webMethods Microgateway User's Guide

Version 10.11

October 2021

WEBMETHODS

Document ID: YAM-UG-1011-20220719

Table of Contents

About this Guide

This guide describes how you can use API Gateway and other API Gateway components to effectively manage APIs for services that you want to expose to applications, whether inside your organization or outside to partners and third parties.

To use this guide effectively, you should have an understanding of the APIs that you want to expose to the developer community and the access privileges you want to impose on those APIs.

Document Conventions

Convention	Description
Bold	Identifies elements on a screen.
Narrowfont	Identifies service names and locations in the format *folder.subfolder.service*, APIs, Java classes, methods, properties.
Italic	Identifies: Variables for which you must supply values specific to your own situation or environment. New terms the first time they occur in the text. References to other documentation sources.
`Monospace font`	Identifies: Text you must type in. Messages displayed by the system. Program code.
{ }	Indicates a set of choices from which you must choose one. Type only the information inside the curly braces. Do not type the { } symbols.
|	Separates two mutually exclusive choices in a syntax line. Type one of these choices. Do not type the | symbol.
[ ]	Indicates one or more options. Type only the information inside the square brackets. Do not type the [ ] symbols.
...	Indicates that you can type multiple options of the same type. Type only the information. Do not type the ellipsis (...).

Online Information and Support

Software AG Documentation Website

You can find documentation on the Software AG Documentation website at https://documentation.softwareag.com.

Software AG Empower Product Support Website

If you do not yet have an account for Empower, send an email to empower@softwareag.com with your name, company, and company email address and request an account.

Once you have an account, you can open Support Incidents online via the eService section of Empower at https://empower.softwareag.com/.

You can find product information on the Software AG Empower Product Support website at https://empower.softwareag.com.

To submit feature/enhancement requests, get information about product availability, and download products, go to Products.

To get information about fixes and to read early warnings, technical papers, and knowledge base articles, go to the Knowledge Center.

If you have any questions, you can find a local or toll-free number for your country in our Global Support Contact Directory at https://empower.softwareag.com/public_directory.aspx and give us a call.

Software AG Tech Community

You can find documentation and other technical information on the Software AG Tech Community website at https://techcommunity.softwareag.com. You can:

- Access product documentation, if you have Tech Community credentials. If you do not, you will need to register and specify "Documentation" as an area of interest.
- Access articles, code samples, demos, and tutorials.
- Use the online discussion forums, moderated by Software AG professionals, to ask questions, discuss best practices, and learn how other customers are using Software AG technology.
- Link to external websites that discuss open standards and web technology.

Data Protection

Software AG products provide functionality with respect to processing of personal data according to the EU General Data Protection Regulation (GDPR). Where applicable, appropriate steps are documented in the respective administration documentation.

1 About webMethods Microgateway

Introduction

The adoption of the micro-service architecture pattern drives the need for lightweight gateways or Microgateways. The Microgateway gives control over a micro-service landscape by enforcing policies which perform authentication, traffic monitoring, and traffic management. The lightweight nature of a Microgateway allows a flexible deployment to avoid gaps or bottlenecks in the policy enforcement.

Microgateway is a gateway that enables micro-services to communicate with each other directly without re-routing the communication channel through an API Gateway. This eases out the traffic overload on API Gateway with communication between micro-services. You can enforce the required protection policies on the Microgateway to have a secure communication channel between the micro-services.

This figure illustrates an API Gateway with micro-services suffering from network latency and security issues as figure 1 followed by figure 2 that depicts a Microgateway with micro-services addressing network latency and security issues by side car deployment.

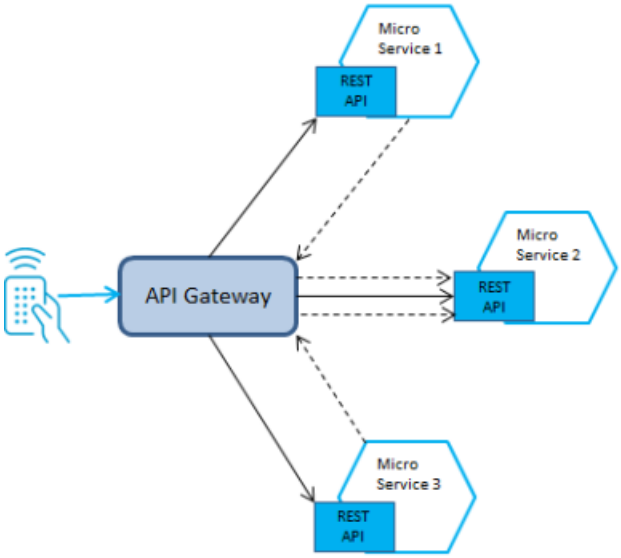


Fig1: API Gateway with micro services suffering from network latency and security issues

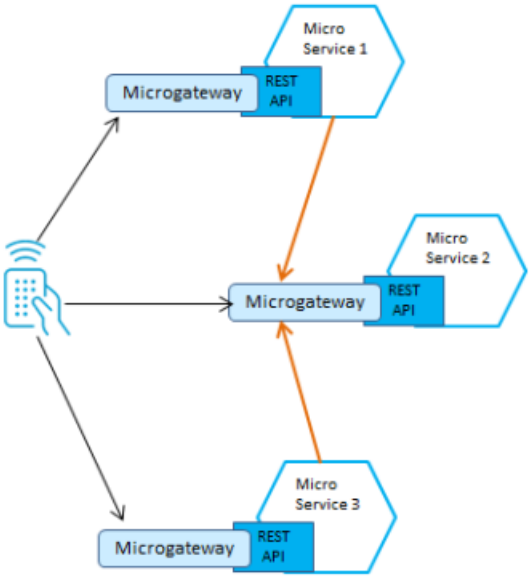


Fig 2: Microgateway with micro services addressing network latency and security issues by side car deployment

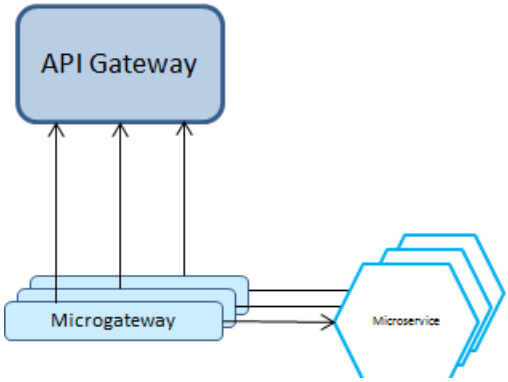
The first part, figure 1, of the diagram depicts micro-services with a single API Gateway that does the enforcement of policies. Here, each micro-service exposes an endpoint where no policy enforcement is done. Moreover, considering that the micro-services are interacting with each other, all the traffic needs to be routed to the API Gateway. This leads to additional network latency and the API Gateway might become a bottleneck. In the second part, figure 2, API Gateway is replaced with a set of Microgateways that are deployed near the micro-services. Such a sidecar deployment does not leave any gaps and avoids bottlenecks, thereby solving the network latency issues and ease of policy enforcement.

Microgateway Components

Microgateway comes with a service performing the policy enforcement on REST APIs. The Microgateway service runs within its own Java runtime environment and is controlled by a simple command line interface that supports basic lifecycle operations like start and stop. The configuration of the service consists of system settings and assets that can be provisioned from a running API Gateway or can be provisioned through a filesystem. The provisioned assets include application, API, policy, and alias definitions. The Microgateway service exposes an administrator REST API to query the status, the system setting, and the provisioned assets and shut down Microgateway as well.

API Gateway Integration

The Microgateway's responsibility is focused on a single micro-service or a small number of micro-services. To manage a micro-service landscape an API Gateway is required. It offers the user interface for configuring the policy configuration and system configurations. Moreover, it is responsible for monitoring the traffic within the micro-service landscape. The following figure shows how Microgateways are interacting with an API Gateway.



The Microgateways pull the assets including APIs, applications, and policies from the API Gateway where they are configured. Also, other administrative settings including SSL configuration and fault configurations are defined in the API Gateway and pulled by the Microgateway during startup. The download of assets is done through the API Gateway REST APIs.

While monitoring the API requests and responses, the Microgateway pushes the runtime analytics information to API Gateway. API Gateway provides a consolidated view through its dashboards.

The download of assets and administrative settings can be done up-front using the Microgateway tooling that allows to provision stand-alone Microgateways, which do not require any connection to a running API Gateway. Stand-alone Microgateways do not allow to perform a consolidated traffic monitoring of a micro-service landscape.

Microgateway Installation

You can install a Microgateway using Software AG Installer. In the Software AG Installer, Microgateway appears as a subnode under the API Gateway node in the installer tree. Microgateway

is selected by default when you select API Gateway in the installer tree. For details on installing using installer, see *Installing Software AG Products*.

Note:
Microgateway can only be installed together with the API Gateway. An independent installation of the Microgateway is not supported. The Microgateway installation becomes operational only if the Microservices feature is active in the API Gateway license.

2 Asset and Configuration Provisioning

Asset Provisioning

Provisioning assets in Microgateway makes assets available for use in Microgateway. The assets you can provision in Microgateway are, APIs including policies and policy properties, API scopes, runtime aliases, applications, and global policies.

Note:
Microgateway only supports the REST APIs and do not support the SOAP and OData APIs.

Asset provisioning in Microgateway covers the following:

- Reading APIs, API scopes, policies, aliases, and applications from API Gateway on Microgateway start
- Updating APIs, API scopes, aliases, and policies from API Gateway
- Updating Applications from API Gateway
- Reading APIs, API scopes, policies, aliases, and applications from a file system by importing an asset archive
- Reading administrative settings from API Gateway

Note:

- Only simple aliases, endpoint aliases, and HTTP transport security aliases are supported in Microgateway.
- If an API has an unsupported policy the provisioning of the API is rejected.
- If a global policy references an unsupported policy property then, the provisioning of the global policy is rejected.

You can provision assets to a Microgateway in one of the following ways:

- API Gateway asset archive-based provisioning
- Pulling assets from API Gateway

Microgateway also supports a mixed provisioning. Asset archives are preferred over the assets being pulled from an API Gateway when there is duplication of APIs.

Alias Provisioning

Simple aliases, Endpoint aliases, and HTTP Transport Security aliases can be configured in one of the following ways:

- By API Gateway asset archive-based provisioning, where the asset archive file exported from API Gateway contains the aliases in API. Microgateway, on startup, imports the aliases from the archive file and replaces all the aliases found in the APIs.
- By configuring the alias values in the custom settings file that is passed as an argument using the -c option during Microgateway startup.

A sample workflow that explains alias provisioning by passing the custom settings YAML file using the -c option during Microgateway start up is as follows:

```
./microgateway.sh downloadSettings -gw hostname:port
--output my-custom-settings.yml
./microgateway.sh createInstance -instance MyInst.zip
--config my-custom-settings.yml
unzip MyInst.zip -d /tmp/inst
/tmp/inst/microgateway.sh start -c my-custom-settings.yml
```

The aliases provided in the custom settings YAML file take precedence over aliases defined in the archive for aliases that have the same ID. A sample custom settings YAML file is as follows:

```
aliases:
  petstore_host:
    type: simple
    value: petstore.swagger.iov2
  petstore_endpoint:
    type: endpoint
    endPointURI: http://petstore.swagger.io/v2/pet/4
    connectionTimeout: 30
    readTimeout: 30
    suspendDurationOnFailure: 0
    optimizationTechnique: None
    passSecurityHeaders: false
    keystoreAlias: ''
  BasicAuth_Custom:
    type: httpTransportSecurityAlias
    authType: HTTP_BASIC
    authMode: INCOMING_HTTP_BASIC_AUTH
    httpAuthCredentials:
      userName: Administrator123
      password: manage
      domain: null
  BasicAuth_Incoming:
    type: httpTransportSecurityAlias
    authType: HTTP_BASIC
    authMode: INCOMING_HTTP_BASIC_AUTH
  JWT_Outbound:
    type: httpTransportSecurityAlias
    authType: JWT
    authMode: INCOMING_JWT
  Oauth_custom:
    type: httpTransportSecurityAlias
    authType: OAUTH2
    authMode: INCOMING_OAUTH_TOKEN
    oauth2Token: 407e5c8d33c54c57bc7932bf5e803979
  Oauth_incomingToken:
    type: httpTransportSecurityAlias
    authType: OAUTH2
    authMode: INCOMING_OAUTH_TOKEN
```

Endpoint configuration through aliases

For a Microgateway server to run in a Kubernetes sidecar environment, the hostname of the native service URL must be changed to localhost. The most convenient way is to use aliases for the

respective Routing policy. Hence the API in API Gateway should use either a Simple alias or an Endpoint alias.

The alias value is included in the config.yml configuration file. You have to configure the alias name and the type, which are mandatory.

Example where API Gateway uses a Simple alias.

```
Name = “MySimpleAlias”, Value = “myhost”
```

Overwrite the alias value as follows in the config.yml file and start Microgateway.

```
…
aliases:
  MySimpleAlias:
    type: simple
    value: localhost
```

Example where API Gateway uses an Endpoint alias.

```
Name = “MyEndpointAlias”, Endpoint = “http://myhost:1234/service”
```

Overwrite the alias value as follows in the config.yml file and start Microgateway.

```
…
aliases:
  MyEndpointAlias:
    type: endpoint
    uri: http://localhost:1234/service
```

These alias values can be used with the generic Microgateway environment variables in the format `mcgw_aliases_`*`name_propery`*` = "value" .`

For example, the endpoint alias with the URI `http://localhost:1234/service` is specified in the environment variables as follows:

```
mcgw_aliases_MySimpleAlias_type = "endpoint"
   mcgw_aliases_MySimpleAlias_uri = “http://localhost:1234/service”
```

Asset Provisioning through the API Gateway Asset Archive

You can start the Microgateway server with one or more API Gateway asset archives exported that contain the assets to be provisioned. Microgateway supports archives exported from version 10.3 or higher. All the supported assets are imported during Microgateway startup.

You can pass on the API Gateway asset archive in the start command through the archive parameter:

```
./microgateway.sh start -p 9090 --archive apigw_archive.zip
```

You can specify multiple archives using a comma separated list. Ensure that there is no space within the comma separated list.

```
./microgateway.sh start -p 9090
--archive apigw_archive1.zip,apigw_archive2.zip
```

The Microgateway reads the archives in the specified order. If there are any assets provisioned by an earlier archive, they are overwritten by the assets present in the later archives.

Creating API Gateway Asset Archives using the Command Line

You can use the `createAssetArchive` command that Microgateway CLI provides for creating an asset archive. The command takes the parameters detailed in the Command Line Reference.

A Sample workflow for Asset Provisioning using Asset Archive

A sample sequence for asset provisioning by importing the package looks as follows. A sample archive EmployeeService.zip is used to describe the example.

1. Start API Gateway with the given archive EmployeeService.zip

   ```
   ./microgateway.sh start -p 9090 -a /tmp/EmployeeService.zip
   ```

2. Check status of the Microgateway instance.

   ```
   GET http://localhost:9090/rest/microgateway/status
   ```

   The status response looks like this:

   ```
   {
       "description": "webMethods Microgateway",
       "publisher": "Software AG",
       "version": "10.4.0.0"
   }
   ```

3. Retrieve details about the deployed APIs, applications, and global policies using the following request.

   ```
   GET http://localhost:9090/rest/microgateway/assets
   ```

   The details display the provisioned APIs.

4. Invoke the Employee API.

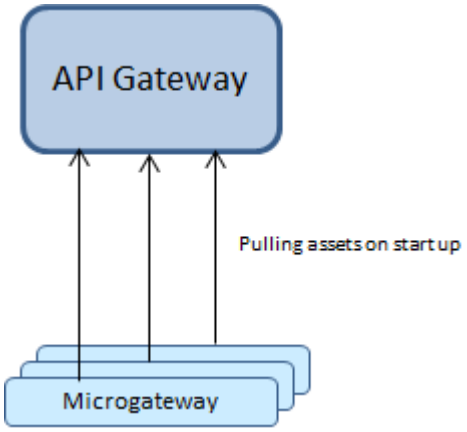
   ```
   GET http://localhost:9090/gateway/EmployeeService/employees
   ```

5. Stop the Microgateway listening on port 9090.

   ```
   ./microgateway.sh stop -p 9090
   ```

Pulling Assets from API Gateway

You can start the Microgateway server with a URL and credentials pointing to an API Gateway. Microgateway pulls the assets from the referenced API Gateway.



Microgateway can only pull assets from API Gateways with version 10.3 or higher. Multiple Microgateways can pull assets from the same API Gateway. To pull assets from API Gateway, use the Microgateway start command with the required command line options detailed in the Command Line Reference.

Connecting to API Gateway

Microgateway connects to the API Gateway during startup. If the API Gateway can't be contacted, then Microgateway terminates with an error message as follows:

```
Start of process failed :
>>> Microgateway server not started <<<
Error during startup: API Gateway not active or accessible: status 503,
errorMessage: server is not active
```

Pulling Specific Assets

An API, policy or application is identified either using its unique identifier or by the combination of name and version. If an asset name or identifier can't be resolved, a respective error message is written to the Microgateway log.

For an identified API, the API along with the API-level policies and policy properties, registered applications, and referenced runtime aliases are pulled from API Gateway.

For an identified global policy, the policy, the policy properties, and referenced runtime aliases are pulled from API Gateway.

For an identified global application, only the application is pulled. For runtime aliases, the default values become effective.

If you do not specify an API, application or policy, then no assets are pulled from the API Gateway.

Note:

- If an API has a unsupported policy, the provisioning of the API is rejected.
- If a global policy references an unsupported policy property then, the provisioning of the global policy is rejected.
- When provisioning runtime aliases the default values become effective.

A Sample workflow for Asset Provisioning by pulling from API Gateway

The following sample sequence shows the flow for provisioning APIs by pulling from API Gateway. It assumes that the API Gateway holds 2 REST APIs: EmployeeService and EmployService2.

1. Start Microgateway with given a API Gateway URL and API name.

```
./microgateway.sh start -p 9090
-gw localhost:5555 -gwu Administrator -gwp password
-apis EmployeeService
```

2. Check the deployed assets.

```
GET http://localhost:9090/rest/microgateway/assets
```

3. Call the API.

```
GET http://localhost:9090/gateway/EmployeeService/employees
```

4. Stop the Microgateway listening on port 9090 .

```
./microgateway.sh stop -p 9090
```

5. Start Microgateway with given a API Gateway URL with multiple API names .

```
./microgateway.sh start -p 9090
-gw localhost:5555 -gwu Administrator
-gwp password -apis EmployeeService,EmployeeService2
```

Configuration Provisioning

The Administration configuration for Microgateway is provisioned in one of the following ways:

- From the default settings file, `system-settings.yml`, which is used for starting a Microgateway server.
- From the user-defined custom settings YAML file, which is used when you want to include some custom settings that needs to override the default settings. The user-defined custom settings YAML file is passed as an argument using the -c option during Microgateway startup.

Note:
From release 10.4, the administration configuration settings are no longer picked up from input archive files.

Default settings file: system-settings.yml

When you start Microgateway server the default settings are read from the system configuration file, which is under `config/system-settings.yml` and contains the following entries:

- `faults`: contains variables for error handling during runtime.
- `extended_settings`: various kinds of settings for runtime.
- `gateway_destination`: API Gateway settings for logging into API Gateway.

- `key_store`: Keystore settings for establishing HTTPS connections.
- `trust_store`: Truststore settings for HTTPS handshake for specific policies.
- `system`: internal settings.

To enable the policy enforcement on a Microgateway the following configurations need to be provisioned in Microgateway:

- extended
- apiFault
- elasticsearchDestinationConfig
- gatewayDestinationConfig

Note:
The external Elasticsearch configuration is optional, and needs to be specified if you have such an Elasticsearch in your environment. You require an external Elasticsearch if you have the Log Invocation policy enforced where Elasticsearch destination is selected. You have to specify the external Elasticsearch configuration settings in the user-defined custom settings YAML file.

The system settings file should not be modified. Any specific changes required are specified in the custom settings file.

The default configuration file `system-settings.yml` looks as follows:

```
---
faults:
   default_error_message: "API Gateway encountered an error.
   Error Message: $ERROR_MESSAGE. Request Details: Service - $SERVICE,
   Operation - $OPERATION, Invocation Time:$TIME, Date:$DATE,
   Client IP - $CLIENT_IP, User - $USER and Application:$CONSUMER_APPLICATION"
   native_provider_fault: "false"
extended_settings:
   defaultEncoding: "UTF-8"
   apiKeyHeader: "x-Gateway-APIKey"
   apig_MENConfiguration_tickInterval: "60"
   events.collectionQueue.size: "10000"
   events.collectionPool.minThreads: "1"
   events.collectionPool.maxThreads: "8"
gateway_destination:
   sendPolicyViolationEvent: "true"
key_store:
   type: JKS
   provider: SUN
   location: config/keystore.jks
   password: password
system:
   version: "10.4.0.0.303"
---
```

User-defined Custom settings YAML file

If you have certain custom settings that you want Microgateway to use by overriding the default settings specified in the `system-settings.yml` file, you can provision these configuration settings

from a user-defined custom settings YAML file. You can create the custom settings file as required. If a particular setting value is not present in the custom settings YAML file, then the appropriate value is taken from the default `config/system-settings.yml` file. The custom settings YAML file contains the configuration values for starting Microgateway and the settings for replacing the system-settings.yml.

The configuration values for starting a Microgateway are as follows in this table.

Configuration values	Settings
`ports`	**Ports configuration section**
	`http`. HTTP port exposed by Microgateway.
	`https`. HTTP port exposed by Microgateway.
	`key_alias`. Key alias for exposing the server certificate on the HTTPS port.
`api_gateway`	**API Gateway configuration section**
	`url`. API Gateway URL.
	`user`. API Gateway user.
	`password`. API Gateway user password.
	`dir`. API Gateway installation folder.
	`download_settings`. Flag to control the download of settings.
`api_endpoint`	**API endpoint section**
	`base_path`. Base path of the APIs exposed by Microgateway.
`admin_api`	**Admin API section**
	`user`. Microgateway user for authenticating requests against the Admin API section.
	`password`. Microgateway user password.
	`admin_path`. base path of the Admin API.
	`downloads`. Asset provisioning section.
	`apis`. APIs to download from API Gateway.
	`applications`. Applications to download form API Gateway.
	`policies`. Global policies to download form API Gateway.
`archive`	**Archive section**
	`file`. Archives to be loaded during startup.

Configuration values	Settings
`policies`	**Policy configuration section**
	`user_auth`. Configuring user configuration.
`logging`	**Logging configuration section**
	`level`. Logging level.
	`path`. File system path for storing log files.
`applications_sync`	**Application synchronization section**
	`enabled`. Flag to enable application synchronization.
	`applications_to_sync`. Applications to synchronize.
	`polling_interval_secs`. Polling interval in seconds.
	`connection_timeout_secs`. Connection time in seconds when synchronizing applications.

A sample user-defined custom settings YAML file looks as follows:

```
ports:
   http: 7000
archive:
   file: /tmp/myarchive.zip
fault:
   ...
extended_settings:
   ...
gateway_destination:
   ...
es_destination
   protocol: "http"
   hostName: "<name of the ElasticSearch host>"
   port: "9240"
   userName: ""
   password: ""
   indexName: "gateway_default_analytics"

   metricsPublishInterval: "60"
   sendErrorEvent: "false"
   sendLifecycleEvent: "false"
   sendPerformanceMetrics: "false"
   sendPolicyViolationEvent: "true"

   sendAuditlogPackageManagementEvent: "false"
   sendAuditlogPlanManagementEvent: "false"
   sendAuditlogApplicationManagementEvent: "false"
   sendAuditlogAliasManagementEvent: "false"
   sendAuditlogRuntimeDataManagementEvent: "false"
   sendAuditlogPolicyManagementEvent: "false"
   sendAuditlogApprovalManagementEvent: "false"
   sendAuditlogUserManagementEvent: "false"
```

```
   sendAuditlogAdministrationEvent: "false"
   sendAuditlogGroupManagementEvent: "false"
   sendAuditlogAccessProfileManagementEvent: "false"
   sendAuditlogAPIManagementEvent: "false"
   sendAuditlogPromotionManagementEvent: "false"
```

Reading Settings from API Gateway

You can read and use the settings from API Gateway during the startup of the Microgateway server, by using the parameter `download_settings` from the YAML configuration file. The default value of the parameter is `false`.

```
# API Gateway configuration
api_gateway:
   url: http://hostname:port
   user: Administrator
   password: password
   dir: /opt/softwareag/IntegrationServer/instances/default
   download_settings: true | false
```

You can also specify download_settings as a command line option during Microgateway startup as follows:

-Shortcut, --Name	Default	Description
-ds, --download_settings	false	Download the settings from API Gateway.

Creating Individual Settings Files

You can create a custom configuration file including all the settings. These settings are pulled from a specified API Gateway.

Use the following command to create a settings file:

```
./microgateway.sh downloadSettings  options
```

where you can use the various command line options detailed in the Command Line Reference.

Example: Use the following command within CLI to create the custom settings YAML file:

```
./microgateway.sh downloadSettings -gw gateway-url -gwu user
-gwp password [--config config-file] --output filename
```

If you do not specify any input settings while creating the custom settings YAML file, then the custom settings file created contains all the API Gateway setting entries, such as fault, extended_settings, and so on. The following invocation creates a custom settings YAML file by downloading settings from the API Gateway running on apigateway-host:

```
./microgateway.sh downloadSettings -gw http://apigateway-host:5555 -gwu Administrator

-gwp password --output config/custom-settings.yml
```

If you have specified an input settings file, the input settings are merged with the settings downloaded form API Gateway. The following invocation creates a merged settings file, `custom-settings.yml`:

```
./microgateway.sh downloadSettings -gw http://apigateway-host:5555 -gwu Administrator

-gwp password --config my-config.yml --output config/custom-config.yml
```

Security Settings

The security settings can be pulled from API Gateway directly or can be configured in the custom settings YAML file. The security settings pulled directly from API Gateway take precedence over the security settings configured in the custom settings YAML file.

A sample configuration file with the aliases looks as follows:

```
---
security_settings:
  providers:
  - !<clientMetadataMapping>
    id: "PingFederate"
    name: "PingFederate"
    type: "clientMetadataMapping"
    owner: "Administrator"
    providerName: "PingFederate"
    implNames:
      grant_types: "grantTypes"
      logo_uri: "logoUrl"
      scope: "restrictedScopes"
      client_secret: "secret"
      redirect_uris: "redirectUris"
      client_name: "name"
      client_id: "clientId"
    extendedValues: {}
    extendedValuesV2:
    - endpointType: "CLIENT_REGISTRATION"
      key: "restrictScopes"
      value: "true"
    - endpointType: "CLIENT_UPDATE"
      key: "restrictScopes"
      value: "true"
  - !<clientMetadataMapping>
    id: "OKTA"
    name: "OKTA"
    type: "clientMetadataMapping"
    owner: "Administrator"
    providerName: "OKTA"
    implNames: {}
    extendedValues: {}
    extendedValuesV2: []
  auth_servers:
  - !<authServerAlias>
    id: "local"
    name: "local"
    description: "Gateway default authorization server"
    type: "authServerAlias"
    owner: "Administrator"
    localIntrospectionConfig:
```

```
      issuer: "JWTISSUER"
    remoteIntrospectionConfig:
     introspectionEndpoint: "http://localhost:5555/invoke/pub.oauth/instrospectToken"
      clientId: "introspection-client"
      clientSecret: "********************************"
      user: "Administrator"
    tokenGeneratorConfig:
      audience: "SAG"
      expiry: 30
      algorithm: "RS256"
      accessTokenExpInterval: 3600
      authCodeExpInterval: 3600
    sslConfig:
      keyStoreAlias: "DEFAULT_IS_KEYSTORE"
      keyAlias: "ssos"
    metadata: {}
    authServerScopes:
    - "Test_LocalOauth"
    - "Dev_LocalOauth"
    supportedGrantTypes:
    - "authorization_code"
    - "password"
    - "client_credentials"
    - "refresh_token"
    - "implicit"
    oauthTokens: []
---
```

3 Microgateway Provisioning

Microgateway Provisioning

Microgateway provisioning allows you to spawn multiple Microgateway instances from a single Microgateway installation. The instances can be defined as self-contained including assets and configuration. You can only provision a pre-configured Microgateway.

You can provision a Microgateway in one of the following ways:

- Instance-based provisioning.
- Docker-based provisioning.

Instance-based Provisioning

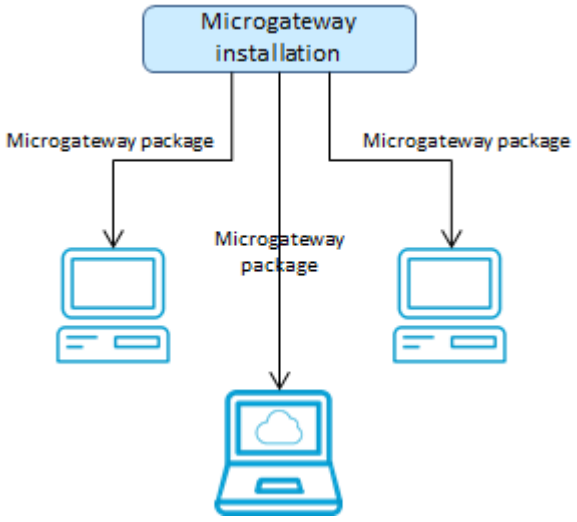
You can create a Microgateway package from an existing installation and copy it to multiple target machines. A Microgateway instance package is a self-contained zip file that contains all artifacts for running a Microgateway. The contents of the zip file are:

- JRE
- Microgateway server
- Microgateway CLI
- Configuration files
- One or more exported API Gateway asset archives

After copying the zip file , you have to extract the zip file contents and run the commandline scripts of the Microgateway CLI to start the Microgateway. A package is configured based on a Microgateway configuration file. The configuration file either points to an API Gateway or to one or more API Gateway asset archives for asset promotion. The asset promotion is performed when you start the Microgateway within the package.

Note:
On Linux it is required that certain permissions must be set after the package has been extracted from the zip. This should be done with the provided script: `./setpermissions.sh`

The figure illustrates creating a Microgateway package from an existing installation and copying it to multiple target machines.



Creating a Microgateway Instance

For the package-based provisioning the Microgateway CLI provides the `createInstance` command. The command creates a Microgateway package that can be transferred to any target environment. The command has the following parameters:

- `config`: optional Microgateway configuration file
- `instance`: the required file or path name of the Microgateway package.

Instance-based Provisioning Flow

A sample sequence for executing an instance-based provisioning is as follows:

1. Create API Gateway archives and set the configuration file accordingly.

```
---
ports:
   http: 5554
   https: 5553

api_gateway:
   url: host:port
   user: Administrator
   password: password
archive:
   file: asset.zip


---
```

2. Create a Microgateway instance.

```
microgateway.bat createInstance -config/config custom-settings.yml
--instance c:/tmp/Microgateway.zip -os win
```

The provided custom-settings.yml is added to the root of MyMicrogateway.zip that can be used for server startup.

3. Copy the Microgateway instance to the target environment.
4. Extract the contents of the zip file using the unzip command and start the Microgateway instance.

```
cd c:/tmp/myinst
unzip c:/tmp/MyMicrogateway.zip
microgateway.bat start --config custom-settings.yml
```

The unzip operation creates the sub-folders of the Microgateway. The start command picks up the custom settings file from the base location. Ensure that on the necessary files the execute-bit is set by applying the setpermissons.sh command (linux-only):

- microgateway.sh
- microgateway-jre-linux/bin/java

A Sample workflow for Instance-based Provisioning with Custom Settings

You can establish a Microgateway instance with a customized configuration setting environment. On start the Microgateway server picks up the specified customized settings. A sample sequence for instance-based provisioning with custom settings looks as follows.

1. Create a Microgateway instance with settings picked up from API Gateway.

```
./microgateway.sh downloadSettings -gw http://hostname:port
--output my-custom-settings.yml
./microgateway.sh createInstance -instance MyInst.zip
--config my-custom-settings.yml
```

2. Publish the Microgateway instance and start it.

```
unzip MyInst.zip -d /tmp/inst
cd /tmp/inst
. ./setpermissions.sh
./microgateway.sh start -c my-custom-settings.yml
```

You can also save a Microgateway environment (used API and all configuration settings) in a version control system, check it out when required, and start as follows:

```
# Checkout a microgateway environment and start it
svn checkout http://repository/... myarchive.zip my-custom-settings.yml
./microgateway.sh start --config my-custom-settings.yml --archive myarchive.zip
```

3. Once the settings are created, you can spawn multiple instances with the same settings on multiple machines.

A Sample workflow for Instance-based Provisioning by Downloading the settings from API Gateway

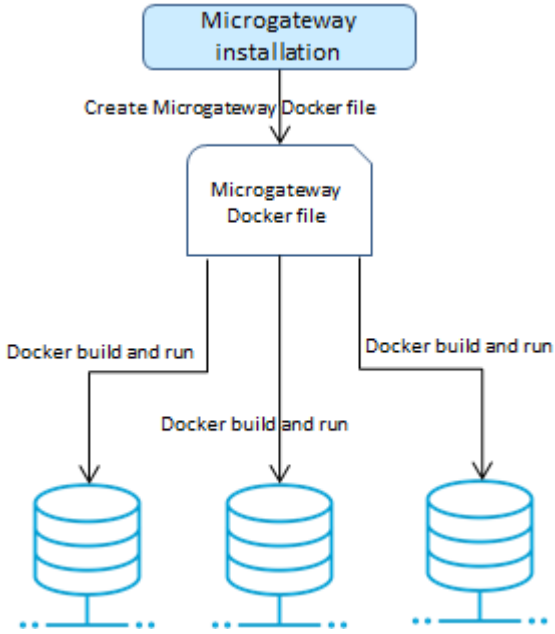
A sample flow is as follows:

```
./microgateway.sh createInstance -instance MyInst.zip -c my-base-settings.yml
unzip MyInst.zip -d /tmp/inst
/tmp/inst/microgateway.sh start -c my-base-settings.yml --download_settings true
```

The download_settings parameter used while starting Microgateway downloads the settings from API Gateway.

Docker-based Provisioning

Docker-based provisioning involves the creation of a Microgateway Docker file from an existing installation, building the image, and running it multiple times in a container environment as depicted in the following figure.



Microgateway Docker image

For the Docker-based provisioning the Microgateway CLI provides the `createDockerFile` command. The command creates a Docker file that can be consumed by docker build for creating a Docker image. The Microgateway Docker image contains an unzipped Microgateway package.

The command takes the command line options detailed in the Command Line Reference.

You can run the Docker image to spawn a Docker container from the created docker image.

The Docker images resulting from Docker files created using the `createDockerFile` command feature the following:

- **Docker logging**

  Microgateway Docker containers log to stdout and stderr. The Microgateway logs can be fetched with the Docker logs command.

- **Docker health check**

Microgateway Docker containers perform health checks.

```
HEALTHCHECK CMD ${MICROGW_DIR}/microgateway.sh status 2>&1 | grep 'Server active'
```

The status command checks the Microgateway availability. If the status command confirms an active Microgateway the container is considered healthy.

- **Graceful shutdown**

  When the docker stop command is used on a Microgateway container it performs a graceful shutdown.

- **Entrypoint support**

  Microgateway Dockerfile exposes an ENTRYPOINT. The options provided to the createDockerFile command are supplied to the ENTRYPOINT through a CMD specification. An example of a generated Docker file is as follows:

```
FROM openjdk:8-jre-alpine
ENV MICROGW_DIR /opt/softwareag/Microgateway
MAINTAINER SoftwareAG
RUN mkdir -p ${MICROGW_DIR}/logs
RUN adduser -u 1724 -g 1724 -D -h ${MICROGW_DIR} sagadmin
RUN chown -R 1724:1724 /opt/softwareag
COPY --chown=1724:1724 ./config/ ${MICROGW_DIR}/config/
COPY --chown=1724:1724 ./lib/ ${MICROGW_DIR}/lib/
COPY --chown=1724:1724 ./files/ ${MICROGW_DIR}/files/
COPY --chown=1724:1724 ./resources/ ${MICROGW_DIR}/resources/
COPY --chown=1724:1724 ./*.jar ${MICROGW_DIR}/
COPY --chown=1724:1724 ./*.sh ${MICROGW_DIR}/
COPY --chown=1724:1724 ./tmp-docker/EmployeeService.zip ${MICROGW_DIR}/config
USER 1724
EXPOSE 4001

WORKDIR ${MICROGW_DIR}

ENTRYPOINT ["java","-jar","microgateway-server.jar"]
CMD [ "-p", "9090", "-a", "config/EmployeeService.zip", "-lv", "INFO"]
```

  When running a Microgateway Docker image the default options may be overridden by supplying them as options to the docker run command line.

```
docker build -t sag:mcgw -f Microgateway_DockerFile .
docker run -d -p 9091:9091 --name mcgw sag:mcgw  -p 9091
-a config/EmployeeService.zip -lv TRACE
```

  This docker run command overrides the default port, asset archive, and logging levels specified by the Dockerfile CMD.

- **JRE support**

  The `createDockerFile` command adds a Microgateway JRE to the Docker file so that Microgateway Docker image can be self-contained. Since the custom base image provides a JRE, the `createDockerFile` command supports the `jre=none` option to reuse the existing JRE and not copy the Microgateway JRE.

Microgateway provides a musl libc compatible JRE to support Alpine Docker-based images. The Microgateway installation provides the musl libc compatible JRE in the `microgateway-jre-linux-musl` folder. You have to specify the `jre=linux-musl` option in the `createDockerFile` command to copy the musl libc compatible JRE. If there is no base image specified the musl libc compatible JRE is copied. The available JRE options are `linux`, `linux-musl`, and `none`. The default value for the jre option depends on the docker_from value:

- If there is no docker_from value specified, then the JRE used is linux-musl as the default base image is Alpine.
- If you specify docker_from value, then the JRE used is linux

A Sample workflow for Docker-based Provisioning with Asset Archive

A sample sequence for executing a Docker-based provisioning with the asset archive looks as follows.

1. Create API Gateway archives and set the configuration file accordingly

```
---
ports:
   http: 5554
   https: 5553


api_endpoint:
    base_path: /gateway

api_gateway:
   url: http://localhost:5555/rest/apigateway
   user: Administrator
   password: <pwd>

admin_api:
    base_path: /rest/microgateway

archive:
   file: asset.zip

---
```

   You can specify the archive zip file in the custom settings YAML file or have it on the command line.

   Note:
   The command line parameters take precedence over the configuration values specified in the custom settings YAML file.

2. Create Microgateway Docker file.

```
./microgateway.sh createDockerFile --docker_dir . -p 9090 -a <file-name>.zip
```

   The command creates the Docker file Microgateway_DockerFile that copies the asset archives referenced by the zip file into the Docker image.

The createDockerFile command allows to configure the Docker file creation using command line parameters. For example, the following command sequence creates a Docker file for a Microgateway container listening on port 9090 and with the assets from the archives apis.zip and policies.zip.

```
./microgateway.sh createDockerFile -p 9090 -c custom-settings.yml
-a apis.zip,policies.zip
```

3. Create the Docker image with asset archive.

```
docker build -t sag:mcgw-static -f Microgateway_DockerFile .
```

The command creates the image sag:mcgw-static from the generated Docker file.

4. Run the Docker image.

```
docker run -d -p 9090:9090 --name mcgw-static sag:mcgw-static
```

The command spawns a Docker container from the image sag:mcgw-static. The Microgateway container listens on the host port 9090.

A Sample workflow for Docker-based Provisioning with pulling from API Gateway

A sample sequence for executing a Docker-based provisioning with pulling from API Gateway looks as follows.

1. Create Microgateway Docker file points to an API Gateway for pulling APIs on startup.

```
./microgateway.sh createDockerFile --docker_dir . -c config/custom-settings.yml
```

The command creates the Docker file Microgateway_DockerFile that copies the API Gateway configurations from the custom settings YAML file into the Docker image.

2. Create the Docker image.

```
docker build -t sag:mcgw-dynamic -f Microgateway_DockerFile .
```

The command creates the image sag:mcgw-dynamic from the generated Docker file.

3. Run the Docker image.

```
docker run -d -p 9090:9090 --name mcgw-dynamic sag:mcgw-dynamic
```

The command spawns a Docker container from the image sag:mcgw. The Microgateway container listens on the host port 9090.

A Sample workflow for Docker-based Provisioning with dynamic configuration

Dynamic configuration is applied by mapping a volume holding the config folder of the Microgateway instance.

A sample sequence for executing a Docker-based provisioning with dynamic configuration looks as follows.

1. Create Docker file pointing to the user-defined custom settings YAML file.

```
./microgateway.sh createDockerFile --docker_dir . -c config/custom-settings.yml
-p 9090
```

The command creates the Docker file Microgateway_DockerFile that copies the configurations from the custom-settings.yml file into the Docker image.

2. Create the Docker image.

```
docker build -t sag:mcgw-dynamic -f Microgateway_DockerFile .
```

The command creates the image sag:mcgw-dynamic from the generated Docker file.

3. Provide config directory with host file system with required Microgateway configurations:

```
> ls ~/mcgw-conf/
custom-settings.yml  keystore.jks  license.xml  system-settings.yml
```

4. Start container with volume mapping pointing to config directory in host file system.

```
docker run -d -v ~/mcgw-conf:/Microgateway/config
-p 9090:9090 --name mcgw-dynamic sag:mcgw-dynamic
```

The command spawns a Docker container from the image sag:mcgw-dynamic. The Microgateway container listens on the host port 9090.

A Sample workflow for Docker-based Provisioning with Custom Settings

A Microgateway Docker instance can be established and run using customized settings. A sample sequence for docker-based provisioning with custom settings looks as follows.

1. Create the custom settings file. These settings are pulled from a specified API Gateway

```
./microgateway.sh downloadSettings -gw http://hostname:port
--output my-custom-settings.yml
```

These settings are pulled from a particular API Gateway instance specified by its hostname and port number.

2. Create Docker file pointing to the customized settings file.

```
./microgateway.sh createDockerFile --http_port 7071 --docker_dir .
--archive myarchive.zip --config my-custom-settings.yml
```

The command creates the Docker file Microgateway_DockerFile that copies the API Gateway configurations from the `my-custom-settings.yml` file into the Docker image.

3. Create the Docker image.

```
docker build -t sag:mgcustomsettings -f Microgateway_DockerFile .
```

The command creates the image sag:mgcustomsettings from the generated Docker file.

4. Run the Docker image.

```
docker run -d -p 7071:7071 --name mgcustomsettings sag:mgcustomsettings
```

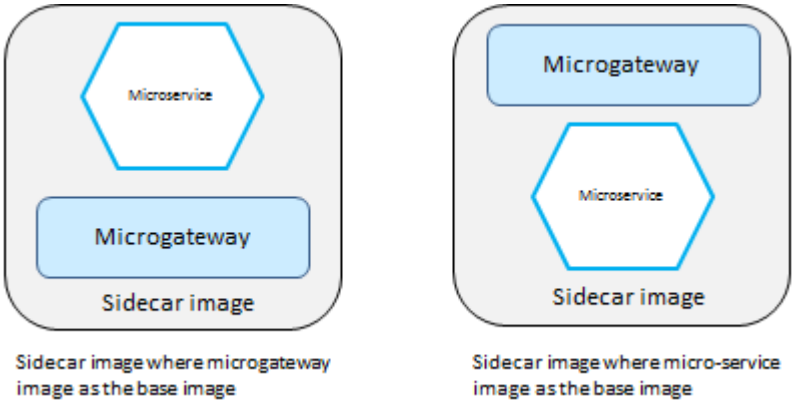
The command spawns a Docker container from the image sag:mgcustomsettings. The contained Microgateway listens at the host port 7071.

Sidecar Container support

You can have Microgateway in a sidecar deployment mode where Microgateway runs with micro-services in the same Docker container. This provides protection to the micro-service as the micro-services can only be reached through the Microgateway. In the sidecar deployment, you can have the Microgateway Docker image in one of the following ways:

- Microgateway Docker image is the base image for the sidecar image.
- Microgateway is added on top of an existing custom image holding the micro-service

This figure depicts the two different ways in which you can have the Microgateway Docker image in the sidecar deployment.



The sidecar deployment requires to spawn multiple processes in Microgateway container. Therefore, the `createDockerFile` command supports the exec option.

The purpose of the `exec` option in the `createDockerFile` command is to start the micro-service process. The `exec` option allows you to specify a shell command that is being added to the Microgateway docker-entrypoint.sh. For example, the following command creates a sidecar Docker image based on a micro-service base image running the SAG node-tours application:

```
./microgateway.sh createDockerFile -dor sag:node-tours
-exec "(cd /usr/src/app; npm start)" -dod . -p 9090 -a ../node-tours.zip
```

When starting the image the command specified in the `exec` option is executed first. The Docker file looks as follows:

```
FROM sag:node-tours

MAINTAINER SoftwareAG

ENV MICROGW_DIR /opt/softwareag/Microgateway

RUN mkdir -p ${MICROGW_DIR}/logs

COPY adduser.sh ${MICROGW_DIR}
RUN ${MICROGW_DIR}/adduser.sh sagadmin 1724 ${MICROGW_DIR}
```

```
RUN chown -R 1724:1724 /opt/softwareag

COPY --chown=1724:1724 ./config/ ${MICROGW_DIR}/config/
COPY --chown=1724:1724 ./lib/ ${MICROGW_DIR}/lib/
COPY --chown=1724:1724 ./files/ ${MICROGW_DIR}/files/
COPY --chown=1724:1724 ./resources/ ${MICROGW_DIR}/resources/
COPY --chown=1724:1724 ./*.jar ${MICROGW_DIR}/
COPY --chown=1724:1724 ./*.sh ${MICROGW_DIR}/

COPY --chown=1724:1724 ./tmp-docker/node-tours.zip ${MICROGW_DIR}/config

COPY --chown=1724:1724 ./microgateway-jre-linux/
${MICROGW_DIR}/microgateway-jre-linux/

EXPOSE 9090

HEALTHCHECK CMD ${MICROGW_DIR}/microgateway.sh status 2>&1 |
grep 'Server active'

WORKDIR ${MICROGW_DIR}

USER 1724

ENTRYPOINT ["./docker-entrypoint.sh"]
CMD ["-p", "9090", "-a", "config/node-tours.zip", "-lv", "ERROR"]
```

The simple docker-entrypoint.sh script is as follows:

```
#!/bin/sh
(cd /usr/src/app; npm start) &
./microgateway-jre-linux/bin/java -jar microgateway-server.jar $@
```

The sample script runs the provided command to start the node.js based micro-service in the background. Then Microgateway is started with the command line parameters.

The Docker file can be used for creating an image using the `docker build` command.

```
docker build -t sag:mcgw-node-tours .
```

The resulting Docker image sag:mcgw-node-tours holds the Microgateway and the NodeTours micro-service. Starting the Docker container starts the Microgateway and the node.js at runtime.

```
docker run -d -p9090:9090--name mcgw-node-tours
sag:mcgw-node-tours
```

The Docker container only exposes the Microgateway port the node.js port is not exposed. Therefore the NodeTours micro-service can not be called directly.

Microgateway image based on an MSR image

You can use the --msr option in the `createDockerFile` command to detect an image holding a webMethods Microservice Runtime (MSR).

If a MSR image is detected, the `createDockerFile` command does not add any jre, since the MSR already provides a jvm. The `createDockerFile` command for creating an MSR based sidecar image is as follows:

```
./microgateway.sh createDockerFile -dod . -dor sag:msr-employee-service
-msr -p 9090 -a ../EmployeeService.zip
```

The Docker file created by the command is as follows:

```
FROM sag:msr-employee-service

MAINTAINER SoftwareAG

ENV MICROGW_DIR /opt/softwareag/Microgateway

RUN mkdir -p ${MICROGW_DIR}/logs

COPY --chown=1724:1724 ./config/ ${MICROGW_DIR}/config/
COPY --chown=1724:1724 ./lib/ ${MICROGW_DIR}/lib/
COPY --chown=1724:1724 ./files/ ${MICROGW_DIR}/files/
COPY --chown=1724:1724 ./resources/ ${MICROGW_DIR}/resources/
COPY --chown=1724:1724 ./*.jar ${MICROGW_DIR}/
COPY --chown=1724:1724 ./*.sh ${MICROGW_DIR}/

COPY --chown=1724:1724 ./tmp-docker/EmployeeService.zip ${MICROGW_DIR}/config

EXPOSE 9090

HEALTHCHECK CMD ${MICROGW_DIR}/microgateway.sh status 2>&1 | grep 'Server active'

WORKDIR ${MICROGW_DIR}

USER 1724

ENTRYPOINT ["./docker-entrypoint.sh"]
CMD ["-p", "9090", "-a", "config/EmployeeService.zip", "-lv", "ERROR"]
```

The Docker image sag:msr-employee-service provides an MSR instance that runs the micro-service EmployeeService. The API definition for the micro-service is provided by the exported API Gateway asset archive, EmployeeService. The image can be created using the `docker build` command.

```
docker build -t
sag:mcgw-msr-employee-service .
```

The resulting Docker image sag:mcgw-msr-employee-service holds the Microgateway and the MSR. Starting the Docker container starts the Microgateway and the MSR.

```
docker run -d -p9090:9090--name mcgw-msr-employee-service
sag:mcgw-msr-employee-service
```

The Docker container only exposes the Microgateway port and the MSR port is not exposed. Therefore the MSR can not be called directly.

Microgateway Docker Environment Variables

You can run a Microgateway docker container with environment variables. This has the advantage that a prepared Docker image can be executed with parameters.

The environment variables can be specified in an generic way based on the YAML configuration layout.

```
Example:
---
ports:
   http: 7071
api_gateway:
   url: “http://localhost:5555”
   dir: "C:\\SoftwareAGapigw"
logging:
   level: "ERROR"
---
```

The variables have the prefix mcgw and every path in the YAML file translates to the environment variables as mcgw_port_http, mcgw_api_gateway_url, mcgw_api_gateway_url,mcgw_logging_level, and so on.

To build and run the Docker image with environment settings, do the following:

1. Create a docker file with a configuration file.

   ```
   ./microgateway.sh createDockerFile --docker_file DockerFileEnv --docker_dir .
   -c myconfig.yml
   ```

   The myconfig.yml file looks as follows:

   ```
   myconfig.yml:
     ports:
       http: 7077
     api_gateway:
       url: "myhost:port"
       user: "myuser"
       password : "mypwd"
     policies:
       user_auth: "internal"
   ```

2. Build Docker image.

   ```
   docker build -t mg-env-image -f DockerFileEnv .
   ```

3. Build Docker image and display the logs.

   ```
   docker run -d -p 7077:7077 --name mg-env-image-bad mg-env-image:latest
   docker logs mg-env-image-bad
   ```

   This displays that the accesses do not work since the API Gateway access properties are invalid and there is no deployed API.

4. Create environment setting file. Prepare a valid environment settings in a file. These are the effective API Gateway access as well as APIs to be pulled from the gateway (env.list):

   ```
   mcgw_api_gateway_url=localhost:5555
   mcgw_api_gateway_user=Administrator
   mcgw_api_gateway_password=pwd
   mcgw_downloads_apis=Employees,EmployeeService
   ```

5. Run docker image with environment settings.

   ```
   docker run -d -p 7077:7077 --env-file env.list --name mg-env-image
   mg-env-image:latest
   ```

```
docker logs mg-env-image
```

This displays that everything is fine.

4 SSL Configuration in Microgateway

SSL Configuration in Microgateway

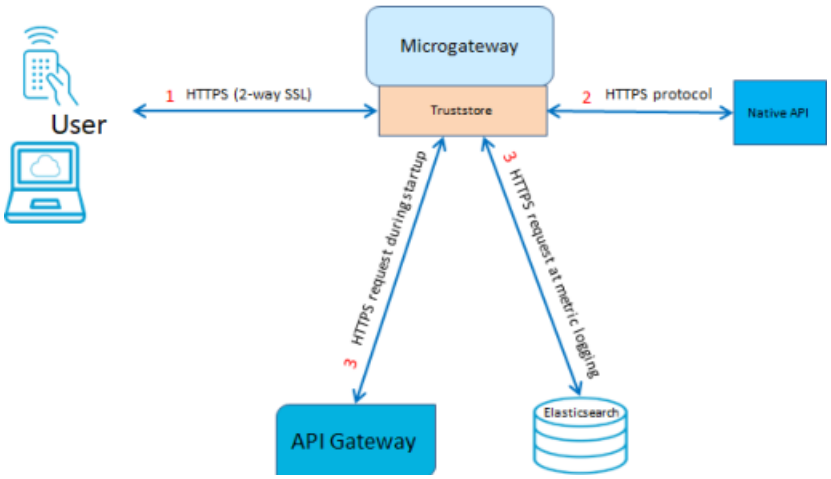
SSL creates a secure connection between servers and clients over the web and internal network, safeguarding and allowing sensitive data to be securely transmitted. HTTPS (Hypertext Transfer Protocol Secure) is an internet communication protocol that protects the integrity and confidentiality of data between the user's computer and the site. The data sent over HTTPS is secured using TLS, which provides protection using encrypted channel.

A Microgateway instance can be communicating with various other components such as, API Gateway server, native services, clients, and Elasticsearch. You must create secure connections between the Microgateway instance and these components in order to enable a secure channel of communication. This article explains SSL configuration in Microgateway.

The article assumes that you have a running Microgateway instance. Additionally, you must have a basic understanding of the following:

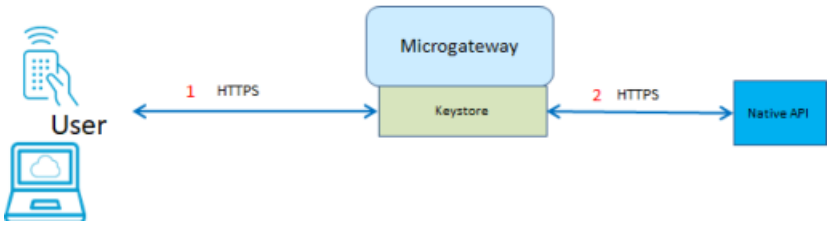
- API Gateway server and administration configuration in API Gateway
- Java security using keystore and truststore certificates

The figure depicts various scenarios where a truststore is used in Microgateway.



1. Two-way SSL connection with the end user
2. HTTPS connection with the native API
3. HTTPS communication with related servers depending on their usage

This figure depicts various scenarios where a keystore is used in Microgateway.



1. HTTPS connection with an end user
2. HTTPS connection with the native API

Managing Certificates in Microgateway

As Microgateway is expected to be scaled up to many instances, managing certificates is important. Microgateway has its own keystore and truststore to manage its certificates. Certificates are required to implement a trust relationship between API consumers, Microgateways, API Gateways and native (micro) services depending on the scenario in which it is used.

Public Certificates

To expose a trusted endpoint, Microgateway has to expose a certificate that has been signed by a certificate authority (CA). Usually the CA is a global widely trusted authority such as GlobalSign, Let's Encrypt, or GeoTrust. For internal usage, CA can also be established within an organization. Using multi-name certificates (SAN) or wild card certificates allow the reuse of a certificate across multiple domains or servers.

Internal Certificates

For internal certificates, you can establish a private CA. You can manually manage certificates through SSL tooling. For example, you can have Vault from Hashicorp that provides a private CA solution, which can be deployed on-premises. Other CA solutions are offered by cloud vendors such as Amazon.

How Do I Secure Microgateway Communication with Clients?

Secure Microgateway to enable various clients to communicate with the Microgateway instance over HTTPS. This use case explains how to secure Microgateway communication using HTTPS protocol with the existing server and client certificates.

The use case starts when you have a Microgateway instance to be secured using HTTPS and you have the required server and client certificates to secure the communication channel between Microgateway and the client. It ends when the secure channel is configured for communication between Microgateway and the client.

To secure Microgateway communication with clients

1. Configure keystore.

   Ensure that the keystore with the required certificates is located at *Install_dir*/Microgateway/config/microgateway_keystore.jks. Open the system-settings.yml file and update the following information.

   ```
   key_store:
      type: JKS
      provider: SUN
      location: config/microgateway_keystore.jks
      password: yourpwd
   ```

   You can use the default keystore or use any other custom key.

2. Configure HTTPS port in Microgateway using the following parameters:

   - Use the parameter `key_alias` in the config.yml file to force using a dedicated certificate, if the keystore contains multiple certificates. For example, to configure the HTTPS port as 9093 and use the certificate microgateway_cert, the entry in the config.yml file should be as follows:

     ```
     ports:
        https: 9093
        key_alias: microgateway_cert
     ```

   - Use the advanced parameters to set Jetty SSLContext parameters for the exposed HTTPS port. For details, see https://wiki.eclipse.org/Jetty/Howto/Configure_SSL. Microgateway supports the following parameters:

     - `include_cipher_suites`. For details, see https://wiki.eclipse.org/Jetty/Howto/CipherSuites.
     - `exclude_cipher_suites`. For details, see https://wiki.eclipse.org/Jetty/Howto/CipherSuites .
     - `need_client_auth`. The default value is `false`.
     - `want_client_auth`. The default value is `false`.
     - `crl_path`. Specifies the path to certificate revocation list file, which is located at *Install_dir*/config/, for SSL certificate validation.
     - `max_cert_path_length`. Specifies the maximum number of intermediate certificates allowed. The default value is `-1`, which denotes that it is unlimited.

3. Configure truststore.

   Microgateway works with one single truststore for all purposes. Manage this truststore with an entry in config/system-settings.yml, similar to the keystore definition, as follows:

   ```
   trust_store:
        type: JKS
        provider: SUN
        location: config/truststore.jks
        password: <pwd>
   ```

4. Enable host name verification.

   Set the Global SSL setting as `default`.

   ```
   ssl_options:
     host_name_verifier: default #none can be specified to deactivate hostname
   verification
   ```

   The available values are:

   - `default`: Provide this value to enable host name verification.
   - `none`: Provide this value to disable host name verification.

   The default value is `none`.

How Do I Secure Microgateway Communication with API Gateway Server?

This use case explains how to secure Microgateway communication with API Gateway server using HTTPS protocol.

The use case starts when you have a Microgateway instance to be secured using HTTPS and you have the required certificates to secure the communication channel between Microgateway and API Gateway. It ends when the secure channel is configured for communication between Microgateway and API Gateway server.

To secure Microgateway communication with API Gateway server

1. Configure truststore.

   Microgateway may use any of the following:

   - Default truststore: The default truststore is located in the cacerts of the Microgateway JRE.
   - Custom truststore: Custom trusstore that may have a truststore configuration imported from API Gateway.

     For details on importing truststore configuration from API Gateway, see "Importing Truststore Configuration Configured in API Gateway" on page 47

2. Ensure that Microgateway communicates over the HTTPS port exposed by the API Gateway server.
3. Configure Keystore.

   Ensure that the required API Gateway server certificates are placed in the Microgateway keystore located at *Install_dir*/Microgateway/config/.

   You can use self-signed certificates or custom CA. For details on configuring keystore for self-signed certificates, see "Configuring Keystore in Microgateway" on page 48

4. Ensure that the Microgateway cacerts are placed in the API Gateway server truststore located at *API Gateway _Install_dir*/common/config/.

How Do I Secure Microgateway Communication with the Native API?

This use case explains how to secure Microgateway communication with the native service using HTTPS protocol.

The use case starts when you have a Microgateway instance to be secured using HTTPS and you have the required certificates to secure the communication channel between Microgateway and the native API. It ends when the secure channel is configured for communication between Microgateway and the native API.

To secure Microgateway communication with native API

1. Configure truststore.

   Microgateway may use any of the following:

   - Default truststore: The default truststore is located in the cacerts of the Microgateway JRE.
   - Custom truststore: Custom truststore that may have a truststore configuration imported from API Gateway.

     For details on importing truststore configuration from API Gateway, see "Importing Truststore Configuration Configured in API Gateway" on page 47

2. Ensure that Microgateway accesses the native API using the API's HTTPS endpoint exposed.

How Do I Secure Microgateway Communication with Elasticsearch?

This use case explains how to secure Microgateway communication with Elasticsearch using the HTTPS protocol.

The use case starts when you have a Microgateway instance to be secured using HTTPS and you have the required certificates to secure the communication channel between Microgateway and Elasticsearch. It ends when the secure channel is configured for communication between Microgateway and Elasticsearch.

To secure Microgateway communication with Elasticsearch

1. Configure truststore.

   Microgateway may use any of the following:

   - Default truststore: The default truststore is located in the cacerts of the Microgateway JRE.
   - Custom truststore: Custom truststore that may have a truststore configuration imported from API Gateway.

     For details on importing truststore configuration from API Gateway, see "Importing Truststore Configuration Configured in API Gateway" on page 47

2. Ensure that Microgateway communicates over the HTTPS port configured on Elasticsearch.

   Configure the port of communication with Elasticsearch, in the system-settings.yml file, as follows:

```
es_destination:
   protocol: "https"
   hostName: "localhost"
   port: "8880"
   userName: ""
   password: ""
```

3. Ensure that the required Elasticsearch certificates are placed in the Microgateway truststore located at *Install_dir*/Microgateway/config/.

   You can use self-signed certificates or custom CA. For details on configuring keystore for self-signed certificates, see "Configuring Keystore in Microgateway" on page 48

Importing Truststore Configuration Configured in API Gateway

When SSL configurations are imported from API Gateway to Microgateway, they must function seamlessly without any disruption. The multiple truststore files used in API Gateway are successfully imported into Microgateway at startup and used across different SSL configurations. You can copy the truststore configurations from API Gateway including the passwords to these files during the Microgateway instance creation. These password files and truststore configurations are loaded during Microgateway startup.

Microgateway works with one single truststore for all purposes. You can manage this truststore with an entry in config/system-settings.yml, similar to the keystore definition:

```
trust_store:
     type: JKS
     provider: SUN
     location: config/truststore.jks
     password: <pwd>
```

By default, Microgateway does not use any particular truststore for communication. In such a case, the default cacerts are located in the Microgateway JVM.

API Gateways may have more than one truststore defined. If the API Gateway instance from where the truststore is imported has multiple truststores, then specify multiple truststore names (and passwords) to import them from respective truststores.

You can also use the import_truststore parameter within createDockerFile, where the truststore used within the image is prepared with importing certificates. The resulting truststore is targeted in the folder for creating the docker image: .../Microgateway/tmp-docker/truststore.jks

To import the truststore configuration from an API Gateway instance

1. You can perform one of the following based on your requirement:

   - To create a new Microgateway instance and import the default truststore from API Gateway, run the following command:

     ```
     ./microgateway.sh createInstance -gwd c:/SoftwareAGapigw
     -itf name -itp pwd
     ```

     Where the name is set to "." . In this case the API Gateway default trust store platform_truststore.jks is imported.

     For example, if you want to import the default truststore, run the following command:

     ```
     microgateway createInstance -gwd c:/SoftwareAGapigw -itf default -itp manage
     ```

Where the default truststore is `default` and password is `manage`.

- To start the Microgateway server and import certificates from multiple truststores configured and available in c:/SoftwareAGapigw/common/conf/mytrust/my_truststore_file.jks, run the following command:

```
./microgateway.sh start -c config.yml -gwd c:/SoftwareAGapigw
-itf name[,name...]
-itp pwd[,pwd...]
```

Where, the *name* argument is the truststore name of the user-configured trust store.

Microgateway reads all the certificates from the API Gateway truststore(s) and saves them to the Microgateway truststore. If a certificate being imported is already present in the Microgateway truststore, then it gets overwritten.

For example, if you want to import certificates from two user-configured truststores from c:/SoftwareAGapigw/common/conf/mytrust/my_truststore_file.jks, run the following command:

```
./microgateway.sh start -c config.yml -gwd c:/SoftwareAGapigw -itf
mytrust,mytrust2
-itp mytrustpwd,mytrustpwd2
```

- To create a Docker file by importing truststore data from API Gateway, run the following command:

```
./microgateway.sh createDockerFile -c config.yml -dod . -gwd c:/SoftwareAGapigw

-itf - -itp manage
```

Here, - denotes that you are importing the default truststore and the truststore is updated in .../Microgateway/tmp-docker/truststore.jks

Configuring Keystore in Microgateway

Microgateway comes with a default keystore, containing a private and a public key for HTTPS communication with the user. The keystore is located in *Install_dir*/Microgateway/config/keystore.jks and you can manage the keystore with an entry in config/system-settings.yml file.

At times, you may want to use a private and public key within a keystore, due to the fact that a Microgateway can be duplicated with the `createInstance` command and the keystore can be copied as well.

To configure keystore in Microgateway

1. Create a self-signed certificate.

Note:
You perform this step only if you want to use self-signed certificates. If you are using the existing certificates, go to Step 2.

a. To create a keystore file, run the following command with the required information:

```
cd …/Microgateway/config
keytool -genkeypair -alias microgateway_cert -keyalg RSA
-keysize 2048 -keystore microgateway_keystore.jks -storepass yourpwd
```

This creates the keystore file.

b. Open the system-settings.yml file and update the following information.

```
key_store:
   type: JKS
   provider: SUN
   location: config/microgateway_keystore.jks
   password: yourpwd
```

On the first access of the Microgateway server, the keystore password gets encrypted and inserted into the Microgateway's passman file to avoid a clear-text password in the file.

2. Start the Microgateway server with an HTTPS port to communicate over HTTPS.

For example:

```
./microgateway.sh start --https_port 7072 -a BayernRest.zip
```

5 Kubernetes Support

Overview

Microgateway can be run within a Kubernetes (k8s) environment. Kubernetes provides a platform for automating deployment, scaling, and operations of services. The basic scheduling unit in Kubernetes is a *pod*. It adds a higher level of abstraction by grouping containerized components. A pod consists of one or more containers that are co-located on the host machine and can share resources. A Kubernetes service is a set of pods that work together, such as one tier of a multi-tier application.

The Kubernetes support includes the following:

- Liveliness check to support Kubernetes pod lifecycle: This helps in verifying that the Microgateway container is up and responding. You can perform the liveliness check by checking the alive file of Microgateway.
- Readiness check to support Kubernetes pod lifecycle: This helps in verifying that the Microgateway container is ready to serve requests.

  For details on pod lifecycle, see *Kubernetes documentation*.
- Prometheus metrics to support the monitoring of Microgateway pods. Microgateway exposes metrics in Prometheus format. The Prometheus based monitoring provides information relevant for the Microgateway operation. You use the metrics endpoint `/rest/microgateway/metrics` to gather the required metrics. The metrics gathered are of two types; the server-level metrics and API-level metrics. For details of the server-level metrics and API-level metrics collected, see "Prometheus Microgateway Metrics" on page 61.

The following sections describe in detail the various ways of deploying Microgateway in Kubernetes. Each of the deployment models described require an existing Kubernetes environment. For details on setting up of a Kubernetes environment, see *Kubernetes documentation*.

Deploying Microgateway as a Kubernetes Service

Microgateway can run as a separate Kubernetes service protecting other Kubernetes services. Microgateway as a Kubernetes service is running in a dedicated pod and protects one or more native services running within the Kubernetes environment. The set of services have to be static since Microgateway can only handle a static set of services.

This section explains how to deploy Microgateway as a Kubernetes service.

To deploy a Microgateway as a Kubernetes service

1. Ensure you have a running Kubernetes environment.

   For details on setting up of a Kubernetes environment, see *Kubernetes documentation*.
2. Create a Microgateway Docker image.

   a. Create a Docker file with any configuration.

For example, prepare a Microgateway image running the EmployeeService:

```
./microgateway.sh createDockerFile --http_port 7071
--docker_file MyDockerFile --docker_dir . --archive EmployeeService.zip
```

b. Build the Microgateway Docker image and prepare it for pushing to a Docker registry to make it available for Kubernetes.

```
docker build -t mg-employees -f MyDockerFile .
docker tag my-microgw reghost:regport/mg-employees
```

c. Push the Docker image to the Docker registry.

```
docker push reghost:regport/mg-employees
```

The Microgateway Docker image is now ready and can be used from Kubernetes.

3. Create and expose Microgateway as Kubernetes deployment as follows:

```
kubectl create deployment mg-employees
--image=reghost:regport/my-employees
kubectl expose deployment mg-employees --type=NodePort --port=7071
```

A Kubernetes pod is created and started, after which a Kubernetes service is exposed through a port that can be accessed from outside the cluster.

4. Verify the Microgateway Kubernetes service definition and the exposed IP and port.

Get the Kubernetes services and find out the IP of the running Microgateway service:

```
kubectl get services
NAME         TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)  …
mg-employees NodePort    10.99.156.94   <none>        7071:31428/TCP
. . .
```

You should now be able to access Microgateway service as follows:

```
GET http://10.99.156.94:31428/rest/microgateway/status
```

Deploying Microgateway as a Kubernetes Service using a YAML file

Microgateway offers capability to generate a deployment YAML file, which can be used to set up pods and services indside a Kubernetes server. WHen accessing API Gateway for APIs to be used, you can specify appropriate parameters, which are then specified as environment settings in the YAML file.

To deploy a Microgateway as a Kubernetes service using a YAML file

1. Ensure you have a running Kubernetes environment.

For details on setting up of a Kubernetes environment, see *Kubernetes documentation*.

2. Ensure that you have a Microgateway Docker image for the containers to be started on Kubernetes.

3. Create the deployment YAML file by running the createKubernetesFile command.

   Example:

   ```
   ./microgateway.sh createKubernetesFile -p 7071
   -di lean-docker-image -pn mymg -gw myhost:5555
   -gwu admin -apis MyAPI -au accessUrl -o MyMicrogatewayDeploy.yaml
   ```

   The arguments used in the command specify the following:

Argument	Description
-p	Microgateway access port to be used
-di <a lean docker image>	Reference to a Microgateway image in a docker registry
-pn	Name for deployment, pod and service
-gw	API Gateway host and port (can also be an address of Kubernetes service)
-gwu	User accessing API Gateway. The password should be specified with standard Kubernetes secrets)
-apis	API to be used that are accessed API Gateway
-au	The access URL through which Microgateway can be reached. If this value is not provided the default value taken would be http://*Kubernetes-service-name*:*port*
-o	The output YAML file, which can be used for deployment

4. Deploy the deployment YAML file to establish a Microgateway pod with a Kubernetes service.

   Example:

   ```
   kubectl -f MyMicrogatewayDeploy.yaml
   ```

   Microgateway now registers to the specified API Gateway, and API Gateway can reach Microgateway through the address given with the -au parameter.

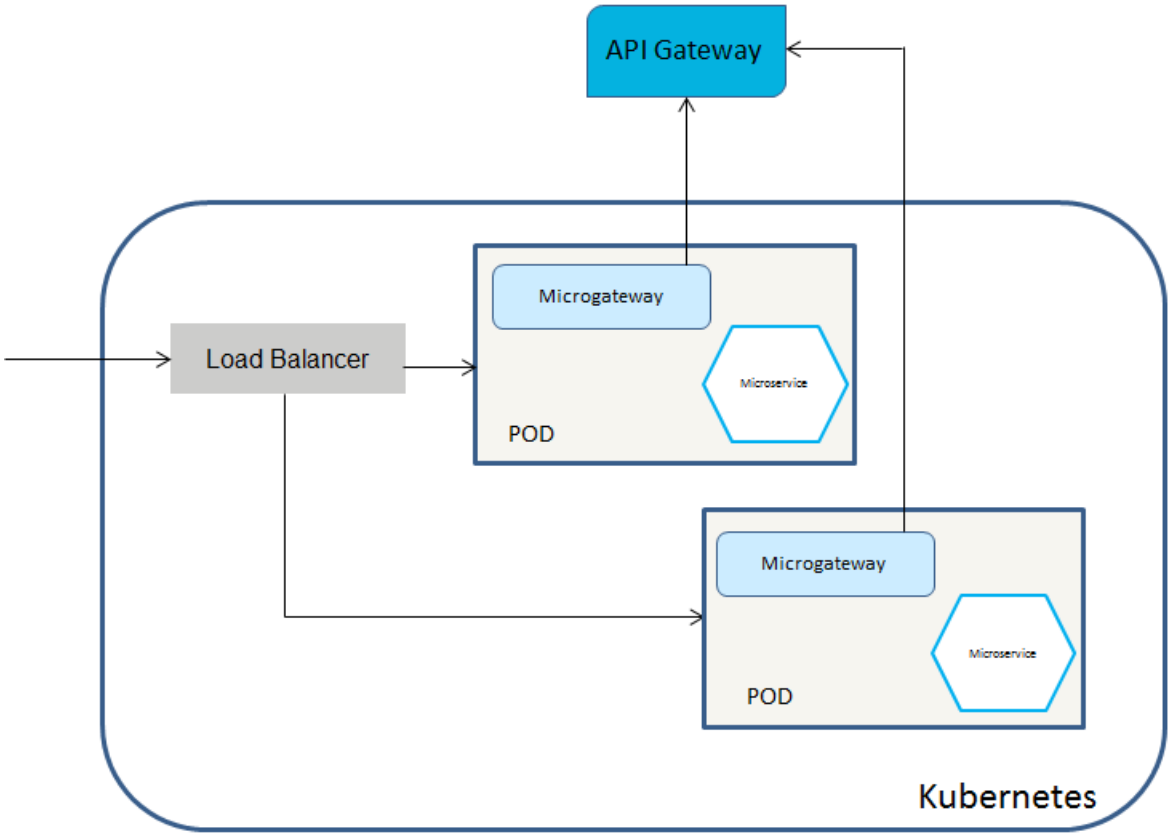
5. Verify that the connections are successful from the Microgateway management section in API Gateway UI.

   For details about Microgateway management section in API Gateway, see *webMethods API Gateway User's Guide*

Kubernetes Sidecar Deployment

A Microgateway Kubernetes sidecar deployment can be established by creating a pod containing two containers; one container runs the native service, and the other container runs the Microgateway protecting the native service.

The diagram depicts Microgateway Kubernetes sidecar deployment.



Each pod consists of a Microgateway container and the native service container. The Microgateway can be connected to an API Gateway to pull API definitions and to push runtime metrics.

The native services are accessed by consumers through the Microgateway endpoint. Since the native services are not exposed by the Kubernetes configuration the Microgateway can't be by-passed. Consumer requests are routed by the Microgateways to the native services.

To access the native service from the Microgateway container, Microgateway has to use `localhost` as URL together with the port exposed by the native service as both the containers are treated as being within the same host.

You can have the following sidecar deployment models:

- A stand-alone Kubernetes sidecar deployment
- A Kubernetes sidecar connected to API Gateway

Deploying a Stand-alone Kubernetes Sidecar

This section describes how you can deploy a stand-alone Kubernetes sidecar. Stand-alone means that the Microgateways are not connected to an API Gateway. The API definitions are provided through API Gateway export archives.

Before you start with the deployment ensure you have done the following:

- Ensure you have a running Kubernetes environment.

  For details on setting up of a Kubernetes environment, see *Kubernetes documentation*.

- The native service Docker images are pushed to a Docker registry to make them available for the Kubernetes environment.

To set up a Kubernetes sidecar deployment

1. Prepare the Microgateway Docker file.

   You first create an archive with the prepared API from an API Gateway that holds the API definition for EmployeeService and then create a Docker file that runs Microgateway with that asset archive.

   ```
   ./microgateway.sh createAssetArchive -gw localhost:5555
   -gwu user -gwp password -apis EmployeeService -a EmployeeService.zip

   ./microgateway.sh createDockerFile --http_port 7076
   --docker_file DockerFileEmployees
   --docker_dir . --archive EmployeeService.zip
   ```

2. Build the Microgateway Docker image and push it to the Docker registry.

   This is achieved by tagging it with the registry URL.

   ```
   docker build -t mg-employees-sidecar -f DockerFileEmployees .

   docker tag mg-employees-sidecar reghost:regport/mg-employees-sidecar

   docker push reghost:regport/mg-employees-sidecar
   ```

3. Create a template for Kubernetes sidecar deployment.

   Microgateway offers a function to generate a Kubernetes YAML file, which can be used for a convenient deployment. You may specify the sidecar parameters together with the Microgateway image parameters to have the two containers created within one pod. There are additional options for number of pods or adjusting health checks.

   ```
   ./microgateway.sh createKubernetesFile
   --docker_image reghost:regport/mg-employees-sidecar
   --pod_name mg-employees-sidecar
   --sidecar_docker_image reghost:regport/employees-sservice
   --sidecar_pod_name employees-sservice
   --output mg-employees-sidecar-deployment.yml
   ```

4. Create and check the Kubernetes deployment.

```
kubectl create -f mg-employees-sidecar-deployment.yml

kubectl get pods
NAME                    READY   STATUS    ...
mg-employees-sidecar    2/2     Running
```

The Kubernetes pod with the 2 containers are created and started. If the deployment is successful you should see 2 out of 2 containers running. You can now expose Microgateway sidecar deployment as a Kubernetes service.

```
kubectl expose deployment mg-employees-sidecar --type=NodePort --port=7076
```

5. Verify the Microgateway Kubernetes service definition including the exposed IP and port.

   Get the Kubernetes services and find out the IP of the running Microgateway sidecar service:

```
kubectl get services
NAME                  TYPE      CLUSTER-IP    EXTERNAL-IP PORT(S)  …
mg-employees-sidecar  NodePort  10.99.106.44  <none>      7076:31238/TCP
```

   You can now access the Microgateway server.

```
GET http://10.99.106.44:31238/rest/microgateway/status
```

Deploying a Kubernetes Sidecar Connected to API Gateway

This section describes a Microgateway Kubernetes sidecar deployment where the Microgateways are connected to an API Gateway for pulling API definitions and runtime metrics data aggregation. The API Gateway instance may run as Kubernetes service.

Before you start with the deployment ensure you have done the following:

- Ensure you have a running Kubernetes environment.

  For details on setting up of a Kubernetes environment, see *Kubernetes documentation*.

- There is an API Gateway instance running in a dedicated Kubernetes service.
- There is a native service Docker image in the Docker registry (employees-service).

To set up a Kubernetes sidecar deployment connected to API Gateway

1. Create a Microgateway Docker image that can be pushed to the registry the Kubernetes cluster is connected to.

```
./microgateway.sh createDockerFile -c ../pull-employee-service.yml
--docker_file DockerFileEmployees -dod .
```

   The command creates the Docker file DockerFileEmployees based on the custom settings file pull-employee-service.yml that holds the configuration for pulling the definition of the EmployeeService API.

```
api_gateway:
  url: 'http://localhost:5555/rest/apigateway'
  user: 'Administrator'
```

```
  password: 'manage'
ports:
  http: '9090'
downloads:
  apis: "EmployeeService"
microgatewayPool:
  microgatewayPoolName: employee-service-mcgw
  microgatewayPoolDescription: Microgateways of EmployeeServices.
```

The configuration also holds entries to register the Microgateway to the Microgateway pool employee-service-mcgw in the referenced API Gateway.

2. Create the image mg-employees-registered, from the resulting Docker file DockerFileEmployees, using docker build.

```
docker build -t mg-employees-registered -f DockerFileEmployees .
```

3. Tag the resulting image and push it to the registry.

```
docker tag mg-employees-sidecar reghost :regport /mg-employees-registered
docker push reghost :regport /mg-employees-registered
```

4. Create the Kubernetes YAML file using the createKubernetesFile command.

```
./microgateway.sh createKubernetesFile
--docker_image reghost:regport/mg-employees-registered
--pod_name mg-employees-registered
--sidecar_docker_image reghost:regport/employees-sservice
--sidecar_pod_name employees-sservice
--output mg-employees-registered-deployment.yml
```

5. Change the Microgateway configuration, in the resulting Kubernetes file, by modifying the Microgateway environment variables.

For example, the API Gateway URL can be reconfigured through the environment variable mcgw_api_gateway_url as shown by the following YAML fragment:

```
containers:
       name: mg-employees-registered
       image: reghost:regport/mg-employees-registered
       imagePullPolicy: IfNotPresent
     - env:
        - name: mcgw_api_gateway_url
          value: http://10.20.198.90:31929/rest/apigateway
        - name: mcgw_api_gateway_user
          value: Administrator
        - name: mcgw_api_gateway_password
          value: manage
        livenessProbe:
          exec:
            command:
            - /bin/sh
            - -c
            - /opt/softwareag/Microgateway/files/k8s-lifenesscheck.sh
          failureThreshold: 3
          initialDelaySeconds: 10
          periodSeconds: 10
          successThreshold: 1
          timeoutSeconds: 5
```

```
        readinessProbe:
          exec:
            command:
            - /bin/sh
            - -c
            - /opt/softwareag/Microgateway/files/k8s-readinesscheck.sh
          failureThreshold: 3
          initialDelaySeconds: 10
          periodSeconds: 10
          successThreshold: 1
          timeoutSeconds: 5
```

Deploying a Kubernetes Sidecar using Helm Charts

If you are using helm charts for your Kubernetes deployments and management, then you can follow the outlined procedure for a convenient Microgateway Kubernetes deployment.

To set up a Microgateway Kubernetes sidecar deployment with Helm charts

1. Install Helm and Tiller.

   For details on installing Helm and Tiller, see *Helm documentation* and *Tiller documentation*.

2. Create your Helm chart.

   ```
   helm create mg-helmchart
   ```

3. Replace deployment template data.

   To achieve this, edit ./mg-helmchart/templates/deployment.yaml and replace the given content.

   ```
   apiVersion: apps/v1
   kind: Deployment
   metadata:
     name: {{ .Release.Name }}
     labels:
       app: {{ .Release.Name  }}
       category: microgateway
   spec:
     replicas: {{ .Values.replicaCount }}
     selector:
       matchLabels:
         app: {{ .Release.Name }}
     template:
       metadata:
         labels:
           app: {{ .Release.Name }}
         annotations:
           prometheus.io/scrape: "{{ .Values.metrics.prometheus }}"
       spec:
         containers:
           - name: {{ .Release.Name }}
             image: "{{ .Values.image.repository }}:{{ .Values.image.tag }}"
             imagePullPolicy: {{ .Values.image.pullPolicy }}
             ports:
               - name: http
                 containerPort: {{ .Values.image.containerPort }}
   ```

```
            protocol: TCP
          livenessProbe:
            exec:
              command:
              - /bin/sh
              - -c
              - /opt/softwareag/Microgateway/files/k8s-lifenesscheck.sh
            initialDelaySeconds: 10
            periodSeconds: 10
            timeoutSeconds: 5
            failureThreshold: 3
          readinessProbe:
            exec:
              command:
              - /bin/sh
              - -c
              - /opt/softwareag/Microgateway/files/k8s-readinesscheck.sh
            initialDelaySeconds: 10
            periodSeconds: 10
            timeoutSeconds: 5
            failureThreshold: 3
        {{- if .Values.sidecarimage }}
        - name: "{{ .Values.sidecarimage.name }}"
          image: "{{ .Values.sidecarimage.repository }}:
                  {{ .Values.sidecarimage.tag }}"
          imagePullPolicy: {{ .Values.sidecarimage.pullPolicy }}
        {{- end }}
          resources:
            {{- toYaml .Values.resources | nindent 12 }}
      {{- with .Values.nodeSelector }}
      nodeSelector:
        {{- toYaml . | nindent 8 }}
      {{- end }}
    {{- with .Values.affinity }}
      affinity:
        {{- toYaml . | nindent 8 }}
    {{- end }}
    {{- with .Values.tolerations }}
      tolerations:
        {{- toYaml . | nindent 8 }}
    {{- end }}
```

4. Replace and adapt the values data.

   To achieve this, edit ./mg-helmchart/values.yaml and adapt values (for example, the Microgateway Docker image or if a sidecar image is to be used).

```
# This is a YAML-formatted file.
# Declare variables to be passed into your templates.

replicaCount: 1

# The Microgateway Docker image
image:
  repository: localhost:5000/mg-node-tours
  tag: latest
  containerPort: 7072
  pullPolicy: IfNotPresent

# The sidecar Docker image of the native service (if used, uncomment
```

```
and adapt values)
sidecarimage:
  repository: localhost:5000/node-tours
  tag: latest
  name: sidecar-node
  pullPolicy: IfNotPresent

metrics:
  prometheus: true

service:
  type: NodePort
  # type: LoadBalancer
  port: 7072

ingress:
  enabled: false
  paths: []
  hosts: []
  tls: []

resources: {}
  # We usually recommend not to specify default resources and to leave this
as a conscious   # choice for the user.

nodeSelector: {}

tolerations: []

affinity: {}
```

5. Check your environment.

```
helm install --dry-run --debug --name mg-test ./mg-helmchart
```

6. Deploy the chart to Kubernetes.

```
helm install --name mg-test ./mg-helmchart
```

7. Check the pods, deployment, and services for the resources created.

You can clear the entire deployment using the following command:

```
helm delete --purge mg-test
```

You have to restart the Kubernetes pods for the environment settings to be used.

Prometheus Microgateway Metrics

The Prometheus Microgateway metrics are provided at the server and API levels. This section provides details of both these metrics.

Server-Level Metrics

All server metrics have the metric type **guage** and a Prometheus label of **host**="*hostname*|*ipaddress*". The table lists the server-level Prometheus metrics for a Microgateway.

Prometheus metric name	Description
sag_microgateway_api_count	The number of deployed APIs.
sag_microgateway_http_requests	The total number of HTTP requests since the last statistics poll.
sag_microgateway_total_http_requests	The total number of HTTP requests since Microgateway startup.
sag_microgateway_max_memory_mb	The maximum memory in MB that the Microgateway JVM instance attempts to use. This indicates the upper memory threshold.
sag_microgateway_total_memory_mb	The total memory in MB allocated in the Microgateway JVM. This value may vary over time depending on the JVM memory handling.
sag_microgateway_free_memory_mb	The free memory in MB in the Microgateway JVM. This value may increase due to JVM's garbage collection.
sag_microgateway_used_memory_mb	The memory in MB that is occupied in the Microgateway JVM. This value is calculated as sag_microgateway_total_memory_mb - sag_microgateway_free_memory_mb.
sag_microgateway_threads	The current number of threads in use in a Microgateway instance.
sag_microgateway_max_threads	The maximum number of threads in a Microgateway JVM within a measured interval.
sag_microgateway_connections	The current number of internal TCP connections allocated by the Microgateway server. This number might increase during parallel multi-user access to the server.

Prometheus metric name	Description
sag_microgateway_max_connections	The maximum number of internal TCP connections the Microgateway server has allocated.
sag_microgateway_total_connections	The total number of internal TCP connections the Microgateway server might allocate. This indicates the upper threshold number of TCP connections allocated by the Microgateway server.

The metrics are stored as time series. Each sample consists of:

- a float64 value
- a millisecond-precision timestamp

Sample for the metric sag_microgateway_api_count

```
# HELP sag_microgateway_api_count The number of deployed APIs.
# TYPE sag_microgateway_api_count gauge
sag_microgateway_api_count{host="microgateway-10-4-cd6d47f85-z2rwq"}
2 1549887734839
```

Sample for the metric sag_microgateway_http_requests

```
# HELP sag_microgateway_http_requests The total number of HTTP requests since
last poll.
# TYPE sag_microgateway_http_requests gauge
sag_microgateway_http_requests{host="microgateway-10-4-cd6d47f85-z2rwq"}
100 1549887734839
```

Sample for the metric sag_microgateway_total_http_requests

```
# HELP sag_microgateway_total_http_requests The total number of HTTP requests
since Microgateway startup.
# TYPE sag_microgateway_total_http_requests gauge
sag_microgateway_total_http_requests{host="microgateway-10-4-cd6d47f85-z2rwq"}
1100 1549887734839
```

API-Level Metrics

All API-level metrics have the metric type **guage** and Prometheus labels of **host**="*hostname|ipaddress*" and **api**="*apiName*". The table lists the API-level Prometheus metrics for a Microgateway.

Prometheus metric name	Description
sag_microgateway_api_error_count	Total number of error events.

Prometheus metric name	Description
sag_microgateway_api_policy_violation_count	Total number of policy violations.
sag_microgateway_api_transaction_count	Total number of transactions.
sag_microgateway_api_average_response_time	The average time taken by the API to complete all invocations in the current interval. This is measured from the moment API Gateway receives the request until the moment it returns the response to the client.
sag_microgateway_api_fault_count	The number of failed invocations since the last statistics poll.
sag_microgateway_api_maximum_response_time	The maximum time taken by the API to complete an invocation since the last statistics poll.
sag_microgateway_api_minimum_response_time	The minimum time taken by the API to complete an invocation since the last statistics poll.
sag_microgateway_api_successful_request_count	The number of successful API invocations since the last statistics poll.
sag_microgateway_api_total_request_count	The total number of requests for each active API in API Gateway since the last statistics poll.

6 Policies

Policies Supported in Microgateway

This section provides information about the runtime policies supported in Microgateway. A policy can be enforced on an API to perform specific tasks, such as transport, authorization, routing of requests to target services, logging, , and error handling of data. For example, a policy could instruct Microgateway to perform any of the following tasks and prevent malicious attacks:

- Verify that the requests submitted to an API come from applications that are authenticated and authorized using only Basic Auth and API Key headers.
- Limits the number of invocations during a specified time interval for a particular API and for applications, and send alerts to API Gateway when these performance conditions are violated.
- Log the request and response messages.

Note:
These policies are configured in API Gateway and provisioned to Microgateway. Microgateway neglects the configurations that are not supported.

Policies are grouped into stages as per their usage. For example, the policies in the Identify and Access stage can be enforced on an API to specify the kind of identifiers that are used to identify the application and authorize it against all applications registered in Microgateway.

Microgateway supports the system-defined policies that are grouped into stages depending on their usage.

- Transport
- Identity and Access
- Request Processing
- Routing
- Traffic Monitoring
- Response Processing
- Error Handling

Transport

The policies in this stage specify the protocol to be used for an incoming request during communication between Microgateway and an application. The policy included in this stage is Enable HTTP/HTTPS.

Enable HTTP/HTTPS

This policy specifies the protocol to use for an incoming request to the API on Microgateway. If you have a native API that requires clients to communicate with the server using the HTTP and

HTTPS protocols, you can use the Enable HTTP or HTTPS policy. This policy allows you to bridge the transport protocols between the client and Microgateway.

For example, you have a native API that is exposed over HTTPS and an API that receives requests over HTTP. If you want to expose the API to the consumers of Microgateway through HTTP, then you configure the incoming protocol as HTTP.

The table lists the properties that are supported for this policy in Microgateway:

Parameter	Description
Protocol	Specifies the protocol (HTTP or HTTPS) to be used to accept and process the requests. The following properties when set specify the following: ■ **HTTP**. Microgateway accepts requests that are sent using the HTTP protocol. This is the default setting. ■ **HTTPS**. Microgateway accepts requests that are sent using the HTTPS protocol.

Identify and Access

The policies in this stage provide different ways of identifying and authorizing an application, and providing the required access rights for the application. Microgateway supports the following identify and access management policies:

- Authorize User
- Identify & Authorize

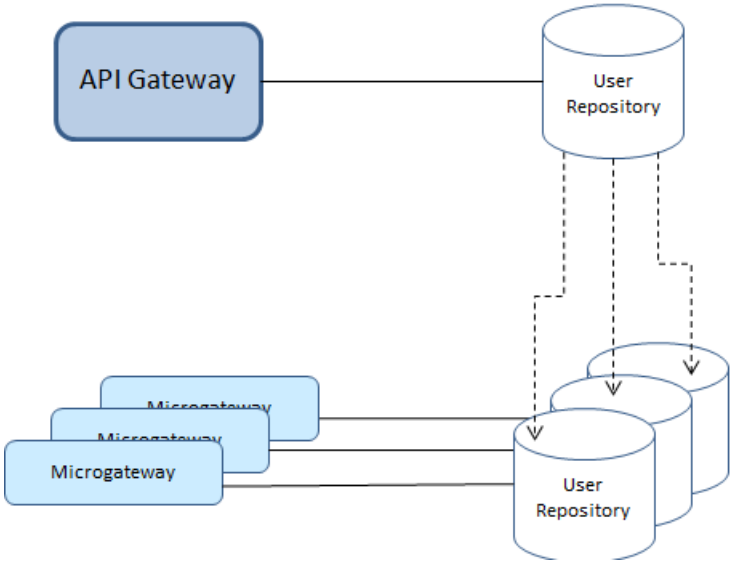
The Authorize User policy authorizes the application against a list of users and a list of groups registered in Microgateway.

The Identify and Authorize policy is used to authorize and allow the client applications to access APIs depending on the identification type specified to validate the client credentials.

User Identification to Support Identity and Access Management Policy

Microgateway supports authentication against users who are defined through API Gateway. The authentication is performed against a read-only user repository. This ensures that users can be authenticated even if Microgateway is not connected to any running API Gateway instance. The Microgateway user repository is populated by copying the API Gateway user repository (users.cnf) when provisioning a Microgateway.

The figure illustrates the Microgateway user repository being populated by copying the API Gateway user repository.



When you provision a Microgateway or start a Microgateway the users.cnf and related configurations are picked up from the location `IntegrationServer\instances\default\config\users.cnf` in the API Gateway installation directory.

The API Gateway installation directory can be specified using the Microgateway configuration parameter `apigw_dir`

The parameter can be specified either as a command line option or through the Microgateway configuration file.

The configuration parameter applies to the following Microgateway commands:

- createInstance
- createDockerFile

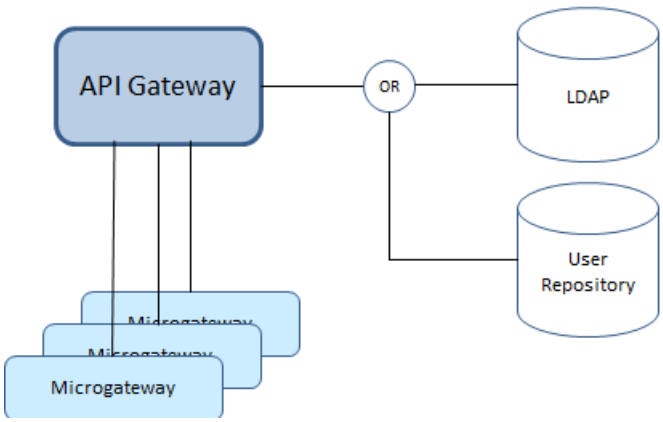
These commands create a copy of the users.cnf. The `apigw_dir` also affects the users.cnf lookup that happens during Microgateway start. The lookup procedure checks for local users.cnf (config/users.cnf). If there is no local users.cnf, then lookup users.cnf using apigw_dir. If apigw_dir or users.cnf is not found, then the startup fails.

Microgateway does not support user authentication by default. To activate user authentication you have to specify the parameter `user_auth = internal` when provisioning or starting a Microgateway.

If the user authentication is not activated, APIs with authentication policies are rejected. The users.cnf lookup is only performed when user authentication is activated.

Delegated authentication using API Gateway

Microgateway supports the delegated authentication to API Gateway. API Gateway performs the authentication against the configured LDAP or the user repository.



The delegated authentication is activated by setting the parameter `user_auth = delegated` . When the delegated authentication is activated, Microgateway talks to the API Gateway authentication API.

The authentication API exposes the resource: /rest/apigateway/authenticate

The resource exposes a POST method. The a user authentication is triggered through the request:

```
POST /rest/apigateway/authenticate
 {
    "user-id":"",
    "password:"",
    "domain":""
    "includeGroups":true
}
```

includeGroups is set to `true` only when Authorize User policy is configured. Based on the value, the groups details of the user are sent in response from API Gateway to Microgateway.

The password and user credentials are transferred in an unencrypted way. Therefore, the delegated authentication must happen through HTTPS.

On successful authentication the API returns a HTTP 200 response with user information and expiry information as follows:

```
HTTP 200 OK
 {
    "status": "Authenticated",
    "accessProfiles": [
        "Administrators",
        "Tyche"
    ],
   "groups": [
        "Administrators",
        "Tyche"
    ],
"user": "user1",
    "expires":"60"
}
```

The response provides information about the user, accessProfiles, groups, and the expiry interval. This tells the Microgateway for how long the delegated authentication result can be cached.

If the authentication fails the API returns a HTTP 401 response.

Authorize User

This policy authorizes incoming requests against the list of users in the users.cnf file in Microgateway, or the list of users in the users.cnf file in API Gateway, or against the list of users, groups or LDAP groups configured in API Gateway. This authentication happens depending on the setting `user_auth` configured in Microgateway and the authentication configuration in API Gateway. For details, see "User Identification to Support Identity and Access Management Policy" on page 67.

Use this policy together with an authentication policy (for example, Require HTTP Basic Authentication).

The table lists the parameters of this policy and how Microgateway applies them to authorize the incoming requests.

Parameter	Description
List of Users	Authorizes incoming requests against a list of users configured in Microgateway in the users.cnf file.
List of Groups	Authorizes incoming requests against a list of groups configured in this policy. This is performed by delegating the authorization to API Gateway to verify if the user identified from the request belongs to any groups configured in the policy. The delegation to API Gateway is achieved when the property `user_auth` is set as `delegated`. When the property `user_auth` is not set as `delegated`, the policy executes this condition as `false`. **Note:** You cannot use the **List of Groups** configuration option to authorize the LDAP groups.
List of Teams	Authorizes incoming requests against a list of teams configured in this policy. This is performed by delegating the authorization to API Gateway to verify if the user identified from the request belongs to any team configured in the policy. The delegation to API Gateway is achieved when the property `user_auth` is set as `delegated`. When the property `user_auth` is not set as `delegated`, the policy executes this condition as `false`.

Identify and Authorize

This policy authorizes and allows the client applications to access APIs depending on the identification type specified to validate the client credentials.

The table lists the parameters of this policy and how Microgateway applies them to validate the client credentials and authorize the client application to access the APIs.

Parameter	Description
Condition	The condition operator for the identification and authentication types specified for validating the client credentials. Select one of the following condition operators: ■ **AND**. Applies all the identification and authentication types. ■ **OR**. Applies one of the specified identification and authentication types.
Allow anonymous	Enable or disable the incoming requests to access the API without any restriction. When you enforce a security policy and select **Allow anonymous**, Microgateway allows all incoming requests to pass through to the native API. The successfully identified requests are grouped under the respective identified application, and all unidentified requests are grouped under a common application named *unknown*. Even when all the incoming requests are allowed to pass through without any restriction you can perform all application-specific actions, such as: ■ Viewing the runtime events for a particular application. ■ Monitoring the service level agreement for a few applications and sending an alert email based on some criteria like request count or availability. ■ Throttling the incoming requests from a particular application and not allowing the request from that application if the number of requests reach the configured hard limit within the configured interval.
Identification Type. Specifies the identification type. You can configure one or more of the following identification types.	
API Key	Denotes using the API key to identify and validate the authenticity of the client's identity against the registered applications for the specified API.

Parameter	Description
Hostname Address	Denotes using the host name to identify the client, extract the client's host name from the HTTP request header, and verify the client's identity against the specified applications in Microgateway. Configure one of the following **Application Lookup** conditions to verify the client's identity: ■ **Registered applications**. Verifies the client's hostname against a list of registered applications for the specified API. ■ **Global applications**. Verifies the client's hostname against a list of global applications. ■ **Global applications and DefaultApplication**. Identifies an application against a list of global applications. If the application is not identified, Microgateway sets this application to **DefaultApplication** and forwards the request to the native service.
HTTP Basic Authentication	Denotes using the Authorization request header to identify and authorize the client application against the specified applications in Microgateway that have the identifier `username`. Configure one of the following **Application Lookup** conditions to verify the client's identity: ■ **Registered applications**. Verifies the client's credentials against the list of registered applications for the specified API. ■ **Global applications**. Verifies the client's credentials against a list of global applications. ■ **Global applications and DefaultApplication**. Identifies an application against a list of global applications. If the application is not identified, Microgateway sets this application to **DefaultApplication** and forwards the request to the native service.
IP Address Range	Denotes using the IP address range to identify the client, extract the client's IP address from the HTTP request header, and verify the client's identity against the specified applications in Microgateway. Configure one of the following **Application Lookup** conditions to verify the client's identity: ■ **Registered applications**. Verifies the client's credentials against a list of registered applications for the specified API. ■ **Global applications**. Verifies the client's credentials against a list of global applications.

Parameter	Description
	■ **Global applications and DefaultApplication**. Identifies an application against a list of global applications. If the application is not identified, Microgateway sets this application to **DefaultApplication** and forwards the request to the native service.
OAuth2 Token	Denotes using the OAuth2 token to identify the client, extract the client's credentials from the OAuth2 token, and verify the client's identity against the specified list of applications in Microgateway. The tokens issued by API Gateway are validated by delegating them to the API Gateway instance. Configure the communication details that are used by Microgateway to introspect the API Gateway-issued OAuth2 tokens. For details on how to configure the communication details, see *webMethods API Gateway User's Guide*. **Note:** The client id and other parameters can be used for further processing using the request transformation policy.
JWT	Denotes using the JSON Web Token (JWT) to identify the client, extract the claims from the JWT and validate the client's claims, and verify the client's identity against the specified applications in Microgateway. Configure one of the following **Application Lookup** conditions to verify the client's identity: ■ **Registered applications**. Verifies the JWT against a list of registered applications for the specified API. ■ **Global applications**. Verifies the JWT against a list of global applications. ■ **Global applications and DefaultApplication**. Identifies an application against a list of global applications. If the application is not identified, Microgateway sets this application to **DefaultApplication** and forwards the request to the native service. **Note:** The claims in the JWT can be used for further processing using the request transformation policy.
OpenID Connect	Denotes using the OpenID (ID) token to identify the client, extract the client's credentials from the ID token, and verify the client's identity against the specified list of applications in Microgateway.

Parameter	Description
	You might have one of the following **Application Lookup** conditions to verify the client's identity: ■ **Registered applications**. Verifies the ID token against a list of registered applications for the specified API. ■ **Global applications**. Verifies the ID token against a list of global applications. ■ **Global applications and DefaultApplication**. Identifies an application against a list of global applications. If the application is not identified, Microgateway sets this application to **DefaultApplication** and forwards the request to the native service. **Note:** The client id and other parameters can be used for further processing using the request transformation policy.
SSL Certificate	Denotes using the SSL certificate to identify the client, extract the client's identity certificate, and verify the client's identity (certificate-based authentication) against the specified applications in Microgateway. Whenever both SSL certificate and custom header certificate are present, the identification is done using the SSL certificate. When the identification fails for the certificate obtained from SSL handshake, the identification using the certificate from the custom header is done. Microgateway extracts the client certificate that is used to identify the client from the request header. The certificate passed in the header should be Base64Encoded or the certificate chain passed in the header should be in the Base64Encoded (.pem) format. If the transport protocol is HTTP or HTTPS, Microgateway checks for the existence of a header and fetches the certificate from the certificate header. If the certificate is from the custom header, Microgateway does not check the validity of the certificate and identifies the application using the certificate. **Note:** Software AG recommends that an external entity validates the certificate sent in the custom header. During asset provisioning at Microgateway start up, the header name is included in the system-settings.yml file. You can customize

Parameter	Description
	the header name by modifying the value and including it in the user-defined custom settings YAML file. Configure one of the following **Application Lookup** conditions to verify the client's identity: ■ **Registered applications**. Verifies the client certificate against a list of registered applications for the specified API. ■ **Global applications**. Verifies the client certificate against a list of global applications. ■ **Global applications and DefaultApplication**. Identifies an application against a list of global applications. If the application is not identified, Microgateway sets this application to **DefaultApplication** and forwards the request to the native service.
Payload Element	Denotes using the payload identifier to identify the client, extract the custom authentication credentials supplied in the request represented using the payload identifier, and verify the client's identity against the specified applications in Microgateway. Configure one of the following **Application Lookup** conditions to verify the client's identity: ■ **Registered applications**. Verifies the client's payload identifier against a list of registered applications for the specified API. ■ **Global applications**. Verifies the client's identity credentials against a list of global applications. ■ **Global applications and DefaultApplication**. Identifies an application against a list of global applications. If the application is not identified, Microgateway sets this application to **DefaultApplication** and forwards the request to the native service. In the Payload identifier section, provide the following information: ■ **Expression type**: Specifies the type of expression that is used for identification. Use one of the following expression types: ■ **XPath**. Contains the following information: ■ **Payload Expression**. The payload expression to which to convert the specified expression type in the request. For example: `/name/id`

Parameter	Description
	■ **Namespace Prefix**. The namespace prefix of the payload expression to be validated. ■ **Namespace URI**. The namespace URI of the payload expression to be validated. ■ **JSONPath**. Specifies the JSONPath for the payload identification. For example: `$.name.id` ■ **Text**. Specifies the regular expression for the payload identification. You can have multiple payload identifiers. However, only one payload of each type is allowed. For example, you can have a maximum of three payload identifiers, each being of a different type.

Application Synchronization to support Identity and Access Management Policy

For Microgateway to support the Identity and Access Management (IAM) policies it is necessary that Microgateway has the recently updated applications from the API Gateway instance from where the applications are provisioned. Microgateway provides a mechanism to synchronize applications between API Gateway and Microgateway to support the IAM policy.

During API provisioning the applications are pulled from the API Gateway instance into Microgateway. After provisioning these applications in Microgateway, these applications have to be in synchronization with those in API Gateway, from where they are provisioned, so that any changes in the applications in the API Gateway instance is reflected in the Microgateway. This helps the IAM policy execution for an API in Microgateway validate with the latest applications instead of validating against the stale application data.

Application synchronization in Microgateway is achieved through a polling mechanism. To avoid the consumption of a considerable amount of memory and CPU, the API Provider provides certain configurations for polling the applications to minimize the memory and CPU utilization. Polling can be done for the following parameters:

- List of application ids
- All registered applications of the APIs in Microgateway
- All global applications

The property `applicationstoSync` is configured for polling the applications where you can specify the parameters in the format `registeredapplication`, `all`, or comma separated ids.

Considerations during application synchronization:

- Microgateway is provisioned with the application synchronization configuration before start up.

- Only one thread runs for synchronization.
- When the thread execution crashes, it starts again.
- A timestamp of the last synchronized application is maintained in the Microgateway instance so that the next polling would be for `applications updated > timestamp`.
- A property to specify the polling interval is added.
- A property to enable or disable synchronization is added.

Request Processing

These policies are used to specify how the request message from an application has to be transformed or pre-processed and configure the masking criteria for the data to be masked before it is submitted to the native API. This is required to protect the data and accommodate differences between the message content that an application is capable of submitting and the message content that a native API expects. The policies included in this stage are:

- Request Transformation
- Validate API Specification
- Data Masking

Request Transformation

This policy enables you to configure several transformations on the request messages from clients into a format required by the native API before it is submitted to the native API.

The transformations include Header, Query Parameter, Path Parameter transformation, HTTP Method transformation, and Payload transformation. You can configure conditions according to which the transformations are executed.

Microgateway supports the following parameter types that can be used to configure the transformation policy:

- request.headers
- request.query.QUERY_NAME (applicable only for REST API)
- request.path(applicable only for REST API)
- request.path.regex[Expression] (applicable only for REST API)
- request.httpmethod (applicable only for REST API)
- request.headers.HEADER_NAME
- request.authorization.clientId
- request.authorization.claims.CLAIM_NAME
- request.authorization.userName

Microgateway supports the following Query types that can be used to configure the transformation policy:

- xpath
- jsonPath
- regex

When you use these syntaxes to extract a value from the payload, the content-types applicable are:

- ${payload.jsonPath} - application/json, application/json/badgerfish
- ${payload.regex} - text/plain
- ${payload.xpath} - application/xml, text/xml, text/html.

The table lists the properties that are supported for this policy in Microgateway:

Parameter	Description
Condition	Conditions are used to specify when the policy has to be executed. You can have multiple conditions with logical operators. Available values are: ■ **AND**. Microgateway transforms the requests that comply with all the configured conditions. ■ **OR**. This is selected by default. Microgateway transforms the requests that comply with at least one configured condition. Various conditions you can define are: ■ **Variable**: Specifies the variable type with a syntax as follows: ■ `${PARAMTYPE}` : This is applicable for variables of string type - path, payload, httpMethod. For example: ${request.path} ■ `${PARAMTYPE.paramName}` : This is applicable for map types - query and headers and also applicable for path. For example: `${request.query.var1}`, `${request.header.Content-Type}`, `${request.path.name}` ■ `${PARAMTYPE.QUERYTYPE[queryValue]}` : This syntax is applicable for payload and path. regex can be applied on path while XPath, JSONPath and regex can be applied on payload. For example: `${request.payload.xpath[//ns:emp/ns:empName]}` where //ns:emp/ns:empName is the xpath to be applied on the payload if contentType is application/xml

Parameter	Description
	`${request.payload.jsonPath[$.cardDetails.number]}` where $.cardDetails.number is the jsonPath to be applied on the payload if contentType is application/json `${request.payload.regex[[0-9]+]}` where [0-9]+ is the regex to be applied on the payload if contentType is text/plain ■ If you want Microgateway to apply xpath, jsonPath, regex based on Content-Type of the payload, use the following common syntax: `${PARAMTYPE.QUERYTYPE[queryValue] || PARAMTYPE.QUERYTYPE2[queryValue2] || ...}` For example: `${request.payload.xpath[//ns:emp/ns:empName] || request.payload.jsonPath[$.cardDetails.number]}` This applies xpath for application/xml and jsonPath for application/json `${request.payload.xpath[//ns:emp/ns:empName] || request.payload.jsonPath[$.cardDetails.number] || request.payload.regex[[0-9]+]}` This applies xpath for application/xml, jsonPath for application/json, and regex for text/plain. ■ **Operator**: Specifies the operator to use to relate variable and the value provided. You can select one of the following: ■ **Equals** ■ **Equals ignore case** ■ **Not equals** ■ **Contains** ■ **Exists** ■ **Value**: Specifies a value with a syntax as follows: ■ `PLAIN VALUE`, for example, application/json ■ `${PARAMTYPE.paramName}` ■ `${PARAMTYPE.QUERYTYPE[queryValue]}` ■ `${PARAMTYPE.QUERYTYPE[queryValue] || PARAMTYPE.QUERYTYPE2[queryValue2] || ...}`
Transformation Configuration: Specifies various transformations to be configured.	
Header/Query/Path Transformation	Specifies the Header, Query or path transformation to be configured for incoming requests.

Parameter	Description
	Various configurations you can define are: ■ **Variable**. Specifies the variable type with a syntax as follows: ■ `${PARAMTYPE}`. This is applicable for variables of string type - path, payload, httpMethod. For example: ${request.path} ■ `${PARAMTYPE.paramName}`. This is applicable for map types - query and headers and also applicable for path. For example: `${request.query.var1}`, `${request.header.Content-Type}`, `${request.path.name}` ■ `${PARAMTYPE.QUERYTYPE[queryValue]}`. This syntax is applicable for payload and path. regex can be applied on path while XPath, JSONPath and regex can be applied on payload. For example: `${request.payload.xpath[//ns:emp/ns:empName]}` where //ns:emp/ns:empName is the xpath to be applied on the payload if contentType is application/xml `${request.payload.jsonPath[$.cardDetails.number]}` where $.cardDetails.number is the jsonPath to be applied on the payload if contentType is application/xml `${request.payload.regex[[0-9]+]}` where [0-9]+ is the regex to be applied on the payload if contentType is anything ■ If you want Microgateway to apply xpath, jsonPath, regex based on Content-Type of the payload, use the following common syntax: `${PARAMTYPE.QUERYTYPE[queryValue] || PARAMTYPE.QUERYTYPE2[queryValue2] || ...}` For example: `${request.payload.xpath[//ns:emp/ns:empName] || request.payload.jsonPath[$.cardDetails.number]}` This applies xpath for application/xml and jsonPath for application/json `${request.payload.xpath[//ns:emp/ns:empName] || request.payload.jsonPath[$.cardDetails.number] || request.payload.regex[[0-9]+]}` This applies xpath for application/xml, jsonPath for application/json, and regex for text/xml. ■ **Value**. Specifies a value with a syntax as follows: ■ `PLAIN VALUE`, for example, application/json ■ `${PARAMTYPE.paramName}` ■ `${PARAMTYPE.QUERYTYPE[queryValue]}`

Parameter	Description
	■ `${PARAMTYPE.QUERYTYPE[queryValue] || PARAMTYPE.QUERYTYPE2[queryValue2] || ...}`
Method transformation for REST API	Specifies the method transformation to be configured for incoming requests. You can have any of the HTTP Method listed: ■ **GET** ■ **POST** ■ **PUT** ■ **DELETE** ■ **HEAD** ■ **CUSTOM** **Note:** ■ When **CUSTOM** is selected, the HTTP method in incoming request is sent to the native service. When other methods are selected, the selected method is used in the request sent to the native service. ■ Only Method Transformation happens when configured, but you have to take care of adding payload during transformations involving method change like GET to POST, and so on.
Payload Transformation	Specifies the payload transformation to be configured for incoming requests. Specifies the following information: ■ An xslt document that contains the following information: ■ **XSLT file**. Specifies the XSLT file used to transform the request messages as required. ■ **Feature Name**. Specifies the name of the XSLT feature. ■ **Feature value**. Specifies the value of the XSLT feature. You can have multiple XSLT features and xslt documents.
Transformation Metadata: Specifies the metadata for transformation of the incoming requests. For example, the namespaces configured in this section can be used when you provide the syntax for XPath ${request.payload.xpath} For example: ${request.payload.xpath[//ns:emp/ns:empName]}	
Namespace	Specifies the namespace information to be configured for transformation. Provide the following information:

Parameter	Description
	■ **Namespace Prefix**. The namespace prefix of the payload expression to be validated. ■ **Namespace URI**. The namespace URI of the payload expression to be validated. **Note:** You can have multiple namespace prefix and URI.

Validate API Specification

This policy validates the incoming request against various components of an API specification such as schema, query parameters, path parameters, content-types, and HTTP headers.

The various components of an API specification are referenced as follows:

- The schema for REST APIs can be imported through a swagger or a RAML file, or a file you upload. The schema is available as part of the API definition.
- The query parameters, path parameters, and content-types are available as part of the API definition.
- The HTTP headers are specified in the Validate API Specification policy page.

The requests sent to the API by a client application must conform with the structure or format expected by the API. The incoming requests are validated against the API specifications selected in this policy to conform to the structure or format expected by the API.

The various components of an API specification that can be validated are:

- **Schema**

  The incoming requests are validated against the schema provided in the API definition. The schema defines the elements and attributes in the request payload and specifies the data types of these elements to ensure that only appropriate data is allowed through to the API.

  For a REST API, the schema validation execution depends on the content-type header in the request. The default content-type header and schema validation type mapping is as follows:

Content-type header	Schema validation type
application/json application/json/badgerfish	JSON schema
application/xml text/xml text/html	XML schema

Content-type header	Schema validation type
text/plain	Regular expression

Note:
If the schema specified for a content-type header in the API definition is not as specified in the mapping table, then the behavior is as follows:

- If there is no schema specified for a content-type header in a REST API, the validation is skipped.
- If application/json is mapped to XML schema in the API definition, then the JSON content in the request is validated against XML schema to provide a backward compatibility support for APIs migrated from API Gateway 10.1 version.
- If only XML schema mappings exist for any of the content-types, the payload is converted into XML and validated against all the XML schemas. If the payload is valid against one of the schemas, the validation is successful.
- If the payload cannot be converted to XML format, the validation is not performed and the request is allowed to reach the native API.

- **Query Parameters**

  The query parameters in the incoming requests are validated against the corresponding query parameter data type specified in the API definition.

- **Path Parameters**

  The path parameters in the incoming requests are validated against the corresponding path parameter data type specified in the API definition.

- **Cookie Parameters**

  The cookie parameters in the incoming requests are validated against the corresponding cookie parameter data type specified in the API definition.

- **Content-types**

  The content-types in the incoming requests are validated against the corresponding content-types specified in the API definition.

- **HTTP Headers**

  The HTTP headers in the incoming requests are validated against the corresponding HTTP headers specified in this policy to conform to the HTTP headers expected by the API.

The API requests that fail the specification validation are considered as policy violations. You can view such policy violation events in the API Gateway dashboard.

The table lists the parameters of this policy and how they are applied to validate API requests.

Parameter	Description
Schema	Validates the request payload against the appropriate schema (based on the content-type header in the request).

Parameter	Description
	Additional features for XML schema validation are: ■ **Feature name**. The name for the schema configuration. For example: TOLERATE_DUPLICATES, NAMESPACE_GROWTH ■ **Feature value**. Specifies whether the feature value is **True** or **False**.
Query Parameters	Validates the query parameters in the incoming request against the query parameters defined in the API definition. **Note:** Except **File** data type, all the other data types, **String**, **Date**, **Date time**, **Integer**, **Double**, and **Boolean**, are supported for the available types.
Path Parameters	Validates the path parameters in the incoming request against the path parameters defined in the API definition. **Note:** Except **File** data type, all the other data types, **String**, **Date**, **Date time**, **Integer**, **Double**, and **Boolean**, are supported for the available types.
Cookie Parameters	Validates the cookie parameters in the incoming request against the cookie parameters defined in the API definition. **Note:** Except **File** data type, all the other data types, **String**, **Date**, **Date time**, **Integer**, **Double**, and **Boolean**, are supported for the available types.
Content-types	Validates the content-types in the incoming request against the content-types defined in the API definition.
HTTP Headers	Validates the HTTP header parameters in the incoming request against the HTTP headers defined in this policy. Various conditions and additional information you can define are: ■ **Condition**. Specifies the logical operator to use to validate multiple HTTP headers in the incoming API requests. Available values are: ■ **AND**. Microgateway accepts only the requests that contain all configured HTTP headers. ■ **OR**. This is selected by default. Microgateway accepts requests that contain at least one configured HTTP header. ■ **HTTP Header Key**. Specifies a key that must be passed through the HTTP header of the incoming API requests.

Parameter	Description
	■ **Header Value**. *Optional.* Specifies the corresponding key value that could be passed through the HTTP header of the incoming API requests. The **Header Value** field type accepts string and regular expression (regex).

Data Masking

Data masking is a technique whereby sensitive data is masked in some way to render it safe and protect the actual data while having a functional substitute for occasions when the real data is not required.

This policy is used to mask sensitive data at the application level. At the application level you must have an Identify and Access policy configured to identify the application for which the masking is applied. If no application is specified then it will be applied for all the other requests. Fields can be masked or filtered in the request messages received. You can configure the masking criteria as required for the XPath, JSONPath, and Regex expressions based on the content-type. This policy can also be applied at the API scope level.

The table lists the content-type and masking criteria mapping.

Content-type	Masking Criteria
application/xml text/xml text/html	XPath
application/json application/json/badgerfish	JSONPath
text/plain	Regex

The table lists the masking criteria properties configured to mask the data in the request messages received:

Parameter	Description
Consumer Applications	*Optional.* Specifies the applications for which the masking criterion has to be applied. For example: If there is a DataMasking(DM1) criteria created for application1 a second DataMasking(DM2) for application2 and a third DataMasking(DM3) with out any application, then for a request that comes from consumer1 the masking criteria DM1 is applied, for a request

Parameter	Description
	that comes from consumer2 DM2 is applied. If a request comes with out any application or from any other application except application1 and application2 DM3 is applied.
XPath: Specifies the masking criteria for XPath expressions in the request messages.	
Masking Criteria	Specifies the masking criteria that contains the following information: ■ **Masking Type**. Specifies the type of masking required. You can have either **Mask** or **Filter**. **Mask** replaces the value with the given value (the default value being ********) and **Filter** removes the field completely. ■ **Query expression**. Specifies the query expression that has to be masked or filtered. For example: /pet/details/status, /user/details/card/ccnumber. ■ **Mask Value**. This is available if masking type selected is **Mask**. Specifies the mask value. For example: sold, any mask value #####. **Note:** You can have multiple masking criteria. ■ **Namespace**. Specifies the following Namespace information: ■ **Namespace Prefix**. The namespace prefix of the payload expression to be validated. ■ **Namespace URI**. The namespace URI of the payload expression to be validated **Note:** You can have multiple namespace prefix and URI specified.
JSONPath: Specifies the masking criteria for JSONPath expressions in the request messages.	
Masking Criteria	Specifies the masking criteria that contains the following information:: ■ **Masking Type**. Specifies the type of masking required. You can have either **Mask** or **Filter**. **Mask** replaces the value with the given value (the default value being ********) and **Filter** removes the field completely. ■ **Query expression**. Specifies the query expression that has to be masked or filtered. For example: $.pet.details.status ■ **Mask Value**. This is available if masking type selected is **Mask**. Specifies the mask value. For example: sold

Parameter	Description
Regex: Specifies the masking criteria for regular expressions in the request messages.	

Masking Criteria	Specifies the masking criteria that contains the following information::
	- **Masking Type**. Specifies the type of masking required. You can have either **Mask** or **Filter**. **Mask** replaces the value with the given value (the default value being ********) and **Filter** removes the field completely.
	  - **Query expression**. Specifies the query expression that has to be masked or filtered. For example: [0-9]+
	  - **Mask Value**. This is available if masking type selected is **Mask**. Specifies the mask value. For example: ########

Apply for transaction Logging	Select this option to apply masking criteria for transactional logs.
Apply for payload	Select this option to apply masking criteria for payload in the incoming request.

Routing

The policies in this stage enforce routing of requests to target APIs based on the rules you can define to route the requests and manage their respective redirections according to the initial request path. The policies included in this stage are

- Straight-through routing
- Content-based routing
- Conditional routing
- Outbound Auth - Transport

Straight Through Routing

When you select the Straight Through routing protocol, the API routes the requests directly to the native service endpoint you specify. If your entry protocol is HTTP or HTTPS, you can select the Straight Through routing policy.

The table lists the properties that are supported for this policy in Microgateway:

Parameter	Value
Endpoint URI	Specifies the URI of the native API endpoint to route the request to in case all routing rules evaluate to False. Service registries that

Parameter	Value
	have been added to the Microgateway instance are also included in the list. **Note:** Only simple aliases, endpoint aliases, and service registry aliases are supported in Microgateway. If you choose a service registry, Microgateway sends a request to the service registry to discover the IP address and port at which the native service is running. Microgateway replaces the service registry alias in the **Endpoint URI** with the IP address and port returned by the service registry. For example, if your service is hosted at the URL: `http://host:port/abc/`, you need to configure the Endpoint URI as: `http://${ServiceRegistryName}/abc/`.
HTTP Method	Specifies the available routing methods: **GET**, **POST**, **PUT**, **DELETE**, and **CUSTOM** (default). When **CUSTOM** is selected, the HTTP method in the incoming request is sent to the native service. When other methods are selected, the selected method is used in the request sent to the native service. **Note:** Software AG recommends to use **Request Transformation > Method Transformation** to achieve this as other transformations can also be done under the same policy.
HTTP Connection Timeout (seconds)	Specifies the time interval (in seconds) after which a connection attempt times out. The precedence of the Connection Timeout configuration is as follows: 1. If you specify a value for the **Connection timeout** field in routing endpoint alias, then the **Connection timeout** value specified in the Endpoint alias section takes precedence over the timeout values defined at the API level and the global level. 2. If you specify a value 0 for the **Connection timeout** field in routing endpoint alias, then Microgateway uses the value specified in the **Connection timeout** field in the routing protocol processing step of an API. The Read Timeout value specified at an API level takes precedence over the global configuration. 3. If you specify a value 0 or do not specify a value for the **Connection timeout** field in the routing protocol processing

Parameter	Value
	step at the API level or specify a value 0 at an alias level, then Microgateway uses the value specified in this `pg.endpoint.connectionTimeout` property. 4. If you do not specify any value for `pg.endpoint.connectionTimeout`, then Microgateway uses the default value of 30 seconds.
Read Timeout (seconds)	Specifies the time interval (in seconds) after which a socket read attempt times out. The precedence of the Read Timeout configuration is as follows: 1. If you specify a value for the **Read timeout** field in routing endpoint alias, then the **Read timeout** value specified in the Endpoint alias section takes precedence over the timeout values defined at the API level and the global level. 2. If you specify a value 0 for the **Read timeout** field in routing endpoint alias, then Microgateway uses the value specified in the **Read Timeout** field in the routing protocol processing step of an API. The Read Timeout value specified at an API level takes precedence over the global configuration. 3. If you specify a value 0 or do not specify a value for the **Read timeout** field in the routing protocol processing step at the API level or specify a value 0 at an alias level, then Microgateway uses the value specified in this `pg.endpoint.readTimeout` property. 4. If you do not specify any value for `pg.endpoint.readTimeout`, then Microgateway uses the default value of 30 seconds.
SSL configuration - Specifies values to enable SSL client authentication that Microgateway uses to authenticate incoming requests for the native API.	
Keystore Alias	Specifies the keystore alias that is present in the system-settings.yml. This value (along with the value of Client Certificate Alias) is used for performing SSL client authentication.
Key Alias	Specifies the alias for the private key, which must be stored in the keystore specified by the keystore alias.
Service Registry Configuration	
Service Discovery Endpoint Parameter	Values required for constructing the discovery service URI. - **Parameter**. An alias that you have included in the discovery service URI while adding the service registry to Microgateway.

Parameter	Value
	■ **Value**. A value for the path parameter. The alias specified in Path Parameter is substituted with this value when invoking the discovery service. For example: if the service registry configuration of the service registry that you have selected in **Endpoint URI** has **Service discovery path** set to `/catalog/service/{serviceName}` (and the `{serviceName}` alias is intended for passing the service name), you must provide `{serviceName}` as **Parameter** and the name of the service as **Value**.

Content-based Routing

If you have a native API that is hosted at two or more endpoints, you can use the Content-based routing protocol to route specific types of messages to specific endpoints. You can route messages to different endpoints based on specific values that appear in the request message. You might use this capability, for example, to determine which operation the consuming application has requested, and route requests for complex operations to an endpoint on a fast machine. For example, if your entry protocol is HTTP or HTTPS, you can select the Content-based routing. The requests are routed according to the content-based routing rules you create. You may specify how to authenticate requests.

The table lists the properties that are supported for this policy in Microgateway:

Parameter	Description
Default Route To: Specifies the URLs of two or more native services in a pool to which the requests are routed.	
Endpoint URI	Specifies the URI of the native API endpoint to route the request to in case all routing rules evaluate to False. Service registries that have been added to the Microgateway instance are also included in the list. **Note:** Only simple aliases, endpoint aliases, and service registry aliases are supported in Microgateway. If you choose a service registry, Microgateway sends a request to the service registry to discover the IP address and port at which the native service is running. Microgateway replaces the service registry alias in the **Endpoint URI** with the IP address and port returned by the service registry. For example, if your service is hosted at the URL: `http://host:port/abc/`, you need to configure the Endpoint URI as: `http://${ServiceRegistryName}/abc/`.

Parameter	Description
HTTP Method	Specifies the available routing methods: **GET**, **POST**, **PUT**, **DELETE**, and **CUSTOM** (default). When **CUSTOM** is selected, the HTTP method in the incoming request is sent to the native service. When other methods are selected, the selected method is used in the request sent to the native service. **Note:** Software AG recommends to use **Request Transformation > Method Transformation** to achieve this as other transformations can also be done under the same policy.
HTTP Connection Timeout (seconds)	Specifies the time interval (in seconds) after which a connection attempt times out. The precedence of the Connection Timeout configuration is as follows: 1. If you specify a value for the **Connection timeout** field in routing endpoint alias, then the **Connection timeout** value specified in the Endpoint alias section takes precedence over the timeout values defined at the API level and the global level. 2. If you specify a value 0 for the **Connection timeout** field in routing endpoint alias, then Microgateway uses the value specified in the **Connection timeout** field in the routing protocol processing step of an API. The Read Timeout value specified at an API level takes precedence over the global configuration. 3. If you specify a value 0 or do not specify a value for the **Connection timeout** field in the routing protocol processing step at the API level or specify a value 0 at an alias level, then Microgateway uses the value specified in this `pg.endpoint.connectionTimeout` property. 4. If you do not specify any value for `pg.endpoint.connectionTimeout`, then Microgateway uses the default value of 30 seconds.
Read Timeout (seconds)	Specifies the time interval (in seconds) after which a socket read attempt times out. The precedence of the Read Timeout configuration is as follows: 1. If you specify a value for the **Read timeout** field in routing endpoint alias, then the **Read timeout** value specified in the

Parameter	Description
	Endpoint alias section takes precedence over the timeout values defined at the API level and the global level. 2. If you specify a value 0 for the **Read timeout** field in routing endpoint alias, then Microgateway uses the value specified in the **Read Timeout** field in the routing protocol processing step of an API. The Read Timeout value specified at an API level takes precedence over the global configuration. 3. If you specify a value 0 or do not specify a value for the **Read timeout** field in the routing protocol processing step at the API level or specify a value 0 at an alias level, then Microgateway uses the value specified in this `pg.endpoint.readTimeout` property. 4. If you do not specify any value for `pg.endpoint.readTimeout`, then Microgateway uses the default value of 30 seconds.
SSL Configuration. Specifies values to enable SSL client authentication that Microgateway uses to authenticate incoming requests for the native API.	
Keystore Alias	Specifies the keystore alias that is present in the system-settings.yml. This value (along with the value of Client Certificate Alias) is used for performing SSL client authentication.
Key Alias	Specifies the alias for the private key, which must be stored in the keystore specified by the keystore alias.
Service Registry Configuration	
Service Discovery Endpoint Parameter	Values required for constructing the discovery service URI. ■ **Parameter**. An alias that you have included in the discovery service URI while adding the service registry to Microgateway. ■ **Value**. A value for the path parameter. The alias specified in Path Parameter is substituted with this value when invoking the discovery service. For example: if the service registry configuration of the service registry that you have selected in **Endpoint URI** has **Service discovery path** set to `/catalog/service/{serviceName}` (and the `{serviceName}` alias is intended for passing the service name), you must provide `{serviceName}` as **Parameter** and the name of the service as **Value**.
Rule: Defines the routing decisions based on one of the following routing options.	
Payload Identifier	Specifies using the payload identifier to identify the client, extract the custom authentication credentials supplied in the request

Parameter	Description
	represented using the payload identifier, and verify the client's identity. The Payload identifier includes the following information. ■ **Expression type**. Specifies the type of expression, which is used for identification. You can have one the following expression type: ■ **XPath**. Provide the following information: ■ **Payload Expression**. Specifies the payload expression that the specified XPath expression type in the request has to be converted to. For example: /name/id ■ **Namespace Prefix**. The namespace prefix of the payload expression to be validated. ■ **Namespace URI**. The namespace URI of the payload expression to be validated. ■ **JSONPath**. Provide the **Payload Expression** that specifies the payload expression that the specified JSONPath expression type in the request has to be converted to. For example: $.name.id ■ **Text**. Provide the **Payload Expression** that specifies the payload expression that the specified Text expression type in the request has to be converted to. For example: any valid regular expression. You can have multiple payload identifiers as required. **Note:** Only one payload identifier of each type is allowed. For example, you can have a maximum of three payload identifiers, each being of a different type.
Route To. Specifies the Endpoint URI of native APIs in a pool to which the requests are routed.	
Endpoint URI	Specifies the URI of the native API endpoint to route the request to. You can use service registries in a similar manner as described in the main **Endpoint URI** above.
HTTP Method	Specifies the available routing methods: **GET**, **POST**, **PUT**, **DELETE**, and **CUSTOM** (default). When **CUSTOM** is selected, the HTTP method in the incoming request is sent to the native service. When other methods are

Parameter	Description
	selected, the selected method is used in the request sent to the native service.
HTTP Connection Timeout (seconds)	Specifies the time interval (in seconds) after which a connection attempt times out. The precedence of the Connection Timeout configuration is as follows: 1. If you specify a value for the **Connection timeout** field in routing endpoint alias, then the **Connection timeout** value specified in the Endpoint alias section takes precedence over the timeout values defined at the API level and the global level. 2. If you specify a value 0 for the **Connection timeout** field in routing endpoint alias, then Microgateway uses the value specified in the **Connection timeout** field in the routing protocol processing step of an API. The Read Timeout value specified at an API level takes precedence over the global configuration. 3. If you specify a value 0 or do not specify a value for the **Connection timeout** field in the routing protocol processing step at the API level or specify a value 0 at an alias level, then Microgateway uses the value specified in this `pg.endpoint.connectionTimeout` property. 4. If you do not specify any value for `pg.endpoint.connectionTimeout`, then Microgateway uses the default value of 30 seconds.
Read Timeout (seconds)	Specifies the time interval (in seconds) after which a socket read attempt times out. The precedence of the Read Timeout configuration is as follows: 1. If you specify a value for the **Read timeout** field in routing endpoint alias, then the **Read timeout** value specified in the Endpoint alias section takes precedence over the timeout values defined at the API level and the global level. 2. If you specify a value 0 for the **Read timeout** field in routing endpoint alias, then Microgateway uses the value specified in the **Read Timeout** field in the routing protocol processing step of an API. The Read Timeout value specified at an API level takes precedence over the global configuration. 3. If you specify a value 0 or do not specify a value for the **Read timeout** field in the routing protocol processing step at the API

Parameter	Description
	level or specify a value 0 at an alias level, then Microgateway uses the value specified in this `pg.endpoint.readTimeout` property. 4. If you do not specify any value for `pg.endpoint.readTimeout`, then Microgateway uses the default value of 30 seconds.
SSL Configuration	Specifies values to enable SSL client authentication that Microgateway uses to authenticate incoming requests for the native API. Provide the following information: ■ **Keystore Alias**. Specifies the keystore alias that is present in the system-settings.yml. This value (along with the value of Client Certificate Alias) is used for performing SSL client authentication. ■ **Key Alias**. Specifies the alias for the private key, which must be stored in the keystore specified by the keystore alias.
Service Registry Configuration	
Service Discovery Endpoint Parameter	Values required for constructing the discovery service URI. ■ **Parameter**. An alias that you have included in the discovery service URI while adding the service registry to Microgateway. ■ **Value**. A value for the path parameter. The alias specified in Path Parameter is substituted with this value when invoking the discovery service. For example: if the service registry configuration of the service registry that you have selected in **Endpoint URI** has **Service discovery path** set to `/catalog/service/{serviceName}` (and the `{serviceName}` alias is intended for passing the service name), you must provide `{serviceName}` as **Parameter** and the name of the service as **Value**.

Conditional Routing

If you have a native API that is hosted at two or more endpoints, you can use the conditional protocol to route specific types of messages to specific endpoints. The requests are routed according to the conditional routing rules you create. For example, if your entry protocol is HTTP or HTTPS, you can select conditional routing. A routing rule specifies where requests should be routed to, and the criteria by which they should be routed to the specified URL. In addition, you may specify how to authenticate the incoming requests.

The table lists the properties that are supported for this policy in Microgateway:

Parameter	Description
Default Route To. Specifies the URLs of two or more native services in a pool to which the requests are routed.	
Endpoint URI	Specifies the URI of the native API endpoint to route the request to in case all routing rules evaluate to False. Service registries that have been added to the Microgateway instance are also included in the list. **Note:** Only simple aliases, endpoint aliases, and service registry aliases are supported in Microgateway. If you choose a service registry, Microgateway sends a request to the service registry to discover the IP address and port at which the native service is running. Microgateway replaces the service registry alias in the **Endpoint URI** with the IP address and port returned by the service registry. For example, if your service is hosted at the URL: `http://host:port/abc/`, you need to configure the Endpoint URI as: `http://${ServiceRegistryName}/abc/`.
HTTP Method	Specifies the available routing methods: **GET**, **POST**, **PUT**, **DELETE**, and **CUSTOM** (default). When **CUSTOM** is selected, the HTTP method in the incoming request is sent to the native service. When other methods are selected, the selected method is used in the request sent to the native service. **Note:** Software AG recommends to use **Request Transformation > Method Transformation** to achieve this as other transformations can also be done under the same policy.
HTTP Connection Timeout (seconds)	Specifies the time interval (in seconds) after which a connection attempt times out. The precedence of the Connection Timeout configuration is as follows: 1. If you specify a value for the **Connection timeout** field in routing endpoint alias, then the **Connection timeout** value specified in the Endpoint alias section takes precedence over the timeout values defined at the API level and the global level. 2. If you specify a value 0 for the **Connection timeout** field in routing endpoint alias, then Microgateway uses the value specified in the **Connection timeout** field in the routing protocol

Parameter	Description
	processing step of an API. The Read Timeout value specified at an API level takes precedence over the global configuration. 3. If you specify a value 0 or do not specify a value for the **Connection timeout** field in the routing protocol processing step at the API level or specify a value 0 at an alias level, then Microgateway uses the value specified in this `pg.endpoint.connectionTimeout` property. 4. If you do not specify any value for `pg.endpoint.connectionTimeout`, then Microgateway uses the default value of 30 seconds.
Read Timeout (seconds)	Specifies the time interval (in seconds) after which a socket read attempt times out. The precedence of the Read Timeout configuration is as follows: 1. If you specify a value for the **Read timeout** field in routing endpoint alias, then the **Read timeout** value specified in the Endpoint alias section takes precedence over the timeout values defined at the API level and the global level. 2. If you specify a value 0 for the **Read timeout** field in routing endpoint alias, then Microgateway uses the value specified in the **Read Timeout** field in the routing protocol processing step of an API. The Read Timeout value specified at an API level takes precedence over the global configuration. 3. If you specify a value 0 or do not specify a value for the **Read timeout** field in the routing protocol processing step at the API level or specify a value 0 at an alias level, then Microgateway uses the value specified in this `pg.endpoint.readTimeout` property. 4. If you do not specify any value for `pg.endpoint.readTimeout`, then Microgateway uses the default value of 30 seconds.
SSL Configuration. Specifies values to enable SSL client authentication that Microgateway uses to authenticate incoming requests for the native API.	
Keystore Alias	Specifies the keystore alias that is present in the system-settings.yml. This value (along with the value of Client Certificate Alias) is used for performing SSL client authentication.
Key Alias	Specifies the alias for the private key, which must be stored in the keystore specified by the keystore alias.
Service Registry Configuration	

Parameter	Description
Service Discovery Endpoint Parameter	Values required for constructing the discovery service URI. ■ **Parameter**: An alias that you have included in the discovery service URI while adding the service registry to Microgateway. ■ **Value**: A value for the path parameter. The alias specified in Path Parameter is substituted with this value when invoking the discovery service. For example: if the service registry configuration of the service registry that you have selected in **Endpoint URI** has **Service discovery path** set to `/catalog/service/{serviceName}` (and the `{serviceName}` alias is intended for passing the service name), you must provide `{serviceName}` as **Parameter** and the name of the service as **Value**.
Rule. Defines the routing decisions based on one of the following routing options.	
Name	Specifies the name for the rule.
Condition Operator	Specifies the condition operator to be used. The operators can be one of the following operators: ■ **OR**. Specifies that one of the set conditions should be applied. ■ **AND**. Specifies all the set conditions should be applied.
Condition	Specify the context variables for processing client requests.
Variable	Select any of the following variables: ■ **Consumer**. Specifies the name of the consumer application in the text box. ■ **Variable Value**. Specifies a value in the Variable Value text box. ■ **Date** ■ **Operator**. Specifies one of the following operators: **After** or **Before** . ■ **Variable Value**. Specifies a date value. ■ **IPV4**. Specifies that IP version to be IPV4. ■ **From IP**.Type an IP address range. ■ **To IP**. Type an IP address range. ■ **IPV6**. Specifies that IP version to be IPV6.

Parameter	Description
	■ **From IP**. Type an IP address range. ■ **To IP**. Type an IP address range. ■ **Predefined Context Variable** ■ **Predefined Context**. Specifies a predefined context. ■ **Operator**. Select one of the following operators: **Equal To** or **Not Equal To**. ■ **Variable Value**. Specifies a value for the selected predefined context. ■ **Time** ■ **Operator**. Specifies one of the following operators: **After** or **Before**. ■ **Hours**. Specifies a time value in hours. ■ **Minutes**. Specifies a time value in minutes.
Route To. Specifies the endpoint URI of native services in a pool to which the requests are routed.	
Endpoint URI	Specifies the URI of the native API endpoint to route the request to. You can use service registries in a similar manner as described in the main **Endpoint URI** above.
HTTP Method	Specifies the available routing methods: **GET**, **POST**, **PUT**, **DELETE**, and **CUSTOM** (default). When **CUSTOM** is selected, the HTTP method in the incoming request is sent to the native service. When other methods are selected, the selected method is used in the request sent to the native service.
HTTP Connection Timeout (seconds)	Specifies the time interval (in seconds) after which a connection attempt times out. The precedence of the Connection Timeout configuration is as follows: 1. If you specify a value for the **Connection timeout** field in routing endpoint alias, then the **Connection timeout** value specified in the Endpoint alias section takes precedence over the timeout values defined at the API level and the global level. 2. If you specify a value 0 for the **Connection timeout** field in routing endpoint alias, then Microgateway uses the value specified in the **Connection timeout** field in the routing protocol

Parameter	Description
	processing step of an API. The Read Timeout value specified at an API level takes precedence over the global configuration. 3. If you specify a value 0 or do not specify a value for the **Connection timeout** field in the routing protocol processing step at the API level or specify a value 0 at an alias level, then Microgateway uses the value specified in this `pg.endpoint.connectionTimeout` property. 4. If you do not specify any value for `pg.endpoint.connectionTimeout`, then Microgateway uses the default value of 30 seconds.
Read Timeout (seconds)	Specifies the time interval (in seconds) after which a socket read attempt times out. The precedence of the Read Timeout configuration is as follows: 1. If you specify a value for the **Read timeout** field in routing endpoint alias, then the **Read timeout** value specified in the Endpoint alias section takes precedence over the timeout values defined at the API level and the global level. 2. If you specify a value 0 for the **Read timeout** field in routing endpoint alias, then Microgateway uses the value specified in the **Read Timeout** field in the routing protocol processing step of an API. The Read Timeout value specified at an API level takes precedence over the global configuration. 3. If you specify a value 0 or do not specify a value for the **Read timeout** field in the routing protocol processing step at the API level or specify a value 0 at an alias level, then Microgateway uses the value specified in this `pg.endpoint.readTimeout` property. 4. If you do not specify any value for `pg.endpoint.readTimeout`, then Microgateway uses the default value of 30 seconds.
SSL Configuration	Specifies values to enable SSL client authentication that Microgateway uses to authenticate incoming requests for the native API. Provide the following information: - **Keystore Alias**. Specifies the keystore alias that is present in the system-settings.yml. This value (along with the value of Client Certificate Alias) is used for performing SSL client authentication.

Parameter	Description
	■ **Key Alias**. Specifies the alias for the private key, which must be stored in the keystore specified by the keystore alias.
Service Registry Configuration	
Service Discovery Endpoint Parameter	Values required for constructing the discovery service URI. ■ **Parameter**. An alias that you have included in the discovery service URI while adding the service registry to Microgateway. ■ **Value**. A value for the path parameter. The alias specified in Path Parameter is substituted with this value when invoking the discovery service. For example: if the service registry configuration of the service registry that you have selected in **Endpoint URI** has **Service discovery path** set to `/catalog/service/{serviceName}` (and the `{serviceName}` alias is intended for passing the service name), you must provide `{serviceName}` as **Parameter** and the name of the service as **Value**.

Outbound Auth - Transport

When the native API is protected and expects the authentication credentials to be passed through transport headers, you can use this policy to provide the credentials that will be added to the request and sent to the native API. Microgateway supports a wide range of authentication schemes, such as Basic Authentication, OAuth, and JWT at the transport-level.

Note:
Transport-level authentication can be used to secure the REST APIs.

The table lists the properties that are supported for this policy in Microgateway:

Parameter	Description
Authentication scheme	Specifies one of the following schemes for outbound authentication at the transport level: ■ **Basic**. Uses basic HTTP authentication details to authenticate the client. ■ **OAuth2**. Uses OAuth token details to authenticate the client. ■ **JWT**. Uses JSON web token details to authenticate the client. ■ **Anonymous**. Authenticates the client without any credentials. ■ **Alias**. Uses the configured alias name for authentication.

Parameter	Description
Authenticate using	Specifies one of the following modes to authenticate the client: ■ **Custom credentials**. Uses the values specified in the policy to obtain the required token to access the native API. ■ **Delegate incoming credentials**. Uses the values specified in the policy by the API providers to select whether to delegate the incoming token or act as a normal client. ■ **Incoming HTTP Basic Auth credentials**. Uses the incoming user credentials to retrieve the authentication token to access the native API. ■ **Incoming OAuth token**. Uses the incoming OAuth2 token to access the native API. ■ **Incoming JWT**. Uses the incoming JSON Web Token (JWT) to access the native API.
Basic	Uses the HTTP authentication details to authenticate the client. Microgateway supports the following modes of HTTP authentication: ■ **Custom credentials** ■ **Incoming HTTP Basic Auth credentials** Provide the following credentials: ■ **User Name**. Specifies the user name. ■ **Password**. Specifies the password of the user. ■ **Domain Name**. Specifies the domain in which the user resides.
OAuth2	Uses the OAuth2 token to authenticate the client. Microgateway supports the following modes of OAuth2 authentication: ■ **Custom credentials** ■ **Incoming OAuth token** **OAuth2 token**. Specifies the client's OAuth2 token.
JWT	Uses the JSON Web Token (JWT) to authenticate the client. If the native API is enforced to use JWT for authenticating the client, then Microgateway enforces the need for a valid JWT in the outbound request while accessing the native API.

Parameter	Description
	Microgateway supports the **Incoming JWT** mode of JWT authentication.
Alias	Name of the configured HTTP Transport Security alias.

Traffic Monitoring

The policy in this stage provides ways to enable logging request and responses to a specified destination. The policy included in this stage is Log Invocation.

The policies in this stage provide ways to enable logging request and responses to a specified destination and enforce limits for the number of service invocations during a specified time interval and send alerts to a specified destination when the performance conditions are violated. The policies included in this stage are:

- Log Invocation
- Traffic Optimization
- Monitor Performance
- Monitor Level Agreement

Log Invocation

This policy enables logging requests or responses to API Gateway and external Elasticsearch. This action also logs other information about the requests or responses, such as the API name, operation name, the Integration Server user, a timestamp, and the response time.

The table lists the properties that are supported for this policy in Microgateway:

Parameter	Description
Store Request	Logs all requests.
Store Response	Logs all responses.
Compress Payload Data	Compresses the logged payload data.
Log Generation Frequency	Specifies how frequently to log the payload. You can have one of the following options: - **Always**. Logs all requests and responses. - **On Failure**. Logs only the failed requests and responses. - **On Success**. Logs only the successful responses and requests.

Parameter	Description
Destination	Specifies the destination where to log the payload. You can have one of the required destinations: ■ **API Gateway** ■ **Elasticsearch** Microgateway does not support destinations other than the ones listed above.

Traffic Optimization

This policy limits the number of API invocations during a specified time interval, and sends alerts to a specified destination when the performance conditions are violated. You can use this policy to avoid overloading the back-end services and their infrastructure, to limit specific clients in terms of resource usage, and so on.

This policy only limits the number of API invocations within a single Microgateway instance. That is, the policy is not applicable across Microgateway instances holding the same APIs.

The table lists the properties that are supported for this policy in Microgateway:

Parameter	Description
Limit Configuration.	
Rule name	Specifies the name of throttling rule to be applied. For example, Total Request Count.
Operator	Specifies the operator that connects the rule to the value specified. The operator specified is **Greater Than**. For example, in this case the throttling rule is applied when the Total Request Count is greater than (exceeds the limit specified for) the value specified in the **Value** field.
Value	Specifies the value of the request count beyond which the policy is violated. **Note:** ■ When multiple requests are made at the same time, it may be possible that this limit applied to trigger an alert is not strictly adhered to. There is no loss observed in the invocation counts data, but there might be a delay in aggregating the count. ■ Aggregation across multiple Microgateway instances is not supported. Aggregation is done at the instance level and the same is considered for throttling limit.

Parameter	Description
Destination	Specifies the destination to log the alerts. You can have one of the following required options: ■ **API Gateway** ■ **Elasticsearch** Microgateway does not support destinations other than the ones listed above.
Alert Interval	Specifies the interval of time for the limit to be reached.
Unit	Specifies the unit for the time interval in minutes, hours, days, or weeks for the alert interval.
Alert Frequency	Specifies the frequency at which the alerts are issued. You can have one of the following options: ■ **Only Once**. Triggers an alert only the first time the specified condition is violated. ■ **Every Time**. Triggers an alert every time the specified condition is violated.
Alert Message	Specifies the text message to be included in the alert.
Consumer Applications	Specifies the application to which this policy applies.

Monitor Performance

This policy is similar to the Monitor Level Agreement policy and does monitor the same set of run-time performance conditions for an API in a Microgateway instance, and sends alerts when the performance conditions are violated. However, this policy monitors run-time performance at the API level. Parameters like success count, fault count, and total request count are immediate monitoring parameters and the evaluation happens immediately after the limit is breached. The rest of the parameters are Aggregated monitoring parameters whose evaluation happens once the configured interval is over. If there is a breach in any of the parameters, an event notification (Monitor event) is sent to the configured destination. In a single policy, multiple action configurations behave as AND condition. The OR condition can be achieved by configuring multiple policies.

This policy only monitors run-time performance conditions within a single Microgateway instance. That is, the policy is not applicable across Microgateway instances holding the same APIs.

The table lists the properties that you can specify for this policy:

Parameter	Value
Action Configuration. Specifies the type of action to be configured.	
Name	Specifies the name of the metric to be monitored. You can have one of the available metrics: ■ **Availability**. Indicates whether the API is available to the specified clients in the current interval. ■ **Average Response Time**. Indicates the average time taken by the API to complete all invocations in the current interval. The average is calculated from the instant the API activation takes place for the configured interval. For example, if you set an alert for Average response time greater than 30 ms with an interval of 1 minute then on API activation, the monitoring interval starts and the average of the response time of all runtime invocations for this API in 1 minute is calculated. If this is greater than 30 ms, then a monitor event is generated. If this is configured under Monitor performance, then all the runtime invokes are taken into account. ■ **Fault Count**. Indicates the number of faults returned in the current interval. ■ **Maximum Response Time**. Indicates the maximum time to respond to a request in the current interval. ■ **Minimum Response Time**. Indicates the minimum time to respond to a request in the current interval. ■ **Success Count**. Indicates the number of successful requests in the current interval. ■ **Total Request Count**. Indicates the total number of requests (successful and unsuccessful) in the current interval.
Operator	Specifies the operator applicable to the metric selected. You can have one of the available operator: **Greater Than**, **Less Than**, **Equals To**.
Value	Specifies the alert value for which the monitoring is applied.
Destination	Specifies the destination where the alert is to be logged. You can have the required options: ■ **API Gateway** ■ **Elasticsearch**

Parameter	Value
	Microgateway does not support destinations other than the ones listed above.
Alert Interval	Specifies the time period in which to monitor performance before sending an alert if a condition is violated. The timer starts once the API is activated and resets after the configured time interval. If an API is deactivated the interval gets reset and on API activation it starts afresh.
Unit	Specifies the unit for the time interval in minutes, hours, days, or weeks for the alert interval.
Alert Frequency	Specifies how frequently to issue alerts for the counter-based metrics (Total Request Count, Success Count, Fault Count). You can have one of the following options: ■ **Only Once**. Triggers an alert only the first time one of the specified conditions is violated. ■ **Every Time**. Triggers an alert every time one of the specified conditions is violated.
Alert Message	Specifies the text to be included in the alert.

Monitor Level Agreement

This policy monitors a set of run-time performance conditions for an API in a Microgateway instance, and sends alerts to a specified destination when the performance conditions are violated. This policy enables you to monitor run-time performance for one or more specified applications. You can configure this policy to define a Service Level Agreement (SLA), which is a set of conditions that defines the level of performance that an application should expect from an API. You can use this policy to identify whether the API threshold rules are met or exceeded. For example, you might define an agreement with a particular application that sends an alert to the application if responses are not sent within a certain maximum response time. You can configure SLAs for each API or application combination.

Parameters like success count, fault count, and total request count are immediate monitoring parameters and the evaluation happens immediately after the limit is breached. The rest of the parameters are Aggregated monitoring parameters whose evaluation happens once the configured interval is over. If there is a breach in any of the parameters, an event notification (Monitor event) is sent to the configured destination. In a single policy, multiple action configurations behave as AND condition. The OR condition can be achieved by configuring multiple policies.

This policy action only monitors run-time performance conditions within a single Microgateway instance. That is, the policy is not applicable across Microgateway instances holding the same APIs.

The table lists the properties that you can specify for this policy:

Parameter	Value
Action Configuration. Specifies the type of action to be configured.	
Name	Specifies the name of the metric to be monitored. You can have one of the available metrics: ■ **Availability**. Indicates whether the API is available to the specified clients in the current interval. ■ **Average Response Time**. Indicates the average time taken by the API to complete all invocations in the current interval. The average is calculated from the instant the API activation takes place for the configured interval. For example, if you set an alert for Average response time greater than 30 ms with an interval of 1 minute then on API activation, the monitoring interval starts and the average of the response time of all runtime invocations for this API in 1 minute is calculated. If this is greater than 30 ms, then a monitor event is generated. If this is configured under Monitor Level Agreement policy with an option to configure applications so that application specific SLA monitoring can be done, then the monitoring for the average response time is done only for the specified application. ■ **Fault Count**. Indicates the number of faults returned in the current interval. ■ **Maximum Response Time**. Indicates the maximum time to respond to a request in the current interval. ■ **Minimum Response Time**. Indicates the minimum time to respond to a request in the current interval. ■ **Success Count**. Indicates the number of successful requests in the current interval. ■ **Total Request Count**. Indicates the total number of requests (successful and unsuccessful) in the current interval.
Operator	Specifies the operator applicable to the metric selected. You can have one of the available operator: **Greater Than**, **Less Than**, **Equals To**.
Value	Specifies the alert value for which the monitoring is applied.
Destination	Specifies the destination where the alert is to be logged. You can have the required options:

Parameter	Value
	■ **API Gateway** ■ **Elasticsearch** Microgateway does not support destinations other than the ones listed above.
Alert Interval	Specifies the time period (in minutes) in which to monitor performance before sending an alert if a condition is violated. The timer starts once the API is activated and resets after the configured time interval. If and API is deactivated the interval gets reset and on API activation its starts afresh.
Alert Frequency	Specifies how frequently to issue alerts for the counter-based metrics (Total Request Count, Success Count, Fault Count). You can have one of the options: ■ **Only Once**. Triggers an alert only the first time one of the specified conditions is violated. ■ **Every Time**. Triggers an alert every time one of the specified conditions is violated.
Alert Message	Specifies the text to be included in the alert.
Consumer Applications	Specifies the application to which this Service Level Agreement applies.

Response Processing

These policies are used to specify how the response message from the API has to be transformed or pre-processed and configure the masking criteria for the data to be masked before it is submitted to the application. This is required to protect the data and accommodate differences between the message content that an API is capable of submitting and the message content that an application expects. The policies included in this stage are:

- Response Transformation
- Validate API Specification
- Data Masking
- CORS

Response Transformation

This policy specifies the properties required to transform response messages from native APIs into a format required by the client.

This policy enables you to configure several transformations on the response messages before it is sent to the client.

Microgateway supports the following parameter types that can be used to configure the transformation policy:

- response.payload
- response.headers
- response.statusCode
- response.statusMessage

Microgateway supports the following Query types that can be used to configure the transformation policy:

- xpath
- jsonPath
- regex

When you use these syntaxes to extract a value from the payload, the content-types applicable are:

- ${payload.jsonPath} - application/json, application/json/badgerfish
- ${payload.regex} - text/plain
- ${payload.xpath} - application/xml, text/xml, text/html.

The table lists the properties that are supported for this policy in Microgateway:

Parameter	Description
Condition	Conditions are used to specify when the policy has to be executed. We can add multiple conditions with logical operators. Available values are: - **AND**. Microgateway transforms the responses that comply with all the configured conditions - **OR**. This is selected by default. Microgateway transforms the responses that comply at least one configured condition. Various conditions you can define are: - **Variable**. Specifies the variable type with a syntax as follows: - `${PARAMTYPE}` : This is applicable for variables of string type - payload, statusCode and statusMessage. For example: ${response.payload}

Parameter	Description
	■ `${PARAMTYPE.paramName}` : This is applicable for map types - headers. For example: , `${response.header.Content-Type}`, ■ `${PARAMTYPE.QUERYTYPE[queryValue]}` : This syntax is applicable for payload. XPath, JSONPath and regex can be applied on payload. For example: `${response.payload.xpath[//ns:emp/ns:empName]}` where //ns:emp/ns:empName is the xpath to be applied on the payload if contentType is application/xml `${response.payload.jsonPath[$.cardDetails.number]}` where $.cardDetails.number is the jsonPath to be applied on the payload if contentType is application/json `${response.payload.regex[[0-9]+]}` where [0-9]+ is the regex to be applied on the payload if contentType is text/plain ■ If you want Microgateway to apply xpath, jsonPath, regex based on Content-Type of the payload, use the following common syntax: `${PARAMTYPE.QUERYTYPE[queryValue] || PARAMTYPE.QUERYTYPE2[queryValue2] || ...}` For example: `${response.payload.xpath[//ns:emp/ns:empName] || response.payload.jsonPath[$.cardDetails.number]}` This applies xpath for application/xml and jsonPath for application/json `${response.payload.xpath[//ns:emp/ns:empName] || response.payload.jsonPath[$.cardDetails.number] || response.payload.regex[[0-9]+]}` This applies xpath for application/xml, jsonPath for application/json, and regex for text/plain. ■ **Operator**. Specifies the operator to use to relate variable and the value provided. You can select one of the following: ■ **Equals** ■ **Equals ignore case** ■ **Not equals** ■ **Contains** ■ **Exists** ■ **Value**. Specifies a value with a syntax as follows: ■ `PLAIN VALUE`, for example, application/json

Parameter	Description
	■ `${PARAMTYPE.paramName}` ■ `${PARAMTYPE.QUERYTYPE[queryValue]}` ■ `${PARAMTYPE.QUERYTYPE[queryValue] || PARAMTYPE.QUERYTYPE2[queryValue2] || ...}`
Transformation Configuration. Specifies various transformations to be configured.	
Header/Query/Path Transformation for REST API	Specifies the Header, Query or path transformation to be configured for the responses received from the native API. Various configurations you can define are: ■ **Variable**. Specifies the variable type with a syntax as follows: ■ `${PARAMTYPE}` : This is applicable for variables of string type - payload, statusCode and statusMessage. For example: ${response.payload} ■ `${PARAMTYPE.paramName}` : This is applicable for map types - headers. For example: , `${response.header.Content-Type}`, ■ `${PARAMTYPE.QUERYTYPE[queryValue]}` : This syntax is applicable for payload. XPath, JSONPath and regex can be applied on payload. For example: `${response.payload.xpath[//ns:emp/ns:empName]}` where //ns:emp/ns:empName is the xpath to be applied on the payload if contentType is application/xml `${response.payload.jsonPath[$.cardDetails.number]}` where $.cardDetails.number is the jsonPath to be applied on the payload if contentType is application/json `${response.payload.regex[[0-9]+]}` where [0-9]+ is the regex to be applied on the payload if contentType is text/plain ■ If you want Microgateway to apply xpath, jsonPath, regex based on Content-Type of the payload, use the following common syntax: `${PARAMTYPE.QUERYTYPE[queryValue] \|\| PARAMTYPE.QUERYTYPE2[queryValue2] \|\| ...}` For example: `${response.payload.xpath[//ns:emp/ns:empName] || response.payload.jsonPath[$.cardDetails.number]}` This applies xpath for application/xml and jsonPath for application/json `${response.payload.xpath[//ns:emp/ns:empName] || response.payload.jsonPath[$.cardDetails.number] ||`

Parameter	Description
	`response.payload.regex[[0-9]+]}` This applies xpath for application/xml, jsonPath for application/json, and regex for text/plain. ■ **Value**. Specifies a value with a syntax as follows: ■ `PLAIN VALUE`, for example, application/json ■ `${PARAMTYPE.paramName}` ■ `${PARAMTYPE.QUERYTYPE[queryValue]}` ■ `${PARAMTYPE.QUERYTYPE[queryValue] || PARAMTYPE.QUERYTYPE2[queryValue2] || ...}`
Status transformation	Specifies the status transformation to be configured for the responses received from the native API. ■ **Code**. Specifies the status code that is sent in the response to the client. ■ **Message**. Specifies the Status message that is sent in the response to the client.
Payload Transformation	Specifies the payload transformation to be configured for the responses received from the native API. Specifies the following information: ■ An xslt document that contains the following information: ■ **XSLT file**. Specifies the XSLT file used to transform the response messages as required. ■ **Feature Name**. Specifies the name of the XSLT feature. ■ **Feature value**. Specifies the value of the XSLT feature. You can have multiple XSLT features and xslt documents.
Transformation Metadata. Specifies the metadata for transformation of the responses received from the native API. For example, the namespaces configured in this section can be used when you provide the syntax for XPath ${response.payload.xpath} For example: ${response.payload.xpath[//ns:emp/ns:empName]}	
Namespace	Specifies the namespace information to be configured for transformation. Provide the following information: ■ **Namespace Prefix**. The namespace prefix of the payload expression to be validated.

Parameter	Description
	■ **Namespace URI**. The namespace URI of the payload expression to be validated. **Note:** You can have multiple namespace prefix and URI.

Validate API Specification

This policy validates the responses against various components of an API specification such as schema, content-types, and HTTP headers.

The various components of an API specification are referenced as follows:

- The schema for REST APIs can be imported through a swagger or a RAML file, or a file you upload. The schema is available as part of the API definition.
- The content-types are available as part of the API definition.
- The HTTP headers are specified in the Validate API Specification policy page.

The response sent to the API by the native service must conform with the structure or format expected by the API. The responses from the native API are validated against the API specifications selected in this policy to conform to the structure or format expected by the API.

The various components of an API specification that can be validated are:

- **Schema**

  The responses from the native API are validated against the schema provided in the API definition. The schema defines the elements and attributes in the response payload and specifies the data types of these elements to ensure that only appropriate data is allowed through to the API.

  For a REST API, the schema validation execution depends on the accept header in the response. The default accept header and schema validation type mapping is as follows:

Accept header	Schema validation type
application/json application/json/badgerfish	JSON schema
application/xml text/xml text/html	XML schema
text/plain	Regular expression

- **Content-types**

  The content-types in the responses from the native API are validated against the corresponding content-types specified in the API definition.

- **HTTP Headers**

  The HTTP headers in the responses from the native API are validated against the corresponding HTTP headers specified in this policy to conform to the HTTP headers expected by the API.

The API requests that fail the specification validation are considered as policy violations. Such policy violation events that are generated can be viewed in the API Gateway dashboard.

The table lists the parameters of this policy and how they are applied to validate API requests:

Parameter	Description
Schema	Validates the response payload against the appropriate schema (based on the accept header in the response). Additional features for XML schema validation are: ■ **Feature name**. The name for the schema configuration. For example: TOLERATE_DUPLICATES, NAMESPACE_GROWTH ■ **Feature value**. Specifies whether the feature value is **True** or **False**.
Content-types	Validates the content-types in the incoming response against the content-types defined in the API definition.
HTTP Headers	Validates the HTTP header parameters in the incoming response against the HTTP headers defined in this policy. Various conditions and additional information you can define are: ■ **Condition**: Specifies the logical operator to use to validate multiple HTTP headers in the incoming API responses. Available values are: ■ **AND**. Microgateway accepts only the responses that contain all configured HTTP headers. ■ **OR**. This is selected by default. Microgateway accepts responses that contain at least one configured HTTP header. ■ **HTTP Header Key**. Specifies a key that must be passed through the HTTP header of the incoming API responses. ■ **Header Value**. *Optional.*. Specifies the corresponding key value that could be passed through the HTTP header of the incoming API responses.

Parameter	Description
	The **Header Value** field type accepts string and regular expression (regex).

Data Masking

Data masking is a technique whereby sensitive data is obscured in some way to render it safe and to protect the actual data while having a functional substitute for occasions when the real data is not required.

This policy is used to mask sensitive data at the application level. At the application level you must have an Identify and Access policy configured to identify the application for which the masking is applied. If no application is specified then it is applied for all the other responses. Fields can be masked or filtered in the response messages to be sent. You can configure the masking criteria as required for the XPath, JSONPath, and Regex expressions based on the content-types. This policy can also be applied at the API scope level.

The table lists the content-type and masking criteria mapping.

Content-type	Masking Criteria
application/xml	XPath
text/xml	
text/html	
application/json	JSONPath
application/json/badgerfish	
text/plain	Regex

The table lists the masking criteria properties that are configured to mask the data in the response messages in Microgateway:

Parameter	Description
Consumer Applications	*Optional.* Specifies the applications for which the masking criterion has to be applied. For example: If there is a DataMasking(DM1) criteria created for application1 a second DataMasking(DM2) for application2 and a third DataMasking(DM3) with out any application, then for a request that comes from consumer1 the masking criteria DM1 is applied, for a request that comes from consumer2 DM2 is applied. If a request comes with out any application or from any other application except application1 and application2 DM3 is applied.

Parameter	Description
XPath: Specifies the masking criteria for XPath expressions in the response messages.	
Masking Criteria	Specifies the masking criteria that contains the following information: ■ **Masking Type**. Specifies the type of masking required. You can have either **Mask** or **Filter**. **Mask** replaces the value with the given value (the default value being ********) and **Filter** removes the field completely. ■ **Query expression**. Specifies the query expression that has to be masked or filtered. For example: /pet/details/status, /user/details/card/ccnumber. ■ **Mask Value**. This is available if masking type selected is **Mask**. Provide a mask value. For example: sold, any mask value #####. **Note:** You can have multiple masking criteria. ■ **Namespace**. Specifies the following Namespace information: ■ **Namespace Prefix**. The namespace prefix of the payload expression to be validated. ■ **Namespace URI**. The namespace URI of the payload expression to be validated **Note:** You can have multiple namespace prefix and URI.
JSONPath. Specifies the masking criteria for JSONPath expressions in the response messages.	
Masking Criteria	Specifies the masking criteria that contains the following information: ■ **Masking Type**. Specifies the type of masking required. You can have either **Mask** or **Filter**. **Mask** replaces the value with the given value (the default value being ********) and **Filter** removes the field completely. ■ **Query expression**. Specify the query expression that has to be masked or filtered. For example: $.pet.details.status ■ **Mask Value**. This is available if masking type selected is **Mask**. Provide a mask value. For example: sold
Regex. Specifies the masking criteria for regular expressions in the response messages.	
Masking Criteria	Specifies the masking criteria that contains the following information:

Parameter	Description
	■ **Masking Type**. Specifies the type of masking required. You can have either **Mask** or **Filter**. **Mask** replaces the value with the given value (the default value being ********) and **Filter** removes the field completely. ■ **Query expression**. Specify the query expression that has to be masked or filtered. For example: [0-9]+ ■ **Mask Value**. This is available if masking type selected is **Mask**. Provide a mask value. For example: ########
Apply for transaction Logging	Select this option to apply masking criteria for transactional logs.
Apply for payload	Select this option to apply masking criteria for payload in the response message.

CORS

The Cross-Origin Resource Sharing (CORS) mechanism supports secure cross-domain requests and data transfers between browsers and web servers. The CORS standard works by adding new HTTP headers that allow servers to describe the set of origins that are permitted to read that information.

This policy uses CORS support that uses additional HTTP headers to let a client or an application gain permission to access selected resources. An application or a client makes a cross-origin HTTP request when it requests a resource from a different domain, protocol, or port than the one from which the current request originated.

The table lists the CORS response specifications that are supported for this policy in Microgateway:

Parameter	Description
Allowed Origins	Specifies the origin from which the responses originating are allowed. syntax for the origin: *scheme*://*host*:*port* You can have multiple origins and you can also provide Regular expressions for allowed origins. Allowed origins of applications registered with this API are also allowed to access this API.
Max Age	Specifies the age for which the preflight response is valid.
Allowed Methods	Specifies the methods that are allowed in the request. Specify one or more of the following: **GET**, **POST**, **PUT**, **DELETE**, and **PATCH**.

Parameter	Description
Allow Headers	Specifies the Headers that are allowed in the request. You can have multiple headers that are to be allowed.
Allow Credentials	Specifies whether the request credentials could be exposed to the user on request failure.
Expose Headers	Specifies the headers that be exposed to the user on request failure. You can have multiple headers that are to be allowed.

A corresponding HTTP header is set for all the values above as per the specification. For additional information, see https://www.w3.org/TR/cors/.

Error Handling

The policy in this stage enables you to specify the error conditions, lets you determine how these error conditions are to be processed. The policy included in this stage is Conditional Error Processing. You can also mask the data while processing the error conditions. The policies included in this stage are:

- Conditional Error Processing
- Data Masking

Conditional Error Processing

Error Handling is the process of passing an exception message issued as a result of a run-time error to take any necessary actions. This policy returns a custom error message (and the native provider's service fault content) to the application when the native provider returns a service fault. You can configure conditional error processing and use variables to create custom error messages.

The table lists the properties that are supported for this policy in Microgateway:

Parameter	Description
Error conditions. Specifies the error conditions and how these error conditions should be processed.	
Status Code Error Criteria	Specify the error status code. Provide a value for the **Code**.
Header Error Criteria	Provide the details of the custom HTTP header(s) included in the client requests. Provide the following information: - **Header Name**. Specifies the name of the HTTP header.

Parameter	Description
	■ **Header Value**. Specifies the value of the HTTP header.
Payload Criteria	Provide the details of the payload criteria in the API request. You can have the following information in the payload identifier section: ■ **Expression type**. Specifies the type of expression, which is used for identification. You can select one the following expression type: ■ **XPath**. Provide the following information: ■ **Payload Expression**. Specifies the payload expression that the specified XPath expression type in the request or the response has to be converted to. For example: /name/id. The response maybe a native service error or Microgateway generated error. ■ **Namespace Prefix**. The namespace prefix of the payload expression to be validated. ■ **Namespace URI**. The namespace URI of the payload expression to be validated. **Note:** You can include multiple namespace prefix and URI. ■ **JSONPath**. Provide the **Payload Expression** that specifies the payload expression that the specified JSONPath expression type in the request or the response has to be converted to. For example: $.name.id. The response maybe a native service error or Microgateway generated error. ■ **Text**. Provide the **Payload Expression** that specifies the payload expression that the specified Text expression type in the request or response has to be converted to. For example: any valid regular expression. The response maybe a native service error or Microgateway generated error. You can add multiple payload identifiers as required. **Note:**

Parameter	Description
	Only one payload identifier of each type is allowed. For example, you can add a maximum of three payload identifiers, each being of a different type. **Value**: Specifies a value that has to match with the value contained in the error Response.
Custom Error Variables. Specifies the error variables to be used in the custom error message.	
Payload Type	Specify the payload type. Available values are: ■ **Request**. Specifies the request payload type. ■ **Response**. Specifies the response payload type.
Name	Provide a name for the payload type.
Payload Identifier	Provide the details of the payload criteria in the API request. Provide the following information in the Payload identifier section: ■ **Expression type**. Specifies the type of expression contained in the payload request. ■ **Payload Expression**. Specifies the payload expression that the specified expression type in the request has to be converted to. ■ **Namespace Prefix**. The namespace prefix of the payload expression to be validated. ■ **Namespace URI**. The namespace URI of the payload expression to be validated. **Note:** You can add multiple namespace prefix and URI. You can add multiple payload identifiers as required.
Failure Message. Specifies the custom failure message format that Microgateway should send to the application. Specify whether the message should be in the **text**, **json**, or **xml** format.	
Send Native Provider Fault Message	Enable this parameter so that Microgateway sends the native REST failure message to the application. When you disable this parameter, the failure message is ignored when a fault is returned by the native API provider.

Data Masking

Data masking is a technique whereby sensitive data is obscured in some way to render it safe and to protect the actual data while having a functional substitute for occasions when the real data is not required.

This policy is used to mask sensitive data in the custom error messages being processed and sent to the application. Fields can be masked or filtered in the error messages. You can configure the masking criteria as required for the XPath, JPath, and Regex expressions. This policy can also be applied at the API scope level.

The table lists the masking criteria properties that are supported for this policy in Microgateway to mask the data in the error messages received:

Parameter	Description
Consumer Applications	Specifies the applications for which the masking criterion has to be applied.
XPath. Specifies the masking criteria for XPath expressions in the error messages.	
Masking Criteria	Specifies the masking criteria that contains the following information: ■ **Masking Type**. Specifies the type of masking required. You can have either **Mask** or **Filter**. **Mask** replaces the value with the given value (the default value being ********) and **Filter** removes the field completely. ■ **Query expression**. Specifies the query expression that has to be masked or filtered. For example: /soapenv:Fault/faultstring ■ **Mask Value**. This is available if masking type selected is **Mask**. Provide a mask value. For example: *Error occurred while processing the request. Please check your input request* or any other meaningful message or string. **Note:** You can have multiple masking criteria. ■ **Namespace**. Specifies the following Namespace information: ■ **Namespace Prefix**. The namespace prefix of the payload expression to be validated. ■ **Namespace URI**. The namespace URI of the payload expression to be validated **Note:** You can have multiple namespace prefix and URI.
JPath. Specifies the masking criteria for JPath expressions in the error messages.	

Parameter	Description
Masking Criteria	Specifies the masking criteria that contains the following information: ■ **Masking Type**. Specifies the type of masking required. You select either **Mask** or **Filter**. **Mask** replaces the value with the given value (the default value being ********) and **Filter** removes the field completely. ■ **Query expression**. Specify the query expression that has to be masked or filtered. For example: $.error.reason ■ **Mask Value**. This is available if masking type selected is **Mask**. Provide a mask value. For example: *Error occurred while processing the request. Please check your input request* or any other meaningful message or string.
Regex. Specifies the masking criteria for regular expressions in the error messages.	
Masking Criteria	Specifies the masking criteria that contains the following information: ■ **Masking Type**. Specifies the type of masking required. You select either **Mask** or **Filter**. **Mask** replaces the value with the given value (the default value being ********) and **Filter** removes the field completely. ■ **Query expression**. Specify the query expression that has to be masked or filtered. For example: (.*) ■ **Mask Value**. This is available if masking type selected is **Mask**. Provide a mask value. For example: *Error occurred while processing the request. Please check your input request* or any other meaningful message or string.
Apply for transaction Logging	Select this option to apply masking criteria for transactional logs.
Apply for payload	Select this option to apply masking criteria for payload.

API Scopes

API definitions can be complex and span across multiple REST resources and methods for an API. To reduce the complexity of an API definition, you can define scopes and impose a set of policies on each scope to suit your requirements. A scope represents a logical grouping of REST resources, methods, or both in an API. An API can have a set of declared scopes.

API scopes are configured in API Gateway and provisioned to Microgateway.

Only the following policies are supported for API Scopes:

- Identify and Access policy: Identify & Authorize

- Traffic Monitoring policies: Log Invocation, Traffic Optimization

Note:
If a scope references an unsupported policy, then the provisioning of the API is rejected.

7 Service Registry Support

Overview

Microgateway supports service discovery and provides capability to publish APIs to runtime service registries.

In a micro-service landscape, service registries provide the information about service instances and their location or endpoints. This information enables the service discovery during runtime. Accordingly you can configure a Microgateway instance to register all its provisioned APIs to a service registry. During startup, it generates one service registry entry per API. The endpoints of the registered APIs are based on the host and port of the Microgateway instance. Multiple Microgateway instances can expose the same APIs and register them to the same service registry. The service registry then shows all the endpoints of the given APIs. During shutdown, a Microgateway instance removes the service registry entries it has generated.

Microgateway currently supports the following service registries.

- **Eureka**

  Eureka is a REST-based service for locating services for the purpose of load balancing and failover of middle-tier servers. It has been primarily designed for applications in the AWS cloud.

- **Service Consul**

  Service Consul is a tool for discovering and configuring services in IT infrastructure.

Service Registry Configuration

You can configure service registry settings in one of the following ways:

- By configuring the service registry settings in the user-defined custom settings YAML file that is passed as an argument using the -c option during Microgateway startup.
- By downloading the service registry settings from API Gateway using the -ds option with the value true during Microgateway startup. The downloaded settings are merged with the settings from the custom settings YAML file. The downloaded settings take preference over the values in the custom settings YAML file in case of conflict during the merge.

To control the registration of APIs in a runtime service registry, the Microgateway custom settings file provides a publish section.

The publish section in the custom settings file is as follows:

```
publish:
  registries:
  - EurekaDefault
```

The publish section contains the registry names referencing the Service Registries. The APIs are only published to the specified service registries during Microgateway startup. To address all the configured service registries, use * as the registry name. Microgateway does not support API-specific registration. Either all or none of the APIs are published to the configured registries. The referenced registries point to the entries in the service_registries section in the custom settings file.

The yaml section for the service registry looks as follows:

```
service_registries:
  EurekaDefault:
    type: serviceRegistryAlias
    description: Eureka is a REST based service that is primarily used in the
                 AWS cloud for locating services for the purpose of load balancing
                 and failover of middle-tier servers
    owner: Administrator
    endpointURI: http://localhost:9091
    heartBeatInterval: 0
    discoveryInfo:
      path: /eureka/apps/{app}
      httpMethod: GET
    registrationInfo:
      path: /eureka/apps/{app}
      httpMethod: POST
    deRegistrationInfo:
      path: /eureka/apps/{app}/{instanceId}
      httpMethod:DELETE
    serviceRegistryType: EUREKA
    connectionTimeout: 30
    readTimeout: 30
```

The table lists the variables in the yaml file, their description and their usage.

Variable	Description
`name`	Name of the registry
`serviceRegistryType`	Type of the registry. It is either EUREKA or SERVICE_CONSUL
`connectionTimeout`	Specifies the time, in seconds, for which the Microgateway tries to connect to the registry.
`readTimeout`	Specifies the time, in seconds, for which the registry tries to connect to the service endpoint.
`endpointURI`	The path of the registry.
`discoveryInfo`	Information required to discover the registry **path**. The rest path of the registry to register services **http Method**. The rest path of the registry to register services.
`registrationInfo`	Information required to register services. **path**. The rest path of the registry to register services. **http Method**. The rest path of the registry to register services.
`deRegistrationInfo`	Information required to deregister services. **path**. The rest path of the registry to deregister services.

Variable	Description
	http Method. The rest path of the registry to deregister services.
`customHeaders`	A map of all custom Headers you need to reach your registry. This is required only if you use consul with "X-Consul-Token" : "" as entry.
`username`	*Optional.* The username of the user authorized to to register services at your registry.
`password`	*Required if you are using the username.* Specifies the password of the user authorized to to register services at your registry.

8 Command Line Reference

Microgateway Command Line Reference

This section describes operations you can perform such as, start and stop Microgateway, retrieve Microgateway status, view the assets provisioned, create a Microgateway instance, create an asset archive, create a docker file, and so on through Command Line Interface(CLI).

Using Microgateway CLI

The Microgateway CLI script comes in 2 flavors: Windows (.bat) and Linux (.sh). Invoking the script provides usage information:

```
Please renew the usage action:

   start                - Start a Microgateway server
   stop                 - Stop a Microgateway server
   status               - Retrieve the Microgateway server status
   assets               - Show the provisioned assets of a running server
   createInstance       - Create a Microgateway instance
   createAssetArchive   - Create an asset archive
   createDockerFile     - Create a Microgateway docker file
   createKubernetesFile - Create a Kubernetes file
   downloadSettings     - Create a custom settings file
   settings             - Show the settings configured in the Microgateway instance
```

Starting a Microgateway

Run the following command to start a Microgateway.

```
./microgateway.sh start options
```

where the options are:

-Shortcut, --Name	Description
-a, --archive <arg>	List of API Gateway archives
-adp,--admin_password <arg>	Password for administration access
-adu,--admin_user <arg>	User for administration access
-apis, --apis <arg>	List of API identifiers (name, unique identifier, name/version).
-apps, --applications <arg>	List of global applications (name, unique identifier)
-as, --apps_sync	Enable Applications sync
-asi, --apps_sync_interval	Polling interval in secs for applications sync
-ast, --apps_sync_timeout	Connection timeout in secs for applications sync
-asv, --apps_to_sync	Applications to synchronize (all, registeredApplications, comma separated ids)

-Shortcut, --Name	Description
-bp,--base_path <arg>	API Gateway base path
-c,--config <arg>	Configuration (YAML) file
-ds,--download_settings	Downloads the settings from API Gateway.
-gw,--api_gateway <arg>	API Gateway URL
-gwp,--api_gateway_password <arg>	API Gateway password
-gwu,--api_gateway_user <arg>	API Gateway user
-gwd,--api_gateway_dir <arg>	API Gateway install directory
-ikf,--import_keystore_file <arg>	To import one or more keystore files into Microgateway. If there are multiple keystore files you can provide the keystore file names as comma separated items.
-ikp,--import_keystore_password <arg>	To import one or more keystore passwords corresponding to the keystore files being imported into Microgateway.
-itf, --import_truststore_file <arg>	To import one or more truststore files into Microgateway. If there are multiple truststore files you can provide the truststore file names as comma separated items.
-itp, --import_truststore_password <arg>	To import one or more truststore passwords corresponding to the truststore files being imported into Microgateway If there are multiple truststore passwords you can provide the password names as comma separated items.
-jvmopt,--jvm_option <arg>	JVM option. Microgateway supports multiple JVMs.
-lv,--log_level <arg>	ERROR, WARN, INFO, DEBUG, TRACE The default value is ERROR
-lp,--log_path <arg>	Path to log files The default value is logs
-mpc,--max_parallel_connections <arg>	Number of parallel HTTP connections to the native API.
-p,--http_port <arg>	HTTP port number
-pols,--policies <arg>	List of global policy identifiers (name, unique identifier).
-sp,--https_port <arg>	HTTPS port number
-sr,--service_registries <arg>	List of service registry names.

-Shortcut, --Name	Description
-ua,--user_auth <arg>	User authentication method (internal or delegated)
-v,--verbose	Print more information to console You will see a status message for each provisioned API. Also, the user authentication status is exposed.

Stopping a Microgateway

Run the following command to stop a Microgateway.

```
./microgateway.sh stop options
```

where the options are:

-Shortcut, --Name	Description
-c,--config <arg>	Configuration (YAML) file
-p,--http_port <arg>	HTTP port number
-sp,--https_port <arg>	HTTPS port number
-adu,--admin_user <arg>	User for administration access
-adp,--admin_password <arg>	Password for administration access

Retrieving Microgateway Status

Run the following command to retrieve the status of a Microgateway.

```
./microgateway.sh status
```

Viewing the Provisioned Assets in Microgateway

Run the following command to view the assets provisioned in Microgateway.

```
./microgateway.sh assets options
```

where the options are:

-Shortcut, --Name	Description
-c,--config <arg>	Configuration (YAML) file
-p,--http_port <arg>	HTTP port number
-sp,--https_port <arg>	HTTPS port number

-Shortcut, --Name	Description
-adu,--admin_user <arg>	User for administration access
-adp,--admin_password <arg>	Password for administration access
-v,--verbose	Print all details

Creating a Microgateway Instance

Run the following command to create a Microgateway instance package.

```
./microgateway.sh createInstance options
```

where the options are:

-Shortcut, --Name	Description
-gwd,--api_gateway_dir <arg>	API Gateway install directory for taking over the user credential file
-c,--config <arg>	Configuration (YAML) file that would be copied into the instance
-os,--os <arg>	Operating system (windows / linux)
-ins,--instance <arg>	Zip filename to hold the resulting Microgateway instance (mandatory)
-ikf,--import_keystore_file <arg>	To import one or more keystore files into Microgateway. If there are multiple keystore files you can provide the keystore file names as comma separated items.
-ikp,--import_keystore_password <arg>	To import one or more keystore passwords corresponding to the keystore files being imported into Microgateway.
-itf, --import_truststore_file <arg>	To import one or more truststore files into Microgateway. If there are multiple truststore files you can provide the truststore file names as comma separated items.
-itp, --import_truststore_password <arg>	To import one or more truststore passwords corresponding to the truststore files being imported into Microgateway If there are multiple truststore passwords you can provide the password names as comma separated items.
-v,--verbose	Print all details

Creating an Asset Archive

Run the following command to create an asset archive from a running API Gateway instance.

```
./microgateway.sh createAssetArchive options
```

where the options are:

-Shortcut, --Name	Description
-a, --archive <arg>	The resulting API Gateway archive
-apis, --apis <arg>	List of API identifiers (name, unique identifier, name/version).
-apps, --applications <arg>	List of global applications (name, unique identifier, name/version).
-gw,--api_gateway <arg>	API Gateway URL
-gwp,--api_gateway_password <arg>	API Gateway password
-gwu,--api_gateway_user <arg>	API Gateway user
-pols,--policies <arg>	List of global policy identifiers (name, unique identifier).

Creating a Microgateway Docker File

Run the following command to create a Microgateway docker file.

```
./microgateway.sh createDockerFile options
```

where the options are:

-Shortcut, --Name	Description
-a, --archive <arg>	List of API Gateway archives
-apis,--apis <arg>	List of API identifiers
-apps,--applications <arg>	List of global application identifiers
-c,--config <arg>	Configuration (YAML) file
-dod,--docker_dir	Microgateway directory to use in Docker file
-dof,--docker_file	Filename to hold Docker file
-dor,--docker_from	FROM image to use in Docker file
-exec, --exec	Command to start micro service
-gw,--api_gateway <arg>	API Gateway URL

-Shortcut, --Name	Description
-gwd,--api_gateway_dir <arg>	API Gateway install directory
-gwp,--api_gateway_password <arg>	API Gateway password
-gwu,--api_gateway_user <arg>	API Gateway user
-ikf,--import_keystore_file <arg>	To import one or more keystore files into Microgateway. If there are multiple keystore files you can provide the keystore file names as comma separated items.
-ikp,--import_keystore_password <arg>	To import one or more keystore passwords corresponding to the keystore files being imported into Microgateway.
-itf, --import_truststore_file <arg>	To import one or more truststore files into Microgateway. If there are multiple truststore files you can provide the truststore file names as comma separated items.
-itp, --import_truststore_password <arg>	To import one or more truststore passwords corresponding to the truststore files being imported into Microgateway If there are multiple truststore passwords you can provide the password names as comma separated items.
-jre, --jre	none, linux or linux-musl
-jvmopt,--jvm_option <arg>	JVM option. Microgateway supports multiple JVMs.
-lv,--log_level <arg>	ERROR, WARN, INFO, DEBUG, TRACE The default value is ERROR
-msr, --msr	Indicates MSR base image
-p,--http_port <arg>	HTTP port number
-pols,--policies <arg>	List of global policy identifiers
-sp,--https_port <arg>	HTTPS port number
-ua,--user_auth <arg>	User authentication method (internal or delegated)

Creating a Microgateway Kubernetes File

You can prepare a Kubernetes deployment file (yml format) for deploying a Microgateway Docker image to Kubernetes. Sidecar deployment is possible and also health-check methods can be selected.

Run the following command to create a Microgateway Kubernetes file:

```
./microgateway.sh createKubernetesFile options
```

where the options are:

-Shortcut, --Name	Description
-pn,--pod_name <arg>	Name for the Kubernetes pod and deployment
-di,--docker_image <arg>	The Microgateway Docker image name (inside a docker registry in the shape: registry/ imagename)
-hm,--health_mode <arg>	Mode for Kubernetes health checks (all, lifeness, readiness) The default value is all.
-p,--http_port <arg>	The exposed port of the Microgateway Docker image
-rep,--replicas <arg>	Number of pod replicas. The default value is 1.
-sdi,--sidecar_docker_image <arg>	*Optional*. Docker image name for the sidecar container (inside a docker registry in the shape: registry/imagename)
-spn,--sidecar_pod_name <arg>	*Optional*. NAme for the sidecar Kubernetes pod.
-o,--output <arg>	Generated output file (yml format)

Creating Settings file

Run the following command to create a custom settings file.

```
./microgateway.sh downloadSettings  options
```

where the options are:

-Shortcut, --Name	Description
-gw,--api_gateway <arg>	API Gateway URL
-gwp,--api_gateway_password <arg>	API Gateway password
-gwu,--api_gateway_user <arg>	API Gateway user
-c,--config <arg>	Optional: input configuration file
-o, - output <arg>	Output settings file

Viewing the Settings in Microgateway

Run the following command to view the settings configured in the Microgateway instance.

```
./microgateway.sh settings options
```

where the options are:

-Shortcut, --Name	Description
-adu,--admin_user <arg>	User for administration access
-adp,--admin_password <arg>	Password for administration access
-c,--config <arg>	Configuration (YAML) file
-p,--http_port <arg>	HTTP port number
-sp,--https_port <arg>	HTTPS port number
-v,--verbose	Print all details

system-settings.yml

The following shows a sample system-settings.yml file structure.

```
---
faults:
   default_error_message: "API Gateway encountered an error.
   Error Message: $ERROR_MESSAGE. Request Details: Service - $SERVICE,
   Operation - $OPERATION, Invocation Time:$TIME, Date:$DATE,
   Client IP - $CLIENT_IP, User - $USER and Application:$CONSUMER_APPLICATION"
   native_provider_fault: "false"
extended_settings:
   defaultEncoding: "UTF-8"
   apiKeyHeader: "x-Gateway-APIKey"
   apig_MENConfiguration_tickInterval: "60"
   events.collectionQueue.size: "10000"
   events.collectionPool.minThreads: "1"
   events.collectionPool.maxThreads: "8"
gateway_destination:
   sendPolicyViolationEvent: "true"
es_destination:
   protocol: "http"
   hostName: "localhost"
   port: "9240"
   indexName: "gateway_default_analytics"
   userName: ""
   password: ""
   sendPolicyViolationEvent: "true"
key_store:
   type: JKS
   provider: SUN
   location: config/keystore.jks
   password: password
system:
   version: "10.4.0.0"
---
```

custom-settings.yml

The following shows a sample custom-settings.yml file structure.

```
---
ports:
   http: 7071
   https: 7072
   key_alias: ssos
api_gateway:
   url: http://localhost:5555
   user: Administrator
   password: password
   dir: "C:\\SoftwareAG"
   download_settings: "false"

api_endpoint:
    base_path: "/gateway"

admin_api:
    user: admin
    password: password
    admin_path: "/rest/microgateway"
downloads:
   apis: EmployeeService
   applications:
   policies:

archive:
   file: "E:/archives/gateway/EmployeeService.zip

policies:
   user_auth: internal | delegated

logging:
   level: "ERROR"
   path: "logs"

applications_sync:
  enabled: true | false
  applications_to_sync: "all | registeredApplications | comma separated ids"
  polling_interval_secs: 10
  connection_timeout_secs: 10
faults:
  default_error_message: "API Gateway encountered an error.
  Error Message: $ERROR_MESSAGE.\
    \ Request Details: Service - $SERVICE, Operation -
    $OPERATION, Invocation Time:$TIME,\
    \ Date:$DATE,  Client IP - $CLIENT_IP, User - $USER and
    Application:$CONSUMER_APPLICATION"
  native_provider_fault: "false"
extended_settings:
  apiKeyHeader: "x-Gateway-APIKey"
  apig_MENConfiguration_tickInterval: "60"
  apig_rest_service_redirect: "false"
  apig_schemaValidationPoolSize: "10"
  customCertificateHeader: "X-Client-Cert"
  decodeAllDelimitersInURI: "false"
  defaultEncoding: "UTF-8"
```

```
  defaultLanguage: "en"
  events.collectionPool.maxThreads: "8"
  events.collectionPool.minThreads: "1"
  events.collectionQueue.size: "10000"
  events.reportingPool.maxThreads: "4"
  events.reportingPool.minThreads: "2"
  events.reportingQueue.size: "5000"
  forwardInternalAPIsRequest: "false"
  pg.3pSnmpSender.sendDelay: "0"
  pg.cs.snmpTarget.base64Encoded: "false"
  pg.cs.snmpTarget.connTimeout: "0"
  pg.cs.snmpTarget.maxRequestSize: "10485760"
  pg.cs.snmpTarget.retries: "1"
  pg.cs.snmpTarget.sendTimeOut: "500"
  pg.endpoint.connectionTimeout: "30"
  pg.endpoint.readTimeout: "30"
  pg.lb.failoverOnDowntimeErrorOnly: "true"
  pg.snmp.communityTarget.base64Encoded: "false"
  pg.snmp.communityTarget.maxRequestSize: "65535"
  pg.snmp.communityTarget.retries: "1"
  pg.snmp.communityTarget.sendTimeOut: "500"
  pg.snmp.customTarget.connTimeout: "0"
  pg.snmp.userTarget.maxRequestSize: "65535"
  pg.snmp.userTarget.retries: "1"
  pg.snmp.userTarget.sendTimeOut: "500"
  pg.uddiClient.publish.maxThreads: "2"
  pg.uddiClient.uddiClientTimeout: "15000"
  pg_Cache_autoScalerRunInterval: "120"
  pg_Cache_averageObjectSize: "64"
  pg_Cache_boundedCacheResizerRunInterval: "30"
  pg_Cache_maxCacheSize: "1048576"
  pg_Cache_minCachePercent: "20"
  pg_Cache_minCacheSize: "1024"
  pg_Cache_statisticsProcessorRunInterval: "15"
  pg_JWT_isHTTPS: "true"  pg_OpenID_isHTTPS: "true"
  pg_oauth2_isHTTPS: "true"
  pg_xslt_disableDoctypeDeclarations: "true"
  pg_xslt_enableDOM: "false"
  pg_xslt_enableSecureProcessing: "true"
  pgmen.quotaSurvival.addLostIntervals: "true"
  pgmen.quotaSurvival.interval: "1"
  retainResponseStatus: "false"
  sendClientRequestURI: "false"
  setDefaultContentType: "true"
  transformerPoolSize: "5"
es_destination:
  metricsPublishInterval: "60"
  port: "9240"
  sendAuditlogAPIManagementEvent: "false"
  sendAuditlogAccessProfileManagementEvent: "false"
  sendAuditlogAdministrationEvent: "false"
  sendAuditlogAliasManagementEvent: "false"
  sendAuditlogApplicationManagementEvent: "false"
  sendAuditlogApprovalManagementEvent: "false"
  sendAuditlogGroupManagementEvent: "false"
  sendAuditlogPackageManagementEvent: "false"
  sendAuditlogPlanManagementEvent: "false"
  sendAuditlogPolicyManagementEvent: "false"
  sendAuditlogPromotionManagementEvent: "false"
  sendAuditlogRuntimeDataManagementEvent: "false"
```

```
  sendAuditlogUserManagementEvent: "false"
  sendErrorEvent: "false"
  sendLifecycleEvent: "false"
  sendPerformanceMetrics: "false"
  sendPolicyViolationEvent: "false"
gateway_destination:
  metricsPublishInterval: "60"
  sendAuditlogAPIManagementEvent: "true"
  sendAuditlogAccessProfileManagementEvent: "true"
  sendAuditlogAdministrationEvent: "true"
  sendAuditlogAliasManagementEvent: "true"
  sendAuditlogApplicationManagementEvent: "true"
  sendAuditlogApprovalManagementEvent: "true"
  sendAuditlogGroupManagementEvent: "true"
  sendAuditlogPackageManagementEvent: "true"
  sendAuditlogPlanManagementEvent: "true"
  sendAuditlogPolicyManagementEvent: "true"
  sendAuditlogPromotionManagementEvent: "true"
  sendAuditlogRuntimeDataManagementEvent: "true"
  sendAuditlogUserManagementEvent: "true"
  sendErrorEvent: "true"
  sendLifecycleEvent: "true"
  sendPerformanceMetrics: "true"
  sendPolicyViolationEvent: "true"
security_settings:
  providers:
  - !<clientMetadataMapping>
    id: "PingFederate"
    name: "PingFederate"
    type: "clientMetadataMapping"
    owner: "Administrator"
    providerName: "PingFederate"
    implNames:
      grant_types: "grantTypes"
      logo_uri: "logoUrl"
      scope: "restrictedScopes"
      client_secret: "secret"
      redirect_uris: "redirectUris"
      client_name: "name"
      client_id: "clientId"
    extendedValues: {}
    extendedValuesV2:
      - endpointType: "CLIENT_REGISTRATION"
        key: "restrictScopes"
        value: "true"
      - endpointType: "CLIENT_UPDATE"
        key: "restrictScopes"
        value: "true"
  - !<clientMetadataMapping>
    id: "OKTA"
    name: "OKTA"
    type: "clientMetadataMapping"
    owner: "Administrator"
    providerName: "OKTA"
    implNames: {}
    extendedValues: {}
    extendedValuesV2: []
  auth_servers:
  - !<authServerAlias>
    id: "local"
```

```
    name: "local"
    description: "API Gateway as an Authorization server."
    type: "authServerAlias"
    owner: "Administrator"
    tokenGeneratorConfig:
      expiry: 0
      accessTokenExpInterval: 3600
      authCodeExpInterval: 600
    authServerScopes: []
    supportedGrantTypes:
      - "authorization_code"
      - "password"
      - "client_credentials"
      - "refresh_token"
      - "implicit"
    oauthTokens: []
    authServerType: "LOCAL_IS"
service_registries:
- !<serviceRegistryAlias>
  id: "ServiceConsulDefault"
  name: "ServiceConsulDefault"
  description: "Service Consul is a tool for discovering and configuring services\

  \ in IT infrastructure."
  type: "serviceRegistryAlias"
  owner: "Administrator"
  endpointURI: "http://localhost:8500/v1"
  heartBeatInterval: 0  password: ""
  customHeaders: {}
  discoveryInfo:
    path: "/catalog/service/{serviceName}"
    httpMethod: "GET"
  registrationInfo:
    path: "/agent/service/register"
    httpMethod: "PUT"
  deRegistrationInfo:
    path: "/agent/service/deregister/{serviceId}"
    httpMethod: "PUT"
  serviceRegistryType: "SERVICE_CONSUL"
  connectionTimeout: 30
  readTimeout: 30
- !<serviceRegistryAlias>
  id: "EurekaDefault"
  name: "EurekaDefault"
  description: "Eureka is a REST based service that is primarily used in the AWS
  cloud\
  \ for locating services for the purpose of load balancing and failover of
  middle-tier servers"
  type: "serviceRegistryAlias"
  owner: "Administrator"
  endpointURI: "http://localhost:8761"
  heartBeatInterval: 0
  password: ""
  customHeaders: {}
  discoveryInfo:
    path: "/eureka/apps/{app}"
    httpMethod: "GET"
  registrationInfo:
    path: "/eureka/apps/{app}"
    httpMethod: "POST"
```

```
  deRegistrationInfo:
    path: "/eureka/apps/{app}/{instanceId}"
    httpMethod: "DELETE"
  serviceRegistryType: "EUREKA"
  connectionTimeout: 30
  readTimeout: 30
---
```

9 REST APIs

Administration API

Microgateway exposes a REST API for administration purpose. The methods of the API allow you to query the status, view the provisioned assets, and the configured settings of a running Microgateway instance.

The Administration API requires a basic authentication, if there are credentials configured in the Microgateway configuration.

GET /rest/microgateway/status

Retrieves a status message that displays the version of the Microgateway instance.

GET /rest/microgateway/settings

Retrieves the configured settings of the Microgateway instance.

GET /rest/microgateway/assets

Retrieves the provisioned assets of the specified Microgateway.