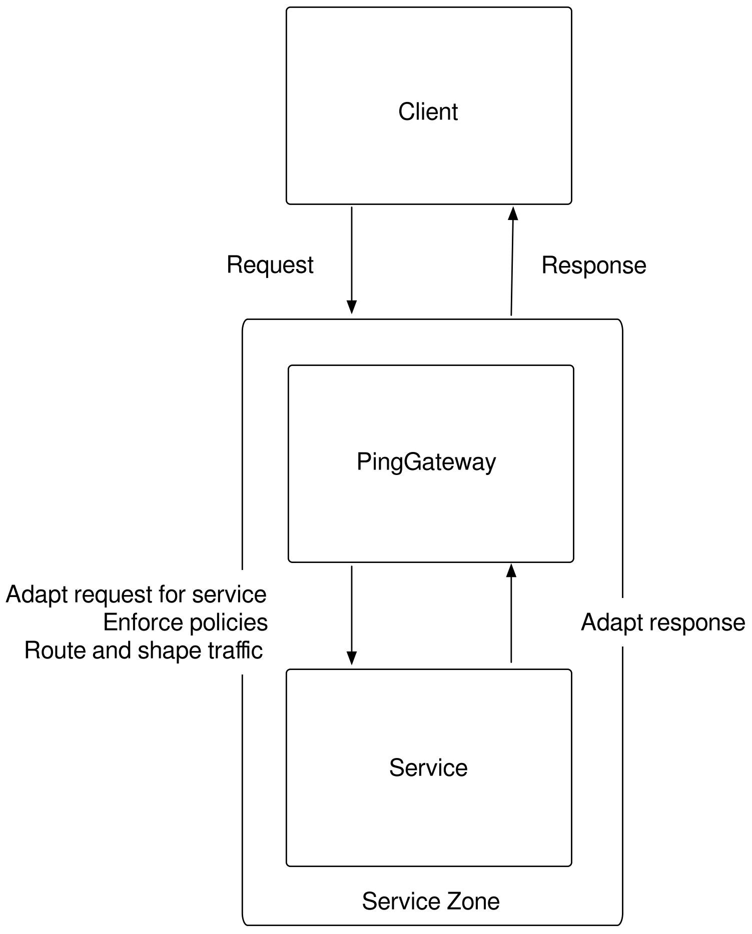
About PingGateway

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:

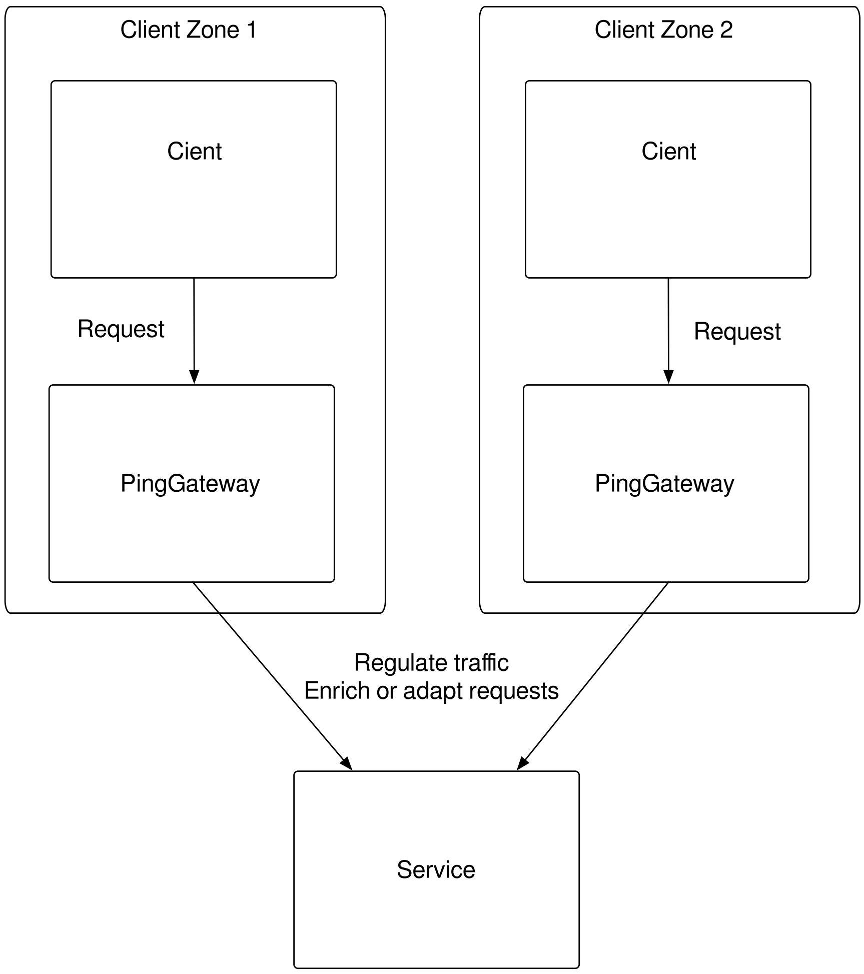


PingGateway provides the following features as a reverse proxy:

- 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:



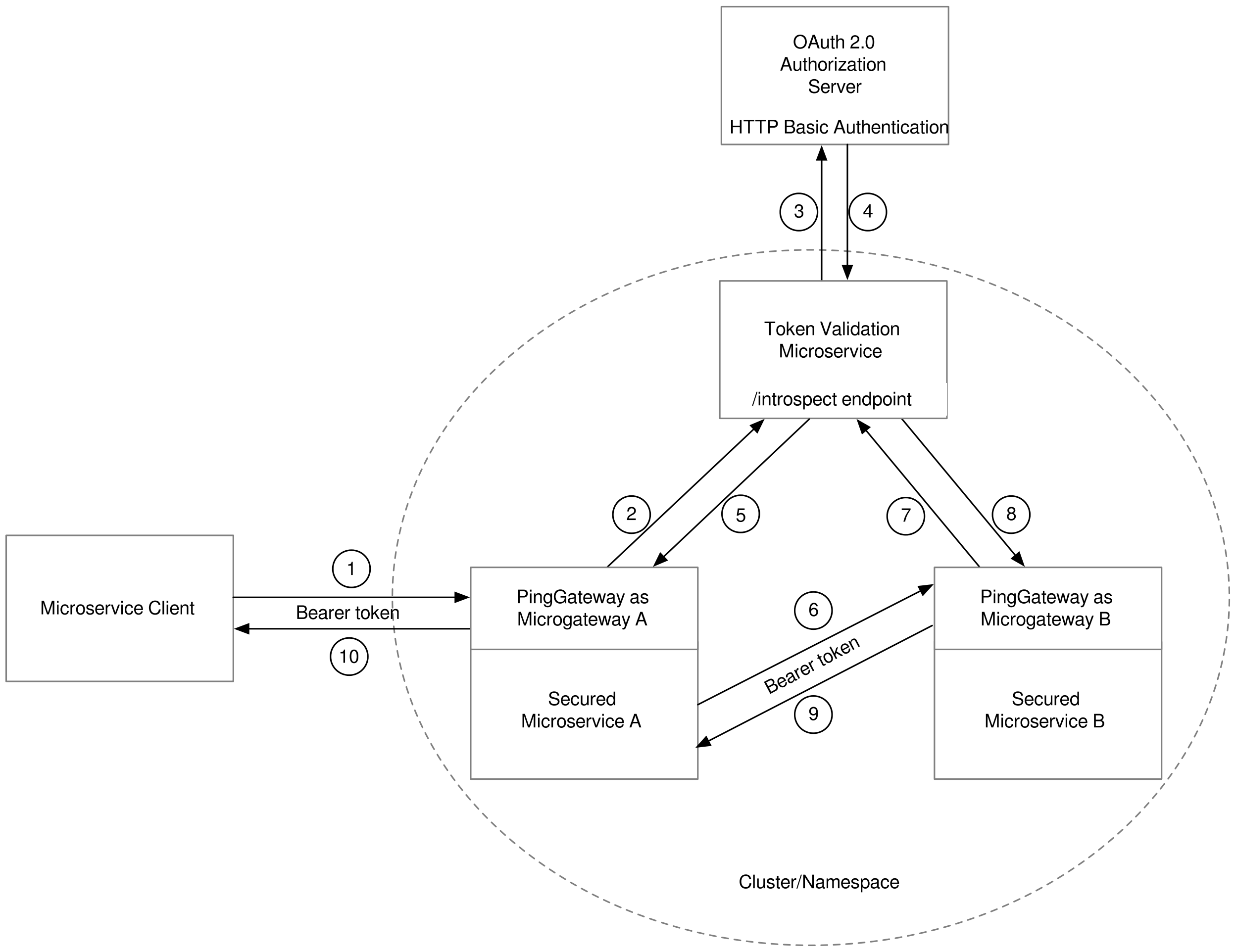
PingGateway provides the following features as a forward proxy:

- 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:

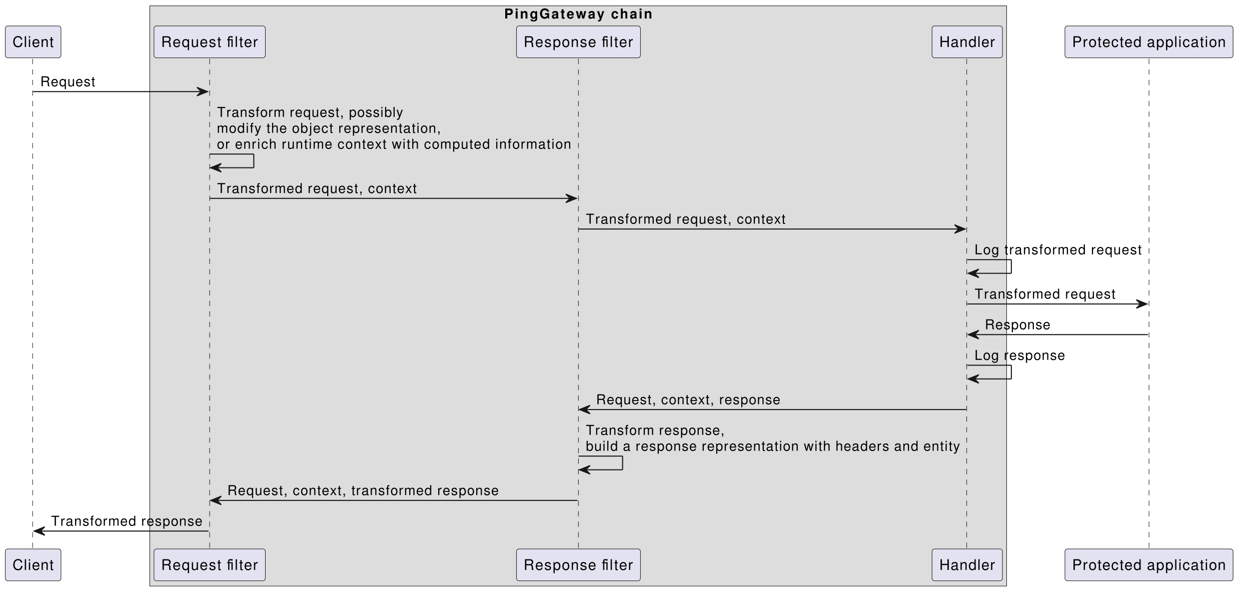
1. A client requests access to Secured Microservice A, providing a stateful OAuth 2.0 access token as credentials.
2. Microgateway A intercepts the request, and passes the access token for validation to the Token Validation Microservice, using the `/introspect` endpoint.

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:


PingGateway chain
Client
Request filter
Response filter
Handler
Protected application
Request
Transform request, possibly modify the object representation, or enrich runtime context with computed information
Transformed request, context
Transformed request, context
Log transformed request
Transformed request
Response
Log response
Request, context, response
Transform response, build a response representation with headers and entity
Request, context, transformed response
Transformed response

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 to collect and share information across the requests.

PingGateway supports stateful and stateless sessions.

- For stateful sessions, PingGateway stores the session data (default). It sets a session cookie on the user-agent that references the session.
- For stateless sessions, the user-agent stores the session data, and PingGateway puts the session data in a JWT stored as one or more session cookies on the user-agent.

Handlers and filters can access session data through the SessionContext.

Session sharing is not thread-safe, so it is not suitable for concurrent exchanges.

Stateful and stateless sessions

Feature	Stateful sessions	Stateless sessions
Cookie size	Unlimited	The maximum size of the JWT session cookie for a user-agent is 4 KBytes. If the cookie exceeds this size, PingGateway automatically splits it into multiple cookies.
Default session cookie name	`IG_SESSIONID`	`openig-jwt-session`
Data types the session can store	All types	JSON-compatible types, such as strings, numbers, booleans, null, and arrays, lists, and maps containing only JSON-compatible types.
Session sharing between PingGateway servers for load balancing and failover	Possible using session stickiness	Possible because the session is a cookie on the user-agent; PingGateway servers must share any encryption keys.
Risk of data inconsistency when simultaneous requests modify the content of a session	Low because PingGateway stores the session content for all exchanges. Processing is not thread-safe.	Higher because the session content is reconstructed from the JWT for each request. Concurrent exchanges don't have the same content.

Stateful sessions

PingGateway uses stateful sessions by default.

To configure sessions explicitly, use an InMemorySessionManager as the `"session"` in the AdminHttpApplication (`admin.json`) for administrative requests and the GatewayHttpApplication (`config.json`) or individual Route for other requests.

The default session duration is 30 minutes. Even if the session is empty, the session remains usable until the timeout.

PingGateway can store any object type in a stateful session.

Because PingGateway stores stateful session content and shares it across all exchanges, simultaneous requests can update the session with low risk of the data becoming inconsistent. Sessions aren’t thread-safe, however; different requests can simultaneously read and modify a shared session.

Stateless sessions

To configure stateless sessions, use a JwtSessionManager for the `"session"` in the AdminHttpApplication (`admin.json`) for administrative requests and the GatewayHttpApplication (`config.json`) or individual Route for other requests.

PingGateway serializes session information as a JWT that is encrypted using authenticated encryption and optionally compressed. PingGateway stores the JWT in one or more session cookies on the user-agent. The JWT contains session attributes as JSON with a marker for the session timeout:

- PingGateway can only serialize JSON-compatible types in JWT session cookies.
- The maximum size of the JWT session cookie for a user-agent is 4 KBytes. If the cookie exceeds this size, PingGateway automatically splits it into multiple cookies.
- When PingGateway serializes an empty session, it marks the supporting cookie as expired, so the user-agent effectively discards it.

  To prevent PingGateway from cleaning up empty session cookies, add information to the session context with an AssignmentFilter.

PingGateway manages stateless sessions as follows:

- When a request enters a route using stateless sessions, PingGateway creates the SessionContext, verifies the cookie signature, decrypts the content of the cookie, and checks the current date is before the session timeout.
- As the request passes through the filters and handlers in the route, the filters and handlers can read and modify the session content.
- When the request returns to the point where the session was created, such the entrance to a route or `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 the cookies using authenticated encryption, assigns them an appropriate expiration time, and returns them in the response.
  - If the session is empty, PingGateway deletes the session, creates a new expired JWT session cookie, and returns the cookie in the response.

    - If the session content has not changed, PingGateway does nothing.

Because the session content is stored on the user-agent, PingGateway servers can easily share stateless sessions. The user-agent returns the JWT cookies and any PingGateway server can unpack and use the session content.

When PingGateway updates stateless sessions in simultaneous requests, there is a high risk of the data becoming inconsistent. PingGateway reconstructs the session content for each exchange. Concurrent exchanges don't have the same content. PingGateway doesn't share sessions across requests. 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 stickiness

Session stickiness helps ensure a client request goes to the server holding the original session data. With session stickiness, a load balancer in front of multiple PingGateway servers sends all requests from the same client session to the same server.

How you configure session stickiness depends on your load balancer.

Configure session stickiness whenever PingGateway stores session data attached to a server-side context. For example, configure session stickiness when using stateful sessions and multiple PingGateway servers.

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?