details](#) and choose [Modify Modules](#).
 2. Select the Istio module and choose [Edit](#).
 3. In the [General](#) section, add the number of trusted proxies.

Due to the variety of network topologies, the Istio CR must specify the number of trusted proxies deployed in front of the Istio Ingress Gateway proxy, so that the client address can be extracted correctly.

4. If you use a Google Cloud or Microsoft Azure cluster, navigate to the [Gateway](#) section and set the Gateway traffic policy to Local. If you use a different cloud service provider, skip this step.

Caution

For production Deployments, it is strongly recommended to deploy Istio Ingress Gateway Pod to multiple nodes if you enable `externalTrafficPolicy : Local`. For more information, see [Network Load Balancer](#).

Default Istio installation profile configures `PodAntiAffinity` to ensure that Ingress Gateway Pods are evenly spread across all nodes. This guarantees that the above requirement is satisfied if your `IngressGateway` autoscaling configuration `minReplicas` is equal to or greater than the number of nodes. You can configure autoscaling options in the Istio CR using the field
`spec.config.components.ingressGateway.k8s.hpaSpec.minReplicas`.

Tip

If you use a Google Cloud or Microsoft Azure cluster, you can find your load balancer's IP address in the field `status.loadBalancer.ingress` of the `ingress-gateway` Service.

5. Choose [Save](#).
6. Go to [Istio > VirtualServices](#) and choose [Create](#).
7. Switch to the [YAML](#) section and paste the following configuration:

```
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: {VIRTUALSERVICE_NAME}
  namespace: {VIRTUALSERVICE_NAMESPACE}
spec:
  hosts:
  - "{SERVICE_NAME}.{DOMAIN_NAME}"
  gateways:
  - {GATEWAY_NAME}/{GATEWAY_NAMESPACE}
  http:
  - match:
    - uri:
        prefix: /
    route:
    - destination:
        port:
          number: {SERVICE_PORT}
        host: {SERVICE_NAME}.{SERVICE_NAMESPACE}.svc.cluster.local
```

When you go to `https://[SUBDOMAIN].[DOMAIN]/[PATH]`, the response contains the X-Forwarded-For and X-Envoy-External-Address headers with your public IP address. See an example response for the Client IP 165.1.187.197:

```
{
  "args": {
    "show_env": "true"
  },
  "headers": {
    ...
    "X-Envoy-Attempt-Count": "...",
    "X-Envoy-External-Address": "165.1.187.197",
    "X-Forwarded-Client-Cert": "...",
    "X-Forwarded-For": "165.1.187.197",
    "X-Forwarded-Proto": "...",
    "X-Request-Id": "..."
  },
  "origin": "165.1.187.197",
  "url": "..."
}
```

→ Tip

You can check your public IP address at <https://api.ipify.org>.

8. Go to **Istio > Authorizaiton Policies** and choose **Create**.
 9. Add a selector to specify the workload for which access should be configured.
 10. Add a rule with a **From** field.
 11. In the **RemoteIpBlocks** field, specify the IP addresses that should be allowed access to the workload.
 12. Choose **Create**.
- Use kubectl.
 1. In the following command, replace the placeholder with the number of trusted proxies deployed in front of the Istio Ingress Gateway proxy. Run the command.


```
kubectl patch istios/default -n kyma-system --type merge -p '{"spec":{"config":{"numTrustedProxies": 3}}}'
```

 Due to the variety of network topologies, the Istio CR must specify the configuration property **numTrustedProxies**, so that the client IP address can be extracted correctly.
 2. If you use a Google Cloud or Microsoft Azure cluster, run the following command to set the traffic policy to Local. If you use a different cloud service provider, skip this step.


```
kubectl patch istios/default -n kyma-system --type merge -p '{"spec":{"config":{"gatewayTrafficPolicy": "Local"}}}'
```

⚠ Caution

For production Deployments, it is strongly recommended to deploy Istio Ingress Gateway Pod to multiple nodes if you enable **externalTrafficPolicy : Local**. For more information, see [Network Load Balancer](#).

Default Istio installation profile configures **PodAntiAffinity** to ensure that Ingress Gateway Pods are evenly spread across all nodes. This guarantees that the above requirement is satisfied if your **IngressGateway** autoscaling configuration **minReplicas** is equal to or greater than the number of nodes. You can configure autoscaling options in the Istio CR using the field `spec.config.components.ingressGateway.k8s.hpaSpec.minReplicas`.

→ Tip

If you use a Google Cloud or Microsoft Azure cluster, you can find your load balancer's IP address in the field `status.loadBalancer.ingress` of the `ingress-gateway` Service.

3. To expose your workload, apply the following VirtualService resource.

You can adjust this sample configuration and use another access strategy, according to your needs.

```
cat <<EOF | kubectl apply -f -
apiVersion: networking.istio.io/v1alpha3
kind: VirtualService
metadata:
  name: {VIRTUALSERVICE_NAME}
  namespace: {VIRTUALSERVICE_NAMESPACE}
spec:
  hosts:
    - "{SERVICE_NAME}.{DOMAIN_NAME}"
  gateways:
    - {GATEWAY_NAME}/{GATEWAY_NAMESPACE}
  http:
    - match:
        - uri:
            prefix: /
      route:
        - destination:
            port:
              number: {SERVICE_PORT}
            host: {SERVICE_NAME}.{SERVICE_NAMESPACE}.svc.cluster.local
EOF
```

When you go to `https://{SUBDOMAIN}.{DOMAIN}/{PATH}`, the response contains the X-Forwarded-For and X-Envoy-External-Address headers with your public IP address. See an example response for the Client IP 165.1.187.197:

```
{
  "args": {
    "show_env": "true"
  },
  "headers": {
    ...
    "X-Envoy-Attempt-Count": "...",
    "X-Envoy-External-Address": "165.1.187.197",
    "X-Forwarded-Client-Cert": "...",
    "X-Forwarded-For": "165.1.187.197",
    "X-Forwarded-Proto": "...",
    "X-Request-Id": "..."
  },
  "origin": "165.1.187.197",
  "url": "..."
}
```

→ Tip

You can check your public IP address at <https://api.ipify.org>.

4. To configure IP-based access to the exposed workload, create an AuthorizationPolicy resource.

The selector specifies the workload for which access should be configured, and the RemoteIpBlocks field specifies the IP addresses for which access should be allowed.

```
cat <<EOF | kubectl apply -f -
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: {AUTHORIZATIONPOLICY_NAME}
  namespace: {AUTHORIZATIONPOLICY_NAMESPACE}
spec:
  action: ALLOW
  rules:
    - from:
        - source:
            ipBlocks: []
            remoteIpBlocks:
              - {ALLOWED_IP}
  selector:
```

```

    matchLabels:
      {KEY}: {VALUE}
EOF

```

Results

You have configured the XFF header in the Istio CR and exposed your workload to the internet. Access to the workload is limited to the IP addresses that you have specified.

Exposing Workloads with Istio VirtualService

Learn how to expose a workload with Istio VirtualService.

Prerequisites

- You have the Istio module added to your Kyma runtime instance.
- You have a deployed workload.
- You have set up your custom domain and a custom TLS Gateway. See [Set Up a Custom Domain](#) and [Set Up a TLS Gateway](#).

Alternatively, you can use the default domain of your Kyma cluster and the default Gateway `kyma-system/kyma-gateway`. To use the default domain and Gateway, the API Gateway module must be added to your cluster.

→ Tip

To get the default domain of your Kyma cluster, run the following command:

```
kubectl get gateway -n kyma-system kyma-gateway -o jsonpath='{.spec.servers[0].hosts}'
```

Context

Kyma's API Gateway module provides the APIRule custom resource (CR), which is the recommended solution for securely exposing workloads. To expose a workload using an APIRule v2, you must include the workload in the Istio service mesh. Including a workload in the mesh brings several benefits, such as secure service-to-service communication, tracing capabilities, or traffic management. For more information, see [Purpose and Benefits of Istio Sidecar Proxies](#) and [The Istio service mesh](#).

However, if you do not need the capabilities provided by the Istio service mesh, you can expose an unsecured workload using Istio VirtualService only. Such an approach might be useful in the following scenarios:

- If you use [Unified Gateway](#) as an entry point for SAP Cloud solutions. In this case, you can configure Unified Gateway to manage API exposure, JWT validation, and routing capabilities, offloading these responsibilities from the service mesh.
- If you want to expose front-end Services that manage their own authentication mechanisms.
- If you require certain features or want to implement specific configurations that APIRule does not support.
- If you require full control over Istio resources and you want to manage them directly without any higher-level abstractions.

The following instructions demonstrate a simple use case where VirtualService exposes an unsecured Service, skipping the requirement to include the Service in the Istio service mesh.

Procedure

- Use Kyma dashboard
 1. Go to the namespace in which you want to create a VirtualService resource.
 2. Go to [Istio > Virtual Services](#).
 3. Choose [Create](#) and provide the following configuration details:

Option	Description
Name	The name of the VirtualService resource you're creating.
Hosts	The address or addresses the client uses when sending requests to the destination Service. Use the fully qualified domain name (FQDN) constructed with the subdomain and domain name in the following format: {SUBDOMAIN}. {DOMAIN_NAME}. The hosts must be defined in the referenced Gateway.
Gateways	The name of the Gateway you want to use and the namespace in which it is created. Use the format {GATEWAY_NAMESPACE}/{GATEWAY_NAME}.

4. Go to [HTTP > Matches > Match](#) and provide the following configuration details:

Option	Description
URI	Add the URI prefix used to match incoming requests. This determines which requests are routed to the specified destination Service.
Port	This is the port number on which the destination Service is listening. It specifies where the VirtualService routes the incoming traffic. If a Service exposes only a single port it is not required to explicitly select the port.

5. Go to [HTTP > Routes > Route > Destinations > Host](#) and add the name and namespace of the Service you want to expose using the following format: {SERVICE_NAME}.{SERVICE_NAMESPACE}.svc.cluster.local. The traffic is routed to this Service.

For more configuration options, see [Virtual Service](#).

↳ Sample Code

See a sample VirtualService configuration that directs all HTTP traffic received at `httpbin.my-domain.com` through the `my-gateway` Gateway to the [HTTPBin Service](#), which is running on port 8000 in the default namespace.

```
apiVersion: networking.istio.io/v1
kind: VirtualService
metadata:
  name: example-vs
  namespace: default
spec:
  hosts:
    - httpbin.my-domain.com
  gateways:
    - default/my-gateway
  http:
    - match:
```

```

- uri:
  prefix: /
route:
- destination:
  port:
    number: 8000
  host: httpbin.default.svc.cluster.local

```

6. To verify if the Service is exposed, run the following command:

```
curl -s -I https://{{SUBDOMAIN}}.{{DOMAIN_NAME}}/
```

If successful, you get code 200 in response.

- Use kubectl

1. To expose a workload using VirtualService, replace the placeholders and run the following command:

Option	Description
{NAME}	The name of the VirtualService resource you're creating.
{NAMESPACE}	The namespace in which you want to create the VirtualService resource.
{SUBDOMAIN}.{{DOMAIN_NAME}}	The address or addresses the client uses when sending requests to the destination Service. Use the fully qualified domain name (FQDN) constructed with the subdomain and domain name in the following format: {SUBDOMAIN}.{{DOMAIN_NAME}}. The hosts must be defined in the referenced Gateway.
{GATEWAY_NAMESPACE}/{GATEWAY_NAME}	The name of the Gateway you want to use and the namespace in which it is created.
{URI_PREFIX}	The URI prefix used to match incoming requests. This determines which requests are routed to the specified destination Service.
{PORT_NUMBER}	This is the port number on which the destination Service is listening. It specifies where the VirtualService routes incoming requests. If a Service exposes only a single port it is not required to explicitly select the port.
{SERVICE_NAME}.{{SERVICE_NAMESPACE}}	The name and namespace of the Service you want to expose. The requests are routed to this Service.

For more configuration options, see [Virtual Service](#).

```

cat <<EOF | kubectl apply -f -
apiVersion: networking.istio.io/v1
kind: VirtualService
metadata:
  name: {{VS_NAME}}
  namespace: {{VS_NAMESPACE}}
spec:
  hosts:
    - {{SUBDOMAIN}}.{{DOMAIN_NAME}}
  gateways:

```

```

    - {GATEWAY_NAMESPACE}/{GATEWAY_NAME}
http:
- match:
  - uri:
    prefix: {URI_PREFIX}
route:
- destination:
  port:
    number: {PORT_NUMBER}
host: {SERVICE_NAME}.{SERVICE_NAMESPACE}.svc.cluster.local

```

Sample Code

See a sample VirtualService configuration that directs all HTTP traffic received at `httpbin.my-domain.com` through the `my-gateway` Gateway to the [HTTPBin Service](#), which is running on port 8000 in the default namespace.

```

apiVersion: networking.istio.io/v1
kind: VirtualService
metadata:
  name: example-vs
  namespace: default
spec:
  hosts:
    - httpbin.my-domain.com
  gateways:
    - default/my-gateway
  http:
    - match:
      - uri:
        prefix: /
      route:
        - destination:
          port:
            number: 8000
          host: httpbin.default.svc.cluster.local

```

2. To verify if the Service is exposed, run the following command:

```
curl -s -I https://{{SUBDOMAIN}}.{{DOMAIN_NAME}}/
```

If successful, you get code 200 in response.

Ordering Rules in APIRule v2

To ensure your service requests are handled as intended, learn how to create and order rules in your APIRule custom resource (CR).

Context

APIRule allows you to define a list of rules that specify how requests to your service are handled. You can define a list of rules in the `spec.rules` field of the APIRule CR. Each rule starts with a hyphen and must contain the following details:

- The path on which the service is exposed
- The list of HTTP methods available for this path
- The specified access strategy

If your APIRule includes multiple rules, their order matters. Follow these steps when creating and ordering the rules to avoid path conflicts.

Procedure

1. Specify the paths.

To define the paths for each rule, use one of the following approaches. You can specify exact path names or, to match multiple paths, use the operators `{*}`, `{**}`, or the `/*` wildcard.

- Specify the exact path name.

For example, `/example/one` specifies the exact path `/example/one`, and `/` specifies the root path.

- Use the operators `{*}`, `{**}`, or the `/*` wildcard.

Pattern	Description	Examples
<code>{*}</code>	It matches a single path component, up to the next path separator <code>/</code> .	<ul style="list-style-type: none"> ■ <code>/example/{*}/one</code> - matches requests with the path prefix <code>example</code>, exactly one additional segment in the middle, and the path suffix <code>one</code>. For example, possible matches include <code>/example/anything/one</code>. ■ <code>/example/{*}</code> - matches requests with path prefix <code>example</code> and exactly one other segment. For example, possible matches include: <code>/example/anything</code>. The paths <code>/example/</code> and <code>/example/anything/</code> don't match.
<code>{**}</code>	<p>If it is the last element of the path, it matches zero or more path segments.</p> <p>If it is not the last element of the path, it matches one or more path segments.</p> <p>If multiple operators are present, <code>{**}</code> must be the last one.</p>	<ul style="list-style-type: none"> ■ <code>/example/{**}/one</code> - matches requests with the path prefix <code>example</code>, one or more additional segments in the middle, and the path suffix <code>one</code>. For example, possible matches include <code>/example/anything/two/one</code> and <code>/example/anything/one</code>. The paths <code>/example//one</code> and <code>/example/one</code> don't match. ■ <code>/example/{**}</code> - matches requests with the path prefix <code>example</code> followed by zero, one, or more additional segments. For example, possible matches include <code>/example/anything</code>, <code>/example/anything/more/</code>, and <code>/example/</code>. ■ <code>/{*}/example/{*}/{**}</code> - matches with any segment at the beginning, the path

Pattern	Description	Examples
		example, and at least one additional path segment. For example, possible matches include /anything/example/anything/ and /anything/example/anything/more.
/*	It matches all paths. It is equivalent to the path /{**}. If defined, /* must be the only element in the entire path. You cannot combine the wildcard path with any other operators or path segments. It was introduced for backward compatibility.	■ /* - matches / and any other path, such as /example/anything/more/ or /example/.

i Note

To be a valid path template, the path must not contain *, {, or } outside of the supported operators or the /* wildcard. No other characters are allowed in the path segment with the path template operator or the wildcard.

Using the wildcard and the operators introduces the possibility of path conflicts. A path conflict occurs when two or more APIRule spec.rules match the same path and share at least one common HTTP method. This is why it is important to consider the order of rules and to understand the connection between rules based on the path prefix and shared HTTP methods. Knowing the expected outcome of a configured rule helps organize and sort the rules.

2. Group paths into rules based on common HTTP methods and access strategies.

Specify HTTP methods that you want to allow for each path. If you want to allow more than one method for a path, you can use one of these approaches:

- Group multiple HTTP methods in a single rule. This is useful when you want to apply the same access strategy to multiple HTTP methods for the same endpoint.

For example, to allow both GET and POST requests to all service endpoints, you specify the /{**} operator in the path without authentication. In such a case, you can define a single rule with both methods:

↳ Sample Code

```
...
rules:
- path: /{**}
  methods:
    - GET
    - POST
  noAuth: true
```

- Create a separate rule for each HTTP method. Use this approach if you want to apply different access strategies or configurations to each method.

In this example, the outcome is the same as grouping multiple HTTP methods in one rule. You still allow both GET and POST requests to all service endpoints without authentication.

↳ Sample Code

```

rules:
  - path: /{**}
    methods:
      - GET
    noAuth: true
  - path: /{**}
    methods:
      - POST
    noAuth: true

```

i Note

You can group multiple HTTP methods in a single rule if they share the same access strategy, or create separate rules for each method to allow for applying different configurations. Both approaches are valid.

3. Order the rules.

Search for the paths that overlap. Overlapping occurs when two or more rules in an APIRule configuration contain paths that can match the same request and share at least one common HTTP method. This might lead to ambiguity about which rule applies to a given request.

When defining the order of rules, remember that each request is evaluated against the list of paths from top to bottom in your APIRule, and the first matching rule is applied. If a request matches the first path, the first rule applies. This is why you must define the rules starting from the most specific path, and leave the most general path as the last one.

i Note

Rules defined earlier in the list have a higher priority than those defined later. The request searches for the first matching rule starting from the top of the APIRule spec.rules list. Therefore, it's recommended to order rules from the most specific path to the most general.

See the following example of an incorrect rules' order:

Sample Code

```

...
rules:
  - path: /anything/{**}
    methods:
      - POST
      - GET
    noAuth: true
  - path: /anything/{*}/one
    methods:
      - POST
  jwt:
    authentications:
      - issuer: https://example.com
        jwksUri: https://example.com/.well-known/jwks.json

```

Applying this configuration causes an APIRule to be in Error state with the description:

Validation errors: Attribute '.spec.rules': Path /anything/{*}/one with method POST conflicts

In the following APIRule, the first rule specifically matches requests to the path `/anything/{*}/one` with the POST method, requiring JWT authentication. The second rule acts as a catch-all for any other paths that start with `/anything/`, allowing unauthenticated POST and GET requests. By placing the more specific rule first, you ensure that requests to `/anything/{*}/one` are handled as intended, while all other matching paths are covered by the more general rule. This approach prevents the more general rule from overshadowing the specific one and ensures the correct access strategy is applied to each path.

Sample Code

```
...
rules:
  - path: /anything/{*}/one
    methods:
      - POST
    jwt:
      authentications:
        - issuer: https://example.com
          jwksUri: https://example.com/.well-known/jwks.json
  - path: /anything/{**}
    methods:
      - POST
      - GET
    noAuth: true
```

You can test the above APIRule configuration using the following requests:

Request	Rule Matched	Access Strategy	Expected Outcome	HTTP Status Code
<code>curl -ik -X GET https://{{SUBDOMAIN}}.{{DOMAIN_NAME}}/anything/more</code>	Second rule (/anything/{**})	noAuth	Unauthenticated GET to /anything/more allowed	200 OK
<code>curl -ik -X POST https://{{SUBDOMAIN}}.{{DOMAIN_NAME}}/anything/more</code>	Second rule (/anything/{**})	noAuth	Unauthenticated POST to /anything/more allowed	200 OK
<code>curl -ik -X POST https://{{SUBDOMAIN}}.{{DOMAIN_NAME}}/anything/more/one --header "Authorization: Bearer <valid_jwt_token>"</code>	First rule (/anything/{*}/one)	JWT (required)	Authenticated POST to /anything/more/one allowed	200 OK
<code>curl -ik -X POST https://{{SUBDOMAIN}}.{{DOMAIN_NAME}}/anything/more/one</code>	First rule (/anything/{*}/one)	JWT (required)	Unauthenticated POST to /anything/more/one denied	403 RBAC: access denied
<code>curl -ik -X GET https://{{SUBDOMAIN}}.{{DOMAIN_NAME}}/anything/more/one</code>	No matching rule	-	GET to /anything/more/one not allowed (no rule matches)	403 RBAC: access denied

4. Check for excluding rules that share common methods.

i Note

Understanding the relationship between paths and methods in a rule is crucial to avoid unexpected behavior. If a rule shares at least one common method with a preceding rule, then the path from preceding rule is excluded from this rule.

For example, the following APIRule configuration excludes the POST and GET methods for the path `/anything/one` with the `noAuth` access strategy. This happens because the rule with the path `/anything/{**}` shares one common HTTP method POST with the preceding rule with the path `/anything/one`.

↳ Sample Code

```
...
rules:
  - methods:
    - POST
    jwt:
      authentications:
        - issuer: https://example.com
          jwksUri: https://example.com/.well-known/jwks.json
      path: /anything/one
  - methods:
    - GET
    - POST
    noAuth: true
    path: /anything/{**}
```

This outcome might be unexpected if you intended to allow GET requests to `/anything/one` without authentication. To achieve that, you must specifically define separate rules for overlapping methods and paths. The following APIRule configuration allows POST requests to `/anything/one` with JWT authentication, while permitting unauthenticated POST requests to all paths starting with `/anything/` except for the `/anything/one` endpoint. Additionally, it allows unauthenticated GET requests to all paths prefixed with `/anything/`. See the following example:

↳ Sample Code

```
...
rules:
  - methods:
    - POST
    jwt:
      authentications:
        - issuer: https://example.com
          jwksUri: https://example.com/.well-known/jwks.json
      path: /anything/one
  - methods:
    - POST
    noAuth: true
    path: /anything/{**}
  - methods:
    - GET
    noAuth: true
    path: /anything/{**}
```

You can test the above APIRule configuration using the following requests:

Request	Rule Matched	Access Strategy	Expected Outcome	HTTP Status Code
<code>curl -ik -X GET https://{{SUBDOMAIN}}.{{DOMAIN_NAME}}/anything/more</code>	Third rule (/anything/{**})	noAuth	Unauthenticated GET to /anything/more allowed	200 OK
<code>curl -ik -X POST https://{{SUBDOMAIN}}.{{DOMAIN_NAME}}/anything/more</code>	Second rule (/anything/{**})	noAuth	Unauthenticated POST to /anything/more allowed	200 OK
<code>curl -ik -X POST https://{{SUBDOMAIN}}.{{DOMAIN_NAME}}/anything/more/one --header "Authorization: Bearer <valid_jwt_token>"</code>	First rule (/anything/{*}/one)	JWT (required)	Authenticated POST to /anything/more/one allowed	200 OK
<code>curl -ik -X POST https://{{SUBDOMAIN}}.{{DOMAIN_NAME}}/anything/more/one</code>	First rule (/anything/{*}/one)	JWT (required)	Unauthenticated POST to /anything/more/one denied	403 RBAC: access denied
<code>curl -ik -X GET https://{{SUBDOMAIN}}.{{DOMAIN_NAME}}/anything/more/one</code>	Third rule (/anything/{**})	noAuth	Unauthenticated GET to /anything/more/one allowed)	200 OK

Using a Short Host Name

Learn how to expose a Service instance using a short host name instead of the full domain name.

Prerequisites

- You have the Istio and API Gateway modules added. See [Adding and Deleting a Kyma Module](#).
- You have installed [curl](#) and [kubectl](#). You have configured kubectl to communicate with your Kyma runtime instance. See [Access a Kyma Instance Using kubectl](#).
- You have a deployed workload.

i Note

To expose a workload using APIRule in version v2, the workload must be a part of the Istio service mesh. See [Enabling Istio Sidecar Proxy Injection](#).

- You have [set up your custom domain](#). Alternatively, you can use the default domain of your Kyma cluster and the default Gateway kyma-system/kyma-gateway.

i Note

Because the default Kyma domain is a wildcard domain, which uses a simple TLS Gateway, it is recommended that you set up your custom domain for use in a production environment.

→ Tip

To learn what the default domain of your Kyma cluster is, run the following command:

```
kubectl get gateway -n kyma-system kyma-gateway -o jsonpath='{.spec.servers[0].hosts}'
```

Context

Using a short host makes it simpler to apply an APIRule custom resource (CR) because the domain name is automatically retrieved from the referenced Gateway, and you don't have to manually set it in each APIRule. This might be particularly useful when reconfiguring resources in a new cluster, as it reduces the risk of errors and streamlines the process.

Procedure

1. To expose your workload using a short host, create an APIRule CR and define only a subdomain in the hosts field. See the following example:

```
cat <<EOF | kubectl apply -f -
apiVersion: gateway.kyma-project.io/v2alpha1
kind: APIRule
metadata:
  name: {APIRULE_NAME}
  namespace: {APIRULE_NAMESPACE}
spec:
  hosts:
    - {SUBDOMAIN}
  service:
    name: {SERVICE_NAME}
    namespace: {SERVICE_NAMESPACE}
    port: {SERVICE_PORT}
  gateway: {NAMESPACE/GATEWAY}
  rules:
    - path: /*
      methods: ["GET"]
      noAuth: true
    - path: /post
      methods: ["POST"]
      noAuth: true
EOF
```

2. Send a GET request to the exposed workload:

```
curl -ik -X GET https://{SUBDOMAIN}.{DOMAIN_NAME}/ip
```

If successful, the call returns the 200 OK response code.

3. Send a POST request to the exposed workload:

```
curl -ik -X POST https://{SUBDOMAIN}.{DOMAIN_NAME}/post -d "test data"
```

If successful, the call returns the 200 OK response code.

Exposing Multiple Workloads

Learn how to expose multiple workloads with one APIRule definition.

Prerequisites

- You have the Istio and API Gateway modules added. See [Adding and Deleting a Kyma Module](#).
- You have access to Kyma dashboard. Alternatively, to use CLI instructions, you must install [kubectl](#) and configure it to communicate with your Kyma runtime instance. See [Access a Kyma Instance Using kubectl](#).

- You have installed [curl](#).
- You have deployed two workloads in one namespace.

i Note

To expose a workload using APIRule in version v2, the workload must be a part of the Istio service mesh. See [Enabling Istio Sidecar Proxy Injection](#).

- You have [set up your custom domain](#). Alternatively, you can use the default domain of your Kyma cluster and the default Gateway kyma-system/kyma-gateway.

i Note

Because the default Kyma domain is a wildcard domain, which uses a simple TLS Gateway, it is recommended that you set up your custom domain for use in a production environment.

→ Tip

To learn what the default domain of your Kyma cluster is, run the following command:

```
kubectl get gateway -n kyma-system kyma-gateway -o jsonpath='{.spec.servers[0].hosts}'
```

Defining Multiple Services on Different Paths

Context

Learn how to expose two Services on different paths at the `spec.rules` level without a root Service defined.

⚠ Caution

Exposing a workload to the outside world is a potential security vulnerability, so be careful. In a production environment, always secure the workload you expose.

Procedure

You can use either Kyma dashboard or kubectl.

- Use Kyma dashboard.
 1. Go to the namespace in which you want to create an APIRule CR.
 2. Go to **Discovery and Network > API Rules** and select **Create**.
 3. Provide the name of the APIRule CR.
 4. Add a Gateway.
 5. Add a rule with the configuration details of one Service.
 6. Add another rule with the configuration details of the other Service.
 7. Choose **Create**.
 8. To call the endpoints, send requests to the exposed Services, for example:

```
curl -ik -X GET https://{{SUBDOMAIN}}.{{DOMAIN_NAME}}/headers
curl -ik -X GET https://{{SUBDOMAIN}}.{{DOMAIN_NAME}}/get
```

If successful, the calls return the 200 OK response code.

- Use kubectl.

1. Add two rules with configuration details of your Services. See an example:

```
cat <<EOF | kubectl apply -f -
apiVersion: gateway.kyma-project.io/v2
kind: APIRule
metadata:
  name: {APIRULE_NAME}
  namespace: {APIRULE_NAMESPACE}
spec:
  hosts:
    - {SUBDOMAIN}.{DOMAIN_NAME}
  gateway: {GATEWAY_NAMESPACE}/{GATEWAY_NAME}
  rules:
    - path: /headers
      methods: ["GET"]
      noAuth: true
      service:
        name: {FIRST_SERVICE_NAME}
        port: {FIRST_SERVICE_PORT}
    - path: /get
      methods: ["GET"]
      noAuth: true
      service:
        name: {SECOND_SERVICE_NAME}
        port: {SECOND_SERVICE_PORT}
EOF
```

2. To call the endpoints, send requests to the exposed Services, for example:

```
curl -ik -X GET https://{SUBDOMAIN}.{DOMAIN_NAME}/headers
curl -ik -X GET https://{SUBDOMAIN}.{DOMAIN_NAME}/get
```

If successful, the calls return the 200 OK response code.

Defining a Service at the Root Level

Context

You can also define a Service at the root level. Such a definition is applied to all the paths specified at **spec.rules** that do not have their own Services defined. Services defined at the **spec.rules** level have precedence over Service definition at the **spec.service** level.

⚠ Caution

Exposing a workload to the outside world is a potential security vulnerability, so be careful. In a production environment, always secure the workload you expose.

Procedure

You can use either Kyma dashboard or kubectl.

- Use Kyma dashboard.

1. Go to the namespace in which you want to create an APIRule CR.
2. Go to **Discovery and Network > API Rules** and select **Create**.
3. Provide the name of the APIRule CR.

4. Add a Gateway.
5. Define one Service in the **Service** section.
6. Add one rule without a Service definition.
7. Add another rule with the Service definition.
8. Choose **Create**.
9. To call the endpoints, send requests to the exposed Services, for example:

```
curl -ik -X GET https://{{SUBDOMAIN}}.{{DOMAIN_NAME}}/headers
curl -ik -X GET https://{{SUBDOMAIN}}.{{DOMAIN_NAME}}/get
```

If successful, the calls return the 200 OK response code.

- Use kubectl.

 1. Replace the placeholders and run the following command:

```
cat <<EOF | kubectl apply -f -
apiVersion: gateway.kyma-project.io/v2
kind: APIRule
metadata:
  name: {{APIRULE_NAME}}
  namespace: {{APIRULE_NAMESPACE}}
spec:
  hosts:
    - {{SUBDOMAIN}}.{{DOMAIN_NAME}}
  gateway: {{GATEWAY_NAMESPACE}}/{{GATEWAY_NAME}}
  service:
    name: {{FIRST_SERVICE_NAME}}
    port: {{FIRST_SERVICE_PORT}}
  rules:
    - path: /headers
      methods: ["GET"]
      noAuth: true
    - path: /get
      methods: ["GET"]
      noAuth: true
      service:
        name: {{SECOND_SERVICE_NAME}}
        port: {{SECOND_SERVICE_PORT}}
EOF
```

2. To call the endpoints, send GET requests to the exposed Services:

```
curl -ik -X GET https://{{SUBDOMAIN}}.{{DOMAIN_NAME}}/headers
curl -ik -X GET https://{{SUBDOMAIN}}.{{DOMAIN_NAME}}/get
```

If successful, the calls return the 200 OK response code.

Exposing Services in Different Namespaces

Context

Expose two workloads deployed in different namespaces with one APIRule definition.

Procedure

You can use either Kyma dashboard or kubectl.

- Use Kyma dashboard.

1. Create a namespace with the Istio sidecar proxy injection enabled.

For more information, see [Enabling Istio Sidecar Proxy Injection](#).

2. Go to **Discovery and Network > API Rules** and select **Create**.

3. Switch to the **YAML** section.

4. Paste the APIRule custom resource (CR) configuration into the editor. You must specify a Service's namespace in each Service definition. See an example:

```
apiVersion: gateway.kyma-project.io/v2
kind: APIRule
metadata:
  name: {APIRULE_NAME}
  namespace: {APIRULE_NAMESPACE}
spec:
  hosts:
    - {SUBDOMAIN}.{DOMAIN_NAME}
  gateway: {GATEWAY_NAMESPACE}/{GATEWAY_NAME}
  rules:
    - path: /headers
      methods: ["GET"]
      service:
        name: {FIRST_SERVICE_NAME}
        namespace: {FIRST_SERVICE_NAMESPACE}
        port: {FIRST_SERVICE_PORT}
        noAuth: true
    - path: /get
      methods: ["GET"]
      service:
        name: {SECOND_SERVICE_NAME}
        namespace: {SECOND_SERVICE_NAMESPACE}
        port: {SECOND_SERVICE_PORT}
        noAuth: true
```

5. Select **Create**.

6. To call the endpoints, send GET requests to the exposed Services:

```
curl -ik -X GET https://{SUBDOMAIN}.{DOMAIN_NAME}/headers
curl -ik -X GET https://{SUBDOMAIN}.{DOMAIN_NAME}/get
```

If successful, the calls return the 200 OK response code.

- Use kubectl.

1. Create a separate namespace for the APIRule CR with enabled Istio sidecar proxy injection.

```
export NAMESPACE={NAMESPACE_NAME}
kubectl create ns $NAMESPACE
kubectl label namespace $NAMESPACE istio-injection=enabled --overwrite
```

2. Expose the Services in their respective namespaces by creating an APIRule custom resource (CR) in its own namespace. You must specify a Service's namespace in each Service definition. See an example:

```
cat <<EOF | kubectl apply -f -
apiVersion: gateway.kyma-project.io/v2
kind: APIRule
metadata:
  name: {APIRULE_NAME}
  namespace: $NAMESPACE
spec:
  hosts:
    - {SUBDOMAIN}.{DOMAIN_NAME}
  gateway: {GATEWAY_NAMESPACE}/{GATEWAY_NAME}
  rules:
    - path: /headers
      methods: ["GET"]
      service:
        name: {FIRST_SERVICE_NAME}
        namespace: {FIRST_SERVICE_NAMESPACE}
```

```

        port: {FIRST_SERVICE_PORT}
        noAuth: true
      - path: /get
        methods: ["GET"]
        service:
          name: {SECOND_SERVICE_NAME}
          namespace: {SECOND_SERVICE_NAMESPACE}
          port: {SECOND_SERVICE_PORT}
        noAuth: true
EOF

```

- To call the endpoints, send requests to the exposed Services, for example:

```

curl -ik -X GET https://{{SUBDOMAIN}}.{{DOMAIN_NAME}}/headers
curl -ik -X GET https://{{SUBDOMAIN}}.{{DOMAIN_NAME}}/get

```

If successful, the calls return the 200 OK response code.

APIRule Migration

APIRule custom resource (CR) v1beta1 has been deprecated and scheduled for deletion. You must migrate all your APIRule CRs to version v2. Learn more about the timeline and see how to perform the migration.

How to Migrate APIRules to Version v2

To migrate to version v2, follow the steps:

- To identify which APIRules must be migrated, run the following command:

```
kubectl get apirules.gateway.kyma-project.io -A -o json | jq '.items[] | select(.metadata.annotations["kyma-project.io/api-rule-type"] == "v1beta1")' | jq '.spec.rules[] | select(.method == "GET")'
```

- To retrieve the complete spec with the rules field of an APIRule in version v1beta1, see [Retrieving the Complete spec of an APIRule in Version v1beta1](#).

- To migrate an APIRule from version v1beta1 to version v2, follow the relevant guide:

- [Migrating APIRule v1beta1 of Type jwt to Version v2](#)
- [Migrating APIRule v1beta1 of Type noop, allow, or no_auth to Version v2](#)
- [Migrating APIRule v1beta1 of type oauth2_introspection to version v2](#)

For more information about APIRule v2, see also [APIRule v2 Custom Resource](#) and [Changes Introduced in APIRule v2](#).

APIRule v1beta1 Migration Timeline

The APIRule v1beta1 deletion process is divided into three steps. See the following timeline:

Step 1: Completed with API Gateway 3.2

Kyma dashboard doesn't display APIRule CRs in version v1beta1. All APIRules v1beta1 are fully operational from the command line, and you are still able to manage them using kubectl. This change does not affect any pipelines that manage APIRules using Kubernetes utilities such as kubectl or helm.

Step 2: Planned for API Gateway 3.3

You won't be able to create APIRule CRs `v1beta1` in new clusters. In existing clusters, you will still be able to create and modify APIRule CRs `v1beta1`.

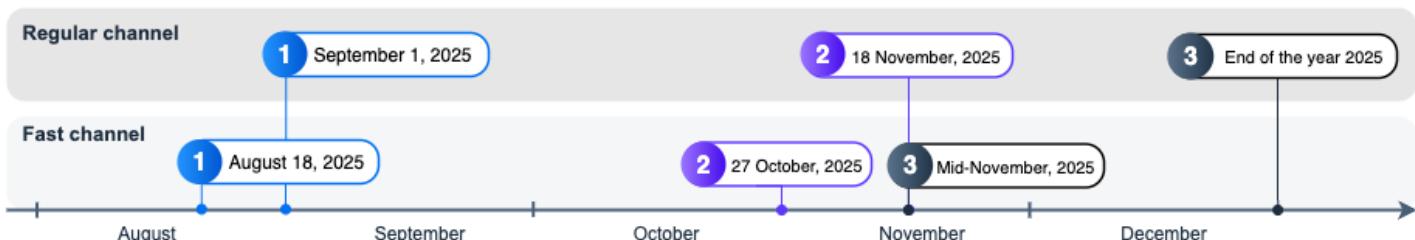
- Fast channel release date: 27 October, 2025
- Regular channel release date: 18 November, 2025

Step 3: Planned for API Gateway 3.4

You won't be able to create APIRule CRs `v1beta1` in new and existing clusters, modify existing APIRule CRs `v1beta1`, or delete them. All APIRule `v1beta1` configurations set up prior to this restriction will remain active and continue to function as expected. The API Gateway module will manage and reconcile resources based on the existing APIRule settings.

- Fast channel release date: Mid-November, 2025
- Regular channel release date: End of the year 2025

Production landscape



Changes Introduced in APIRule v2

Learn about the changes that APIRule v2 introduces and the actions you must take to adjust your `v1beta1` resources.

A Workload Must Be in the Istio Service Mesh

To use APIRules in version v2, the workload that an APIRule exposes must be in the Istio service mesh. If the workload is not inside the Istio service mesh, the APIRule does not work as expected.

Required action: To add a workload to the Istio service mesh, see [Enabling Istio Sidecar Proxy Injection](#).

Internal Traffic to Workloads Is Blocked by Default

By default, access to the workload from internal traffic is blocked if APIRule CR in version v2 is applied. This approach aligns with Kyma's "secure by default" principle.

CORS Policy Is Not Applied by Default

Version `v1beta1` applied the following CORS configuration by default:

```
corsPolicy:
  allowOrigins: ["*"]
  allowMethods: ["GET", "POST", "PUT", "DELETE", "PATCH"]
  allowHeaders: ["Authorization", "Content-Type", "*"]
```

Version v2 does not apply these default values. If the `corsPolicy` field is empty, the CORS configuration is not applied. For more information, see [architecture decision record #752](#).

Required action: Configure CORS policy in the `corsPolicy` field.

→ Remember

Since not all APIs require the same level of access, adjust your CORS policy configuration according to your application's specific needs and security requirements.

If you decide to use the default CORS values defined in the APIRule `v1beta1`, you must explicitly define them in your APIRule v2. For preflight requests to work, you must allow the "OPTIONS" method in the `rules.methods` field of your APIRule.

Path Specification Must Not Contain Regexp

APIRule in version v2 does not support regexp in the `spec.rules.path` field of APIRule CR. Instead, it supports the use of the `{*}` and `{**}` operators. See the supported configurations:

- Use the exact path (for example, `/abc`). It matches the specified path exactly.
- Use the `{*}` operator (for example, `/foo/{*}` or `/foo/{*}/bar`). This operator represents any request that matches the given pattern, with exactly one path segment replacing the operator.
- Use the `{**}` operator (for example, `/foo/{**}` or `/foo/{**}/bar`). This operator represents any request that matches the pattern with zero or more path segments in the operator's place. It must be the last operator in the path.
- Use the wildcard path `/*`, which matches all paths. It's equivalent to the `/{**}` path. If your configuration in APIRule `v1beta1` used such a path as `/foo(.*)`, when migrating to the new versions, you must define configurations for two separate paths: `/foo` and `/foo/{**}`.

i Note

The order of rules in the APIRule CR is important. Rules defined earlier in the list have a higher priority than those defined later. Therefore, we recommend defining rules from the most specific path to the most general.

Operators allow you to define a single APIRule that matches multiple request paths. However, this also introduces the possibility of path conflicts. A path conflict occurs when two or more APIRule resources match the same path and share at least one common HTTP method. This is why the order of rules is important.

For more information on the APIRule specification, see [APIRule v2 Custom Resource](#).

Required action: Replace regexp expressions in the `spec.rules.path` field of your APIRule CRs with the `{*}` and `{**}` operators.

JWT Configuration Requires Explicit Issuer URL

Version v2 of APIRule introduces an additional mandatory configuration field for JWT-based authorization - `issuer`. You must provide an explicit issuer URL in the APIRule CR. See an example configuration:

```
rules:
- jwt:
  authentications:
    - issuer: {YOUR_ISSUER_URL}jwksUri: {YOUR_JWKS_URI}
```

If you use Cloud Identity Services, you can find the issuer URL in the OIDC well-known configuration at https://{{YOUR_TENANT}}.accounts.ondemand.com/.well-known/openid-configuration.

Required action: Add the `issuer` field to your APIRule specification. For more information, see [Migrating APIRule v1beta1 of Type jwt to Version v2](#).

Removed Support for Oathkeeper OAuth2 Handlers

The APIRule CR in version v2 does not support Oathkeeper OAuth2 handlers. Instead, it introduces the `extAuth` field, which you can use to configure an external authorizer.

Required action: Migrate your Oathkeeper-based OAuth2 handlers to use an external authorizer. To learn how to do this, see [Migrating APIRule v1beta1 of type oauth2_introspection to version v2](#) and [Configuration of the extAuth Access Strategy](#).

Removed Support for Oathkeeper Mutators

The APIRule CR in version v2 does not support Oathkeeper mutators. Request mutators are replaced with request modifiers defined in the `spec.rule.request` section of the APIRule CR. This section contains the request modification rules applied before the request is forwarded to the target workload. Token mutators are not supported in APIRule v2. For that, you must define your own `extAuth` configuration.

Required action: Migrate your rules that rely on Oathkeeper mutators to use request modifiers or an external authorizer. For more information, see [Configuration of the extAuth Access Strategy](#).

Removed Support for Opaque Tokens

The APIRule CR in version v2 does not support the usage of Opaque tokens. Instead, it introduces the `extAuth` field, which you can use to configure an external authorizer.

Required action: Migrate your rules that use Opaque tokens to use an external authorizer. For more information, see [Configuration of the extAuth Access Strategy](#).

Retrieving the Complete spec of an APIRule in Version v1beta1

Learn how to retrieve the complete `spec` of an APIRule originally applied in version v1beta1 when the displayed `spec` does not contain the `rules` field. To do this, you can either use Kyma dashboard or the `kubectl get` command.

Prerequisites

- You have the Istio and API Gateway modules added.
- You have a deployed workload exposed by an APIRule in the deprecated v1beta1 version.
- To use CLI instructions, you must have [kubectl](#) and [yq](#). Alternatively, you can use Kyma dashboard.

Context

APIRule in version v1beta1 is deprecated and scheduled for removal. Once the APIRule custom resource definition (CRD) stops serving version v1beta1, the API server will no longer respond to requests for APIRules in this version. Consequently, you will encounter errors attempting to access the APIRule custom resource at the deprecated v1beta1 version.

This creates a migration challenge: If your APIRule was originally created using `v1beta1` and you have not yet migrated to `v2`, you may notice that the `spec` is missing the `rules` field when viewed in Kyma dashboard or via the `kubectl get` command.

For example, suppose you have applied the following APIRule in version `v1beta1`:

↳ Sample Code

```
apiVersion: gateway.kyma-project.io/v1beta1
kind: APIRule
metadata:
  name: httpbin
  namespace: test
spec:
  gateway: kyma-system/kyma-gateway
  host: httpbin
  rules:
    - accessStrategies:
        - handler: noop
          methods:
            - POST
          path: /anything
    - accessStrategies:
        - handler: allow
          methods:
            - HEAD
          path: /headers
    - accessStrategies:
        - handler: no_auth
          methods:
            - GET
          path: /.*
  service:
    name: httpbin
    namespace: test
    port: 8000
```

When retrieving this APIRule, the resulting `spec` does not include the `rules` field, as these rules cannot be converted to `v2`:

```
kubectl get apirules.gateway.kyma-project.io -n $NAMESPACE $APIRULE_NAME -oyaml
```

↳ Sample Code

```
...
spec:
  gateway: kyma-system/kyma-gateway
  hosts:
    - httpbin
  service:
    name: httpbin
    namespace: test
    port: 8000
...
...
```

In this case, you must access the original APIRule `v1beta1` configuration through an annotation. To learn how to do this, follow the procedure.

Procedure

- Use Kyma dashboard.
 1. Go to **Discovery and Network > API Rules** and select specific APIRule CR in version v1beta1.
 2. Go to the **Edit** tab.
 3. Copy the value of `metadata.annotations.gateway.kyma-project.io/v1beta1-spec` as it stores the full original configuration of the APIRule created in version v1beta1.
- Use kubectl.

1. To get the full original **spec** of the APIRule created in version v1beta1, use the annotation that stores the original configuration. Run:

```
kubectl get apirules.gateway.kyma-project.io -n $NAMESPACE $APIRULE_NAME -o jsonpath='{.r
```

See a sample output in the JSON format:

Sample Code

```
{"host": "httpbin", "service": {"name": "httpbin", "namespace": "test", "port": 8000}, "gatewa
```

2. To format the output as YAML for better readability, use the `yq` command.

```
kubectl get apirules.gateway.kyma-project.io -n $NAMESPACE $APIRULE_NAME -o jsonpath='{.r
```

See a sample output in the YAML format:

Sample Code

```
host: httpbin
service:
  name: httpbin
  namespace: test
  port: 8000
gateway: kyma-system/kyma-gateway
rules:
  - path: /anything
    methods:
      - POST
    accessStrategies:
      - handler: noop
  - path: /headers
    methods:
      - HEAD
    accessStrategies:
      - handler: allow
  - path: /.*
    methods:
      - GET
    accessStrategies:
      - handler: no_auth
```

Next Steps

Next, adjust the obtained configuration of the APIRule to migrate it to version v2. To learn how to do this, follow the relevant tutorial:

- [Migrating APIRule v1beta1 of Type jwt to Version v2](#)
- [Migrating APIRule v1beta1 of Type noop, allow, or no_auth to Version v2](#)

- [Migrating APIRule v1beta1 of type oauth2_introspection to version v2](#)

Migrating APIRule v1beta1 of Type jwt to Version v2

Learn how to migrate an APIRule created in version v1beta1 using the jwt handler to version v2

Prerequisites

- You have read [Changes Introduced in APIRule v2](#), which details the updates implemented in the new version of APIRule. If any of these changes affect your setup, you must consider them when migrating to APIRule v2 and make the necessary adjustments.
- You have the Istio and API Gateway modules added.
- You have installed [kubectl](#) and [curl](#).
- You have a deployed workload exposed by an APIRule in the deprecated v1beta1 version. The APIRule uses the jwt handler.

i Note

To expose a workload using APIRule in version v2, the workload must be a part of the Istio service mesh. See [Enabling Istio Sidecar Proxy Injection](#).

Context

APIRule in version v1beta1 is deprecated and scheduled for removal. Once the APIRule custom resource definition (CRD) stops serving version v1beta1, the API server will no longer respond to requests for APIRules in this version. As a result, you will encounter errors when attempting to access the APIRule custom resource using the deprecated v1beta1 version. Therefore, you must migrate to version v2.

This example demonstrates a migration from an APIRule v1beta1 with the jwt handler to an APIRule v2 with the jwt handler. The example uses an HTTPBin service, exposing the /anything and /. * endpoints. The HTTPBin service is deployed in its own namespace, with Istio enabled, ensuring the workload is part of the Istio service mesh.

Procedure

1. Retrieve a configuration of the APIRule in version v1beta1 and save it for further modifications. For instructions, see [Retrieving the Complete spec of an APIRule in Version v1beta1](#).

See a sample of the retrieved spec in the YAML format. The following configuration uses the jwt handler to expose the HTTPBin service's /anything and /. * endpoints.

☰ Sample Code

```
host: httpbin.local.kyma.dev
service:
  name: httpbin
  namespace: test
  port: 8000
gateway: kyma-system/kyma-gateway
rules:
  - path: /anything
    methods:
      - POST
    accessStrategies:
      - handler: jwt
```

```

config:
  jwks_urls:
    - https://IAS_TENANT.accounts.ondemand.com/oauth2/certs
- path: /.*
  methods:
    - GET
  accessStrategies:
    - handler: jwt
      config:
        jwks_urls:
          - https://IAS_TENANT.accounts.ondemand.com/oauth2/certs

```

2. Adjust the retrieved configuration to align with the `jwt` configuration of APIRule v2.

To ensure the APIRule specification is compatible with version v2, you must include a mandatory field named `issuer` in the `jwt` handler's configuration.

You can find the `issuer` URL in the OIDC well-known configuration of your tenant, located at `https://YOUR_TENANT.accounts.ondemand.com/.well-known/openid-configuration`. Additionally, note that the value of the `jwks_urls` field is now stored in the `jwksUri` field. Tokens do not need to be reissued unless they have expired.

See an example of the adjusted APIRule configuration for version v2:

Sample Code

```

apiVersion: gateway.kyma-project.io/v2
kind: APIRule
metadata:
  name: httpbin
  namespace: test
spec:
  hosts:
    - httpbin
  service:
    name: httpbin
    namespace: test
    port: 8000
  gateway: kyma-system/kyma-gateway
  rules:
    - jwt:
        authentications:
          - issuer: https://YOUR_TENANT.accounts.ondemand.com
            jwksUri: https://YOUR_TENANT.accounts.ondemand.com/oauth2/certs
        methods:
          - POST
        path: /anything
    - jwt:
        authentications:
          - issuer: https://YOUR_TENANT.accounts.ondemand.com
            jwksUri: https://YOUR_TENANT.accounts.ondemand.com/oauth2/certs
        methods:
          - GET
        path: /{**}

```

Note

The `hosts` field accepts a short host name (without a domain). Additionally, the path `/.*` has been changed to `/{**}` because APIRule v2 does not support regular expressions in the `spec.rules.path` field.

For more information, see [Changes Introduced in APIRule v2](#).

3. Update the APIRule to version v2 by applying the adjusted configuration.

To verify the version of the applied APIRule, check the value of the `gateway.kyma-project.io/original-version` annotation in the APIRule spec. A value of v2 indicates that the APIRule has been successfully migrated. To see the value, run:

```
kubectl get apirules.gateway.kyma-project.io -n $NAMESPACE $APIRULE_NAME -oyaml
```

The following output indicates that the APIRule has been successfully migrated to version v2:

↳ Sample Code

```
apiVersion: gateway.kyma-project.io/v2
kind: APIRule
metadata:
  annotations:
    gateway.kyma-project.io/original-version: v2
  ...
  
```

⚠ Caution

Do not manually change the `gateway.kyma-project.io/original-version` annotation. This annotation is automatically updated when you apply your APIRule in version v2. Modifying the annotation's value manually causes your APIRule `v1beta1` to be handled and configured as version v2, potentially leading to reconciliation errors.

4. To preserve the internal traffic policy from the APIRule `v1beta1`, you must apply the following AuthorizationPolicy.

In APIRule v2, internal traffic is blocked by default. Without this AuthorizationPolicy, attempts to connect internally to the workload cause an RBAC: `access denied` error. Ensure that the selector label is updated to match the target workload.

Option	Description
{NAMESPACE}	The namespace to which the AuthorizationPolicy applies. This namespace must include the target workload for which you allow internal traffic. The selector matches workloads in the same namespace as the AuthorizationPolicy.
{LABEL_KEY}: {LABEL_VALUE}	To further restrict the scope of the AuthorizationPolicy, specify label selectors that match the target workload. Replace these placeholders with the actual key and value of the label. The label indicates a specific set of Pods to which a policy should be applied. The scope of the label search is restricted to the configuration namespace in which the AuthorizationPolicy is present. For more information, see Authorization Policy .

```
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
  name: allow-internal
  namespace: {NAMESPACE}
spec:
  selector:
    matchLabels:
      {LABEL_KEY}: {LABEL_VALUE}
  action: ALLOW
  rules:
  - from:
    - source:
        notPrincipals: ["cluster.local/ns/istio-system/sa/istio-ingressgateway-service-account"]
```

- 5. To retain the CORS configuration from the APIRule v1beta1, update the APIRule in version v2 to include the same CORS settings.

For preflight requests to work correctly, you must explicitly add the "OPTIONS" method to the `rules.methods` field of your APIRule v2. For guidance, see the [sample APIRule CRs](#).

Results

You have migrated your APIRule CR to version v2. Now, access your workload:

- Send a GET request to the exposed workload using JWT authentication:

Sample Code

```
curl -ik -X GET https://{{SUBDOMAIN}}.{{DOMAIN_NAME}}/ip --header "Authorization: Bearer $ACCESS_TOKEN"
```

If successful, the call returns the 200 OK response code.

- Send a POST request to the exposed workload using JWT authentication:

Sample Code

```
curl -ik -X POST https://{{SUBDOMAIN}}.{{DOMAIN_NAME}}/anything -d "test data" --header "Authorization: Bearer $ACCESS_TOKEN"
```

If successful, the call returns the 200 OK response code.

- Send a POST request to the exposed workload without JWT authentication:

Sample Code

```
curl -ik -X POST https://{{SUBDOMAIN}}.{{DOMAIN_NAME}}/anything -d "test data"
```

The call returns the 403 error code.

Migrating APIRule v1beta1 of Type noop, allow, or no_auth to Version v2

Learn how to migrate an APIRule created in version v1beta1 using the `noop`, `allow`, or `no_auth` handlers to version v2. In APIRule v2, the `noAuth` handler replaces all of the above handlers from the v1beta1 version.

Prerequisites

- You have read [Changes Introduced in APIRule v2](#), which details the updates implemented in the new version of APIRule. If any of these changes affect your setup, you must consider them when migrating to APIRule v2 and make the necessary adjustments.
- You have the Istio and API Gateway modules added.
- You have installed [kubectl](#) and [curl](#).
- You have a deployed workload exposed by an APIRule in the deprecated v1beta1 version. The APIRule uses the `noop`, `allow`, or `no_auth` handler.

i Note

To expose a workload using APIRule in version v2, the workload must be a part of the Istio service mesh. See [Enabling Istio Sidecar Proxy Injection](#).

Context

APIRule in version v1beta1 is deprecated and scheduled for removal. Once the APIRule custom resource definition (CRD) stops serving version v1beta1, the API server will no longer respond to requests for APIRules in this version. As a result, you will encounter errors when attempting to access the APIRule custom resource using the deprecated v1beta1 version. Therefore, you must migrate to version v2.

This example demonstrates a migration from an APIRule v1beta1 with noop, allow, and no_auth handlers to an APIRule v2 with the noAuth handler. The example uses an HTTPBin service, exposing the /anything, /headers, and /.* endpoints. The HTTPBin service is deployed in its own namespace, with Istio enabled, ensuring the workload is part of the Istio service mesh.

Procedure

1. Retrieve a configuration of the APIRule in version v1beta1 and save it for further modifications. For instructions, see [Retrieving the Complete spec of an APIRule in Version v1beta1](#).

See a sample of the retrieved spec in the YAML format. The following configuration uses these handlers to expose the HTTPBin endpoints:

- The noop handler to expose /anything
- The allow handler to expose /headers
- The no_auth handler to expose /.*

Sample Code

```
host: httpbin.local.kyma.dev
service:
  name: httpbin
  namespace: test
  port: 8000
gateway: kyma-system/kyma-gateway
rules:
  - path: /anything
    methods:
      - POST
    accessStrategies:
      - handler: noop
  - path: /headers
    methods:
      - HEAD
    accessStrategies:
      - handler: allow
  - path: /.*
    methods:
      - GET
    accessStrategies:
      - handler: no_auth
```

2. Adjust the obtained configuration to match the v2 APIRule specification by replacing the noop, allow, and no_auth handlers with the noAuth handler.

To do this, you must modify the existing APIRule spec and ensure it is valid for the v2 version of the noAuth type. See an example of the adjusted APIRule:

↳ Sample Code

```
apiVersion: gateway.kyma-project.io/v2
kind: APIRule
metadata:
  name: httpbin
  namespace: test
spec:
  hosts:
    - httpbin
  service:
    name: httpbin
    namespace: test
    port: 8000
  gateway: kyma-system/kyma-gateway
  rules:
    - path: /anything
      methods: ["POST"]
      noAuth: true
    - path: /headers
      methods: ["HEAD"]
      noAuth: true
    - path: /{**}
      methods: ["GET"]
      noAuth: true
```

i Note

The `hosts` field accepts a short host name (without a domain). Additionally, the path `/ . *` has been changed to `/ {**}` because APIRule v2 does not support regular expressions in the `spec.rules.path` field.

For more information, see [Changes Introduced in APIRule v2](#).

- To update the APIRule to version v2, apply the adjusted configuration.

To verify the version of the applied APIRule, check the value of the `gateway.kyma-project.io/original-version` annotation in the APIRule spec. If the APIRule has been successfully migrated, you see the value v2. Use the following command:

```
kubectl get apirules.gateway.kyma-project.io -n $NAMESPACE $APIRULE_NAME -oyaml
```

The following output indicates that the APIRule has been successfully migrated to version v2:

↳ Sample Code

```
apiVersion: gateway.kyma-project.io/v2
kind: APIRule
metadata:
  annotations:
    gateway.kyma-project.io/original-version: v2
...
...
```

⚠ Caution

Do not manually change the `gateway.kyma-project.io/original-version` annotation. This annotation is automatically updated when you apply your APIRule in version v2. Modifying the annotation's value manually causes your APIRule `v1beta1` to be handled and configured as version v2, potentially leading to reconciliation errors.

- To preserve the internal traffic policy from the APIRule `v1beta1`, you must apply the following AuthorizationPolicy.

In APIRule v2, internal traffic is blocked by default. Without this AuthorizationPolicy, attempts to connect internally to the workload cause an RBAC: `access denied` error. Ensure that the selector label is updated to match the target workload.

Option	Description
{NAMESPACE}	The namespace to which the AuthorizationPolicy applies. This namespace must include the target workload for which you allow internal traffic. The selector matches workloads in the same namespace as the AuthorizationPolicy.
{LABEL_KEY}: {LABEL_VALUE}	<p>To further restrict the scope of the AuthorizationPolicy, specify label selectors that match the target workload. Replace these placeholders with the actual key and value of the label. The label indicates a specific set of Pods to which a policy should be applied. The scope of the label search is restricted to the configuration namespace in which the AuthorizationPolicy is present.</p> <p>For more information, see Authorization Policy.</p>

```

apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
  name: allow-internal
  namespace: {NAMESPACE}
spec:
  selector:
    matchLabels:
      {LABEL_KEY}: {LABEL_VALUE}
  action: ALLOW
  rules:
  - from:
    - source:
        notPrincipals: ["cluster.local/ns/istio-system/sa/istio-ingressgateway-service-account"]

```

5. To retain the CORS configuration from the APIRule v1beta1, update the APIRule in version v2 to include the same CORS settings.

For preflight requests to work correctly, you must explicitly add the "OPTIONS" method to the rules.methods field of your APIRule v2. For guidance, see the [sample APIRule CRs](#).

Results

You have migrated your APIRule CR to version v2. Now, access your workload:

- Send a POST request to the exposed workload:

Sample Code

Send a POST request to the exposed workload:

If successful, the call returns the 200 OK response code.

- Send a HEAD request to the exposed workload:

Sample Code

```
curl -ik -I https://{{SUBDOMAIN}}.{{DOMAIN_NAME}}/headers
```

If successful, the call returns the 200 OK response code.

- Send a GET request to the exposed workload:

Sample Code

```
curl -ik -X GET https://{{SUBDOMAIN}}.{{DOMAIN_NAME}}/ip
```

If successful, the call returns the 200 OK response code.

Migrating APIRule v1beta1 of type oauth2_introspection to version v2

Learn how to migrate an APIRule created in version v1beta1 using the oauth2_introspection handler to the extAuth handler in version v2. In APIRule v2, the extAuth handler replaces all Ory Oathkeeper-based handlers used in the v1beta1 version. The instructions focus on oauth2_introspection because it is the most popular Ory Oathkeeper-based handler.

Prerequisites

- You have read [Changes Introduced in APIRule v2](#), which details the updates implemented in the new version of APIRule. If any of these changes affect your setup, you must consider them when migrating to APIRule v2 and make the necessary adjustments.
- You have the Istio and API Gateway modules added.
- You have installed [kubectl](#) and [curl](#).
- You have a deployed workload exposed by an APIRule in the deprecated v1beta1 version. The APIRule uses the jwt handler.

Note

To expose a workload using APIRule in version v2, the workload must be a part of the Istio service mesh. See [Enabling Istio Sidecar Proxy Injection](#).

Context

APIRule in version v1beta1 is deprecated and scheduled for removal. Once the APIRule custom resource definition (CRD) stops serving version v1beta1, the API server will no longer respond to requests for APIRules in this version. As a result, you will encounter errors when attempting to access the APIRule custom resource using the deprecated v1beta1 version. Therefore, you must migrate to version v2.

In this example, the APIRule v1beta1 was created with the oauth2_introspection handler, so the migration targets an APIRule v2 using the extAuth handler. To illustrate the migration, the HTTPBin service is used, exposing the /anything and /.* endpoints. The HTTPBin service is deployed in its own namespace, with Istio enabled, ensuring the workload is part of the Istio service mesh.

Procedure

1. Retrieve a configuration of the APIRule in version v1beta1 and save it for further modifications. For instructions, see [Retrieving the Complete spec of an APIRule in Version v1beta1](#).

See a sample of the retrieved spec in the YAML format. The following configuration uses the oauth2_introspection handler to expose HTTPBin service's /anything and /.* endpoints:

Sample Code

```

host: httpbin.local.kyma.dev
service:
  name: httpbin
  namespace: test
  port: 8000
gateway: kyma-system/kyma-gateway
rules:
  - path: /anything
    methods:
      - POST
    accessStrategies:
      - handler: oauth2_introspection
        config:
          introspection_request_headers:
            Authorization: Basic {ENCODED_CREDENTIALS}
          introspection_url: https://{IAS_TENANT}.accounts.ondemand.com/oauth2/introspect
          required_scope:
            - write
  - path:.*/
    methods:
      - GET
    accessStrategies:
      - handler: oauth2_introspection
        config:
          introspection_request_headers:
            Authorization: Basic {ENCODED_CREDENTIALS}
          introspection_url: https://{IAS_TENANT}.accounts.ondemand.com/oauth2/introspect
          required_scope:
            - read

```

2. In order for the extAuth handler in APIRule v2 to work, you must first deploy a service that acts as an external authorizer for Istio.

The following instructions use [OAuth2 Proxy](#) with an OAuth2.0-compliant authorization server supporting OIDC discovery.

a. Replace the placeholders and create the `values.yaml` file with the OAuth2 Proxy configuration.

The following example shows the configuration of OAuth2 Proxy with the following parameters:

- `CLIENT_SECRET`, `CLIENT_ID`, and `OIDC_ISSUER_URL`. To get them, follow [Get a JSON Web Token \(JWT\)](#).
- `DOMAIN_TO_EXPOSE_WORKLOADS` refers to either a custom domain or, as in this example, the default domain `local.kyma.dev`.
- `COOKIE_SECRET` that you can generate using the following command:

```
openssl rand -base64 32 | tr -- '+/' '_'
```

- `TOKEN_SCOPES` specifies the OAuth scopes. Each provider has a default set of scopes that are used if you haven't configured custom scopes.

For a list of options and further details, refer to the [OAuth2 Proxy documentation](#).

↳ Sample Code

```

cat <<EOF > values.yaml
config:
  clientId: {CLIENT_ID}
  clientSecret: {CLIENT_SECRET}
  cookieName: ""
  cookieSecret: {COOKIE_SECRET}

extraArgs:
  auth-logging: true
  cookie-domain: "{DOMAIN_TO_EXPOSE_WORKLOADS}"

```

```

cookie-samesite: lax
cookie-secure: false
force-json-errors: true
login-url: static://401
oidc-issuer-url: {OIDC_ISSUER_URL}
pass-access-token: true
pass-authorization-header: true
pass-host-header: true
pass-user-headers: true
provider: oidc
request-logging: true
reverse-proxy: true
scope: "{TOKEN_SCOPES}"
set-authorization-header: true
set-xauthrequest: true
skip-jwt-bearer-tokens: true
skip-oidc-discovery: false
skip-provider-button: true
standard-logging: true
upstream: static://200
whitelist-domain: "*.{DOMAIN_T0_EXPOSE_WORKLOADS}:*"
EOF

```

- b. To install OAuth2 Proxy with your configuration, use [OAuth2 Proxy helm chart](#) :

```

kubectl create namespace oauth2-proxy
helm repo add oauth2-proxy https://oauth2-proxy.github.io/manifests
helm upgrade --install oauth2-proxy oauth2-proxy/oauth2-proxy -f values.yaml -n oauth2-proxy

```

- c. Register OAuth2 Proxy as an authorization provider in the Istio module:

```

kubectl patch istio -n kyma-system default --type merge --patch '{"spec":{"config":{"auth-provider": "oidc","oidc-provider": "https://:401","oidc-client-id": "","oidc-client-secret": "","oidc-username-claim": "sub","oidc-groups-claim": "groups","oidc-groups-prefix": "group:","oidc-groups-suffix": ""}}}'

```

3. Adjust the obtained configuration of the APIRule to use the extAuth handler in version v2.

The following APIRule example delegates token validation to the previously configured OAuth2 Proxy. Existing tokens stay valid throughout the migration, ensuring that the process does not disrupt any exposed or secured workloads.

Sample Code

```

apiVersion: gateway.kyma-project.io/v2
kind: APIRule
metadata:
  name: httpbin
  namespace: test
spec:
  hosts:
    - httpbin
  service:
    name: httpbin
    namespace: test
    port: 8000
  gateway: kyma-system/kyma-gateway
  rules:
    - extAuth:
        authorizers:
          - oauth2-proxy
      methods:
        - POST
      path: /anything
    - extAuth:
        authorizers:

```

```

    - oauth2-proxy
  methods:
    - GET
  path: /{**}

```

i Note

The `hosts` field accepts a short host name (without a domain). Additionally, the path `/ .*` has been changed to `/{**}` because APIRule v2 does not support regular expressions in the `spec.rules.path` field.

For more information, see [Changes Introduced in APIRule v2](#).

4. Update the APIRule to version v2 by applying the adjusted configuration.

To verify the version of the applied APIRule, check the value of the `gateway.kyma-project.io/original-version` annotation in the APIRule spec. A value of v2 indicates that the APIRule has been successfully migrated. To see the value, run:

```
kubectl get apirules.gateway.kyma-project.io -n $NAMESPACE $APIRULE_NAME -oyaml
```

The following output indicates that the APIRule has been successfully migrated to version v2:

Sample Code

```

apiVersion: gateway.kyma-project.io/v2
kind: APIRule
metadata:
  annotations:
    gateway.kyma-project.io/original-version: v2
...

```

⚠ Caution

Do not manually change the `gateway.kyma-project.io/original-version` annotation. This annotation is automatically updated when you apply your APIRule in version v2. Modifying the annotation's value manually causes your APIRule `v1beta1` to be handled and configured as version v2, potentially leading to reconciliation errors.

5. To preserve the internal traffic policy from the APIRule `v1beta1`, you must apply the following AuthorizationPolicy.

In APIRule v2, internal traffic is blocked by default. Without this AuthorizationPolicy, attempts to connect internally to the workload cause an RBAC: `access denied` error. Ensure that the selector label is updated to match the target workload.

Option	Description
{NAMESPACE}	The namespace to which the AuthorizationPolicy applies. This namespace must include the target workload for which you allow internal traffic. The selector matches workloads in the same namespace as the AuthorizationPolicy.

Option	Description
{LABEL_KEY}: {LABEL_VALUE}	<p>To further restrict the scope of the AuthorizationPolicy, specify label selectors that match the target workload. Replace these placeholders with the actual key and value of the label. The label indicates a specific set of Pods to which a policy should be applied. The scope of the label search is restricted to the configuration namespace in which the AuthorizationPolicy is present.</p> <p>For more information, see Authorization Policy.</p>

```

apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
  name: allow-internal
  namespace: {NAMESPACE}
spec:
  selector:
    matchLabels:
      {LABEL_KEY}: {LABEL_VALUE}
  action: ALLOW
  rules:
  - from:
    - source:
        notPrincipals: ["cluster.local/ns/istio-system/sa/istio-ingressgateway-service-account"]

```

- 6. To retain the CORS configuration from the APIRule v1beta1, update the APIRule in version v2 to include the same CORS settings.

For preflight requests to work correctly, you must explicitly add the "OPTIONS" method to the `rules.methods` field of your APIRule v2. For guidance, see the [sample APIRule CRs](#).

Results

You have migrated your APIRule CR to version v2. Now, access your workload:

- Send a GET request to the exposed workload using JWT authentication::

Sample Code

```
curl -ik -X GET https://{{SUBDOMAIN}}.{{DOMAIN_NAME}}/ip --header "Authorization:Bearer $ACCESS_TOKEN"
```

If successful, the call returns the 200 OK response code.

- Send a POST request to the exposed workload using JWT authentication:

Sample Code

```
curl -ik -X POST https://{{SUBDOMAIN}}.{{DOMAIN_NAME}}/anything -d "test data" --header "Authorization:Bearer $ACCESS_TOKEN"
```

If successful, the call returns the 200 OK response code.

- Send a POST request to the exposed workload without JWT authentication:

Sample Code

```
curl -ik -X POST https://{{SUBDOMAIN}}.{{DOMAIN_NAME}}/anything -d "test data"
```

The call returns the 403 error code.

FAQ

APIRule CRD v2 is the latest stable version. Version v1beta1 has been deprecated and scheduled for deletion. See the frequently asked questions related to the migration process.

Displaying an APIRule's Spec

Why doesn't my APIRule contain any rules?

This APIRule is not migrated to version v2. Since version v2 is now the default version, when you request an APIRule, kubectl converts it to version v2. This conversion only affects the displayed resource's textual format and does not modify the resource in the cluster. If the full conversion is possible, the `rules` field is presented in the output. However, if the conversion cannot be completed, the `rules` are missing, and the original rules are stored in the resource's annotation `gateway.kymaproject.io/v1beta1-spec`. For more information, see [Retrieving the Complete spec of an APIRule in Version v1beta1](#).

Why doesn't my APIRule contain a Gateway?

If your APIRule doesn't contain the Gateway when displayed using kubectl, this means that your APIRule is in version v1beta1 and uses an unsupported Gateway format. The APIRule v2 supports only the Gateway format `namespace/gateway-name`. When you try to display the APIRule v1beta1 using kubectl, its textual format is converted to version v2. Since the Gateway format you're using is neither available in version v2 nor v1beta1, it is not included in the output.

How do I request an APIRule in a particular version?

Specify the version you want to request in the kubectl command.

To get version v1beta1, run:

```
kubectl get apirules.v1beta1.gateway.kymaproject.io -n $NAMESPACE $APIRULE_NAME -o yaml
```

To get version v2alpha1, run:

```
kubectl get apirules.v2alpha1.gateway.kymaproject.io -n $NAMESPACE $APIRULE_NAME -o yaml
```

To get version v2, run:

```
kubectl get apirules.gateway.kymaproject.io -n $NAMESPACE $APIRULE_NAME -o yaml
```

Version v2 is the stored version, so kubectl uses it by default to display your APIRules, no matter if you specify version v2 in the command or do not specify any version.

Why doesn't Kyma dashboard display all my APIRules?

APIRule v1beta1 deletion is divided into phases. As part of the first one, APIRule v1beta1 support has been removed from Kyma dashboard. This means that you can no longer view, edit, or create APIRules v1beta1 using Kyma dashboard. For more information on the deletion timeline for SAP BTP, Kyma runtime, see [APIRule v1beta1 Migration Timeline](#).

Checking an APIRule's Version

Why does the `kubectl get` command return my APIRule in version v2?

APIRule v2 is now the default version displayed by kubectl. This means that no matter in which version the APIRule was actually created in the cluster, kubectl converts the APIRule's displayed textual format to the latest stable version v2. It does not modify the resource in the cluster.

How do I check which version of APIRule I'm using?

To check the version of your APIRule, run the following command:

```
kubectl get apirules.gateway.kyma-project.io -n $NAMESPACE $APIRULE_NAME -o yaml
```

The annotation `gateway.kyma-project.io/original-version` specifies the version of your APIRule.

Migrating an APIRule v1beta1 to Version v2

How do I know which APIRules must be migrated?

You must migrate all APIRules v1beta1 to version v2. To list all your APIRules v1beta1, run the following command:

```
kubectl get apirules.gateway.kyma-project.io -A -o json | jq '.items[] | select(.metadata.annotations["gateway.kyma-project.io/original-version"] == "v1beta1")'
```

If `kubectl get` returns an APIRule in version v2, does it mean that my APIRule is migrated to v2?

No. APIRule v2 is now the default version displayed by kubectl. Kubectl converts the textual format of each APIRule, no matter what its original version is. So, if your APIRule is in version v1beta1, kubectl converts it to version v2 and displays it in the command's output. This conversion does not affect the resource itself.

To verify if your APIRule is migrated, check the annotation `gateway.kyma-project.io/original-version`. If it specifies version v2, your APIRule is migrated. If the annotation is `gateway.kyma-project.io/original-version: v1beta1`, this means that the resource is in version v1beta1 even though in the command line it is converted to match the v2 specification.

i Note

Do not manually change the `gateway.kyma-project.io/original-version` annotation. This annotation is automatically updated when you migrate your APIRule to version v2.

I used `oauth2-introspection` in APIRule v1beta1. How do I migrate it to v2?

The `oauth2-introspection` handler is removed from APIRule v2. To migrate your APIRule that uses this handler, you must first deploy a service that acts as an external authorizer for Istio and then define the `extAuth` access strategy in your APIRule CR. See [Migrating APIRule v1beta1 of type oauth2_introspection to version v2](#).

I used `regexp` in the paths of APIRule v1beta1. How do I migrate it to v2?

APIRule v2 does not support `regexp` in the `spec.rules.path` field of APIRule CR. Instead, it supports using the `{*}` and `{**}` operators and the `/*` wildcard. For more information, see [Changes Introduced in APIRule v2](#).

Why do I get a validation error for the legacy gateway format while trying to migrate to v2?

In APIRule v2, you must provide the Gateway using the format namespace/gateway-name. The legacy formats are not supported.

Using APIRules v1beta1

Why can't I create an APIRule v1beta1 in a new cluster?

APIRule v1beta1 deletion is divided into phases. In the second phase, you can no longer create APIRule v1beta1 in new clusters. For more information on the deletion timeline for SAP BTP, Kyma runtime, see [APIRule v1beta1 Migration Timeline](#).

Why are my APIRules v1beta1 in the Warning state?

When a resource is in the Warning state, it signifies that user action is required. All APIRules v1beta1 are set to this state to indicate that you must migrate these resources to version v2.

Troubleshooting for the API Gateway Module

Troubleshoot problems related to the API Gateway module.

[Issues when Creating APIRule CRs](#)

[Certificate Management: Issuer Not Created](#)

[Kyma Gateway Not Reachable](#)

[401 Unauthorized or 403 Forbidden](#)

[Connection Refused or Timeout](#)

[Blocked In-Cluster Communication](#)

[APIRule v2 Doesn't Contain Rules](#)

[APIRule v2alpha1 Doesn't Contain Rules](#)

[APIRule v2 Contains a Changed Status Schema](#)

[APIRules Are Missing from Kyma Dashboard](#)

If you created APIRules using version v1beta1, and you have not yet migrated them to version v2, they are not displayed in Kyma dashboard's [API Rules](#) view. To display APIRules in Kyma dashboard, you must migrate them to version v2.

Issues when Creating APIRule CRs

This document offers solutions for common problems that may arise when creating an APIRule custom resource (CR) and outlines possible reasons why the CR might be in an error state after creation.

To check the state of an APIRule CR, run:

```
kubectl get apirules httpbin
```

If the CR is in the error state, you get a response similar to this one:

NAME	STATUS	HOST
httpbin	ERROR	httpbin.xxx.shoot.canary.k8s-hana.ondemand.com

The error may signify that your APIRule CR is in an inconsistent state and the service cannot be properly exposed. To check the error message of the APIRule CR, run:

```
kubectl get apirules -n {NAMESPACE} {APIRULE_NAME} -o=jsonpath='{.status.description}'
```

The **issuer** Configuration of the **jwt** Access Strategy is Missing

Symptom

You get the following error:

```
{"code": "ERROR", "description": "Validation error: Attribute \".spec.rules[0].jwt\": supplied config
```

Cause

Your APIRule is missing the **issuer** configuration for the **jwt** access strategy. See an example:

```
apiVersion: gateway.kyma-project.io/v2
kind: APIRule
metadata:
  ...
spec:
  ...
  rules:
    - path: /.*
      jwt:
```

Solution

Add JWT configuration for the **issuer** or use the `` symbols. Here's an example of a valid configuration:

```
spec:
  ...
  rules:
    - path: /.*
      methods: [ "GET" ]
      jwt:
        authentications:
          - issuer: "https://dev.kyma.local"
            jwksUri: "https://example.com/.well-known/jwks.json"
```

Invalid **issuer** Configuration for the **jwt** Access Strategy

Symptom

You get the following error and the APIRule CR is not created:

```
The APIRule "httpbin" is invalid: .spec.rules[0].jwt.authentications[0].issuer: value is empty or r
```

Cause

If the **issuer** contains a colon ':', it must be a valid URI. Here's an example of the APIRule CR with an invalid **issuer** URL configuration:

```
apiVersion: gateway.kyma-project.io/v2
kind: APIRule
metadata:
  ...
spec:
  ...
  rules:
    - path: /.*
      jwt:
        authentications:
          - issuer: ://unsecured.or.not.valid.url
            jwksUri: https://example.com/.well-known/jwks.json
```

Solution

The JWT **issuer** must not be empty and must be a valid URI, for example:

```
apiVersion: gateway.kyma-project.io/v2
kind: APIRule
metadata:
  ...
spec:
  ...
  rules:
    - path: /.*
      jwt:
        authentications:
          - issuer: https://dev.kyma.local
            jwksUri: https://dev.kyma.local/.well-known/jwks.json
```

Both **noAuth** and **jwt** Access Strategies Defined on the Same Path

Symptom

You get the following error:

```
{"code": "ERROR", "description": "Validation error: Attribute \".spec.rules[0].noAuth\": noAuth access strategy defined on the same path as jwt access strategy"}
```

Cause

You must not set the **noAuth** access strategy to **noAuth** and define the **jwt** configuration on the same path. See an example of APIRule CR with invalid configuration:

```
spec:
  ...
  rules:
    - path: /.*
```

```

rules:
  - path: /.*
    noAuth: true
    jwt:
      authentications:
        - issuer: https://dev.kyma.local
          jwksUri: https://dev.kyma.local/.well-known/jwks.json

```

Solution

Decide on one configuration you want to use. You can either use noAuth access to the specific path or restrict it using a JWT security token.

Occupied Host

Symptom

You get the following error:

```
{"code": "ERROR", "description": "Validation error: Attribute \".spec.host\": This host is occupied by
```

Cause

Your APIRule CR specifies a host that is already used by another APIRule or Virtual Service. See an example of two APIRule CRs that use the same host:

```

spec:
  host: httpbin.xxx.shoot.canary.k8s-hana.ondemand.com

```

```

spec:
  host: httpbin.xxx.shoot.canary.k8s-hana.ondemand.com

```

Solution

Use a different host for one of the APIRule CRs. See an example:

```

spec:
  httpbin-new.xxx.shoot.canary.k8s-hana.ondemand.com

```

```

spec:
  host: httpbin.xxx.shoot.canary.k8s-hana.ondemand.com

```

Certificate Management: Issuer Not Created

Symptom

When you try to create an Issuer custom resource (CR) using `cert.gardener.cloud/v1alpha1`, the resource is not created. There are no logs in the `cert-management` controller.

Cause

The cert-management controller does not watch the namespace in which you created the Issuer CR. By default, cert-management watches the default namespace for all Issuer CRs.

Solution

To make sure that you created the Issuer CR in the default namespace, run: `kubectl get issuers -A`. If you want to create the Issuer CR in a different namespace, adjust cert-management settings during the installation.

Kyma Gateway Not Reachable

Symptom

You cannot access Services or Functions exposed using APIRules. Kyma Gateway refuses the connection.

Cause

The issue occurs when the default kyma-gateway is either renamed or duplicated. Once you have multiple Gateway resources pointing to the same host, the Gateway CR created first takes precedence over the others.

Solution

Having two Gateway resources pointing to the same host is not recommended. To resolve the issue, make sure the default kyma-gateway exists and is not renamed or duplicated. If there are multiple Gateway resources pointing to the same host, delete the duplicated Gateway. To delete the Gateway resource, run: `kubectl -n kyma-system delete gateway {DUPLICATED_GATEWAY_NAME}`.

401 Unauthorized or 403 Forbidden

Symptom

When you try to reach a service, you get 401 Unauthorized or 403 Forbidden in response.

Cause

The error 401 Unauthorized occurs when you try to access a Service that requires authentication, and the appropriate credentials were not provided or were incorrect. You get the error 403 Forbidden when you try to access a Service or perform an action for which you lack permission.

Solution

Make sure that you are using an active JSON Web Token (JWT) with proper scopes.

1. Decode the JWT.
2. Check the validity and scopes of the JWT:

```
{
  "sub": *****,
  "scp": "test",
  "aud": *****,
  "iss": *****,
  "exp": 1697120462,
  "iat": *****,
  "jti": *****
}
```

3. [Generate a new JWT](#) if needed.

Connection Refused or Timeout

Symptom

After you have finished all the steps required to set up your custom domain, you receive the `connection refused` or `connection timeout` error when you try to expose a Service. It shows up when you call the Service endpoint by sending a GET request. The error looks as follows:

```
curl: (7) Failed to connect to httpbin.mydomain.com port 443: Connection refused
```

Cause

DNS resolves to an incorrect IP address.

Solution

Check if the IP address provided as the value of the `spec.targets` parameter of the `DNSEntry` custom resource is the IP address of the Ingress Gateway you are using. To check the Ingress Gateway IP, run:

```
kubectl -n istio-system get service istio-ingressgateway -o jsonpath='{.status.loadBalancer.ingress[0].ip}'
```

In addition, ensure that your OS resolves the target host name to the same Ingress Gateway IP address. Run:

```
host {YOUR_SUBDOMAIN}
```

i Note

`YOUR_SUBDOMAIN` specifies the name of your subdomain, for example, `httpbin.mydomain.com`.

Blocked In-Cluster Communication

Symptom

After switching from `APIRule` custom resource (CR) `v1beta1` to version `v2`, in-cluster communication is blocked. Attempts to connect internally to the workload result in the error RBAC: `access denied`.

Cause

By default, access to the workload from internal traffic is blocked if an APIRule CR in version v2 or v2alpha1 is applied. This approach aligns with Kyma's "secure by default" principle.

Solution

If APIRule is applied, internal traffic is blocked by default. To allow it, you must create an ALLOW-type AuthorizationPolicy.

See the following example of an AuthorizationPolicy that allows internal traffic to the given workload. Note that it excludes traffic coming from `istio-ingressgateway` not to interfere with policies applied by APIRule to external traffic.

To use this code sample, replace `{NAMESPACE}`, `{LABEL_KEY}`, and `{LABEL_VALUE}` with the values appropriate for your environment.

Option	Description
<code>{NAMESPACE}</code>	The namespace to which the AuthorizationPolicy applies. This namespace must include the target workload for which you allow internal traffic. The selector matches workloads in the same namespace as the AuthorizationPolicy.
<code>{LABEL_KEY}: {LABEL_VALUE}</code>	<p>To further restrict the scope of the AuthorizationPolicy, specify label selectors that match the target workload. Replace these placeholders with the actual key and value of the label. The label indicates a specific set of Pods to which a policy should be applied. The scope of the label search is restricted to the configuration namespace in which the AuthorizationPolicy is present.</p> <p>For more information, see Authorization Policy.</p>

```
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
  name: allow-internal
  namespace: {NAMESPACE}
spec:
  selector:
    matchLabels:
      {LABEL_KEY}: {LABEL_VALUE}
  action: ALLOW
  rules:
  - from:
    - source:
        notPrincipals: ["cluster.local/ns/istio-system/sa/istio-ingressgateway-service-account"]
```

For example, let's suppose you have deployed the [HTTPBin Service](#) and exposed it using an APIRule v2 or v2alpha1. To apply this policy to the HTTPBin Service labeled with `app: httpbin` deployed in the `default` namespace, you must set `{NAMESPACE}` to `default`, `{LABEL_KEY}` to `app`, and `{LABEL_VALUE}` to `httpbin`.

```
apiVersion: security.istio.io/v1
kind: AuthorizationPolicy
metadata:
  name: allow-internal
  namespace: default
spec:
  selector:
    matchLabels:
      app: httpbin
  action: ALLOW
```

```

rules:
- from:
  - source:
    notPrincipals: ["cluster.local/ns/istio-system/sa/istio-ingressgateway-service-account"]
  
```

Verification

To test if the internal traffic reaches the exposed workload, follow the steps:

1. Create a sample namespace with enabled Istio sidecar proxy injection:

```

kubectl create ns curl-test
kubectl label namespace curl-test istio-injection=enabled --overwrite
  
```

2. Run a Pod inside the Istio service mesh:

```
kubectl run -n curl-test --image=curlimages/curl test-internal -- /bin/sleep 3600
```

3. To test the internal communication, replace the placeholders and run the following command:

Option	Description
{SERVICE_NAME}	The name of the exposed Service.
{NAMESPACE}	The namespace in which the Service is deployed.
{PORT}	The port number on which the Service is listening for incoming traffic. Defined in the port field of the Service's configuration.
{EXPOSED_PATH}	A path exposed by the Service that you want to access.

```
kubectl exec -ti -n curl-test test-internal -- curl http://{SERVICE_NAME}.{NAMESPACE}.svc.cluster.local{EXPOSED_PATH}
```

For example, to test the connection to the [HTTPBin Service](#) ↗ exposed using an APIRule v2 or v2alpha1 on the /get path, you can use the following command:

```
kubectl exec -ti -n curl-test test-internal -- curl http://httpbin.default.svc.cluster.local:80/get
```

If successful, the command returns the contents of the specified path.

4. To clean up the resources created for testing, delete the namespace curl-test from the cluster. Deleting the namespace removes all resources within it.

```
kubectl delete namespace curl-test
```

APIRule v2 Doesn't Contain Rules

Symptom

An APIRule custom resource (CR) does not contain the rules field, for example:

```
kubectl get apirules.gateway.kyma-project.io -n $NAMESPACE $APIRULE_NAME -o yaml
```

```

apiVersion: gateway.kyma-project.io/v2
kind: APIRule
metadata:
  name: httpbin
  namespace: test
spec:
  gateway: kyma-system/kyma-gateway
  hosts:
    - httpbin.local.kyma.dev
  service:
    name: httpbin
    namespace: test
    port: 8000
status:
  lastProcessedTime: "2025-04-25T11:16:11Z"
  state: Ready

```

Cause

The APIRule was originally created using version `v1beta1`, and you haven't yet migrated it to version `v2`. Since the latest stable version of the APIRule in the Kubernetes API is now `v2`, running the `kubectl get` command without specifying a version of APIRule assumes version `v2`.

To display the resource in version `v2`, a conversion from `v1beta1` to `v2` is performed. This conversion only affects the displayed resource's textual format and does not modify the resource in the cluster. If the full conversion is possible, the `rules` field is presented in the output. However, if the conversion cannot be completed, the `rules` are missing, and the original rules are stored in the resource's annotations.

Solution

Specify explicitly `v1beta1` version when requesting the APIRule resource:

```
kubectl get apirules.v1beta1.gateway.kyma-project.io -n $NAMESPACE $APIRULE_NAME -o yaml
```

APIRule v2alpha1 Doesn't Contain Rules

Symptom

An APIRule custom resource (CR) does not contain the `rules` field, for example:

```
kubectl get apirules.v2alpha1.gateway.kyma-project.io -n $NAMESPACE $APIRULE_NAME -o yaml
```

```

apiVersion: gateway.kyma-project.io/v2alpha1
kind: APIRule
metadata:
  name: httpbin
  namespace: test
spec:
  gateway: kyma-system/kyma-gateway
  hosts:
    - httpbin.local.kyma.dev
  service:

```

11/18/25, 11:56 AM

```
name: httpbin
namespace: test
port: 8000
status:
  lastProcessedTime: "2025-04-25T11:16:11Z"
  state: Ready
```

Cause

The APIRule was originally created using version **v1beta1**, and you haven't yet migrated it to version **v2**.

To display the resource in version **v2alpha1**, a conversion from **v1beta1** to **v2alpha1** is performed. This conversion only affects the displayed resource's textual format and does not modify the resource in the cluster. If the full conversion is possible, the **rules** field is presented in the output. However, if the conversion cannot be completed, the **rules** are missing, and the original rules are stored in the resource's annotations.

Solution

Specify explicitly **v1beta1** version when requesting the APIRule resource:

```
kubectl get apirules.v1beta1.gateway.kyma-project.io -n $NAMESPACE $APIRULE_NAME -o yaml
```

APIRule v2 Contains a Changed Status Schema

Symptom

There is a changed **status** schema in an APIRule custom resource (CR), for example:

```
kubectl get apirules.gateway.kyma-project.io -n $NAMESPACE $APIRULE_NAME -o yaml
```

```
...
status:
  lastProcessedTime: "2025-04-25T11:16v:11Z"
  state: Ready
```

Cause

The schema of the **status.state** field in the v2 APIRule CR introduces a unified approach, similar to the one used in the API Gateway CR. The possible states of the **status.state** field are **Ready**, **Warning**, **Error**, **Processing**, or **Deleting**.

Solution

To get the APIRule in its original version, run:

```
kubectl get apirules.v1beta1.gateway.kyma-project.io -n $NAMESPACE $APIRULE_NAME -o yaml
```

You get an output similar to this one:

```
...
status:
```

```

APIRuleStatus:
  code: OK
accessRuleStatus:
  code: OK
lastProcessedTime: "2025-04-25T11:16:11Z"
observedGeneration: 1
virtualServiceStatus:
  code: OK

```

APIRules Are Missing from Kyma Dashboard

If you created APIRules using version `v1beta1`, and you have not yet migrated them to version `v2`, they are not displayed in Kyma dashboard's [API Rules](#) view. To display APIRules in Kyma dashboard, you must migrate them to version `v2`.

Symptom

Kyma dashboard's [API Rules](#) view does not display APIRules created in version `v1beta1`. To check if the APIRules are present in the cluster, run the following command:

```
kubectl get apirules.v2.gateway.kyma-project.io -A
```

This command lists all APIRules available in your Kyma cluster, regardless of their original version. To get a specific APIRule and check the version in which it was created, run the following command:

```
kubectl get apirules.v2.gateway.kyma-project.io -n $NAMESPACE $APIRULE_NAME -o yaml
```

If the APIRule was created using version `v1beta1`, the output contains the annotation `gateway.kyma-project.io/original-version: v1beta1`.

See the following example:

```

apiVersion: gateway.kyma-project.io/v2
kind: APIRule
metadata:
  annotations:
    gateway.kyma-project.io/original-version: v1beta1
  name: httpbin
  namespace: test
spec:
  gateway: kyma-system/kyma-gateway
  hosts:
    - httpbin.local.kyma.dev
  service:
    name: httpbin
    namespace: test
    port: 8000
status:
  lastProcessedTime: "2025-04-25T11:16:11Z"
  state: Warning

```

Cause

APIRules that are not displayed in Kyma dashboard were originally created using version `v1beta1`, and you haven't yet migrated them to version v2. The APIRule `v1beta1` API is no longer available via Kyma dashboard.

Solution

To make sure that support for your APIRules is maintained, you must migrate them to version v2. To learn how to do this, see [APIRule Migration](#).

Default Configuration and Limitations of the API Gateway Module

APIRule Controller Limitations

APIRule Controller relies on Istio and Ory custom resources (CRs) to provide routing capabilities. In terms of persistence, the controller depends only on APIRules stored in the Kubernetes cluster. In terms of the resource configuration, the following requirements are set on APIGateway Controller:

- CPU Requests: 10 m
- CPU Limits: 100 m
- Memory Requests: 64 Mi
- Memory Limits: 128 Mi

You can create an unlimited number of APIRule CRs.

Ory Resources' Configuration

The configuration of Ory resources depends on the cluster capabilities. If your cluster has fewer than 5 total virtual CPU cores or its total memory capacity is less than 10 gigabytes, the default setup for resources is lighter. If your cluster exceeds both of these thresholds, the higher resource configuration is applied.

The default configuration for larger clusters includes the following settings for the Ory components' resources:

Ory Resources' Configuration

Component	CPU Requests	CPU Limits	Memory Requests	Memory Limits
Oathkeeper	100 m	10000 m	64 Mi	512 Mi
Oathkeeper Maester	10 m	400 m	32 Mi	1 Gi

The default configuration for smaller clusters includes the following settings for the Ory components' resources:

Ory Resources' Configuration

Component	CPU Requests	CPU Limits	Memory Requests	Memory Limits
Oathkeeper	10 m	100 m	64 Mi	128 Mi
Oathkeeper Maester	10 m	100 m	20 Mi	50 Mi

Autoscaling Configuration

The default configuration in terms of autoscaling of Ory components is as follows:

Ory Resources' Configuration

Component	Min Replicas	Max Replicas
Oathkeeper	3	10
Oathkeeper Maester	3	10

Oathkeeper Maester is a separate container running in the same Pod as Oathkeeper. Because of that, the autoscaling configuration of the Oathkeeper and Oathkeeper Master components is similar. The autoscaling configuration is based on CPU utilization, with HorizontalPodAutoscaler set up for 80% average CPU request utilization.