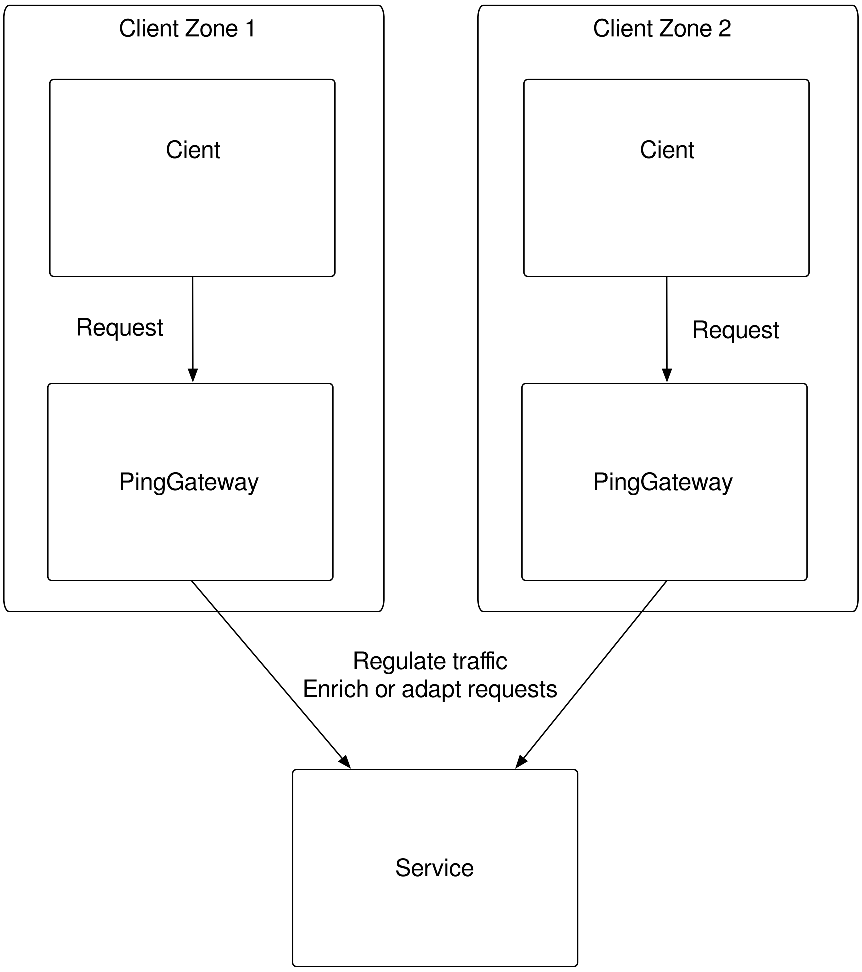
About PingGateway

PingGateway integrates web applications, APIs, and microservices with the Ping Identity Platform. PingGateway enforces security and access control in conjunction with PingAM modules.

This guide shows you an overview of PingGateway.

PingGateway as a reverse proxy

PingGateway as a reverse proxy server is an intermediate connection point between external clients and internal services. PingGateway intercepts client requests and server responses, enforcing policies, and routing and shaping traffic. The following image illustrates PingGateway as a reverse proxy:

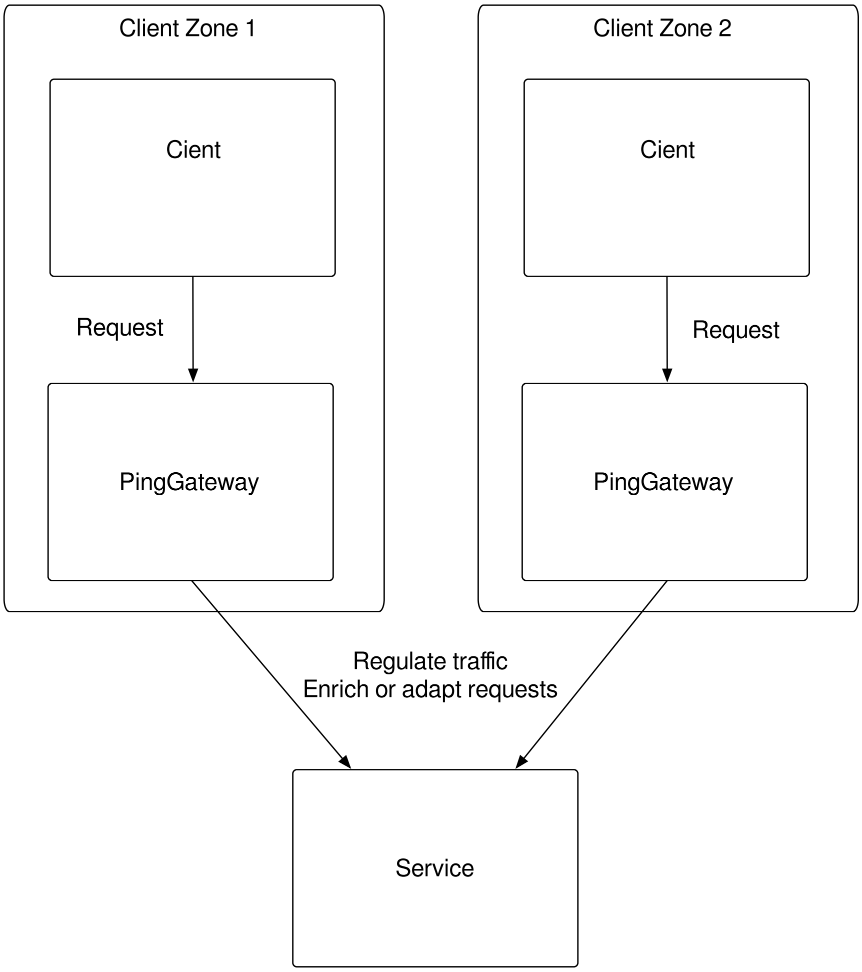


PingGateway provides the following features as a reverse proxy:

- Access management integration
- Application and API security
- Credential replay
- OAuth 2.0 support
- OpenID Connect 1.0 support
- Network traffic control
- Proxy with request and response capture
- Request and response rewriting
- SAML 2.0 federation support
- Single sign-on (SSO)

PingGateway as a forward proxy

In contrast, PingGateway as a forward proxy is an intermediate connection point between an internal client and an external service. PingGateway regulates outbound traffic to the service, and can adapt and enrich requests. The following image illustrates PingGateway as a forward proxy:



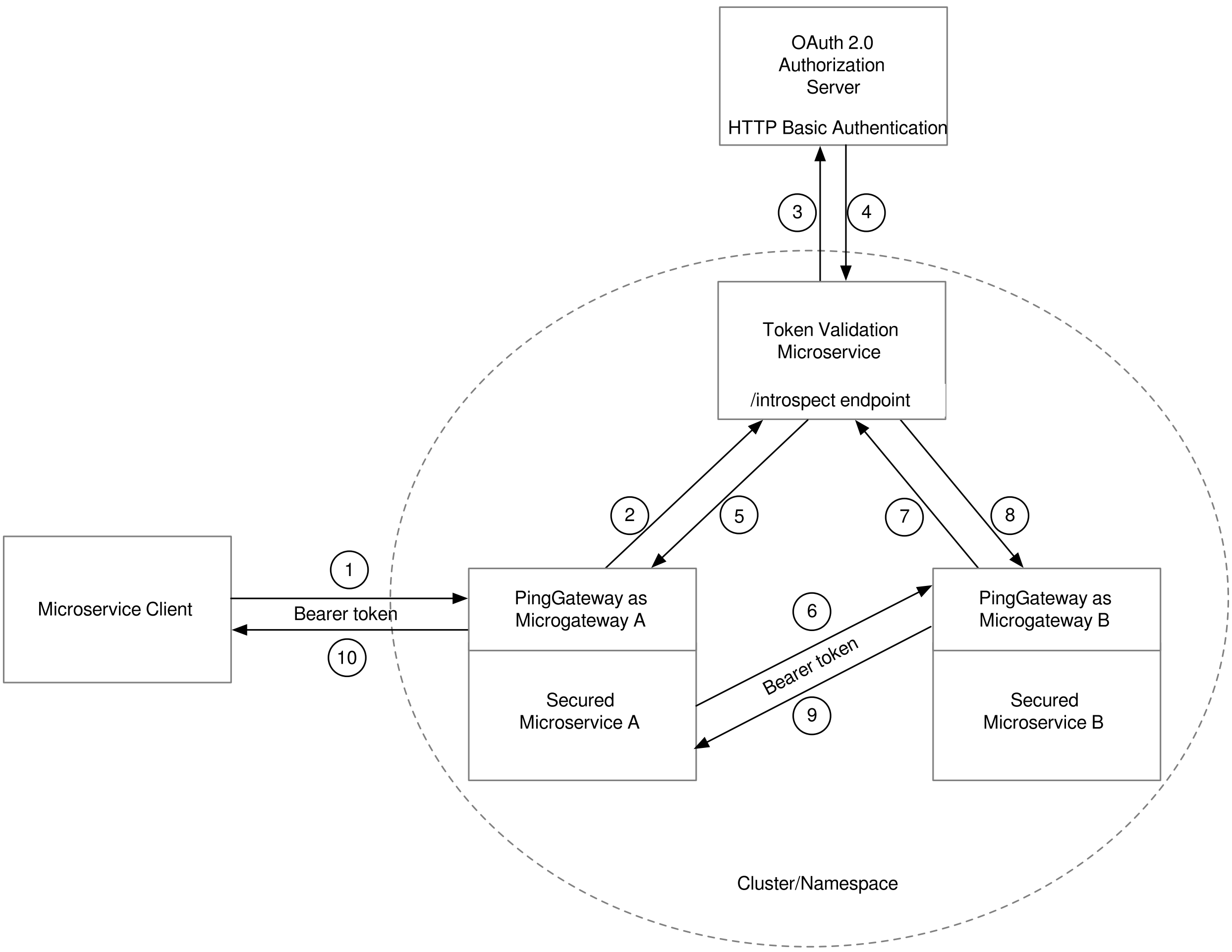
PingGateway provides the following features as a forward proxy:

- Addition of authentication or authorization to the request
- Addition of tracer IDs to the requests
- Addition or removal of request headers or scopes

PingGateway as a microgateway

PingGateway is optimized to run as a microgateway in containerized environments. Use PingGateway with business microservices to separate the security concerns of your applications from their business logic. For example, use PingGateway with the ForgeRock Token Validation Microservice to provide access token validation at the edge of your namespace.

For an example, refer to PingGateway as a microgateway. The following image illustrates the request flow in an example deployment:



PingGateway processes the request in the following steps:

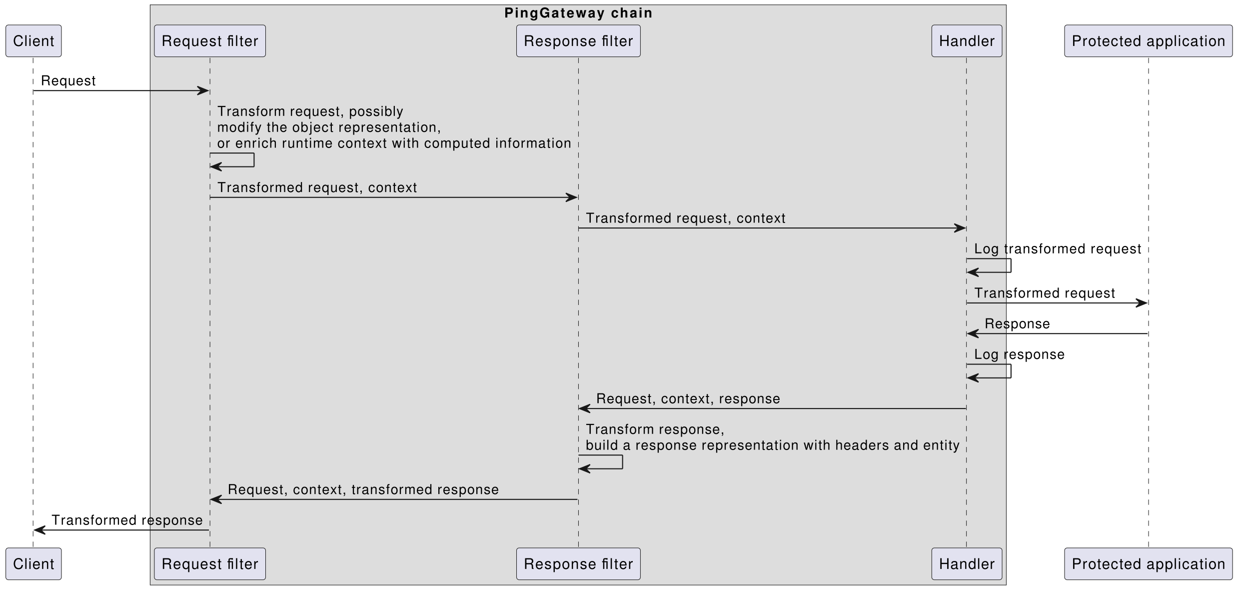
1. A client requests access to Secured Microservice A, providing a stateful OAuth 2.0 access token as credentials.
2. Microgateway A intercepts the request, and passes the access token for validation to the Token Validation Microservice, using the `/introspect` endpoint.

3. The Token Validation Microservice requests the Authorization Server to validate the token.
4. The Authorization Server introspects the token, and sends the introspection result to the Token Validation Microservice.
5. The Token Validation Microservice caches the introspection result, and sends it to Microgateway A, which forwards the result to Secured Microservice A.
6. Secured Microservice A uses the introspection result to decide how to process the request. In this case, it continues processing the request. Secured Microservice A asks for additional information from Secured Microservice B, providing the validated token as credentials.
7. Microgateway B intercepts the request, and passes the access token to the Token Validation Microservice for validation, using the `/introspect` endpoint.
8. The Token Validation Microservice retrieves the introspection result from the cache, and sends it back to Microgateway B, which forwards the result to Secured Microservice B.
9. Secured Microservice B uses the introspection result to decide how to process the request. In this case it passes its response to Secured Microservice A, through Microgateway B.
10. Secured Microservice A passes its response to the client, through Microgateway A.

Object model

PingGateway processes HTTP requests and responses by passing them through user-defined chains of filters and handlers. The filters and handlers provide access to the request and response at each step in the chain, and make it possible to alter the request or response, and collect contextual information.

The following image illustrates a typical sequence of events when PingGateway processes a request and response through a chain:



When PingGateway processes a request, it first builds an object representation of the request, including parsed query/form parameters, cookies, headers, and the entity. PingGateway initializes a runtime context to provide additional metadata about the request and applied transformations. PingGateway then passes the request representation into the chain.

In the request flow, filters modify the request representation and can enrich the runtime context with computed information. In the ClientHandler, the entity content is serialized, and additional query parameters can be encoded as described in RFC 3986: Query.

In the response flow, filters build a response representation with headers and the entity.

The route configuration in Adding headers and logging results shows the flow through a chain to a protected application.

Sessions

PingGateway uses sessions to group requests from a user agent or other source, and collect information from the requests. When multiple requests are made in the same session, the requests can share the session information. Because session sharing is not thread-safe, it is not suitable for concurrent exchanges.

The following table compares stateful and stateless sessions:

Feature	Stateful sessions	Stateless sessions
Cookie size.	Unlimited.	Maximum size of the JWT session cookie is 4 KBytes, as defined by the browser. If the cookie exceeds this size, PingGateway automatically splits it into multiple cookies.
Default name of the session cookie.	`IG_SESSIONID`.	`openig-jwt-session`.
Object types that can be stored in the session.	All types.	JSON-compatible types, such as strings, numbers, booleans, null, structures such as arrays, and list and maps containing only JSON-compatible types.
Session sharing between instances of PingGateway, for load balancing and failover.	Possible when session stickiness is configured.	Possible because the session content is a cookie on the user agent, that can be copied to multiple instances of PingGateway.
Risk of data inconsistency when simultaneous requests modify the content of a session.	Low because the session content is stored on PingGateway and shared by all exchanges. Processing is not thread-safe.	Higher because the session content is reconstructed for each request. Concurrent exchanges don't have the same content.

Stateful sessions

To configure stateful sessions, update the `session` property AdminHttpApplication (`admin.json`).

When a JwtSession isn't configured for a request, stateful sessions are created automatically. Session information is stored in the PingGateway cookie, called `IG_SESSIONID` by default. When the user agent sends a request with the cookie, the request can access the session information on PingGateway.

When a JwtSession object is configured in the route that processes a request, or in its ascending configuration (a parent route or `config.json`), the session is always stateless and can't be stateful.

When a request enters a route without a JwtSession object in the route or its ascending configuration, a stateful session is created lazily.

Session duration is defined by the `session` property in AdminHttpApplication (`admin.json`) with a default of 30 minutes.

Even if the session is empty, the session remains usable until the timeout.

Any object type can be stored in a stateful session.

Because session content is stored on PingGateway and shared by all exchanges, when PingGateway processes simultaneous requests in a stateful session there is low risk that the data becomes inconsistent. However, sessions aren't thread-safe; different requests can simultaneously read and modify a shared session.

Session information is available in SessionContext to downstream handlers and filters. For more information, refer to SessionContext.

Stateless sessions

Stateless sessions are provided when a JwtSession object is configured in `config.json` or in a route. For more information about configuring stateless sessions, refer to JwtSession.

PingGateway serializes stateless session information as JSON, stores it in a JWT that can be encrypted and then signed, and places the JWT in a cookie. The cookie contains all of the information about the session, including the session attributes as JSON, and a marker for the session timeout.

The maximum size of the JWT session cookie is 4 KBytes, as defined by the browser. If the cookie exceeds this size, PingGateway automatically splits it into multiple cookies.

Only JSON-compatible object types can be stored in stateless sessions. These object types include strings, numbers, booleans, null, structures such as arrays, and list and maps containing only JSON-compatible types.

Stateless sessions are managed as follows:

- When a request enters a route with a JwtSession object in the route or its ascending configuration, PingGateway creates the SessionContext, verifies the cookie signature, decrypts the content of the cookie, and checks that the current date is before the session timeout.
- When the request passes through the filters and handlers in the route, the request can read and modify the session content.

- When the request returns to the the point where the session was created, for example, at the entrance to a route or at `config.json`, PingGateway updates the cookie as follows:
  - If the session content has changed, PingGateway serializes the session, creates one or more new JWT session cookies with the new content, encrypts and then signs the cookies, assigns them an appropriate expiration time, and returns them in the response.
  - If the session is empty, PingGateway deletes the session, creates a new JWT session cookie with an expiration time that has already passed, and returns the cookie in the response.
  - If the session content has not changed, PingGateway does nothing.

Because the session content is stored in one or more JWT session cookies on the user agent, stateless sessions can be shared easily between PingGateway instances. The cookies are automatically carried over in requests, and any PingGateway instance can unpack and use the session content.

When PingGateway processes simultaneous requests in stateless sessions, there is a high risk that the data becomes inconsistent. This is because the session content is reconstructed for each exchange, and concurrent exchanges don’t have the same content.

PingGateway does not share sessions across requests. Instead, each request has its own session objects that it modifies as necessary, writing its own session to the session cookie regardless of what other requests do.

Session information is available in SessionContext to downstream handlers and filters. For more information, refer to SessionContext.

Session stickiness

Session stickiness helps to ensure that a client request goes to the server holding the original session data.

If data attached to a context must be stored on the server-side, configure session stickiness so that the load balancer sends all requests from the same client session to the same server.

The way you configure session stickiness and session replication depends on your load balancer.

Learn more from the example configuration in Share JWT sessions between multiple instances of PingGateway.

API descriptors

Common REST endpoints in PingGateway serve API descriptors at runtime. When you retrieve an API descriptor for an endpoint, a JSON that describes the API for that endpoint is returned.

To discover and understand APIs, use the API descriptor with a tool such as Swagger UI to generate a web page that helps you to view and test the different endpoints.

When you start PingGateway, or add or edit routes, registered endpoint locations for the routes hosted by the main router are written in `$HOME/.openig/logs/route-system.log`, where `$HOME/.openig` is the instance directory. Endpoint locations for subroutes are written to other log files. To retrieve the API descriptor for a specific endpoint, append one of the following query string parameters to the endpoint:

- `_api`, to represent the API accessible over HTTP. This OpenAPI descriptor can be used with endpoints that are complete or partial URLs.

  The returned JSON respects the OpenAPI specification and can be consumed by Swagger tools, such as Swagger UI.
- `_crestapi`, to provide a compact representation that is independent of the transport protocol. This ForgeRock® Common REST (Common REST) API descriptor can't be used with partial URLs.

  The returned JSON respects a ForgeRock proprietary specification dedicated to describe Common REST endpoints.

For more information about Common REST API descriptors, refer to Common REST API documentation.

Retrieve API descriptors for a router

IMPORTANT

Switch to development mode to retrieve these API descriptors.

With PingGateway running as described in the Quick install, run the following query to generate a JSON that describes the router operations supported by the endpoint:

```
$ curl
http://ig.example.com:8080/openig/api/system/objects/_router/route
s\?_api

{
    "swagger": "2.0",
    "info": {
```

```
        "version": "IG version",
        "title": "IG"
        },
        "host": "0:0:0:0:0:0:0:1",
        "basePath": "/openig/api/system/objects/_router/routes",
        "tags": [{
        "name": "Routes Endpoint"
        }],
        . . .
```

Alternatively, generate a Common REST API descriptor by using the `?_crestapi` query string.

Retrieve API descriptors for the UMA service

> IMPORTANT
>
> Switch to development mode to retrieve these API descriptors.

With the UMA tutorial running as described in UMA support, run the following query to generate a JSON that describes the UMA share API:

```
$ curl
http://ig.example.com:8080/openig/api/system/objects/_router/route
s/00-uma/objects/umaservice/share\?_api

{
        "swagger": "2.0",
        "info": {
        "version": "IG version",
        "title": "IG"
        },
        "host": "0:0:0:0:0:0:0:1",
        "basePath": "/openig/api/system/objects/_router/routes/00-
uma/objects/umaservice/share",
        "tags": [{
        "name": "Manage UMA Share objects"
        }],
        . . .
```

Alternatively, generate a Common REST API descriptor by using the `?_crestapi` query string.

Retrieve API descriptors for the main router

Run a query to generate a JSON that describes the API for the main router and its subsequent endpoints. For example:

```
$ curl
http://ig.example.com:8080/openig/api/system/objects/_router\?_api

{
     "swagger": "2.0",
     "info": {
     "version": "IG version",
     "title": "IG"
     },
     "host": "ig.example.com:8080",
     "basePath": "/openig/api/system/objects/_router",
     "tags": [{
     "name": "Monitoring endpoint"
     }, {
     "name": "Manage UMA Share objects"
     }, {
     "name": "Routes Endpoint"
     }],
     . . .
```

Because the above URL is a partial URL, you cannot use the `?_crestapi` query string to generate a Common REST API descriptor.

Retrieve API descriptors for PingGateway instances

Run a query to generate a JSON that describes the APIs provided by the PingGateway instance that's responding to a request. For example:

```
$ curl http://ig.example.com:8080/openig/api\?_api

{
     "swagger": "2.0",
     "info": {
     "version": "IG version",
     "title": "IG"
     },
     "host": "ig.example.com:8080",
     "basePath": "/openig/api",
     "tags": [{
     "name": "Internal Storage for UI Models"
     }, {
```

```
"name": "Monitoring endpoint"
}, {
"name": "Manage UMA Share objects"
}, {
"name": "Routes Endpoint"
}, {
"name": "Server Info"
}],
. . .
```

If routes are added after the request is performed, they aren't included in the returned JSON.

Because the above URL is a partial URL, you can't use the `?_crestapi` query string to generate a Common REST API descriptor.

Was this helpful?