The Book of Apigee Edge Antipatterns

Avoid common pitfalls, maximize the power of your APIs

Version 2.0

Contents

Introduction to Antipatterns

Everything in the universe, every phenomenon has two polar opposite expressions. The physicist Robert Dirac's equation of the electron concludes that for every electron, there is an Antielectron.   Matter therefore has its twin - the Antimatter. Unsurprisingly, Software Design **'Patterns'** have **'Antipatterns'**.

Andrew Koenig coined the word antipattern as early as 1955. Literature says that he was inspired by the Gang of Fours book, *Design Patterns*, which developed the titular concept in the software field.

Wikipedia defines a software antipattern as:

> *"In software engineering, an anti-pattern is a pattern that may be commonly used but is ineffective and/or counterproductive in practice."*

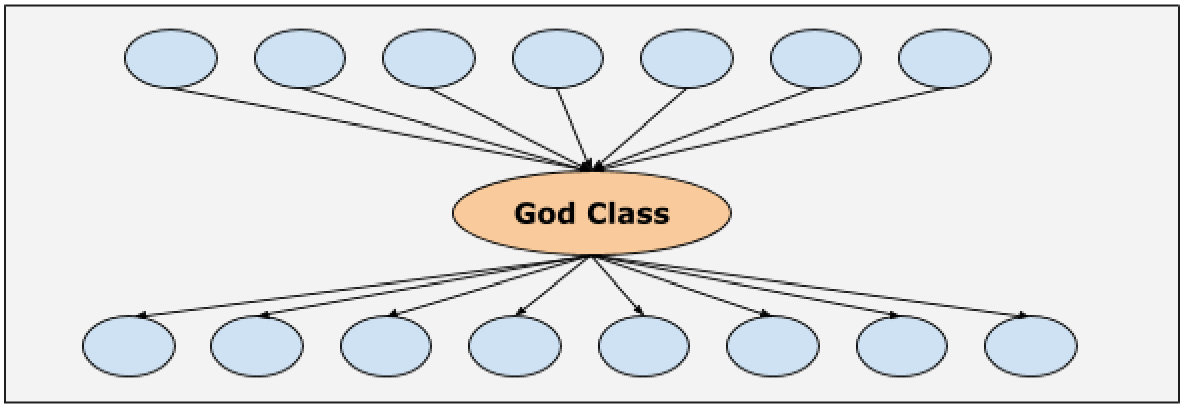
Simply put, an antipattern is something that the software allows its 'user' to do, but is something that may have adverse functional, serviceable or performance affect.

Let's take an example:

Consider the exotically named - "The God Class/Object"

In Object Oriented parlance, A God Class is a class that controls too many classes for a given application.

So, if an application has a class that has the following reference tree:



Each oval blob above represents a class. As the image illustrates, the class depicted as 'God Class' uses and references too many classes.

The framework on which the application was developed does not prevent the creation of such a class, but it has many disadvantages, the primary one's being :

1) Hard to maintain
2) Single point of failure when the application runs

Consequently, creation of such a class should be avoided. It is an antipattern.

What is this book about?

This book is about common antipatterns that are observed as part of the API proxies deployed on Apigee Edge platform.

Apigee Edge is a platform for developing and managing API proxies. Think of a proxy as an abstraction layer that "fronts" for your backend service APIs and provides value-added features like security, rate limiting, quotas, analytics, and more.

Having interacted with numerous customers using the Edge over the years, we have gathered a unique view of the different kinds of problems our customers encounter. This view has helped shape our thoughts around the do's and don'ts of API proxy development on the Edge.

This book is specifically about the don'ts on Apigee Edge - the Antipatterns.

Why did we write it?

The primary motivation was to share our learnings for the benefit of all the users and developers of Apigee Edge.

The good news is that each of these antipatterns can be clearly identified and rectified with appropriate good practices. Consequently, the APIs deployed on Edge would serve their intended purpose and be more performant.

If this book helps bring antipatterns to the forethought of API proxy developers, architects and testers, it would have served its purpose.

Antipattern Context

It is important to note that most antipatterns of evolving platforms are a point in time phenomenon. Therefore any antipatterns documented in this book may cease to remain as such with changes in the design of the platform. We will be constantly updating the book with any such changes.

It is important therefore that the latest version of the book be referred to.

Target Audience

This book would best serve 'Apigee Edge developers' as they progress through the lifecycle of designing and developing API proxies for their services. It should ideally be used as a reference guide during the API Development Lifecycle.

The other context that would be served well by the book is for troubleshooting. It would definitely be worth it for teams to quickly refer to this book, if any of the policies are not working as per documentation.

For eg. - If a team is encountering a problem with caching, they could scan the book for any antipatterns related to caching to understand if they have implemented the same and also get an idea on how to resolve the problem.

The book presupposes that the Target Audience comprises of people who have used Apigee Edge and are therefore aware of standard terminologies like proxies, policies, management services, gateway services etc. It does not venture to define or explain Apigee Edge concepts or common terminologies.
Those details can be found at : https://docs.apigee.com

Authors

Amar Devegowda
Akash Tumkur Prabhashankar
Debora Elkin
Mark Eccles
Muthukkannan Alagappan
Senthil Kumar Tamizhselvan
Uday Joshi
Venkataraghavan Lakshminarayanachar

Acknowledgements

We would like to acknowledge the significant contributions made by the following Googlers in reviewing, proofreading and shaping this book - Arghya Das, Alex Koo, Bala Kasiviswanathan, Carlos Eberhardt, Dane Knezic, Dave Newman, David Allen, Divya Achan, Gregory Brail, Greg Kuelgen, Floyd Jones, Janice Hunt, Jennifer Bennett, Joshua Norrid, Liz

Lynch, Matthew Horn, Marsh Gardiner, Mike Dunker, Mukunda Gnanasekharan, Peter Johnson, Prashanth Subrahmanyam, Rajesh Doda, Stephen Gilson, Steven Richardson, Steve Traut, Will Witman and many others.

Edge Antipatterns

1.Policy Antipatterns

1.1. Use waitForComplete() in JavaScript code

The JavaScript policy in Apigee Edge allows you to add custom code that executes within the context of an API proxy flow. For example, the custom code in the JavaScript policy can be used to:

- Get and set flow variables
- Execute custom logic and perform fault handling
- Extract data from requests or responses
- Dynamically edit the backend target URL
- Dynamically add or remove headers from a request or a response
- Parse a JSON response

HTTP Client

A powerful feature of the Javascript policy is the HTTP client. The HTTP client (or the `httpClient` object) can be used to make one or multiple calls to backend or external services. The HTTP client is particularly useful when there's a need to make calls to multiple external services and mash up the responses in a single API.

Sample JavaScript Code making a call to backend with httpClient object

```
var headers = {'X-SOME-HEADER' : 'some value' };
var myRequest = new Request("http://www.example.com","GET",headers);
var exchange = httpClient.send(myRequest);
```

The `httpClient` object exposes two methods `get` and `send` (`send` is used in the above sample code) to make HTTP requests. Both methods are asynchronous and return an `exchange` object before the actual HTTP request is completed.

The HTTP requests may take a few seconds to a few minutes. After an HTTP request is made, it's important to know when it is completed, so that the response from the request can be processed. One of the most common ways to determine when the HTTP request is complete is by invoking the `exchange` object's `waitForComplete()` method.

waitForComplete()

The `waitForComplete() method` pauses the thread until the HTTP request completes and a response (success/failure) is returned. Then, the response from a backend or external service can be processed.

Sample JavaScript code with waitForComplete()

```
var headers = {'X-SOME-HEADER' : 'some value' };
var myRequest = new Request("http://www.example.com","GET",headers);
var exchange = httpClient.send(myRequest);
// Wait for the asynchronous GET request to finish
exchange.waitForComplete();

// Get and Process the response
if (exchange.isSuccess()) {
    var responseObj = exchange.getResponse().content.asJSON;
    return responseObj.access_token;
} else if (exchange.isError()) {
    throw new Error(exchange.getError());
}
```

Antipattern

Using `waitForComplete()` after sending an HTTP request in JavaScript code will have performance implications.

Consider the following JavaScript code that calls `waitForComplete()` after sending an HTTP request.

Code for sample.js

```
// Send the HTTP request
var exchangeObj = httpClient.get("http://example.com");
// Wait until the request is completed
exchangeObj.waitForComplete();

// Check if the request was successful
if (exchangeObj.isSuccess())  {
   response = exchangeObj.getResponse();
   context.setVariable('example.status', response1.status);
} else {
  error = exchangeObj.getError();
  context.setVariable('example.error', 'Woops: ' + error);
}
```

In this example:

1. The JavaScript code sends an HTTP request to a backend API.
2. It then calls `waitForComplete()` to pause execution until the request completes.
   The `waitForComplete()` API causes the thread that is executing the JavaScript code to be blocked until the backend completes processing the request and responds back.

There's an upper limit on the number of threads (30%) that can concurrently execute JavaScript code on a Message Processor at any time. After that limit is reached, there will not be any threads available to execute the JavaScript code. So, if there are too many concurrent requests executing the `waitForComplete()` API in the JavaScript code, then subsequent requests will fail with a 500 Internal Server Error and 'Timed out' error message even before the JavaScript policy times out.

In general, this scenario may occur if the backend takes a long time to process requests and/or if there is high traffic.

Impact

1. The API requests will fail with **500 Internal Server Error** and with error message 'Timed out' when the number of concurrent requests executing `waitForComplete()` in the JavaScript code exceeds the predefined limit.

2. Diagnosing the cause of the issue can be tricky as the JavaScript fails with 'Timed out' error even though the time limit for the specific JavaScript policy has not elapsed.

Best Practice

Use callbacks in the HTTP client to streamline the callout code and improve performance and avoid using `waitForComplete()` in JavaScript code. This method ensures that the thread executing JavaScript is not blocked until the HTTP request is completed.

When a callback is used, the thread sends the HTTP requests in the JavaScript code and returns back to the pool. Because the thread is no longer blocked, it is available to handle other requests. After the HTTP request is completed and the callback is ready to be executed, a task will be created and added to the task queue. One of the threads from the pool will execute the callback based on the priority of the task.

Sample JavaScript code using Callbacks in httpClient

```
function onComplete(response,error)
 // Check if the HTTP request was successful
    if (response) {
      context.setVariable('example.status', response.status);
     } else {
      context.setVariable('example.error', 'Woops: ' + error);
     }
}
// Specify the callback Function as an argument
httpClient.get("http://example.com", onComplete);
```

Further reading

- JavaScript policy
- JavaScript callback support in httpClient for improved callouts
- Making JavaScript callouts with httpClient

1.2. Set Long Expiration time for OAuth Access and Refresh Token

Apigee Edge provides the OAuth 2.0 framework to secure APIs. OAuth2 is one of the most popular open-standard, token-based authentication and authorization schemes. It enables client applications to access APIs on behalf of users without requiring users to divulge their username and password.

Apigee Edge allows developers to generate access and/or refresh tokens by implementing any one of the four OAuth2 grant types - client credentials, password, implicit and authorization code using the OAuthV2 policy. Client applications use access tokens to consume secure APIs. Each access token has its own expiry time, which can be set in the OAuthV2 policy.

Refresh tokens are optionally issued along with access tokens with some of the grant types. Refresh tokens are used to obtain new, valid access tokens after the original access token has expired or been revoked. The expiry time for refresh tokens can also be set in OAuthV2 policy.

Antipattern

Setting a long expiration time for an Access Token and/or Refresh Token in the OAuthV2 policy leads to accumulation of OAuth tokens and increased disk space use on Cassandra nodes.

Sample OAuthV2 policy showing a long expiration time of 200 days for Refresh Tokens

```
<OAuthV2 name="GenerateAccessToken">
    <Operation>GenerateAccessToken</Operation>
    <ExpiresIn>1800000</ExpiresIn> <!-- 30 minutes -->
    <RefreshTokenExpiresIn>17280000000</RefreshTokenExpiresIn> <!--
200 days -->
    <SupportedGrantTypes>
      <GrantType>password</GrantType>
    </SupportedGrantTypes>
    <GenerateResponse enabled="true"/>
</OAuthV2>
```

In the above example:

- The Access token is set with a reasonably lower expiration time of 30 mins.
- The Refresh Token is set with a very long expiration time of 200 days.
- If the traffic to this API is 10 requests/second, then it can generate as many as 864,000 tokens in a day.
- Since the refresh tokens expire only after 200 days, they persist in the data store (Cassandra) for a long time leading to continuous accumulation.

Impact

- Leads to significant growth of disk space usage on the data store (Cassandra).

- For Private Cloud users, this could increase storage costs, or in the worst case, the disk could become full and lead to runtime errors or outage.

Best Practice

Use an appropriate lower expiration time for OAuth access and refresh tokens depending on your specific security requirements, so that they get purged quickly and thereby avoid accumulation.

Set the expiration time for refresh tokens in such a way that it is valid for a little longer period than the access tokens. For example, if you set 30 mins for access token and then set 60 mins for refresh token.

This ensures that:

- There’s ample time to use a refresh token to generate new access and refresh tokens after the access token is expired.
- The refresh tokens will expire a little while later and can get purged in a timely manner to avoid accumulation.

Further reading

- OAuth 2.0 framework
- Secure APIs with OAuth
- Edge Product Limits

1.3. Use Greedy Quantifiers in RegularExpressionProtection policy

The RegularExpressionProtection policy defines regular expressions that are evaluated at runtime on input parameters or flow variables. You typically use this policy to protect against content threats like SQL orJavaScript injection, or to check against malformed request parameters like email addresses or URLs.

The regular expressions can be defined for request paths, query parameters, form parameters, headers, XML elements (in an XML payload defined using XPath), JSON object attributes (in a JSON payload defined using JSONPath).

The following example RegularExpressionProtection policy protects the backend from SQL injection attacks:

```
<!-- /antipatterns/examples/greedy-1.xml -->
<RegularExpressionProtection async="false" continueOnError="false"
enabled="true"
  name="RegexProtection">
    <DisplayName>RegexProtection</DisplayName>
    <Properties/>
    <Source>request</Source>
    <IgnoreUnresolvedVariables>false</IgnoreUnresolvedVariables>
    <QueryParam name="query">
      <Pattern>[\s]*(?i)((delete)|(exec)|(drop\s*table)|
        (insert)|(shutdown)|(update)|(\bor\b))</Pattern>
    </QueryParam>
</RegularExpressionProtection>
```

Antipattern

The default quantifiers (`*`, `+`, `?`) are greedy in nature - they start to match with the longest possible sequence; when no match is found they backtrack gradually to try to match the pattern. If the resultant string matching the pattern is very short, then using greedy quantifiers can take more time than necessary. This is especially true if the payload is large (in the tens or hundreds of KBs).

The following example expression uses multiple instances of `.*`, which are greedy operators:

```
<Pattern>.*Exception in thread.*</Pattern>
```

In this example, the RegularExpressionProtection policy first tries to match the longest possible sequence -- the entire string. If no match is found, the policy then backtracks gradually. If the matching string is close to the start or middle of the payload, then using a greedy quantifier like `.*` can take a lot more time and processing power than reluctant qualifiers like `.*?` or (less commonly) possessive quantifiers like `.*+`.

Reluctant quantifiers (like `X*?`, `X+?`, `X??`) start by trying to match a single character from the beginning of the payload and gradually add characters. *Possessive* quantifiers (like `X?+`, `X*+`, `X++`) try to match the entire payload only once.

Given the following sample text for the above pattern:

```
Hello this is a sample text with Exception in thread with lot of
text after the Exception text.
```

Using greedy `.*` is non-performant in this case. The pattern `.*Exception in thread.*` takes 141 steps to match. If you used the pattern `.*?Exception in thread.*` (with a reluctant quantifier) instead, the evaluation takes only 55 steps.

Impact

Using greedy quantifiers like wildcards (`*`) with the RegularExpressionProtection policy can lead to:

- Increase in overall latency for API requests for a moderate payload size (up to 1MB)
- Longer time to complete the execution of the RegularExpressionProtection policy
- API requests with large payloads (>1MB) failing with 504 Gateway Timeout Errors if the predefined timeout period elapses on the Edge Router
- High CPU utilization on Message Processors due to large amount of processing which can further impact other API requests

Best Practice

- Avoid using greedy quantifiers like `.*` in regular expressions with the ReqularExpressionProtection policy. Instead, use reluctant quantifiers like `.*?` or possessive quantifiers like `.*+` (less commonly) wherever possible.

Further reading

- Regular Expression Quantifiers
- Regular Expression Protection policy

1.4. Cache Error Responses

Caching is a process of storing data temporarily in a storage area called **cache** for future reference. Caching data brings significant performance benefits because it:

- Allows faster retrieval of data
- Reduces processing time by avoiding regeneration of data again and again
- Prevents API requests from hitting the backend servers and thereby reduces the overhead on the backend servers
- Allows better utilization of system/application resources
- Improves the response times of APIs

Whenever we have to frequently access some data that doesn't change too often, we highly recommend to use a **cache** to store this data.

Apigee Edge provides the ability to store data in a cache at runtime for persistence and faster retrieval. The caching feature is made available through the PopulateCache policy, LookupCache policy, InvalidateCache policy, and ResponseCache policy.

In this section, let's look at **Response Cache** policy. The **Response Cache** policy in Apigee Edge platform allows you to cache the responses from backend servers. If the client applications make requests to the same backend resource repeatedly and the resource gets updated periodically, then we can cache these responses using this policy. The Response Cache policy helps in returning the cached responses and consequently avoids forwarding the requests to the backend servers unnecessarily.

The Response Cache policy

- Reduces the number of requests reaching the backend
- Reduces network bandwidth
- Improves API performance and response times

Antipattern

The Response Cache policy allows to cache HTTP responses with any possible Status code, by default. This means that both success and error responses can be cached. Here's a sample Response Cache policy with default configuration:

```
<ResponseCache async="false" continueOnError="false" enabled="true"
name="TargetServerResponseCache">
  <DisplayName>TargetServer ResponseCache</DisplayName>
  <CacheKey>
    <Key Fragment ref="request.uri" /></CacheKey>
    <Scope>Exclusive</Scope>
    <ExpirySettings>
      <TimeoutInSec ref="flow.variable.here">600</TimeoutInSec>
    </ExpirySettings>
  <CacheResource>targetCache</CacheResource>
</ResponseCache>
```

The Response Cache policy caches error responses in its default configuration. However, it is not advisable to cache error responses without considerable thought on the adverse implications because:

- Scenario 1: Failures occur for a temporary, unknown period and we may continue to send error responses due to caching even after the problem has been fixed
OR
- Scenario 2: Failures will be observed for a fixed period of time, then we will have to modify the code to avoid caching responses once the problem is fixed

Let’s explain this by taking these two scenarios in more detail.

Scenario 1: Temporary Backend/Resource Failure

Consider that the failure in the backend server is because of one of the following reasons:
- A temporary network glitch
- The backend server is extremely busy and unable to respond to the requests for a temporary period
- The requested backend resource may be removed/unavailable for a temporary period of time
- The backend server is responding slow due to high processing time for a temporary period, etc

In all these cases, the failures could occur for a unknown time period and then we may start getting successful responses. If we cache the error responses, then we may continue to send error responses to the users even though the problem with the backend server has been fixed.

Scenario 2: Protracted or fixed Backend/Resource Failure

Consider that we know the failure in the backend is for a fixed period of time. For instance, you are aware that either:

- A specific backend resource will be unavailable for 1 hour
  OR
- The backend server is removed/unavailable for 24 hours due to a sudden site failure, scaling issues, maintenance, upgrade, etc.

With this information, we can set the cache expiration time appropriately in the Response Cache policy so that we don't cache the error responses for a longer time. However, once the backend server/resource is available again, we will have to modify the policy to avoid caching error responses. This is because if there is a temporary/one off failure from the backend server, we will cache the response and we will end up with the problem explained in scenario 1 above.

Impact

- Caching error responses can cause error responses to be sent even after the problem has been resolved in the backend server
- Users may spend a lot of effort troubleshooting the cause of an issue without knowing that it is caused by caching the error responses from the backend server

Best Practice

1. Don't store the error responses in the response cache. Ensure that the `<ExcludeErrorResponse>` element is set to `true` in the ResponseCache policy to prevent error responses from being cached as shown in the below code snippet.

With this configuration only the responses for the default success codes 200 to 205 (unless the success codes are modified) will be cached.

```
<ResponseCache async="false" continueOnError="false"
enabled="true" name="TargetServerResponseCache">
  <DisplayName>TargetServerResponseCache</DisplayName>
  <CacheKey>
    <KeyFragment ref="request.uri" />
  </CacheKey>
  <Scope>Exclusive</Scope>
  <ExpirySettings>
    <TimeoutinSec ref="flow.variable.here">600</TimeoutinSec>
  </ExpirySettings>
  <CacheResource>targetCache</CacheResource>
  <ExcludeErrorResponse>true</ExcludeErrorResponse>
</ResponseCache>
```

2. If you have the requirement to cache the error responses for some specific reason, then you can do the following, *only if you are absolutely sure that the backend server failure is not for a brief/temporary period*:
    a. Determine the maximum/exact duration of time for which the failure will be observed (if possible).
        i. Set the Expiration time appropriately to ensure that you don't cache the error responses longer than the time for which the failure can be seen.
        ii. Use the ResponseCache policy to cache the error responses without the **<ExcludeErrorResponse>** element.
    b. **Note:** Once the failure has been fixed, remove/disable the ResponseCache policy. Otherwise, the error responses for temporary backend server failures may get cached.

It is not advisable to cache 5XX responses from the backend servers.

Further reading

- Response Cache policy

1.5. Store data greater than 512kb size in Cache

Apigee Edge provides the ability to store data in a cache at runtime for persistence and faster retrieval.

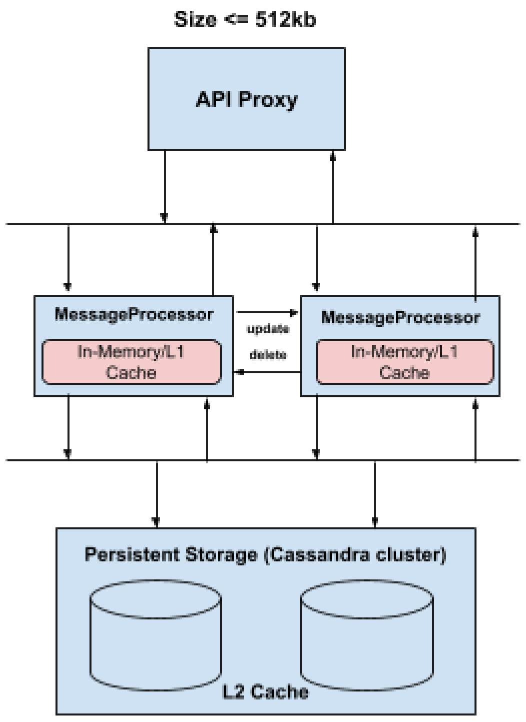
- The data is initially stored in the Message Processor's in-memory cache, referred to as **L1 cache**.
- The L1 cache is limited by the amount of memory reserved for it as a percentage of the JVM memory.
- The cached entries are later persisted in **L2 cache**, which is accessible to all message processors. More details can be found in the section below.
- The L2 cache does not have any hard limit on the number of cache entries, however the maximum **size** of the entry that can be cached is restricted to 512kb. The cache size of 512kb is the recommended size for optimal performance.

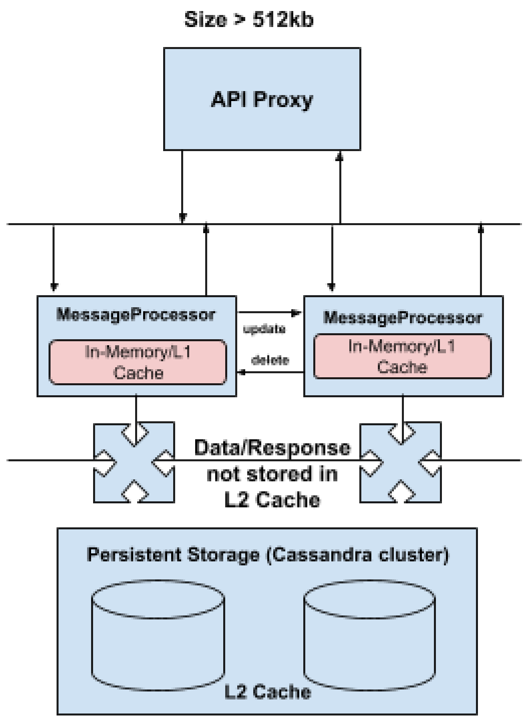
Antipattern

This particular antipattern talks about the implications of exceeding the current cache size restrictions within the Apigee Edge platform.

When data > 512kb is cached, the consequences are as follows :

- API requests executed for the first time on each of the Message Processors need to get the data independently from the original source (policy or a target server), as entries > 512kb are not available in L2 cache.
- Storing larger data (> 512kb) in L1 cache tends to put more stress on the platform resources. It results in the L1 cache memory being filled up faster and hence lesser space being available for other data. As a consequence, one will not be able to cache the data as aggressively as one would like to.
- Cached entries from the Message Processors will be removed when the limit on the number of entries is reached. This causes the data to be fetched from the original source again on the respective Message Processors.





Impact

- Data of size > 512kb will not be stored in L2/persistent cache.
- More frequent calls to the original source (either a policy or a target server) leads to increased latencies for the API requests.

Best Practice

1. It is preferred to store data of size < 512kb in cache to get optimum performance.
2. If there's a need to store data > 512kb, then consider
   a. Using any appropriate database for storing large data or
   b. Compressing the data (if feasible) and store the data only if the compressed size <= 512kb.

Further reading

- Edge Caching Internals

1.6. Log data to third party servers using JavaScript policy

Logging is one of the most efficient ways for debugging problems. The information about the API request such as headers, form params, query params, dynamic variables, etc. can all be logged for later reference. The information can be logged locally on the Message Processors (Edge for Private Cloud only) or to third party servers such as SumoLogic, Splunk, or Loggly.

Antipattern

In the code below, the `httpClient` object in a JavaScript policy is used to log the data to a Sumo Logic server. This means that the process of logging actually takes place during request/response processing. This approach is counterproductive because it adds to the processing time, thereby increasing overall latencies.

LogData_JS:

```
<Javascript async="false" continueOnError="false" enabled="true"
timelimit="2000" name="LogData_JS">
  <DisplayName>LogData_JS</DisplayName>
  <ResourceURL>jsc://LogData.js</ResourceURL>
</Javascript>
```

LogData.js:

```
var sumoLogicURL = "...";
httpClient.send(sumoLogicURL);
waitForComplete();
```

The culprit here is the call to `waitForComplete()`, which suspends the caller's operations until the logging process is complete. A valid pattern is to make this asynchronous by eliminating this call.

Impact

- Executing the logging code via the JavaScript policy adds to the latency of the API request.
- Concurrent requests can stress the resources on the Message Processor and consequently adversely affecting the processing of other requests.

Best Practice

1. Use the Message Logging policy to transfer/log data to Log servers or third party log management services such as Sumo Logic, Splunk, Loggly, etc.
2. The biggest advantage of the Message Logging policy is that it can be defined in the Post Client Flow, which gets executed after the response is sent back to the requesting client app.
3. Having this policy defined in Post Client flow helps separate the logging activity and the request/response processing thereby avoiding latencies.
4. If there is a specific requirement or a reason to do logging via javascript (not the Post Client flow option mentioned above), then it needs to be done asynchronously i.e if you are using a httpclient object, you would need to ensure that the javascript does -not- implement “waitForComplete”

Further reading

- JavaScript policy
- Message Logging policy

1.7. Invoke the MessageLogging policy multiple times in an API proxy

Apigee Edge's MessageLogging policy lets API proxy developers log custom messages to syslog or to disk (Edge for Private Cloud only). Any important information related to the API request such as input parameters, request payload, response code, error messages (if any), and so on, can be logged for later reference or for debugging. While the policy uses a background process to perform the logging, there are caveats to using the policy.

Antipattern

The MessageLogging policy provides an efficient way to get more information about an API request and debugging any issues that are encountered with the API request. However, using the same MessageLogging policy more than once or having multiple MessageLogging policies log data in chunks in the same API proxy in flows other than the PostClientFlow may have adverse implications. This is because Apigee Edge opens a connection to an external syslog server for a MessageLogging policy. If the policy uses TLS over TCP, there is additional overhead of establishing a TLS connection.

Let's explain this with the help of an example API proxy:

API proxy

In the following example, a MessageLogging policy named "LogRequestInfo" is placed in the Request flow, and another MessageLogging policy named "LogResponseInfo" is added to the Response flow. Both are in the ProxyEndpoint PreFlow. The LogRequestInfo policy executes in the background as soon as the API proxy receives the request, and the LogResponseInfo policy executes **after** the proxy has received a response from the target server but **before** the proxy returns the response to the API client. This will consume additional system resources as two TLS connections are potentially being established.

In addition, there's a MessageLogging policy named "LogErrorInfo" that gets executed only if there's an error during API proxy execution.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ProxyEndpoint name="default">
  ...
<FaultRules>
```

```
    <FaultRule name="fault-logging">
        <Step>
            <Name>LogErrorInfo</Name>
        </Step>
    </FaultRule>
</FaultRules>
<PreFlow name="PreFlow">
    <Request>
      <Step>
        <Name>LogRequestInfo</Name>
      </Step>
    </Request>
  </PreFlow>
  <PreFlow name="PreFlow">
    <Response>
      <Step>
        <Name>LogResponseInfo</Name>
      </Step>
    </Response>
  </PreFlow>
  ...
</ProxyEndpoint>
```

Message Logging policy

In the following example policy configurations, the data is being logged to third-party log servers using TLS over TCP. If more than one of these policies is used in the same API proxy, the overhead of establishing and managing TLS connections would occupy additional system memory and CPU cycles, leading to performance issues at scale.

LogRequestInfo policy

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
```

```
<MessageLogging name="LogRequestInfo">
  <Syslog>
    <Message>[3f509b58
tag="{organization.name}.{apiproxy.name}.{environment.name}"] Weather
request for WOEID {request.queryparam.w}.</Message>
    <Host>logs-01.loggly.com</Host>
    <Port>6514</Port>
    <Protocol>TCP</Protocol>
    <FormatMessage>true</FormatMessage>
    <SSLInfo>
        <Enabled>true</Enabled>
    </SSLInfo>
  </Syslog>
  <logLevel>INFO</logLevel>
</MessageLogging>
```

LogResponseInfo policy

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<MessageLogging name="LogResponseInfo">
  <Syslog>
    <Message>[3f509b58
tag="{organization.name}.{apiproxy.name}.{environment.name}"] Status:
{response.status.code}, Response {response.content}.</Message>
    <Host>logs-01.loggly.com</Host>
    <Port>6514</Port>
    <Protocol>TCP</Protocol>
    <FormatMessage>true</FormatMessage>
    <SSLInfo>
        <Enabled>true</Enabled>
    </SSLInfo>
  </Syslog>
  <logLevel>INFO</logLevel>
```

```
</MessageLogging>
```

LogErrorInfo policy

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<MessageLogging name="LogErrorInfo">
  <Syslog>
    <Message>[3f509b58
tag="{organization.name}.{apiproxy.name}.{environment.name}"] Fault
name: {fault.name}.</Message>
    <Host>logs-01.loggly.com</Host>
    <Port>6514</Port>
    <Protocol>TCP</Protocol>
    <FormatMessage>true</FormatMessage>
    <SSLInfo>
        <Enabled>true</Enabled>
    </SSLInfo>
  </Syslog>
  <logLevel>ERROR</logLevel>
</MessageLogging>
```

Impact

- Increased networking overhead due to establishing connections to the log servers multiple times during API proxy flow.
- If the syslog server is slow or cannot handle the high volume caused by multiple syslog calls, then it will cause back pressure on the message processor, resulting in slow request processing and potentially high latency or 504 Gateway Timeout errors.
- Increased number of concurrent file descriptors opened by the message processor on Private Cloud setups where file logging is used.
- If the MessageLogging policy is placed in flows other than PostClient flow, there's a possibility that the information may not be logged, as the MessageLogging policy will not be executed if any failure occurs before the execution of this policy.

In the previous ProxyEndpoint example, the information will not be logged under the following circumstances:

- If any of the policies placed ahead of the LogRequestInfo policy in the request flow fails.
  or
- If the target server fails with any error (HTTP 4XX, 5XX). In this situation, when a successful response isn't returned, the LogResponseInfo policy LogResponseInfo won't get executed.

In both of these cases, the LogErrorInfo policy will get executed and logs only the error-related information.

Best Practice

1. Use an ExtractVariables policy or JavaScript policy to set all the flow variables that are to be logged, making them available for the MessageLogging policy.

2. Use a single MessageLogging policy to log all the required data in the PostClientFlow, which gets executed unconditionally.

3. Use the UDP protocol where guaranteed delivery of messages to the syslog server is not required and TLS/SSL is not mandatory.

The MessageLogging policy was designed to be decoupled from actual API functionality, including error handling. Therefore, invoking it in the PostClientFlow, which is outside of request/response processing, means that it will always log data regardless of whether the API failed or not.

Here's an example invoking MessageLogging policy in PostClientFlow

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
 ...
<PostClientFlow>
        <Request/>
        <Response>
            <Step>
                <Name>LogInfo</Name>
            </Step>
        </Response>
</PostClientFlow>
 ...
```

Here’s an example of MessageLogging policy LogInfo that logs all the data

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<MessageLogging name="LogInfo">
  <Syslog>
    <Message>[3f509b58
tag="{organization.name}.{apiproxy.name}.{environment.name}"] Weather
request for WOEID {woeid} Status: {weather.response.code}, Response
{weather.response}, Fault: {fault.name:None}.</Message>
    <Host>logs-01.loggly.com</Host>
    <Port>6514</Port>
    <Protocol>TCP</Protocol>
    <FormatMessage>true</FormatMessage>
    <SSLInfo>
        <Enabled>true</Enabled>
    </SSLInfo>
  </Syslog>
  <logLevel>INFO</logLevel>
</MessageLogging>
```

Because response variables are not available in PostClientFlow following an Error Flow, it's important to explicitly set `woeid` and `weather.response*` variables using ExtractVariables or JavaScript policies.

Further reading

- JavaScript policy
- ExtractVariables policy
- Having code execute after proxy processing, including when faults occur, with the PostClientFlow

1.8. Configure a Non Distributed Quota

Apigee Edge provides the ability to configure the number of allowed requests to an API proxy for a specific period of time using the Quota policy.

Antipattern

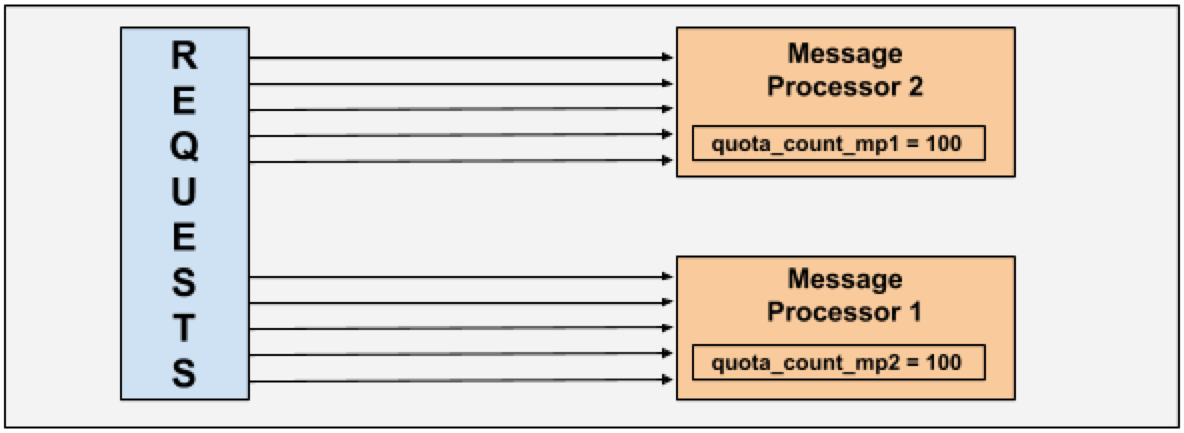
An API proxy request can be served by one or more distributed Edge components called Message Processors. If there are multiple Message Processors configured for serving API requests, then the quota will likely be exceeded because each Message Processor keeps it's own 'count' of the requests it processes.

Let's explain this with the help of an example. Consider the following Quota policy for an API proxy -

```
<!-- /antipatterns/examples/1-6.xml -->
<Quota name="CheckTrafficQuota">
  <Interval>1</Interval>
  <TimeUnit>hour</TimeUnit>
  <Allow count="100"/>
</Quota>
```

The above configuration should allow a total of 100 requests per hour.

However, in practice when multiple message processors are serving the API requests, the following happens



In the above illustration,

- The quota policy is configured to allow 100 requests per hour.
- The requests to the API proxy are being served by two Message Processors.
- Each Message Processor maintains its own quota count variable, **quota_count_mp1** and **quota_count_mp2,** to track the number of requests they are processing.
- Therefore each of the Message Processor will allow 100 API requests separately. The net effect is that a total of 200 requests are processed instead of 100 requests.

Impact

This situation defeats the purpose of the quota configuration and can have detrimental effects on the backend servers that are serving the requests.
The backend servers can:

- be stressed due to higher than expected incoming traffic
- become unresponsive to newer API requests leading to 503 errors

Best Practice

Consider, setting the `<Distributed>` element to `true` in the Quota policy to ensure that a common counter is used to track the API requests across all Message Processors. The `<Distributed>` element can be set as shown in the code snippet below:

```
<!-- /antipatterns/examples/1-7.xml -->
<Quota name="CheckTrafficQuota">
  <Interval>1</Interval>
  <TimeUnit>hour</TimeUnit>
  <Distributed>true</Distributed>
  <Allow count="100"/>
</Quota>
```

Further reading

- Quota policy

1.9. Re-use a Quota policy

Apigee Edge provides the ability to configure the number of allowed requests for an API proxy for a specific period of time using the Quota policy.

Antipattern

If a Quota policy is reused, then the quota counter will be decremented each time the Quota policy gets executed irrespective of where it is used. That is, if a Quota policy is reused:

- Within the same flow or different flows of an API proxy
- In different target endpoints of an API proxy

Then the quota counter is decremented each time it is executed and we will end up getting Quota violation errors much earlier than expected for the specified interval of time.

Let's use the following example to explain how this works.

API proxy

Let's say we have an API proxy named **"TestTargetServerQuota",** which routes traffic to two different target servers based on the resource path. And we would like to restrict the API traffic to 10 requests per minute for each of these target servers. Here's the table that depicts this scenario :

Resource Path	Target Server	Quota
/target-us	target-US.somedomain.com	10 requests per minute
/target-eu	target-EU.somedomain.com	10 requests per minute

Quota policy

Since the traffic quota is same for both the target servers, we define single quota policy named **"Quota-Minute-Target-Server"** as shown below:

```xml
<Quota name="Quota-Minute-Target-Server">
  <Interval>1</Interval>
  <TimeUnit>minute</TimeUnit>
  <Distributed>true</Distributed>
  <Allow count="10"/>
</Quota>
```

Target Endpoints

Let's use the quota policy **"Quota-Minute-Target-Server"** in the preflow of the target endpoint **"Target-US"**

```xml
<TargetEndpoint name="Target-US">
  <PreFlow name="PreFlow">
    <Request>
      <Step>
        <Name>Quota-Minute-Target-Server</Name>
      </Step>
    </Request>
  </PreFlow>
  <HTTPTargetConnection>
    <URL>http://target-us.somedomain.com</URL>
  </HTTPTargetConnection>
</TargetEndpoint>
```

And reuse the same quota policy **"Quota-Minute-Target-Server"** in the preflow of the other target endpoint **"Target-EU"** as well:

```xml
<TargetEndpoint name="Target-EU">
  <PreFlow name="PreFlow">
    <Request>
      <Step>
        <Name>Quota-Minute-Target-Server</Name>
      </Step>
    </Request>
  <Response/>
  </PreFlow>
  <HTTPTargetConnection>
    <URL>http://target-us.somedomain.com</URL>
  </HTTPTargetConnection>
</TargetEndpoint>
```

Incoming Traffic Pattern

Let's say we get a total of 10 API requests for this API proxy within the first 30 seconds in the following pattern:

Resource Path	/target-us	/target-eu	All Resources
No. of Requests	4	6	10

A little later, we get the 11th API request with the resource path as `/target-us`, let's say after 32 seconds.

We expect the request to go through successfully assuming that we still have 6 API requests for the target endpoint "target-us" as per the quota allowed.

However, in reality, we get a `Quota violation error`.
Reason: Since we are using the same quota policy in both the target endpoints, a single quota counter is used to track the API requests hitting both the target endpoints. Thus we exhaust the quota of 10 requests per minute collectively rather than for the individual target endpoint.

Impact

This antipattern can result in a fundamental mismatch of expectations, leading to a perception that the quota limits got exhausted ahead of time.

Best Practice

1. Use the `<Class>` or `<Identifier>` elements to ensure multiple, unique counters are maintained by defining a single Quota policy. Let's redefine the Quota policy "Quota-Minute-Target-Server" that we just explained in the previous section by using the header `target_id` as the `<Identifier>` for as shown below:

```
<Quota name="Quota-Minute-Target-Server">
  <Interval>1</Interval>
  <TimeUnit>minute</TimeUnit>
  <Allow count="10"/>
  <Identifier ref="request.header.target_id"/>
  <Distributed>true</Distributed>
</Quota>
```

   a. We will continue to use this Quota policy in both the target endpoints "Target-US" and "Target-EU" as before.

   b. Now let's say if the header `target_id` has a value "US" then the requests are routed to the target endpoint "Target-US".

   c. Similarly if the header `target_id` has a value "EU" then the requests are routed to the target endpoint "Target-EU".

   d. So even if we use the same quota policy in both the target endpoints, separate quota counters are maintained based on the `<Identifier>` value.

   e. Therefore, by using the `<Identifier>` element we can ensure that each of the target endpoints get the allowed quota of 10 requests.

2. Use separate Quota policy in each of the flows/target endpoints/API Proxies to ensure that you always get the allowed count of API requests. Let's now look at the same example used in the above section to see how we can achieve the allowed quota of 10 requests for each of the target end points.
   a. Define a separate Quota policy, one each for the target endpoints "Target-US" and "Target-EU"

Quota policy for Target Endpoint "Target-US"

```
<Quota name="Quota-Minute-Target-Server-US">
  <Interval>1</Interval>
  <TimeUnit>minute</TimeUnit>
  <Distributed>true</Distributed>
  <Allow count="10"/>
</Quota>
```

Quota policy for Target Endpoint "Target-EU"

```
<Quota name="Quota-Minute-Target-Server-EU">
  <Interval>1</Interval>
  <TimeUnit>minute</TimeUnit>
  <Distributed>true</Distributed>
  <Allow count="10"/>
</Quota>
```

b. Use the respective quota policy in the definition of the target endpoints as shown below:
Target Endpoint "Target-US"

```
<TargetEndpoint name="Target-US">
  <PreFlow name="PreFlow">
    <Request>
      <Step>
        <Name>Quota-Minute-Target-Server-US</Name>
      </Step>
    </Request>
    <Response/>
  </PreFlow>
  <HTTPTargetConnection>
    <URL>http://target-us.somedomain.com</URL>
  </HTTPTargetConnection>
</TargetEndpoint>
```

Target Endpont “Target-EU”

```
<TargetEndpoint name="Target-EU">
  <PreFlow name="PreFlow">
    <Request>
      <Step>
        <Name>Quota-Minute-Target-Server-EU</Name>
      </Step>
    </Request>
    <Response/>
  </PreFlow>
  <HTTPTargetConnection>
    <URL>http://target-us.somedomain.com</URL>
  </HTTPTargetConnection>
</TargetEndpoint>
```

c. Since we are using separate Quota policy in the target endpoints **“Target-US”** and **“Target-EU”,** a separate counter will be maintained. This ensures that we get the allowed quota of 10 API requests per minute for each of the target endpoints.

Use <Class> or <Identifier> element to ensure multiple, unique counters are maintained.

Further reading

- Quota policy

1.10. Use the RaiseFault policy under inappropriate conditions

The Raise Fault policy lets API developers initiate an error flow, and set error variables in a response body message and set the appropriate response status codes. You can also use the Raise Fault policy to set flow variables pertaining to the fault, such as `fault.name`, `fault.type`, and `fault.category`. Because these variables are visible in analytics data and Router access logs used for debugging, it's important to identify the fault accurately.

You can use the RaiseFault policy to treat specific conditions as errors, even if an actual error has not occurred in another policy or in the backend server of the API proxy. For example, if you want the proxy to send a custom error message to the client app whenever the backend response body contains the string `unavailable`, then you could invoke the RaiseFault policy as shown in the code snippet below:

```
<TargetEndpoint name="default">
...
  <Response>
    <Step>
      <Name>RF-Service-Unavailable</Name>
      <Condition>(message.content Like "*unavailable*")</Condition>
   </Step>
  </Response>
...
```

The RaiseFault policy's name is visible as the `fault.name` in API Monitoring and as `x_apigee_fault_policy` in the Analytics and Router access logs. This helps to diagnose the cause of the error easily.

Antipattern

Scenario 1: Using the RaiseFault policy within FaultRules after another policy has already thrown an error

Consider the example below, where an OAuthV2 policy in the API proxy flow has failed with an `InvalidAccessToken` error. Upon failure, Edge will set the `fault.name` as `InvalidAccessToken`, enter into the error flow, and execute any defined FaultRules. In the FaultRule, there is a RaiseFault policy named `RaiseFault` that sends a customized error response whenever an `InvalidAccessToken` error occurs. However, the use of RaiseFault policy in a FaultRule means that the `fault.name` variable is overwritten and masks the true cause of the failure.

```
<FaultRules>
  <FaultRule name="generic_raisefault">
    <Step>
        <Name>RaiseFault</Name>
        <Condition>(fault.name equals "invalid_access_token") or
(fault.name equals "InvalidAccessToken")</Condition>
    </Step>
  </FaultRule>
</FaultRules>
```

Scenario 2: Using RaiseFault policy in a FaultRule under all conditions

In the example below, a RaiseFault policy named `RaiseFault` executes if the `fault.name` is not `RaiseFault`:

```
<FaultRules>
    <FaultRule name="fault_rule">
        ....
        <Step>
            <Name>RaiseFault</Name>
            <Condition>!(fault.name equals "RaiseFault")</Condition>
        </Step>
    </FaultRule>
</FaultRules>
```

As in the first scenario, the key fault variables `fault.name`, `fault.code`, and `fault.policy` are overwritten with the RaiseFault policy's name. This behavior makes it almost impossible to determine which policy actually caused the failure without accessing a trace file showing the failure or reproducing the issue.

Scenario 3: Using RaiseFault policy to return a HTTP 2xx response outside of the error flow

In the example below, a RaiseFault policy named `HandleOptionsRequest` executes when the request verb is `OPTIONS`:

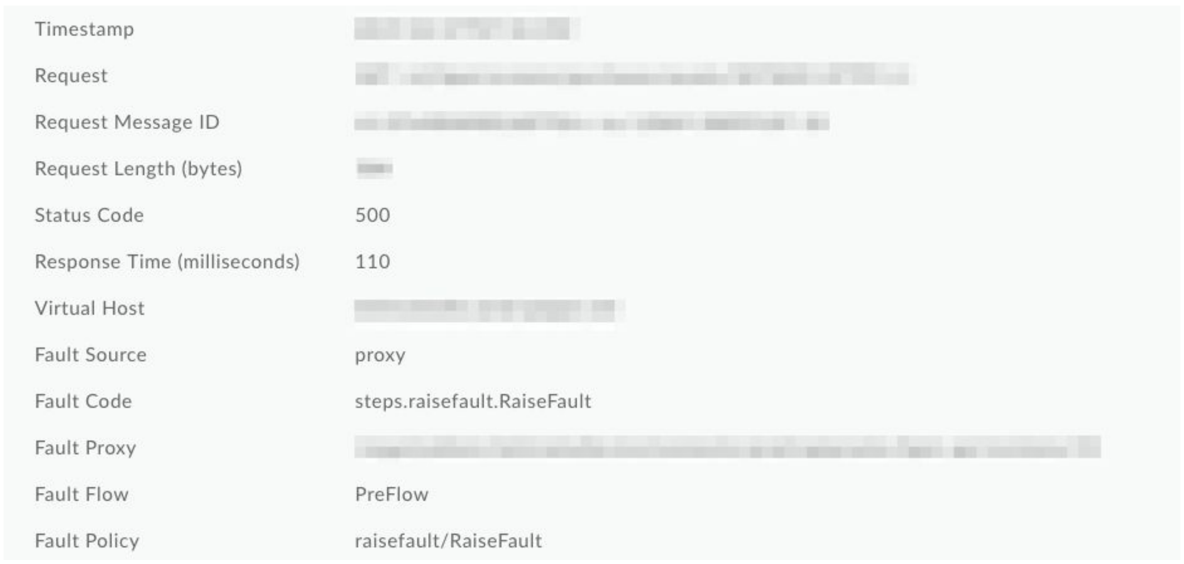
```
<PreFlow name="PreFlow">
    <Request>
        …
        <Step>
            <Name>HandleOptionsRequest</Name>
            <Condition>(request.verb Equals "OPTIONS")</Condition>
        </Step>
        …
</PreFlow>
```

The intent is to return the response to the API client immediately without processing other policies. However, this will lead to misleading analytics data because the fault variables will contain the RaiseFault policy's name, making the proxy more difficult to debug. The correct way to implement the desired behavior is to use Flows with special conditions, as described in Adding CORS support.

Impact

Use of the RaiseFault policy as described above results in overwriting key fault variables with the RaiseFault policy's name instead of the failing policy's name. In Analytics and Nginx Access logs, the `x_apigee_fault_code` and `x_apigee_fault_policy` variables are overwritten. In API Monitoring, the `Fault Code` and `Fault Policy` are overwritten. This behavior makes it difficult to troubleshoot and determine which policy is the true cause of the failure.

In the screenshot below from API Monitoring, you can see that the Fault Code and Fault policy were overwritten to generic `RaiseFault` values, making it impossible to determine the root cause of the failure from the logs:



Timestamp	[redacted]
Request	[redacted]
Request Message ID	[redacted]
Request Length (bytes)	[redacted]
Status Code	500
Response Time (milliseconds)	110
Virtual Host	[redacted]
Fault Source	proxy
Fault Code	steps.raisefault.RaiseFault
Fault Proxy	[redacted]
Fault Flow	PreFlow
Fault Policy	raisefault/RaiseFault

Best Practice

1. When an Edge policy raises a fault and you want to customize the error response message, use the AssignMessage or JavaScript policies instead of the RaiseFault policy.

2. The RaiseFault policy should be used in a non-error flow. That is, only use RaiseFault to treat a specific condition as an error, even if an actual error has not occurred in a policy or in the backend server of the API proxy. For example, you could use the RaiseFault policy to signal that mandatory input parameters are missing or have incorrect syntax.

3. You can also use RaiseFault in a fault rule if you want to detect an error during the processing of a fault. For example, your fault handler itself could cause an error that you want to signal by using RaiseFault.

Further reading

- Handling Faults
- Raise Fault policy
- Community discussion on Fault Handling Patterns

1.11. Access multi-value HTTP Headers incorrectly in an API proxy

The HTTP headers are the name value pairs that allows the client applications and backend services to pass additional information about requests and responses respectively. Some simple examples are:

- Authorization request header passes the user credentials to the server
    - `Authorization: Basic YWxhZGRpbjpvcGVuc2VzYW1l`
- Content-Type header indicates the type of the request/response content being sent
    - `Content-Type: application/json`

The HTTP Headers can have one or more values depending on the header field definitions. A multi-valued header will have comma separated values. Here are a few examples of headers that contain multiple values:

- `Cache-Control: no-cache, no-store, must-revalidate`
- `Accept: text/html, application/xhtml+xml, application/xml;q=0.9, */*;q=0.8`
- `X-Forwarded-For: 10.125.5.30, 10.125.9.125`

Note that the order of the multiple header values is important and must be preserved. For example with "X-Forwarded-For", the IP addresses are listed in order of network hops from first to last.

Apigee Edge allows the developers to access headers easily using flow variables in any of the Edge policies or conditional flows. Here are the list of variables that can be used to access a specific request or response header in Apigee Edge:

Flow variables:

- `message.header.header-name`
- `request.header.header-name`
- `response.header.header-name`
- `message.header.header-name.N`
- `request.header.header-name.N`
- `response.header.header-name.N`

Javascript objects:

- `context.proxyRequest.headers.header-name`

- `context.targetRequest.headers.header-name`
- `context.proxyResponse.headers.header-name`
- `context.targetResponse.headers.header-name`

Here’s a sample AssignMessage policy showing how to read the value of a request header and store it into a variable:

```
<AssignMessage continueOnError="false" enabled="true"
name="assign-message-default">
  <AssignVariable>
      <Name>reqUserAgent</Name>
      <Ref>request.header.User-Agent</Ref>
  </AssignVariable>
</AssignMessage>
```

Antipattern

Accessing the values of HTTP headers in Edge policies in a way that returns only the first value is incorrect, especially if the specific HTTP Header(s) has more than one value and all these values are required for processing by target servers, client applications or a policy within Apigee Edge.

The following sections contain examples of header access.

Example 1: Read a multi-valued Accept header using JavaScript code

Consider that the `Accept` header has multiple values as shown below:

```
Accept: text/html, application/xhtml+xml, application/xml
```

Here's the JavaScript code that reads the value from `Accept` header:

```
// Read the values from Accept header
var acceptHeaderValues =
context.getVariable("request.header.Accept");
```

The above JavaScript code returns only the first value from the `Accept` header, such as `text/html`.

Example 2: Read a multi-valued Access-Control-Allow-Headers header in AssignMessage or RaiseFault policy

Consider that the `Access-Control-Allow-Headers` header has multiple values as shown below:

```
Access-Control-Allow-Headers: content-type, authorization
```

Here's the part of code from AssignMessage or RaiseFault policy setting the `Access-Control-Allow-Headers` header:

```
<Set>
  <Headers>
    <Header
name="Access-Control-Allow-Headers">{request.header.Access-Control-Re
quest-Headers}</Header>
  </Headers>
</Set>
```

The above code sets the Header `Access-Control-Allow-Headers` with only the first value from the request header `Access-Control-Allow-Headers`, in this example `content-type`.

Impact

- In both the examples above, notice that only the first value from multi-valued headers are returned. If these values are subsequently used by another policy in the API proxy flow or by the backend service to perform some function or logic, then it could lead to an unexpected outcome or result.
- When request header values are accessed and passed onto the target server, API requests could be processed by the backend incorrectly and hence they may give incorrect results.

- If the client application is dependent on specific header values from the Edge response, then it may also process incorrectly and give incorrect results.

Best Practice

1. Use the appropriate built-in flow variables:
   `request.header.header_name.values.count`,
   `request.header.header_name.N`,
   `response.header.header_name.values.count`,
   `response.header.header_name.N`.

   Then iterate to fetch all the values from a specific header in JavaScript or Java callout policies.

   Example: Sample JavaScript code to read a multi-value header

   ```
   for (var i = 1; i
   <=context.getVariable('request.header.Accept.values.count');
   i++)
   {
     print(context.getVariable('request.header.Accept.' + i));
   }
   ```

   Note: Edge splits the header values considering comma as a delimiter and not through other delimiters like semicolon etc.

   For example "application/xml;q=0.9, */*;q=0.8" will appear as one value with the above code.

   If the header values need to be split using semicolon as a delimiter, then use `string.split(";")` to separate these into values.

2. Use the `substring()` function on the flow variable `request.header.header_name.values` in RaiseFault or AssignMessage policy to read all the values of a specific header.

   Example: Sample RaiseFault or AssignMessage policy to read a multi-value header

```
<Set>
  <Headers>
   <Header
name="Access-Control-Allow-Headers">{substring(request.header.Ac
cess-Control-Request-Headers.values,1,-1)}</Header>
  </Headers>
</Set>
```

Further reading

- How to handle multi-value headers in Javascript?
- List of HTTP header fields
- Request and response variables
- Flow Variables

1.12. Use Service Callout policy to invoke a backend service in a No Target API proxy

An API proxy is a managed facade for backend services. At its simplest, an API proxy consists of a ProxyEndpoint (defining the URL of the API) and a TargetEndpoint (defining the URL of the backend service).

Apigee Edge offers a lot of flexibility for building sophisticated behaviour on top of this pattern. For instance, you can add policies to control the way the API processes a client request before sending it to the backend service, or manipulate the response received from the backend service before forwarding it to the client. You can invoke other services using service callout policies, add custom behaviour by adding Javascript code and even create an API proxy that doesn't invoke a backend service.

Antipattern

Using service callouts to invoke a backend service in an API proxy with no routes to a target endpoint is technically feasible, but will result in the loss of analytics data about the performance of the external service.

An API proxy that does not contain target routes can be useful in cases where the request message does not need to be forwarded to the TargetEndpoint. Instead, the ProxyEndpoint performs all of the necessary processing. For example, the ProxyEndpoint could retrieve data from a lookup to the API Services's key/value store and return the response without invoking a backend service.

You can define a null Route in an API proxy, as shown here:

```
<RouteRule name="noroute"/>
```

A proxy using a null route is a "no target" proxy, because it does not invoke a target backend service.

It is technically possible to add a service callout to a no target proxy to invoke an external service, as shown in the example below:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ProxyEndpoint name="default">
    <Description/>
    <FaultRules/>
    <PreFlow name="PreFlow">
        <Request>
            <Step>
                <Name>ServiceCallout-InvokeBackend</Name>
            </Step>
        </Request>
        <Response/>
    </PreFlow>
    <PostFlow name="PostFlow">
        <Request/>
        <Response/>
    </PostFlow>
    <Flows/>
    <HTTPProxyConnection>
        <BasePath>/no-target-proxy</BasePath>
        <Properties/>
        <VirtualHost>secure</VirtualHost>
    </HTTPProxyConnection>
    <RouteRule name="noroute"/>
</ProxyEndpoint>
```

However, the proxy can not provide analytics information about the external service behaviour (such as processing time or error rates), making it difficult to assess the performance of the target server.

Impact

- Analytics information on the interaction with the external service (error codes, response time, target performance, etc.) is unavailable.
    -

- Any specific logic required before or after invoking the service callout is included as part of the overall proxy logic, making it harder to understand and reuse.

Best Practice

If an API proxy interacts with only a single external service, the API proxy should follow the basic design pattern, where the backend service is defined as the target endpoint of the API proxy. A proxy with no routing rules to a target endpoint should not invoke a backend service using the ServiceCallout policy.

The following proxy configuration implements the same behaviour as the example above, but follows best practices:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ProxyEndpoint name="default">
    <Description/>
    <FaultRules/>
    <PreFlow name="PreFlow">
        <Request/>
        <Response/>
    </PreFlow>
    <PostFlow name="PostFlow">
        <Request/>
        <Response/>
    </PostFlow>
    <Flows/>
    <HTTPProxyConnection>
        <BasePath>/simple-proxy-with-route-to-backend</BasePath>
        <Properties/>
        <VirtualHost>secure</VirtualHost>
    </HTTPProxyConnection>
    <RouteRule name="default">
        <TargetEndpoint>default</TargetEndpoint>
    </RouteRule>
</ProxyEndpoint>
```

Use Service callouts to support mashup scenarios, where you want to invoke external services before or after invoking the target endpoint. Service callouts are not meant to replace target endpoint invocation.

Further reading

- Understanding APIs and API proxies
- How to configure Route Rules
- Null Routes
- Service Callout policy

2. Performance Antipatterns

2.1. Leave unused NodeJS API Proxies deployed

One of the unique and useful features of Apigee Edge is the ability to wrap a NodeJS application in an API proxy. This allows developers to create event-driven server-side applications using Edge.

Antipattern

Deployment of API Proxies is the process of making them available to serve API requests. Each of the deployed API Proxies is loaded into Message Processor's runtime memory to be able to serve the API requests for the specific API proxy. Therefore, the runtime memory usage increases with the increase in the number of deployed API Proxies. Leaving any unused API Proxies deployed can cause unnecessary use of runtime memory.

In the case of NodeJS API Proxies, there is a further implication.

The platform launches a "Node app" for every deployed NodeJS API proxy. A Node app is akin to a standalone node server instance on the Message Processor JVM process.
In effect, for every deployed NodeJS API proxy, Edge launches a node server each, to process requests for the corresponding proxies. If the same NodeJS API proxy is deployed in multiple environments, then a corresponding node app is launched for each environment. In situations where there are a lot of deployed but unused NodeJS API Proxies, multiple Node apps are launched. Unused NodeJS proxies translate to idle Node apps which consume memory and affect start up times of the application process.

Used Proxies			**Unused Proxies**		
No. of proxies	**Number of deployed envs**	**Number of nodeapps launched**	**No. of proxies**	**Number of deployed envs**	**Number of nodeapps launched**

10	Dev,test,prod (3)	10x3=30	12	Dev,test,prod (3)	12x3=36

In the illustration above, 36 unused nodeapps are launched, which uses up system memory and has an adverse effect on start up times of the process.

Impact

- High Memory usage and cascading effect on application’s ability to process further requests.

- Likely Performance impact on the API Proxies that are actually serving traffic.

Best Practice

Undeploy any unused API Proxies.

You can use Analytics Proxy Performance dashboard to determine which proxies are not serving traffic for a certain period of time, then you can choose to undeploy them if they don't need to be deployed.

Further reading

- [Overview of Node.js on Apigee Edge]
- [Getting started with Node.js on Apigee Edge]
- [Debugging and troubleshooting Node.js proxies]

3.Generic Antipatterns

3.1. Invoke Management API calls from an API proxy

Edge has a powerful utility called “management APIs” which offers services such as:

- Deploying or undeploying API Proxies
- Configuring virtual hosts, keystores and truststores, etc.
- Creating, deleting and/or updating entities such as KeyValueMaps, API Products, Developer Apps, Developers, Consumer Keys, etc.
- Retrieving information about these entities

These services are made accessible through a component called **“Management Server”** in the Apigee Edge platform. These services can be invoked easily with the help of simple management API calls.

Sometimes we may need to use one or more of these services from API Proxies at runtime. This is because the entities such as KeyValueMaps, OAuth Access Tokens, API Products, Developer Apps, Developers, Consumer Keys, etc. contain useful information in the form of key-value pairs, custom attributes or as part of its profile. For instance, you can store the following information in KeyValueMap to make it more secure and accessible at runtime:

- Back-end target URLs
- Environment properties
- Security credentials of backend or third party systems etc

Similarly, you may want to get the list of API Products or developer’s email address at runtime. This information will be available as part of the Developer Apps profile.

All this information can be effectively used at runtime to enable dynamic behaviour in policies or custom code within Apigee Edge.

Antipattern

The management APIs are preferred and useful for administrative tasks and should not be used for performing any runtime logic in API Proxies flow. This is because:

- Using management APIs to access information about the entities such as KeyValueMaps, OAuth Access Tokens or for any other purpose from API Proxies leads to dependency on Management Servers.
- Management Servers are not a part of Edge runtime component and therefore, they may not be highly available.
- Management Servers also may not be provisioned within the same network or data center and may therefore introduce network latencies at runtime.
- The entries in the management servers are cached for longer period of time, so we may not be able to see the latest data immediately in the API Proxies if we perform writes and reads in a short period of time.
- Increases network hops at runtime.

In the code sample below, management API call is made via the custom JavaScript code to retrieve the information from the KeyValueMap.

Code Sample : Accessing the KeyValueMap entries from within the JavaScript code using Management API

```
var response =
httpClient.send('https://api.enterprise.apigee.com/v1/o/org_name/e/env_name/keyvaluemap
s/kvm_name');
```

If the management server is unavailable, then the JavaScript code invoking the management API call fails. This subsequently causes the API request to fail.

Impact

- Introduces additional dependency on Management Servers during runtime. Any failure in Management servers will affect the API calls.
- User credentials for management APIs need to be stored either locally or in some secure store such as Encrypted KVM.

- Performance implications owing to invoking the management service over the network.
- May not see the updated values immediately due to longer cache expiration in management servers.

Best practice

There are more effective ways of retrieving information from entities such as KeyValueMaps, API Products, DeveloperApps, Developers, Consumer Keys, etc. at runtime. Here are a few examples:

- Use KeyValueMapOperations policy to access information from KeyValueMaps. Here's a sample code that shows how to retrieve information from the KeyValueMap:

  ```
  <KeyValueMapOperations mapIdentifier="urlMap" async="false"
      continueOnError="false" enabled="true" name="GetURLKVM">
    <DisplayName>GetURLKVM</DisplayName>
    <ExpiryTimeInSecs>86400</ExpiryTimeInSecs>
    <Scope>environment</Scope>
    <Get assignTo="urlHost1" index="2">
      <Key>
        <Parameter>urlHost_1</Parameter>
      </Key>
    </Get>
  </KeyValueMapOperations>
  ```

- To access information about API Products, Developer Apps, Developers, Consumer Keys, etc. in the API proxy, you can do either of the following:
  - If your API proxy flow has a **VerifyAPIKey** policy, then you can access the information using the flow variables populated as part of this policy. Here is a sample code that shows how to retrieve the name and created_by information of a Developer App using JavaScript:

```
print("Application Name ",
context.getVariable(""verifyapikey.
VerifyAPIKey.app.name"));
print("Created by:", context.getVariable("verifyapikey.
VerifyAPIKey.app.created_by"));
```

- If If your API proxy flow doesn't have a VerifyAPIKey policy, then you can access the profiles of API Products, Developer Apps, etc. using the Access Entity and Extract Variables policies:
  - Retrieve the profile of DeveloperApp with the AccessEntity policy:

Code Sample : Retrieve the DeveloperApp profile using AccessEntity policy

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<AccessEntity async="false" continueOnError="false"
enabled="true" name="GetDeveloperApp">
  <DisplayName>GetDeveloperApp</DisplayName>
  <EntityType value="app"></EntityType>
  <EntityIdentifier ref="developer.app.name"
type="appname"/>
  <SecondaryIdentifier ref="developer.id"
type="developerid"/>
</AccessEntity>
```

  - Extract the `appId` from DeveloperApp with the ExtractVariables policy:

Code Sample : Extract the appId from DeveloperApp using ExtractVariables policy

```
<ExtractVariables name="Extract-Developer App-Info">
  <!--
    The source element points to the variable populated by
AccessEntity policy.
    The format is <policy-type>.<policy-name>
    In this case, the variable contains the whole
developer profile.
  -->
  <Source>AccessEntity.GetDeveloperApp"</Source>
  <VariablePrefix>developerapp</VariablePrefix>
  <XMLPayload>
    <Variable name="appld" type="string">
      <!-- You parse elements from the developer profile
using XPath. -->
      <XPath>/App/AppId</XPath>
    </Variable>
  </XMLPayload>
</ExtractVariables>
```

Further reading

- Key Value Map Operations policy
- VerifyAPIKey policy
- Access Entity policy

3.2. Invoke a Proxy within Proxy using custom code or as a Target

Edge allows you to invoke one API proxy from another API proxy. This feature is useful especially if we have an API proxy that contains reusable code that can be used by other API Proxies.

Antipattern

Invoking one API proxy from another either using HTTPTargetConnection in the target endpoint or custom JavaScript code results in additional network hop.

In the code sample 1 below, we are invoking the Proxy 2 from Proxy 1 using HTTPTargetConnection

Invoke Proxy 2 from Proxy 1 using HTTPTargetConnection

The following code sample invokes Proxy 2 from Proxy 1 using HTTPTargetConnection:

```
<HTTPTargetConnection>
  <URL>http://myorg-test.apigee.net/proxy2</URL>
</HTTPTargetConnection>
```

Invoke the Proxy 2 from Proxy 1 from the JavaScript code

The next code sample invokes Proxy 2 from Proxy 1 using JavaScript:

```
var response =
httpClient.send('http://myorg-test.apigee.net/proxy2');
```

Code Flow

To understand why this has an inherent disadvantage, we need to understand the route a request takes as illustrated by the diagram below:

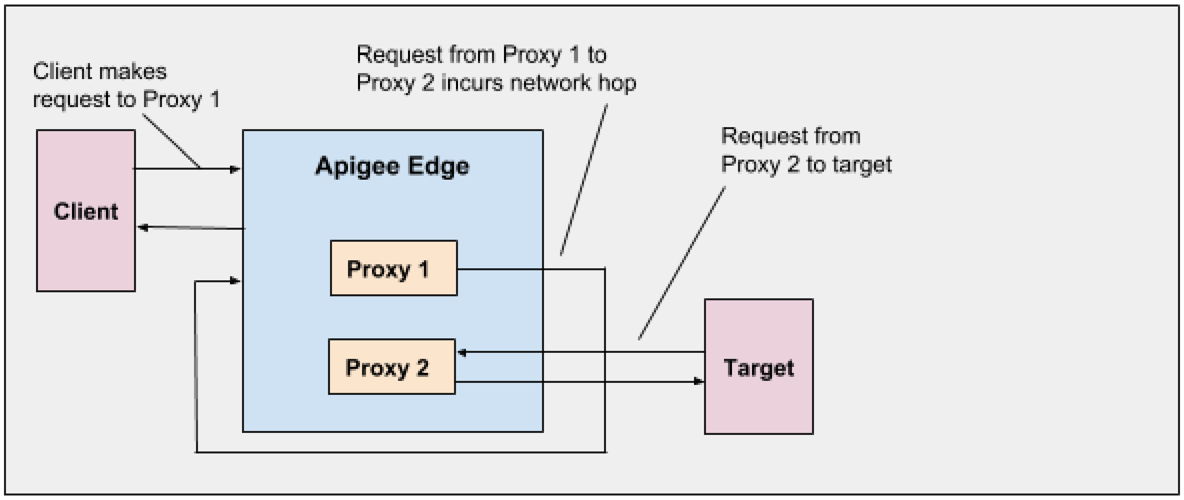


Figure 1: **Code Flow**

As depicted in the diagram, a request traverses multiple distributed components, including the Router and the Message Processor.

In the code samples above , invoking Proxy 2 from Proxy 1 means that the request has to be routed through the traditional route i.e Router > MP, at runtime. This would be akin to invoking an API from a client thereby making multiple network hops that add to the latency. These hops are unnecessary considering that Proxy 1 request has already ‘reached’ the MP.

Impact

Invoking one API proxy from another API proxy incurs unnecessary network hops, that is the request has to be passed on from one Message Processor to another Message Processor.

Best Practice

1. Use the proxy chaining feature for invoking one API proxy from another. Proxy chaining is more efficient as it uses local connection to reference the target endpoint (another API proxy).

   The code sample shows proxy chaining using LocalTargetConnection:

   ```
   <LocalTargetConnection>
     <APIProxy>proxy2</APIProxy>
     <ProxyEndpoint>default</ProxyEndpoint>
   </LocalTargetConnection>
   ```

   The invoked API proxy gets executed within the same Message Processor; as a result, it avoids the network hop as shown in the following figure:

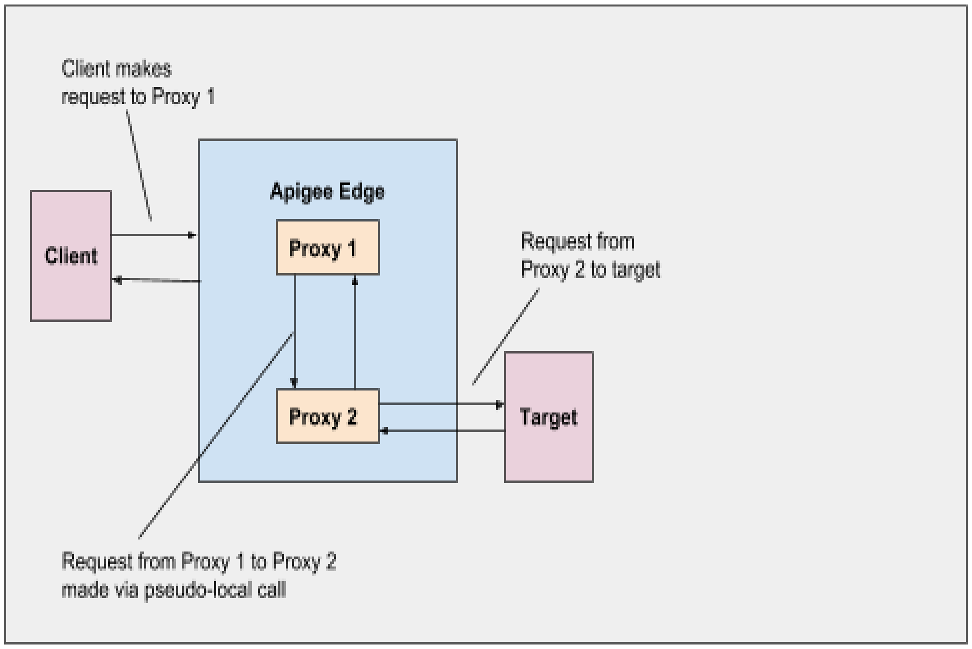

Client makes request to Proxy 1
Apigee Edge
Client
Proxy 1
Request from Proxy 2 to target
Proxy 2
Target
Request from Proxy 1 to Proxy 2 made via pseudo-local call

Figure 2: Code Flow with Proxy Chaining

Further reading

- Proxy Chaining

3.3. Manage Edge Resources without using Source Control Management

Apigee Edge provides many different types of resources and each of them serve a different purpose. There are certain resources that can be configured (i.e., created, updated, and/or deleted) only through the Edge UI, management APIs, or tools that use management APIs, and by users with the prerequisite roles and permissions. For example, only org admins belonging to a specific organization can configure these resources. That means, these resources cannot be configured by end users through developer portals, nor by any other means. These resources include:

- API proxies
- Shared flows
- API products
- Caches
- KVMs
- Keystores and truststores
- Virtual hosts
- Target servers
- Resource files

While these resources do have restricted access, if any modifications are made to them even by the authorized users, then the historic data simply gets overwritten with the new data. This is due to the fact that these resources are stored in Apigee Edge only as per their current state. The main exceptions to this rule are API proxies and shared flows.

API Proxies and Shared Flows under Revision Control

API proxies and shared flows are managed -- in other words, created, updated and deployed -- through revisions. Revisions are sequentially numbered, which enables you to add new changes and save it as a new revision or revert a change by deploying a previous revision of the API proxy/Shared Flow. At any point in time, there can be only one revision of an API proxy/Shared Flow deployed in an environment unless the revisions have a different base path.

Although the API proxies and shared flows are managed through revisions, if any modifications are made to an existing revision, there is no way to roll back since the old changes are simply overwritten.

Audits and History

Apigee Edge provides the Audits and API, Product, and organization history features that can be helpful in troubleshooting scenarios. These features enable you to view information like who performed specific operations (create, read, update, delete, deploy, and undeploy) and when the operations were performed on the Edge resources. However, if any update or delete operations are performed on any of the Edge resources, the audits cannot provide you the older data.

Antipattern

Managing the Edge resources (listed above) directly through Edge UI or management APIs without using source control system

There's a misconception that Apigee Edge will be able to restore resources to their previous state following modifications or deletes. However, Edge Cloud does not provide restoration of resources to their previous state. Therefore, it is the user’s responsibility to ensure that all the data related to Edge resources is managed through source control management, so that old data can be restored back quickly in case of accidental deletion or situations where any change needs to be rolled back. This is particularly important for production environments where this data is required for runtime traffic.

Let’s explain this with the help of a few examples and the kind of impact that can be caused if the data in not managed through a source control system and is modified/deleted knowingly or unknowingly:

Example 1: Deletion or modification of API proxy

When an API proxy is deleted, or a change is deployed on an existing revision, the previous code won’t be recoverable. If the API proxy contains Java, JavaScript, Node.js, or Python code that is not managed in a source control management (SCM) system outside Apigee, a lot of development work and effort could be lost.

Example 2: Determination of API proxies using specific virtual hosts

A certificate on a virtual host is expiring and that virtual host needs updating. Identifying which API proxies use that virtual host for testing purposes may be difficult if there are many API proxies. If the API proxies are managed in an SCM system outside Apigee, then it would be easy to search the repository.

Example 3: Deletion of keystore/truststore

If a keystore/truststore that is used by a virtual host or target server configuration is deleted, it will not be possible to restore it back unless the configuration details of the keystore/truststore, including certificates and/or private keys, are stored in source control.

Impact

- If any of the Edge resources are deleted, then it's not possible to recover the resource and its contents from Apigee Edge.
- API requests may fail with unexpected errors leading to outage until the resource is restored back to its previous state.
- It is difficult to search for inter-dependencies between API proxies and other resources in Apigee Edge.

Best Practice

1. Use any standard SCM coupled with a continuous integration and continuous deployment (CICD) pipeline for managing API proxies and shared flows.

2. Use any standard SCM for managing the other Edge resources, including API products, caches, KVMs, target servers, virtual hosts, and keystores.
    - If there are any existing Edge resources, then use management APIs to get the configuration details for them as a JSON/XML payload and store them in source control management.
    - Manage any new updates to these resources in source control management.
    - If there’s a need to create new Edge resources or update existing Edge resources, then use the appropriate JSON/XML payload stored in source control management and update the configuration in Edge using management APIs.

* Encrypted KVMs cannot be exported in plain text from the API. It is the user's responsibility to keep a record of what values are put into encrypted KVMs.

Further reading

- Source Control Management
- Source Control for API proxy Development
- Guide to implementing CI on Apigee Edge
- Maven Deploy Plugin for API Proxies
- Maven Config Plugin to manage Edge Resources
- Apigee Management API (for automating backups)

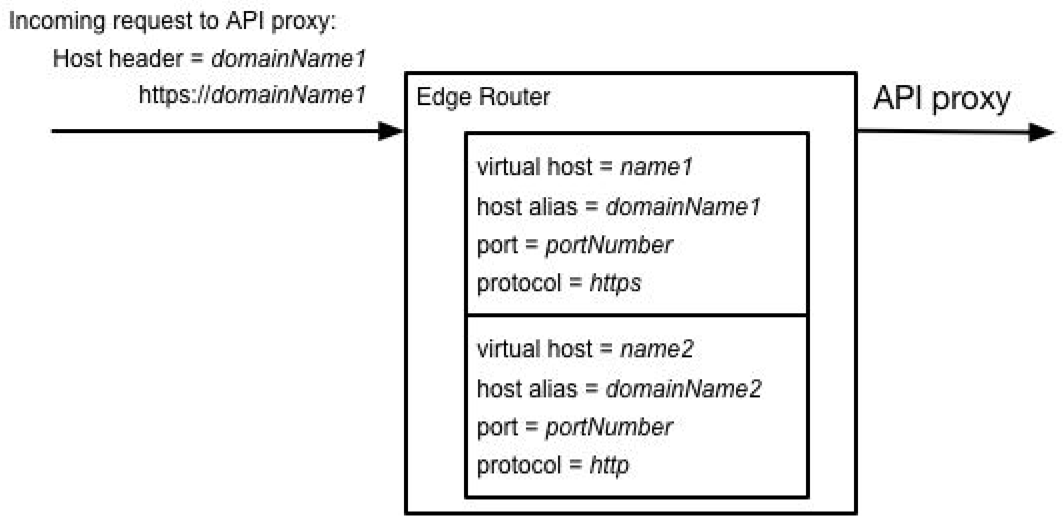
3.4. Define multiple virtual hosts with same host alias and port number

In Apigee Edge, a *Router* handles all incoming API traffic. That means all HTTP and HTTPS requests to an Edge API proxy are first handled by an Edge Router. Therefore, API proxy request must be directed to the IP address and open port on a Router.

A *virtual host* lets you host multiple domain names on a single server or group of servers. For Edge, the servers correspond to Edge Routers. By defining virtual hosts on a Router, you can handle requests to multiple domains.

A virtual host on Edge defines a protocol (HTTP or HTTPS), along with a Router port and a host alias. The host alias is typically a DNS domain name that maps to a Router's IP address.

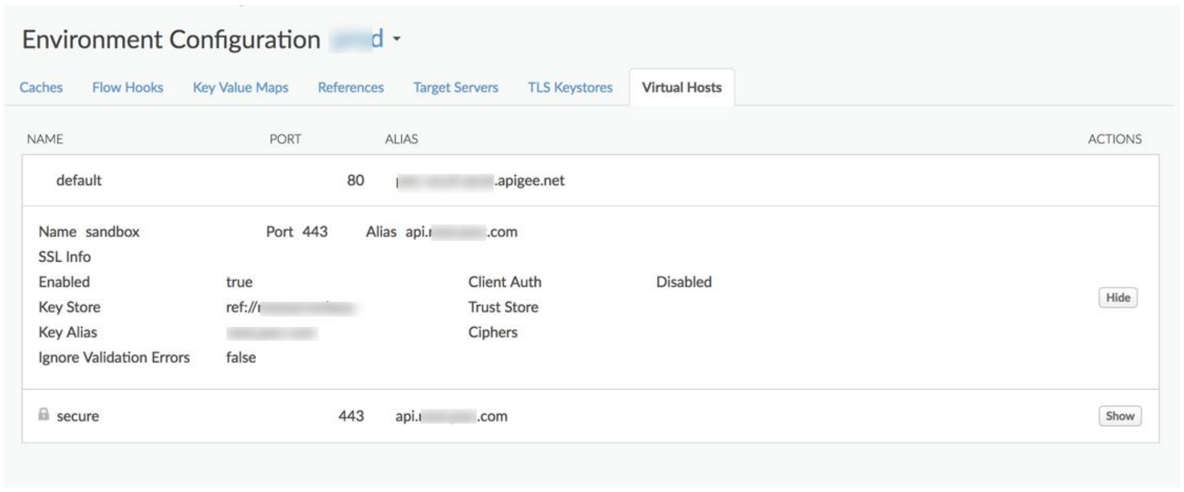
For example, the following image shows a Router with two virtual host definitions:



In this example, there are two virtual host definitions. One handles HTTPS requests on the domain *domainName1*, the other handles HTTP requests on *domainName2*.

On a request to an API proxy, the Router compares the Host header and port number of the incoming request to the list of **host aliases** defined by all virtual hosts to determine which virtual host handles the request.

Sample configuration for virtual hosts `are` shown below:



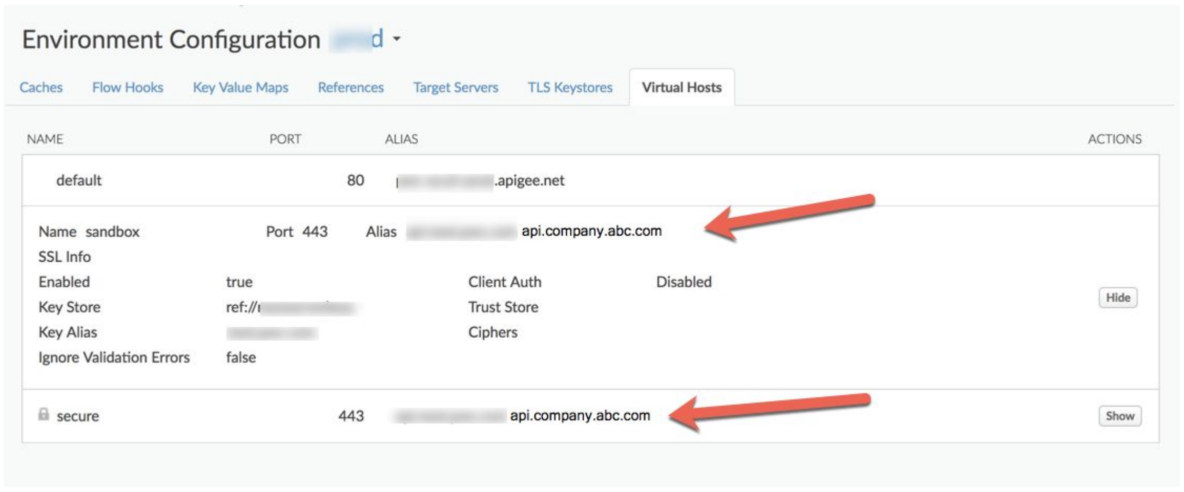
NAME	PORT	ALIAS	ACTIONS
default	80	.apigee.net	
sandbox	443	api..com	Hide
secure	443	api..com	Show

SSL Info			
Enabled	true	Client Auth	Disabled
Key Store	ref://	Trust Store	
Key Alias		Ciphers	
Ignore Validation Errors	false		

Antipattern

Defining multiple virtual hosts with the same host alias and port number in the same/different environments of an organization or across organizations will lead to confusion at the time of routing API requests and may cause unexpected errors/behaviour.
Let's use an example to explain the implications of having multiple virtual hosts with same host alias.

Consider there are two virtual hosts `sandbox` *and* `secure` defined with the same host alias i.e., `api.company.abc.com` in an environment:



NAME	PORT	ALIAS	ACTIONS
default	80	.apigee.net	
sandbox	443	api.company.abc.com	Hide
secure	443	api.company.abc.com	Show

SSL Info			
Enabled	true	Client Auth	Disabled
Key Store	ref://	Trust Store	
Key Alias		Ciphers	
Ignore Validation Errors	false		

With the above setup, there could be two scenarios:

Scenario 1 : An API proxy is configured to accept requests to only one of the virtual hosts sandbox

```
<ProxyEndpoint name="default">
  ...
  <HTTPProxyConnection>
    <BasePath>/demo</BasePath>
    <VirtualHost>sandbox</VirtualHost>
  </HTTPProxyConnection>
  ...
</ProxyEndpoint>
```

In this scenario, when the client applications make the calls to specific API proxy using the host alias `api.company.abc.com`, they will intermittently get 404 errors with the message:

```
Unable to identify proxy for host: secure
```

This is because the Router sends the requests to both `sandbox` and `secure` virtual hosts. When the requests are routed to `sandbox` virtual host, the client applications will get a successful response. However, when the requests are routed to `secure` virtual host, the client applications will get 404 error as the API proxy is not configured to accept requests on `secure` virtual host.

Scenario 2 : An API proxy is configured to accept requests to both the virtual hosts sandbox and secure

```
<ProxyEndpoint name="default">
  ...
  <HTTPProxyConnection>
```

```
    <BasePath>/demo</BasePath>
    <VirtualHost>sandbox</VirtualHost>
    <VirtualHost>secure</VirtualHost>
  </HTTPProxyConnection>
  ...
</ProxyEndpoint>
```

In this scenario, when the client applications make the calls to specific API proxy using the host alias `api.company.abc.com`, they will get a valid response based on the proxy logic.
However, this causes incorrect data to be stored in Analytics as the API requests are routed to both the virtual hosts, while the actual requests were aimed to be sent to only one virtual host.
This can also affect the logging information and any other data that is based on the virtual hosts.

Impact

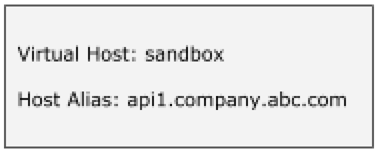
- 404 Errors as the API requests may get routed to a virtual host to which the API proxy may not be configured to accept the requests.

- Incorrect Analytics data since the API requests get routed to all the virtual hosts having the same host alias while the requests were made only for a specific virtual host.

Best Practice

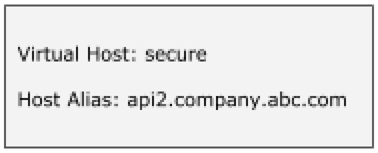
1. Don't define multiple virtual hosts with same host alias and port number in the same environment, or different environments of an organization.

2. If there's a need to define multiple virtual hosts, then use different host aliases in each of the virtual hosts as shown below :



Virtual Host: sandbox

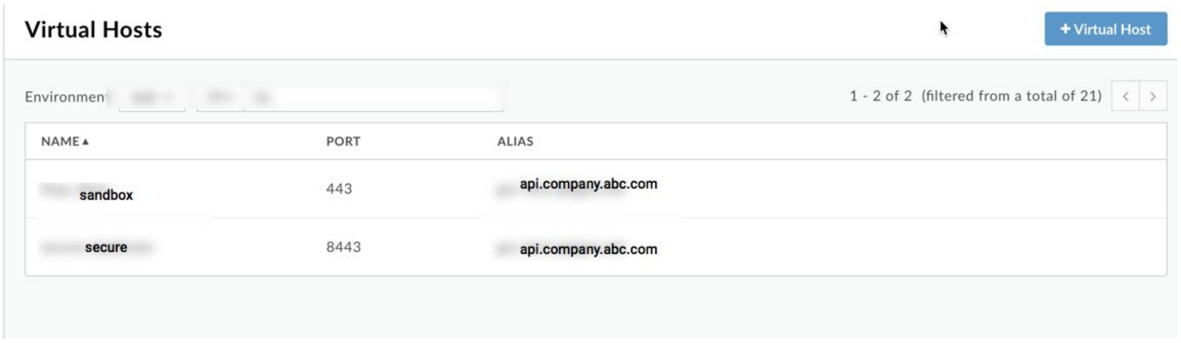
Host Alias: api1.company.abc.com



Virtual Host: secure

Host Alias: api2.company.abc.com

If there's a need to define multiple virtual hosts with same host alias, then it can be achieved by having different ports numbers for each of the virtual hosts. For example `sandbox` virtual host has the host alias api.company.abc.com with port 443 `secure` virtual host also has the same host alias api.company.abc.com but with a different port 8443



NAME ▲	PORT	ALIAS
sandbox	443	api.company.abc.com
secure	8443	api.company.abc.com

Further reading

- Virtual hosts

3.5. Load Balance with a single Target Server with MaxFailures set to a non-zero value

The TargetEndpoint configuration defines the way Apigee Edge connects to a backend service or API. It sends the requests and receives the responses to/from the backend service. The backend service can be a HTTP/HTTPS server, NodeJS, or Hosted Target.

The backend service in the TargetEndpoint can be invoked in one of the following ways:

- Direct URL to an HTTP or HTTPS server
- ScriptTarget to an Edge-hosted Node.js script
- HostedTarget to NodeJS deployed on Hosted Target Environment
- TargetServer configuration

Likewise, the Service Callout policy can be used to make a call to any external service from the API proxy flow. This policy supports defining HTTP/HTTPS target URLs either directly in the policy itself or using a TargetServer configuration.

TargetServer

TargetServer configuration decouples the concrete endpoint URLs from TargetEndpoint configurations or in Service Callout policies. A TargetServer is referenced by a name instead of the URL in TargetEndpoint. The TargetServer configuration will have the hostname of the backend service, port number, and other details.

Here is a sample TargetServer configuration:

```xml
<TargetServer name="target1">
  <Host>www.mybackendservice.com</Host>
  <Port>80</Port>
  <IsEnabled>true</IsEnabled>
</TargetServer>
```

The TargetServer enables you to have different configurations for each environment. A TargetEndpoint/Service Callout policy can be configured with one or more named TargetServers using a LoadBalancer. The built-in support for load balancing enhances the availability of the APIs and failover among configured backend server instances.

Here is a sample TargetEndpoint configuration using TargetServers:

```
<TargetEndpoint name="default">
    <HTTPTargetConnection>>
      <LoadBalancer>
        <Server name="target1"/>
        <Server name="target2"/>
      </LoadBalancer>
    </HTTPTargetConnection>
</TargetEndpoint>
```

MaxFailures

The `MaxFailures` configuration specifies maximum number of request failures to the target server after which the target server shall be marked as down and removed from rotation for all subsequent requests.

An example configuration with `MaxFailures` specified:

```
<TargetEndpoint name="default">
    <HTTPTargetConnection>
      <LoadBalancer>
        <Server name="target1"/>
       <Server name="target2"/>
       <MaxFailures>5</MaxFailures>
      </LoadBalancer>
    </HTTPTargetConnection>
</TargetEndpoint>
```

In the above example, if five consequent requests failed for target1 then target1 will be removed from rotation and all subsequent requests will be sent only to target2.

Antipattern

Having single TargetServer in a `LoadBalancer` configuration of the TargetEndpoint or Service Callout policy with `MaxFailures` set to a non-zero value is not recommended as it can have adverse implications.

Consider the following sample configuration that has a single TargetServer named “target1” with `MaxFailures` set to 5 (non-zero value):

```
<TargetEndpoint name="default">
  <HTTPTargetConnection>
      <LoadBalancer>
        <Algorithm>RoundRobin</Algorithm>
        <Server name="target1" />
        <MaxFailures>5</MaxFailures>
      </LoadBalancer>
  </HTTPTargetConnection>
```

If the requests to the TargetServer "target1" fails five times (number specified in `MaxFailures`), the TargetServer is removed from rotation. Since there are no other TargetServers to fail over to, all the subsequent requests to the API proxy having this configuration will fail with `503 Service Unavailable` error.

Even if the TargetServer "target1" gets back to its normal state and is capable of sending successful responses, the requests to the API proxy will continue to return 503 errors. This is because Edge does not automatically put the TargetServer back in rotation even after the target is up and running again. To address this issue, ***the API proxy must be redeployed*** for Edge to put the TargetServer back into rotation.

If the same configuration is used in the Service Callout policy, then the API requests will get 500 Error after the requests to the TargetServer “target1” fails for 5 times.

Impact

Using a single TargetServer in a `LoadBalancer` configuration of TargetEndpoint or Service Callout policy with `MaxFailures` set to a non-zero value causes:

- API Requests to fail with 503/500 Errors continuously (after the requests fail for MaxFailures number of times) until the API proxy is redeployed.
- Longer outage as it is tricky and can take more time to diagnose the cause of this issue (without prior knowledge about this antipattern).

Best Practice

1. Have more than one TargetServer in the `LoadBalancer` configuration for higher availability.

2. **Always define a Health Monitor** when `MaxFailures` is set to a non-zero value. A target server will be removed from rotation when the number of failures reaches the number specified in `MaxFailures`. Having a HealthMonitor ensures that the TargetServer is put back into rotation as soon as the target server becomes available again, ***meaning there is no need to redeploy the proxy***.

   To ensure that the health check is performed on the same port number that Edge uses to connect to the target servers, Apigee recommends that you omit the `<Port>` child element under `<TCPMonitor>` unless it is different from TargetServer port. By default `<Port>` is the same as the TargetServer port.

   Sample configuration with HealthMonitor:

```
<TargetEndpoint name="default">
  <HTTPTargetConnection>
    <LoadBalancer>
      <Algorithm>RoundRobin</Algorithm>
      <Server name="target1" />
      <Server name="target2" />
      <MaxFailures>5</MaxFailures>
    </LoadBalancer>
    <Path>/test</Path>
    <HealthMonitor>
      <IsEnabled>true</IsEnabled>
      <IntervalInSec>5</IntervalInSec>
      <TCPMonitor>
        <ConnectTimeoutInSec>10</ConnectTimeoutInSec>
      </TCPMonitor>
    </HealthMonitor>
  </HTTPTargetConnection>
</TargetEndpoint>
```

3. If there's some constraint such that only one TargetServer and if the HealthMonitor is not used, then don't specify `MaxFailures` in the `LoadBalancer` configuration.

    ○ **The default value of MaxFailures is 0.** This means that Edge always tries to connect to the target for each request and never removes the target server from the rotation.

Further reading

- Load Balancing across Backend Servers
- How to use Target Servers in your API Proxies

3.6. Access the Request/Response payload when streaming is enabled

In Edge, the default behavior is that HTTP request and response payloads are stored in an in-memory buffer before they are processed by the policies in the API proxy. If streaming is enabled, then request and response payloads are streamed without modification to the client app (for responses) and the target endpoint (for requests). Streaming is useful especially if an application accepts or returns large payloads, or if there's an application that returns data in chunks over time.

Antipattern

Accessing the request/response payload with streaming enabled causes Edge to go back to the default buffering mode.

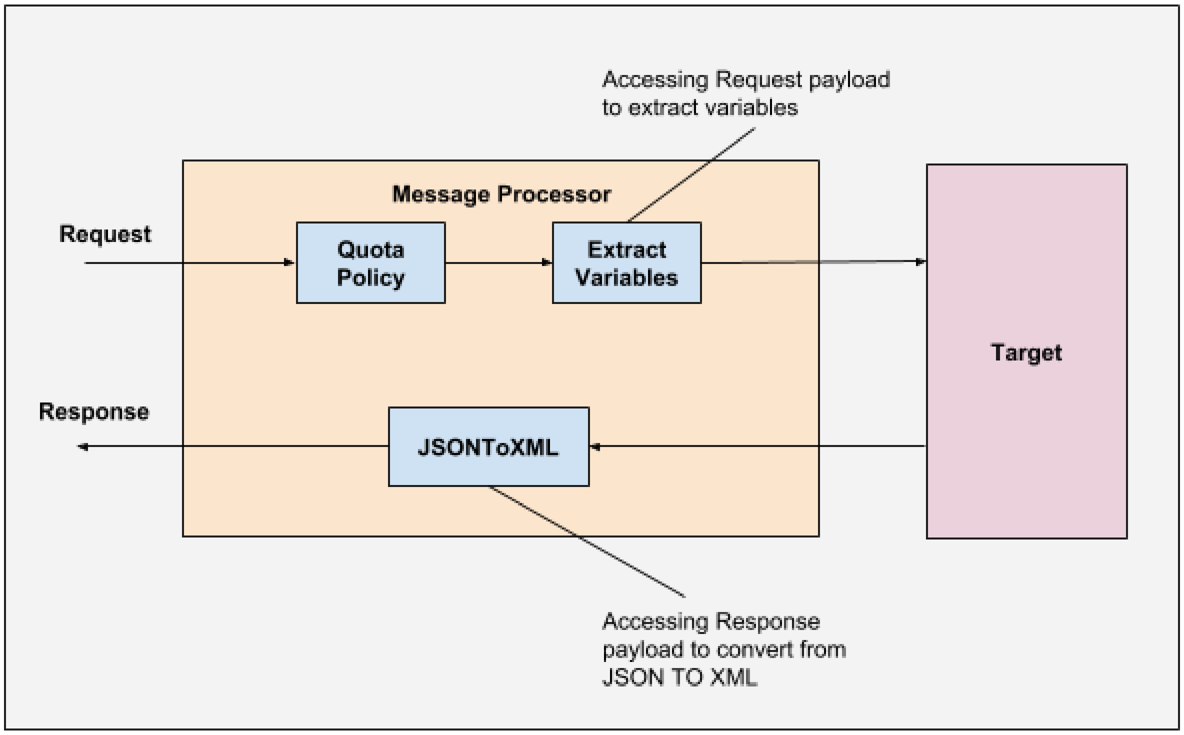

Accessing Request payload to extract variables
Message Processor
Request
Quota Policy
Extract Variables
Target
Response
JSONToXML
Accessing Response payload to convert from JSON TO XML

Figure 1: Accessing request/response payload with streaming enabled

The illustration above shows that we are trying to extract variables from the request payload and converting the JSON response payload to XML using JSONToXML policy. This will disable the streaming in Edge.

Impact

- Streaming will be disabled which can lead to increased latencies in processing the data.
- Increase in the heap memory usage or OutOfMemory Errors can be observed on Message Processors due to use of in-memory buffers especially if we have large request/response payloads.

Best Practice

Don't access the request/response payload when streaming is enabled.

Further reading

- Streaming requests and responses
- How does APIGEE Edge Streaming works ?
- How to handle streaming data together with normal request/response payload in a single API proxy
- Best practices for API proxy design and development

3.7. Define multiple ProxyEndpoints in an API proxy

The ProxyEndpoint configuration defines the way client apps consume the APIs through Apigee Edge. The ProxyEndpoint defines the URL of the API proxy and how a proxy behaves: which policies to apply and which target endpoints to route to, and the conditions that need to be met for these policies or route rules to be executed.
In short, the ProxyEndpoint configuration defines all that needs to be done to implement an API.

Antipattern

An API proxy can have one or more proxy endpoints. Defining multiple ProxyEndpoints is an easy and simple mechanism to implement multiple APIs in a single proxy. This lets you reuse policies and/or business logic before and after the invocation of a TargetEndpoint.

On the other hand, when defining multiple ProxyEndpoints in a single API proxy, you end up conceptually combining many unrelated APIs into a single artefact. It makes the API Proxies harder to read, understand, debug, and maintain. This defeats the main philosophy of API Proxies: making it easy for developers to create and maintain APIs.

Impact

Multiple ProxyEndpoints in an API proxy can:

- Make it hard for developers to understand and maintain the API proxy.
- Obfuscate analytics. By default, analytics data is aggregated at the proxy level. There is no breakdown of metrics by proxy endpoint unless you create custom reports.
- Make it difficult to troubleshoot problems with API Proxies.

Best Practice

When you are implementing a new API proxy or redesigning an existing API proxy, use the following best practices:

1. Implement one API proxy with a single ProxyEndpoint.

2. If there are multiple APIs that share common target server and/or require the same logic pre or post invocation of the target server, consider using shared flows to implement such logic in different API Proxies.
3. If there are multiple APIs that share a common starting basepath, but differ in the suffix, use conditional flows in a single ProxyEndpoint.

4. If there exists an API proxy with multiple ProxyEndpoints and if there are no issues with it, then there is no need to take any action.

Using one ProxyEndpoint per API proxy leads to:
1. Simpler, easier to maintain proxies

2. Better information in Analytics; proxy performance, target response time, etc. will be reported separately instead of rolled up for all ProxyEndpoints

3. Faster troubleshooting and issue resolution

Further reading

- API proxy Configuration Reference
- Reusable Shared Flows

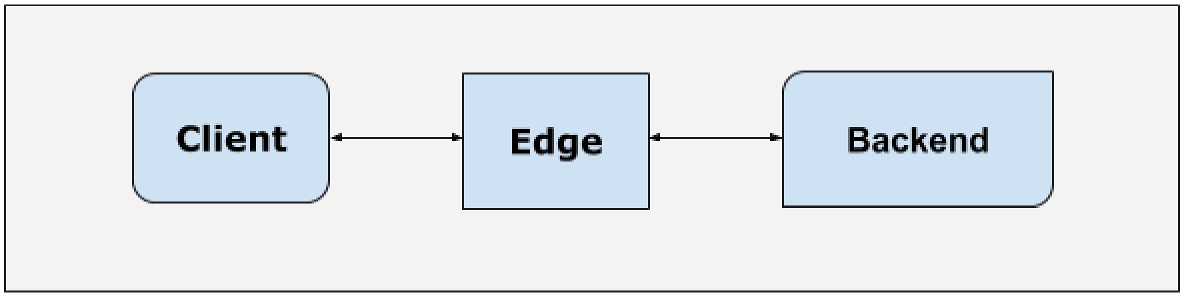
4. Backend Antipatterns

4.1. Allow a Slow Backend

Backend systems run the services that API Proxies access. In other words, they are the fundamental reason for the very existence of APIs and the API Management Proxy layer.

Any API request that is routed via the Edge platform traverses a typical path before it hits the backend:

- The request originates from a client which could be anything from a browser to an app.
- The request is then received by the Edge gateway.
- It is processed within the gateway. As a part of this processing, the request passes onto a number of distributed components.
- The gateway then routes the request to the backend that responds to the request.
- The response from the backend then traverses back the exact reverse path via the Edge gateway back to the client.



In effect, the performance of API requests routed via Edge is dependent on both Edge and the backend systems. In this anti pattern, we will focus on the impact on API requests due to badly performing backend systems.

Antipattern

Let us consider the case of a problematic backend. These are the possibilities:

- Inadequately sized backend
- Slow backend

Inadequately sized backend

The challenge in exposing the services on these backend systems via APIs is that they are accessible to a large number of end users. From a business perspective, this is a desirable challenge, but something that needs to be dealt with.

Many times backend systems are not prepared for this extra demand on their services and are consequently under sized or are not tuned for efficient response.

The problem with an 'inadequately sized' backend is that if there is a spike in API requests, then it will stress the resources like CPU, Load and Memory on the backend systems. This would eventually cause API requests to fail.

Slow backend

The problem with an improperly tuned backend is that it would be very slow to respond to any requests coming to it, thereby leading to increased latencies, premature timeouts and a compromised customer experience.

The Edge platform offers a few tunable options to circumvent and manage the slow backend. But these options have limitations.

Impact

- In the case of an inadequately sized backend, increase in traffic could lead to failed requests.
- In the case of a slow backend, the latency of requests will increase.

Best Practice

1. Use caching to store the responses to improve the API response times and reduce the load on the backend server.

2. Resolve the underlying problem in the slow backend servers.

Further reading

- Edge Caching Internals

4.2. Disable HTTP persistent (Reusable keep-alive) connections

An API proxy is an interface for client applications that can be used to connect with backend services. Apigee Edge provides multiple ways to connect to backend services through an API proxy:

- TargetEndpoint to connect to any HTTP/HTTPs, NodeJS, or Hosted Target services.
- ServiceCallout policy to invoke any external service pre- or post- invocation of the target server in TargetEndpoint.
- Custom code can be added to JavaScript policy or JavaCallout policy to connect to backend services

Persistent Connections

HTTP persistent connection, also called HTTP keep-alive or HTTP connection reuse, is a concept that allows a singleTCP connection to send and receive multiple HTTP requests/responses, instead of opening a new connection for every request/response pair.

Apigee Edge uses persistent connections for communicating with the backend services by default. A connection is kept alive for 60 seconds by default. That is, if a connection is kept idle in the connection pool for more than 60 seconds, then the connection will be closed.

The keep alive timeout period is configurable through a property named `keepalive.timeout.millis`, which can be specified in the TargetEndpoint configuration of an API proxy. For example, the keep alive time period can be set to 30 seconds for a specific backend service in the TargetEndpoint.

In the example below, the `keepalive.timeout.millis` is set to 30 seconds in the TargetEndpoint configuration:

```
<TargetEndpoint name="default">
  <HTTPTargetConnection>
    <URL>http://mocktarget.apigee.net</URL>
    <Properties>
```

```
      <Property name="keepalive.timeout.millis">30000</Property>
    </Properties>
  </HTTPTargetConnection>
</TargetEndpoint>
```

In the example above, `keepalive.timeout.millis` controls the keep alive behavior for a specific backend service in an API proxy. There is also a property that controls keep alive behavior for all backend services in all proxies. The `HTTPTransport.keepalive.timeout.millis` is configurable in the Message Processor component. This property also has a default value of 60 seconds. Making any modifications to this property affects the keep alive connection behavior between Apigee Edge and all the backend services in all the API proxies.

Antipattern

Disabling persistent (keep alive) connections by setting the property `keepalive.timeout.millis` to 0 in the TargetEndpoint configuration of a specific API proxy or setting the `HTTPTransport.keepalive.timeout.millis` to 0 on Message Processors is not recommended as it will impact performance.

In the example below, the TargetEndpoint configuration disables persistent (keep alive) connections for a specific backend service by setting `keepalive.timeout.millis` to 0:

```
<TargetEndpoint name="default">
  <HTTPTargetConnection>
    <URL>http://mocktarget.apigee.net</URL>
    <Properties>
      <Property name="keepalive.timeout.millis">0</Property>
     </Properties>
  </HTTPTargetConnection>
</TargetEndpoint>
```

If the keep alive connections are disabled for one or more backend services, then Edge will need to open a new connection every time a new request is to be made to the specific backend service(s). In addition, SSL handshake must be performed for every new request if the backend is HTTPs, adding to the overall latency of the API requests.

Impact

- Increases the overall response time of the API requests as Apigee Edge will need to open a new connection, as SSL handshake must be performed for every new request.

- Connections may get exhausted under high traffic conditions, as it takes some time for the connections to be released back to the system.

Best Practice

1. Backend services should honor and handle HTTP persistent connection in accordance with HTTP 1.1 standards.

2. Backend services should respond with a `Connection:keep-alive` header if they are able to handle persistent (keep alive) connections.

3. Backend services should respond with a `Connection:close` header if they are unable to handle persistent connections

Implementing this pattern will ensure that Apigee Edge can automatically handle persistent or non-persistent connection with backend services, without requiring changes to the API proxy.

Further reading

- HTTP persistent connections
- Keep-Alive
- Endpoint Properties Reference

5. Private Cloud Antipatterns

5.1. Add Custom Information to Apigee owned Schema in Postgres Database

Applicable for Private Cloud Users Only

Analytics Services is a very powerful built-in feature provided by Apigee Edge. It collects and analyzes a broad spectrum of data that flows across APIs. The analytics data captured can provide very useful insights. For instance, How is the API traffic volume trending over a period of time? Which is the most used API? Which APIs are having high error rates?

Regular analysis of this data and insights can be used to take appropriate actions such as future capacity planning of APIs based on the current usage, business and future investment decisions, and many more.

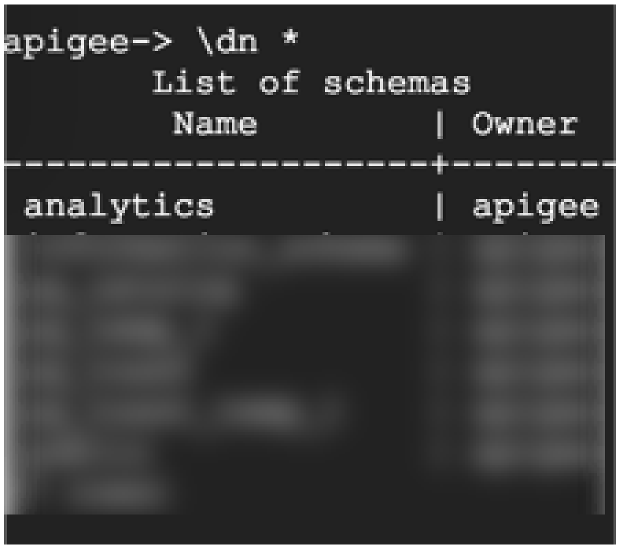
Analytics Data and its storage

Analytics Services captures many different types of data such as:

1. Information about an API - Request URI, Client IP address, Response Status Codes, etc.
2. API proxy Performance - Success/Failure rate, Request and Response Processing Time, etc.
3. Target Server Performance - Success/Failure rate, Processing Time
4. Error Information - Number of Errors, Fault Code, Failing Policy, Number of Apigee and Target Server caused errors
5. Other Information - Number of requests made by Developers, Developer Apps, etc

All these data are stored in an `analytics` schema created and managed within a Postgres Database by Apigee Edge.

Typically, in a vanilla Edge installation, Postgres will have following schemata :



```
apigee-> \dn *
      List of schemas
       Name        | Owner
-------------------+-------
 analytics         | apigee
```

[redacted]

The schema named `analytics` is used by Edge for storing all the analytics data for each organization and environment. If monetization is installed there will be a `rkms` schema. Other schemata are meant for Postgres internals.

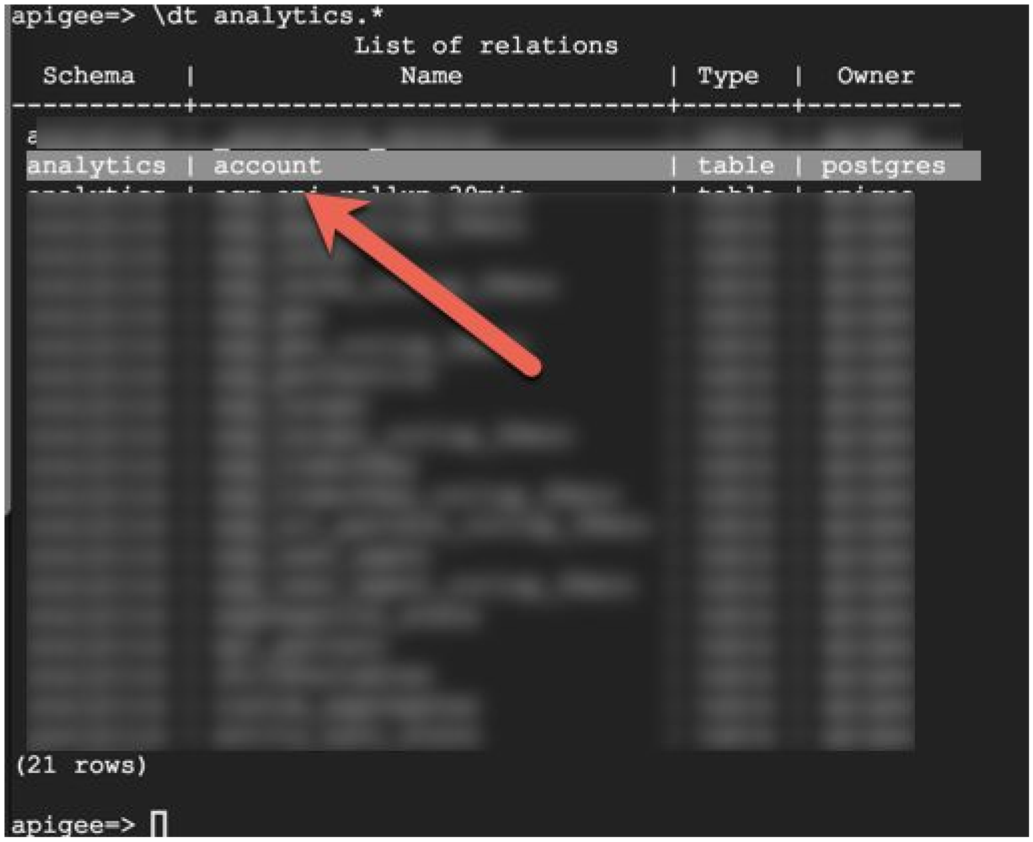
The `analytics` schema will keep changing as Apigee Edge will dynamically add new fact tables to it at runtime. The Postgres server component will aggregate the fact data into aggregate tables which get loaded and displayed on the Edge UI.

Antipattern

Adding custom columns, tables, and/or views to any of the Apigee owned schemata in Postgres Database on Private Cloud environments directly using SQL queries is not advisable, as it can have adverse implications.

Let’s take an example to explain this in detail.

Consider a custom table named *account* has been created under analytics schema as shown below:



```
apigee=> \dt analytics.*
                    List of relations
  Schema    |            Name              | Type  |  Owner
-----------+------------------------------+-------+----------
 analytics | account                      | table | postgres
(21 rows)

apigee=>
```

After a while, let's say there's a need to upgrade Apigee Edge from a lower version to a higher version. Upgrading Private Cloud Apigee Edge involves upgrading Postgres amongst many other components. If there are any custom columns, tables, or views added to the Postgres Database, then Postgres upgrade fails with errors referencing to custom objects as they are not created by Apigee Edge. Thus, the Apigee Edge upgrade also fails and cannot be completed.

Similarly errors can occur during Apigee Edge maintenance activities where in backup and restore of Edge components including Postgres Database are performed.

Impact

- Apigee Edge upgrade cannot be completed because the Postgres component upgrade fails with errors referencing to custom objects not created by Apigee Edge.

- Inconsistencies (and failures) while performing Apigee Analytics service maintenance (backup/restore).

Best Practice

1. Don't add any custom information in the form of columns, tables, views, functions, and procedures directly to any of the Apigee owned schema such as `analytics` etc.

2. If there's a need to support custom information, it can be added as columns (fields) using a Statistics Collector policy to `analytics` schema.

Further reading

- Analytics Services Overview
- Analytics Stats APIs