Google Cloud | apigee

NOTE: For the latest version of “The Book of Apigee Edge Antipatterns”, please visit:

https://docs.apigee.com/api-platform/antipatterns

The Book of Apigee Edge Antipatterns

Avoid common pitfalls, maximize the power of your APIs

Contents

(c)2017 Apigee Corp. All Rights Reserved

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 it's 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 it's 'user' to do, but is something that may have adverse functional, serviceable or performance affect.

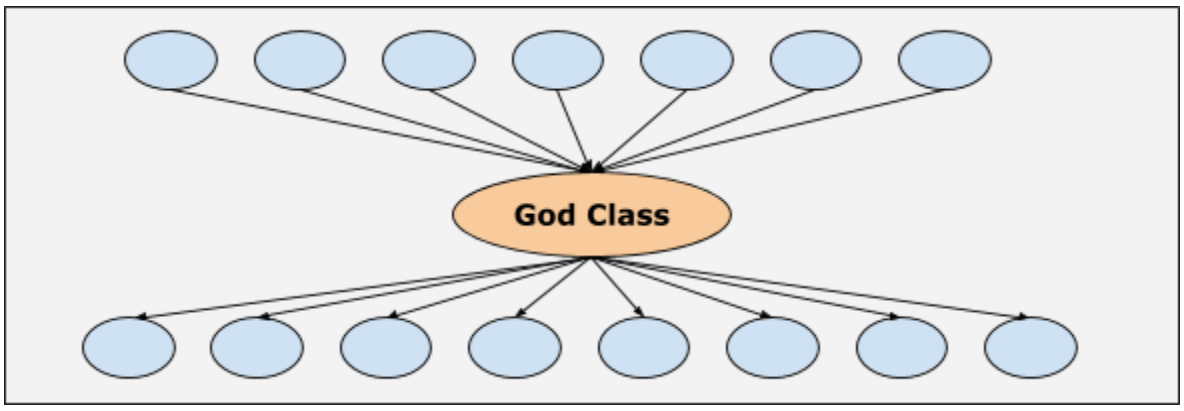
Let's take an example:

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:

(c)2017 Apigee Corp. All Rights Reserved



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 dont’s of API proxy development on the Edge.

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

(c)2017 Apigee Corp. All Rights Reserved

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 it's 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.

(c)2017 Apigee Corp. All Rights Reserved

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 the 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 : http://docs.apigee.com

Authors

Amar Devegowda
Senthil Kumar Tamizhselvan
Akash Tumkur Prabhashankar
Venkataraghavan Lakshminarayanachar

Acknowledgements

We would like to acknowledge the significant contributions made by the following Googlers in reviewing, proofreading and shaping this book - Rajesh Doda, Prashanth Subrahmanyam, Carlos Eberhardt, Mukunda Gnanasekharan, Joshua Norrid, Arghya Das, Gregory Brail, Bala Kasiviswanathan, Marsh Gardiner, Stephen Gilson, Divya Achan and many others.

(c)2017 Apigee Corp. All Rights Reserved

Edge Antipatterns

1.Policy Antipatterns

1.1. Caching 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 policies **Populate Cache policy**, **LookupCache policy**, **InvalidateCache policy**, and **Response Cache 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.

(c)2017 Apigee Corp. All Rights Reserved

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:

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

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

- ➔ The failure occurs for a temporary unknown period and we may continue to send error responses due to caching even after the problem has been fixed, OR
- ➔ The failure 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:

(c)2017 Apigee Corp. All Rights Reserved

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
- ➔ 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

(c)2017 Apigee Corp. All Rights Reserved

Best Practice

➔ Don't store the error responses in the response cache. Ensure that the **<ExcludeErrorResponse>** element is set to true in the Response Cache 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.

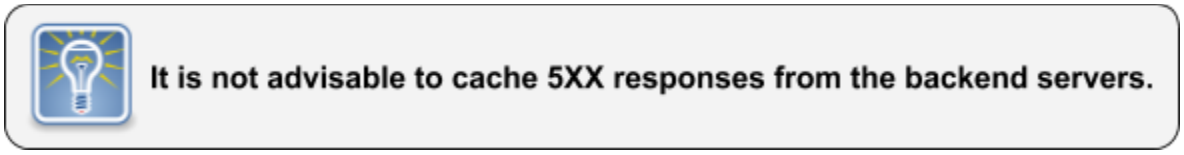
```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ResponseCache async="false" continueOnError="false" enabled="true"
name="TargetServerResponseCache">
    <DisplayName>TargetServerResponseCache</DisplayName>
    <CacheKey>
        <KeyFragment ref="request.uri" />
    </CacheKey>
    <Scope>Exclusive</Scope>
    <ExpirySettings>
        <TimeoutInSec ref="flow.variable.here">600</TimeoutInSec>
    </ExpirySettings>
    <CacheResource>targetCache</CacheResource>
    <ExcludeErrorResponse>true</ExcludeErrorResponse>
</ResponseCache>
```

➔ 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*:

- Determine the maximum/exact duration of time for which the failure will be observed (if possible).
  - 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.
  - Use the ResponseCache policy to cache the error responses without the **<ExcludeErrorResponse>** element.

(c)2017 Apigee Corp. All Rights Reserved

- **Note:** Once the failure has been fixed, remove/disable the ResponseCache policy. Otherwise, the error responses for temporary backend server failures may get cached.



References

Response Cache policy

(c)2017 Apigee Corp. All Rights Reserved

1.2. Storing 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.

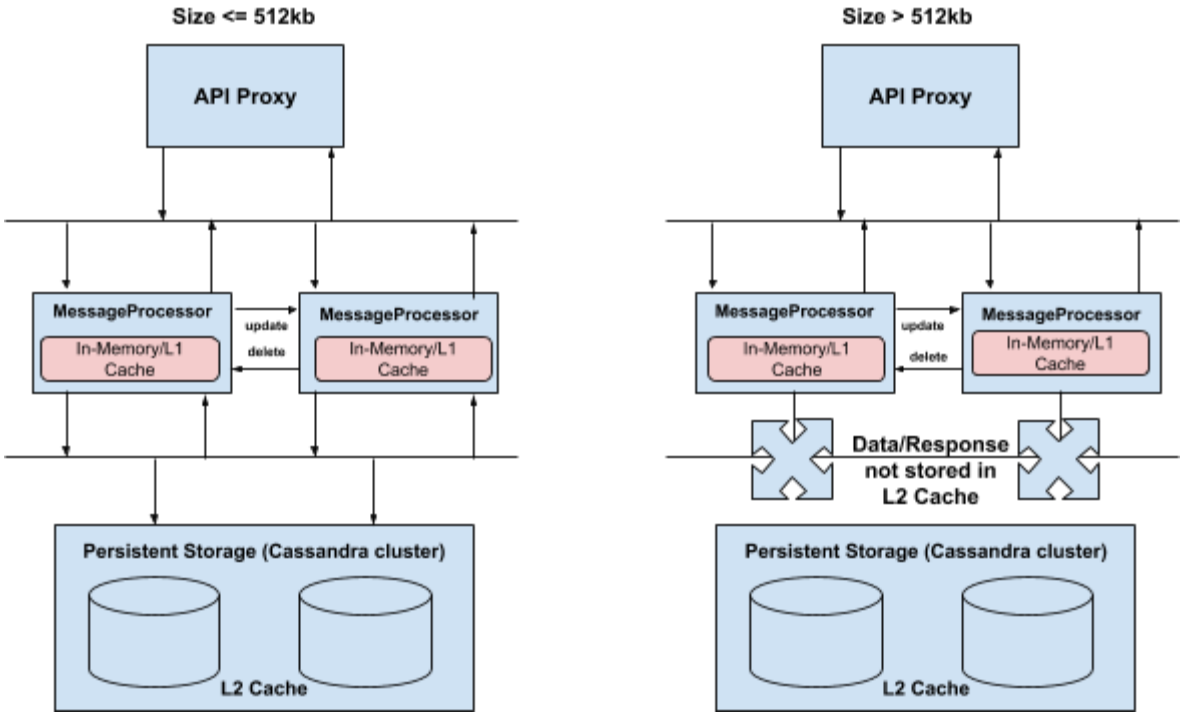
Antipattern

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.

(c)2017 Apigee Corp. All Rights Reserved



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

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

References

Edge Caching Internals

(c)2017 Apigee Corp. All Rights Reserved

1.3. Logging data to third party servers using JavaScript policy

Logging is one of the 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 SumoLogic 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.

JavaScript Policy - LogData_JS

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

JavaScript Code - LogData.js

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

(c)2017 Apigee Corp. All Rights Reserved

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

- ➔ 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.
- ➔ 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.
- ➔ Having this policy defined in Post Client flow helps separate the logging activity and the request/response processing thereby avoiding latencies.
- ➔ 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”

References

JavaScript Policy

Message Logging Policy

(c)2017 Apigee Corp. All Rights Reserved

1.4. 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.

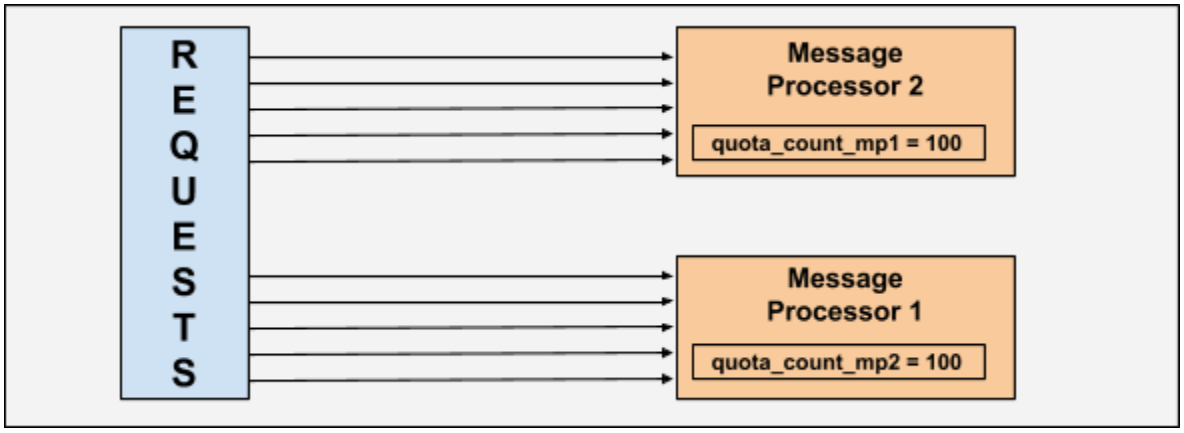
Let's explain this with the help of an example. Consider the following Quota policy for an API Proxy -

```
<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.

(c)2017 Apigee Corp. All Rights Reserved

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:

(c)2017 Apigee Corp. All Rights Reserved

```
<Quota name="CheckTrafficQuota">
  <Interval>1</Interval>
  <TimeUnit>hour</TimeUnit>
  <Distributed>true</Distributed>
  <Allow count="100"/>
</Quota>
```

References

Quota policy

(c)2017 Apigee Corp. All Rights Reserved

1.5. Re-using 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 gets decremented each time it is executed and we will end up getting Quota violation errors much earlier than the 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:

(c)2017 Apigee Corp. All Rights Reserved

```
<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"**

```
<TargetEndpoint name="Target-US">
  <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>
```

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

(c)2017 Apigee Corp. All Rights Reserved

```
<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.

(c)2017 Apigee Corp. All Rights Reserved

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

➔ 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>
```

- We will continue to use this Quota policy in both the target endpoints "Target-US" and "Target-EU" as before.
- Now let's say if the header target_id has a value "US" then the requests are routed to the target endpoint "Target-US".
- Similarly if the header target_id has a value "EU" then the requests are routed to the target endpoint "Target-EU".
- So even if we use the same quota policy in both the target endpoints, separate quota counters are maintained based on the **Identifier** value.
- Therefore, by using the **<Identifier>** element we can ensure that each of the target endpoints get the allowed quota of 10 requests.

➔ 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.

(c)2017 Apigee Corp. All Rights Reserved

- 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>
```

- Use the respective quota policy in the definition of the target endpoints as shown below:

(c)2017 Apigee Corp. All Rights Reserved

Target Endpoint “Target-US”

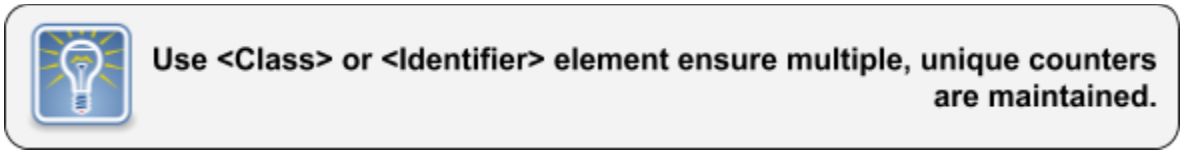
```
<TargetEndpoint name="Target-US">
  <PreFlow name="PreFlow">
  <Request>
     <Step>
        <Name>Quota-Minute-Target-Server-US</Name>
     </Step>
  <Request/>
  </Response>
  </Preflow>
  <HTTPTargetConnection>
  <URL>http://target-us.somedomain.com</URL>
  </HTTPTargetConnection>
</TargetEndpoint>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

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)2017 Apigee Corp. All Rights Reserved

- 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.



References

Quota policy

(c)2017 Apigee Corp. All Rights Reserved

2. Performance Antipatterns

2.1. Leaving 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.

(c)2017 Apigee Corp. All Rights Reserved

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.

References

Overview of Node.js on Apigee Edge

Getting started with Node.js on Apigee Edge

Debugging and troubleshooting Node.js proxies

(c)2017 Apigee Corp. All Rights Reserved

3. Generic Antipatterns

3.1. Invoking 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

Code Sample 1: Invoking Proxy 2 from Proxy 1 using HTTPTargetConnection

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

(c)2017 Apigee Corp. All Rights Reserved

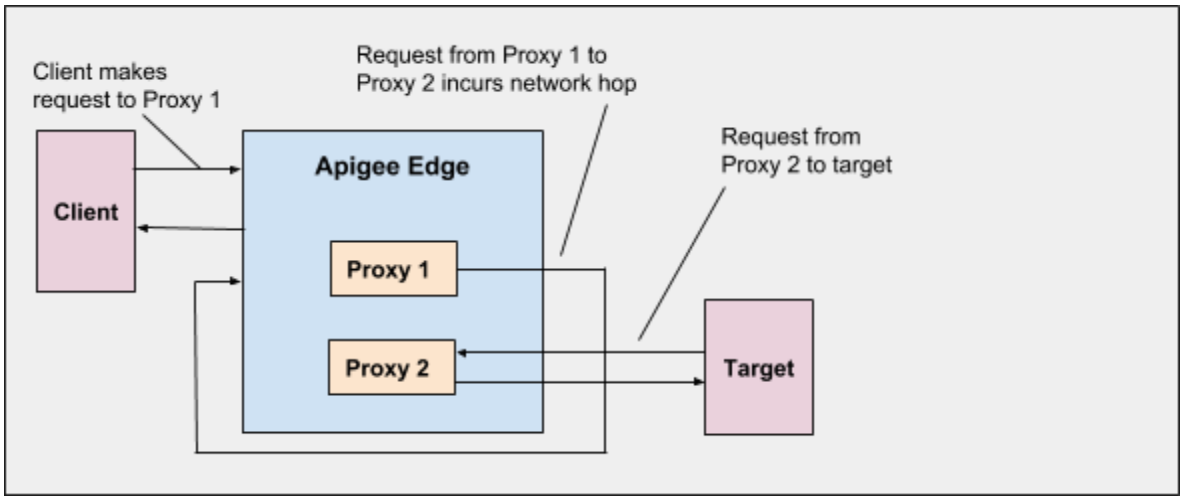
In the code sample 2 below, we are invoking the Proxy 2 from Proxy 1 using JavaScript

Code Sample 2: Invoking the Proxy 2 from Proxy 1 from the JavaScript code

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

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

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.

(c)2017 Apigee Corp. All Rights Reserved

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

➔ 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:

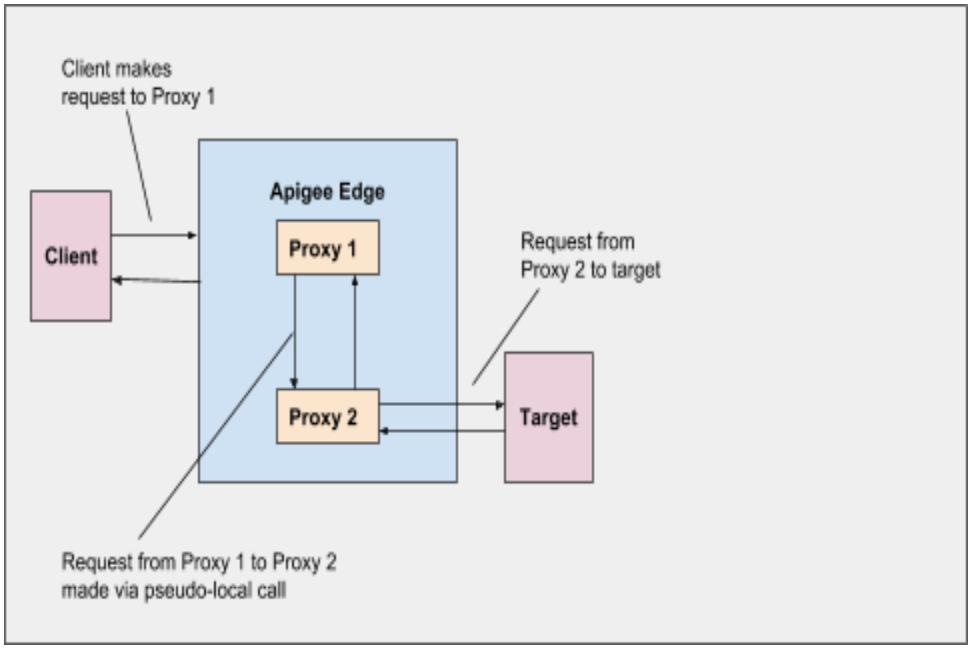
Code Sample: Proxy Chaining with LocalTargetConnection

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

- As you can see in the following figure, the invoked API Proxy gets executed within the same Message Processor and hence avoids the network hop as shown in the figure below.

(c)2017 Apigee Corp. All Rights Reserved

Code Flow with Proxy Chaining



References

Proxy Chaining

(c)2017 Apigee Corp. All Rights Reserved

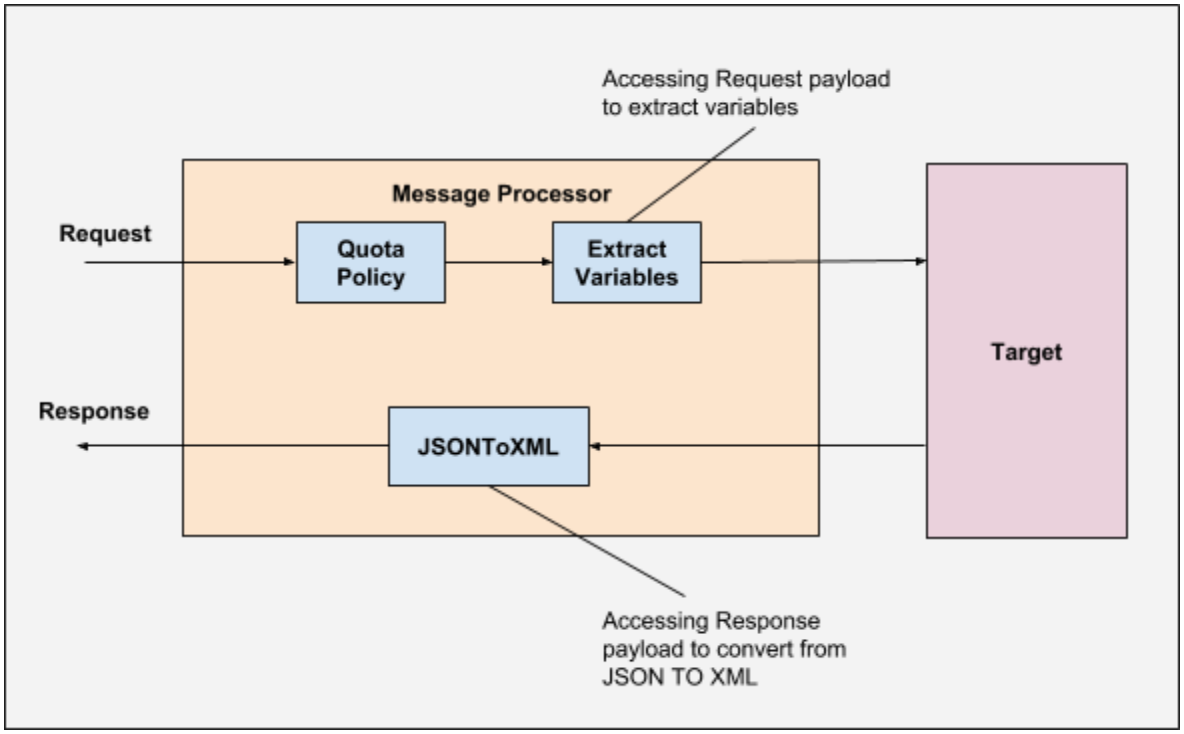
3.2. Accessing the request/response payload when streaming is enabled

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/response payload with streaming enabled



(c)2017 Apigee Corp. All Rights Reserved

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.

References

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

(c)2017 Apigee Corp. All Rights Reserved

3.3. Invoking 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.

(c)2017 Apigee Corp. All Rights Reserved

- ➔ 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>/keyv
aluemaps/<kvm_name>');
response.waitForComplete();
```

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.

(c)2017 Apigee Corp. All Rights Reserved

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:

(c)2017 Apigee Corp. All Rights Reserved

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

- If your API Proxy flow doesn't have a VerifyAPIKey policy, then you can access the profiles of API Products, Developer Apps, etc. using **AccessEntity** policy:
    - The below code sample shows how to use AccessEntity policy to retrieve the profile of DeveloperApp
    - This is followed by using ExtractVariable policy to retrieve the values of the attributes from the DeveloperApp profile

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>
```

(c)2017 Apigee Corp. All Rights Reserved

Code Sample : Extract the appId from DeveloperApp using ExtractVariables policy

```
<ExtractVariables name="Extract-DeveloperApp-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="appId" type="string">
      <!-- You parse elements from the developer profile using XPath. -->
    <XPath>/App/AppId</XPath>
  </Variable>
 </XMLPayload>
</ExtractVariables>
```

References

Key Value Map Operations policy

VerifyAPIKey policy

Access Entity Policy

(c)2017 Apigee Corp. All Rights Reserved

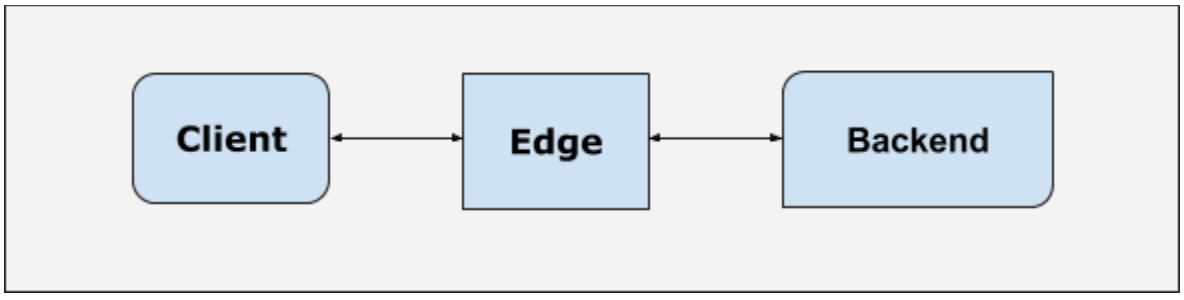
4. Backend Antipatterns

4.1. 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.

(c)2017 Apigee Corp. All Rights Reserved

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.

(c)2017 Apigee Corp. All Rights Reserved

Best Practice

- ➔ Use caching to store the responses to improve the API response times and reduce the load on the backend server.
- ➔ Resolve the underlying problem in the slow backend servers.

References

Edge Caching Internals

(c)2017 Apigee Corp. All Rights Reserved