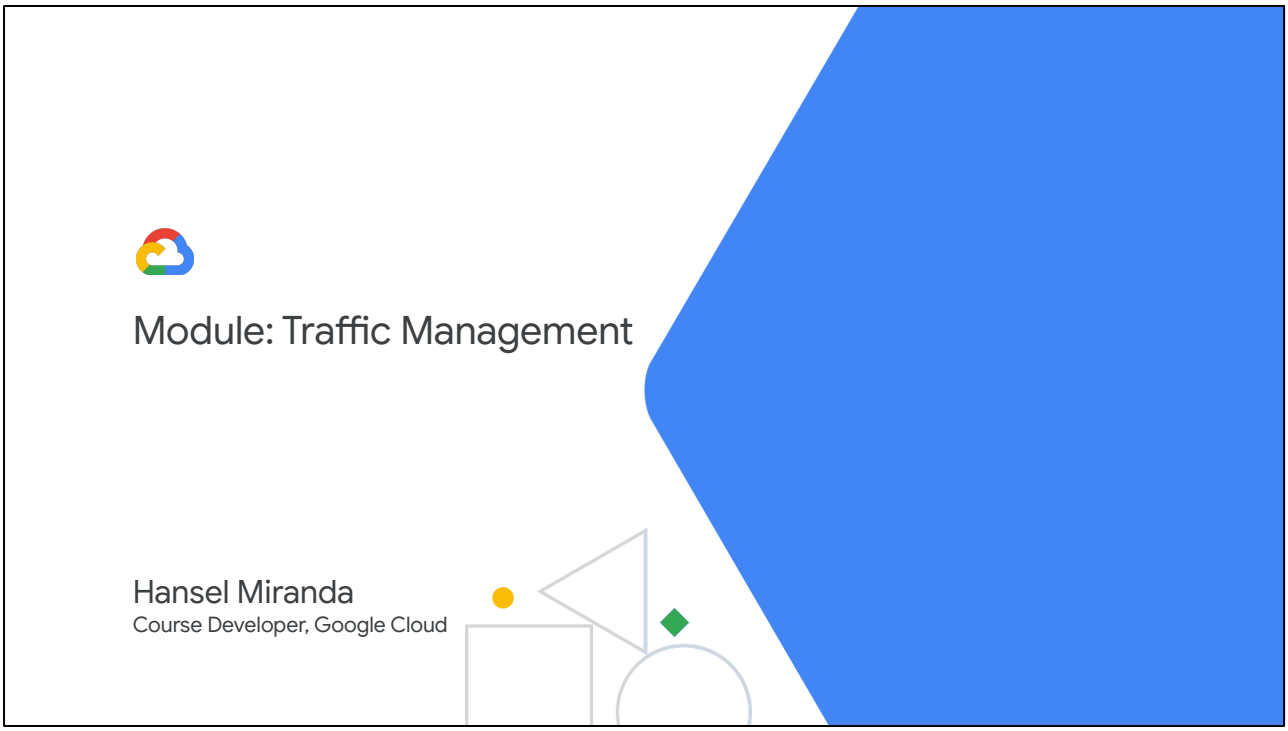
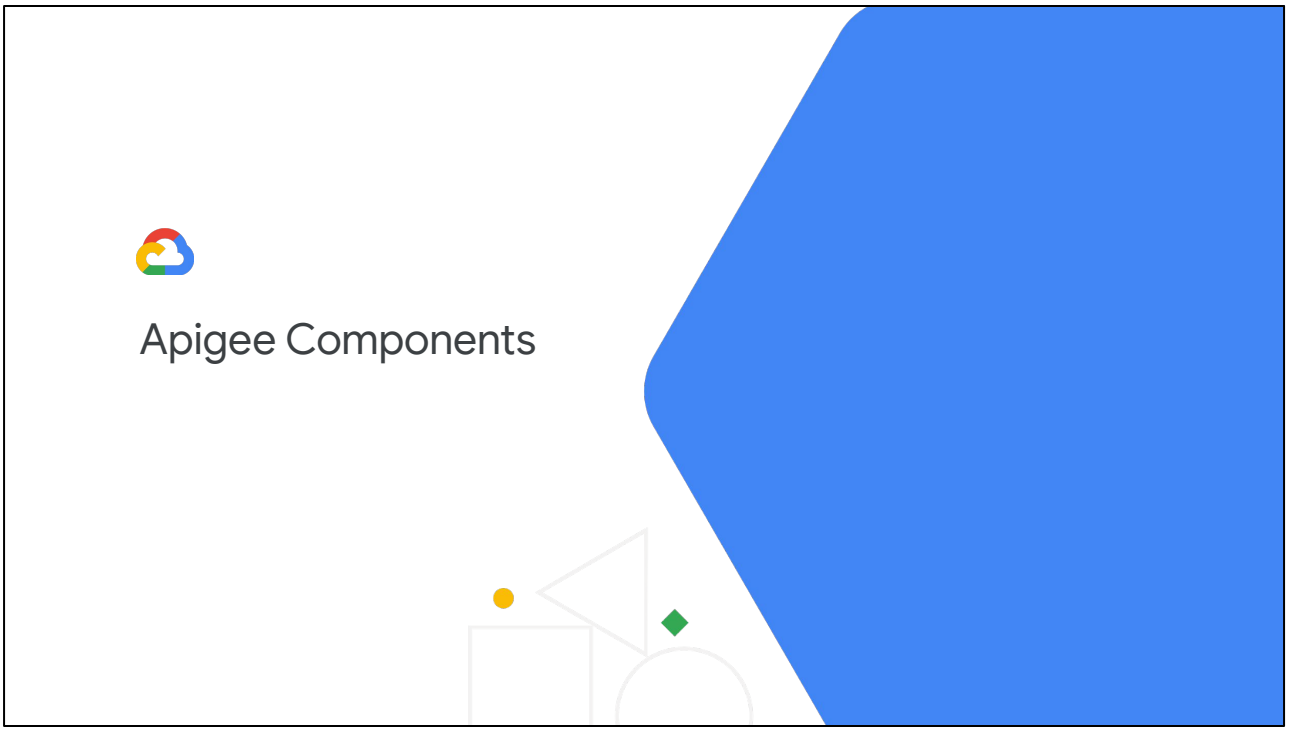


In this module, you will learn about API traffic management.

You will learn how to use spike arrests to control the rate of traffic, and how to use quotas to limit the number of requests over a specified period of time.

You'll also learn how to use caching in your API proxies, reducing unnecessary traffic to backend services by caching HTTP responses.

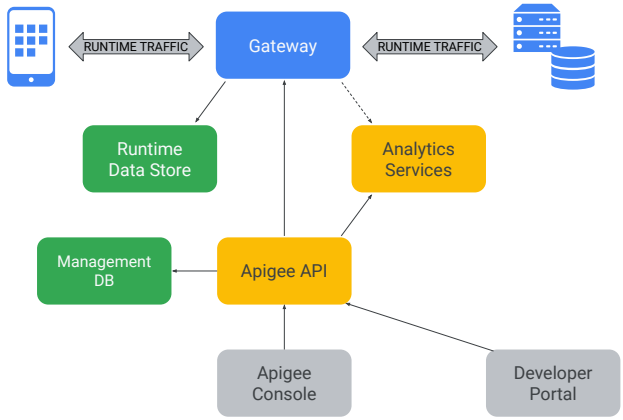
You will also complete labs that add a spike arrest, quota, and response cache policy to your retail API proxy.



In order to really understand Apigee traffic management, you need to understand the system components of Apigee.

This lecture will discuss these components.

Logical components

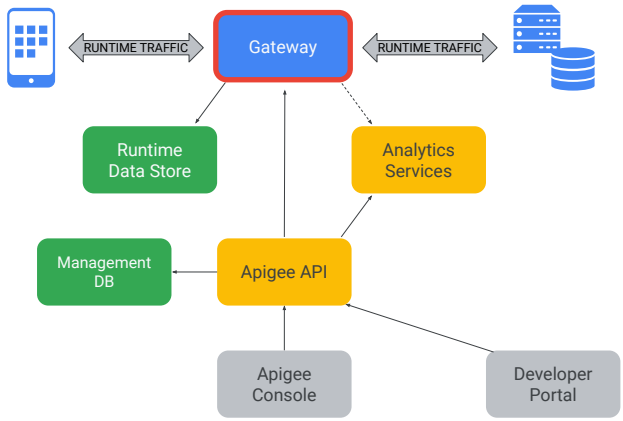


This is a simplified logical diagram of the primary Apigee components for Apigee.

Although implementations differ depending on the deployment topology, the logical components remain the same regardless of how Apigee is deployed.

Gateway

- Route and process runtime traffic.
- Logically composed of router and message processor.



Gateways are responsible for routing and processing runtime traffic flowing through the proxies.

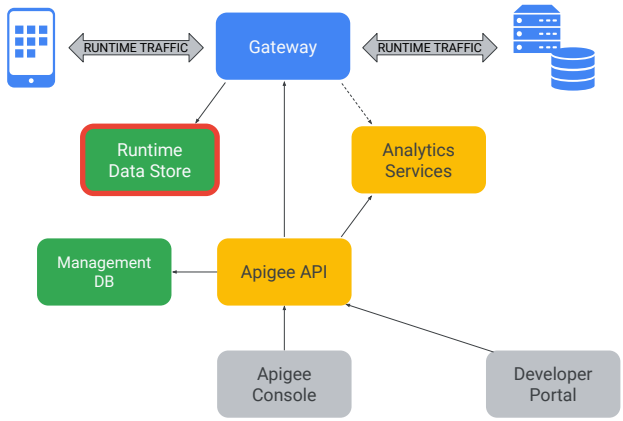
Logically we think of the gateway containing a router and a message processor.

The router routes the request to the correct proxy. We have seen that environment groups, environments, and base paths are used to determine which deployed proxy receives the request.

The message processor handles the API traffic running within the proxies.

Runtime data store

- Manages persistence of configuration and runtime data.
- Only gateways and runtime data store required for runtime traffic.
- Data propagated between regions automatically via database replication.

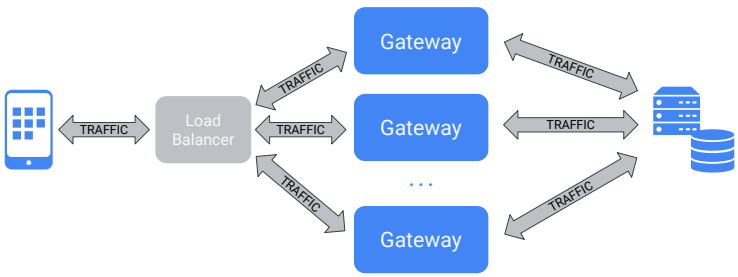


The runtime data store manages persistence of runtime data. Gateways are stateless and all runtime data, like tokens and cache entries, are persisted in the runtime data store.

Only gateways and the runtime data store are required for runtime traffic to be successfully processed. Other components can fail or lose connectivity, and API traffic will still be handled.

For multi-region deployments, runtime data propagation between regions is handled using database replication.

Runtime scaling



- Add or remove gateways to scale capacity.
- Runtime data store and other components scaled separately.
- Can allow increased traffic to backend services.

Gateways are logically fronted by a load balancer. The load balancer distributes traffic among the gateways.

Runtime API capacity can be scaled up and down by scaling up and down the number of gateways.

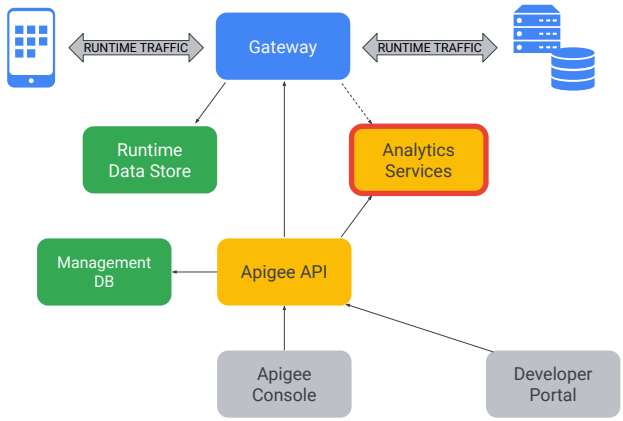
In managed Google Cloud deployments of Apigee, the number of gateways is automatically scaled up and down based on how loaded the gateways are.

The runtime data store and other components can also be separately scaled when necessary.

You should be aware that increasing the API traffic capacity of Apigee may result in a corresponding increase in traffic being sent to backend services.

Analytics services

- Stores runtime metrics and provides reporting services.



The Analytics Services component stores runtime metrics and provides reporting for those metrics.

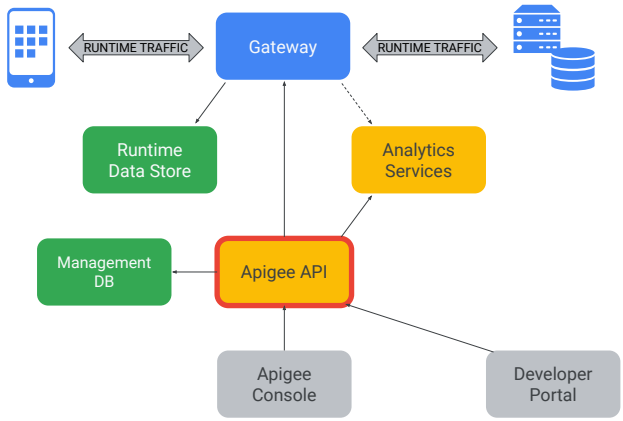
At the end of an API call, the gateway sends an analytics record to Analytics Services.

This record includes standard metrics about the API call and can also contain custom metrics as specified in the proxy.

This record is queued, and later processed by Analytics Services, making it available for reporting.

Apigee API

- Used by external tools or automation pipelines.
- Responsible for storing changes to an organization and updating gateways to reflect changes.



The Apigee API allows configuration of organization entities and the API lifecycle. It also allows retrieval of runtime analytics data for the organization.

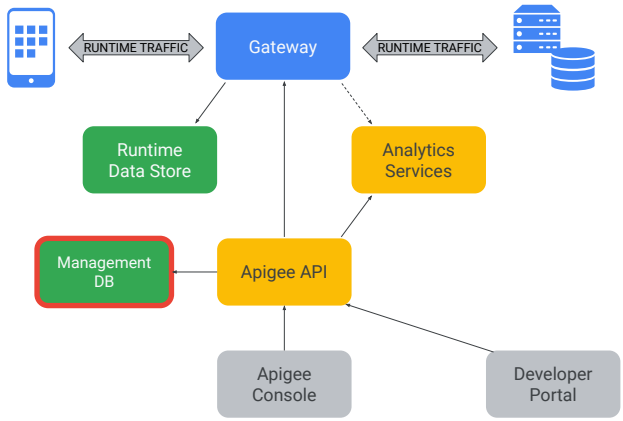
The Apigee API may be used by automation pipelines like Continuous Integration/Continuous Deployment, or CI/CD.

Tools can also use the Apigee API to export analytics data or integrate with customer dashboards.

The Apigee API stores requested changes to an organization in the management database.

Management database

- Stores non-runtime data for an organization.
- Runtimes poll for changes in this database.

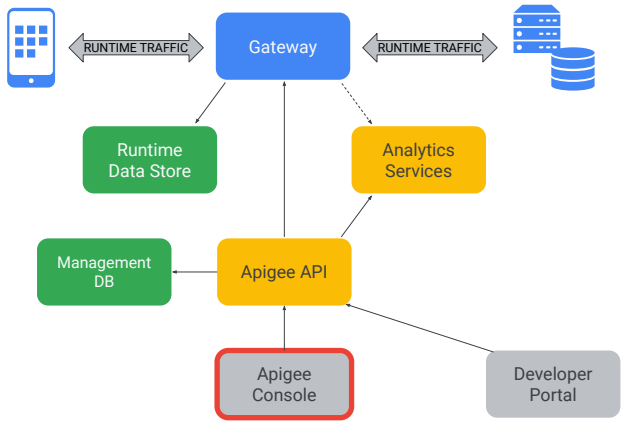


The **Management database** stores non-runtime data for an organization.

The runtime detects when deployments and other runtime entities are changed.

Apigee console

- Used to manage organizations and environments and deploy and trace proxies.
- Integrates with Apigee using the Apigee API.

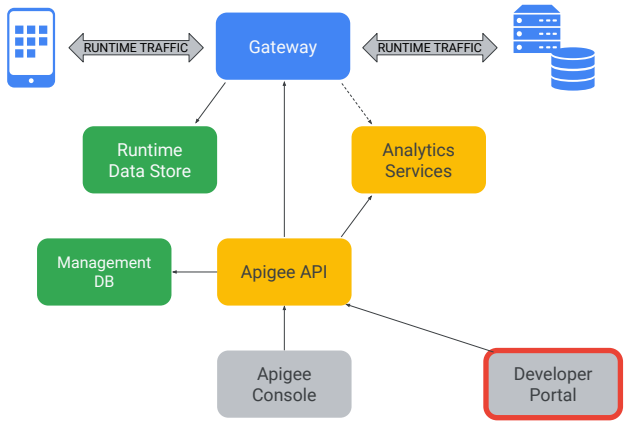


Apigee users, including organization administrators, use Apigee's **console** to manage organizations and environments and deploy and trace proxies.

The Apigee console uses the Apigee API to integrate with Apigee. Most of the tasks performed in the console can also be done using the Apigee API.

Developer portal

- Used by application developers to discover and sign up for APIs.
- Two types of developer portal: Drupal-based and integrated.
- Integrates with Apigee by using Apigee API.



A developer portal is a web interface designed to support app developers in the use of APIs implemented on Apigee.

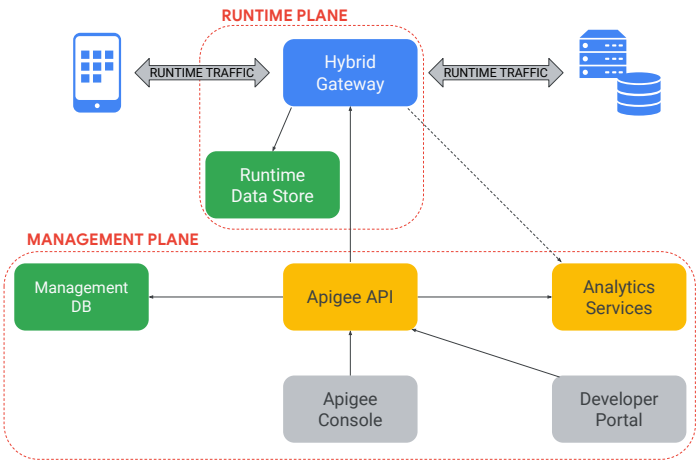
App developers can sign up for the portal, where they can discover APIs, exploring the APIs using live documentation, and register apps to use the APIs.

Apigee supports two types of developer portals: a flexible Drupal-based portal and a lightweight integrated portal. We will discuss the two types of portals in a later lecture.

Developer portals use the Apigee API to integrate with Apigee.

Hybrid deployment

- Fully featured.
- Management plane in Google Cloud.
- Containerized runtime components managed by customer in private data centers or other private clouds.
- Identical API developer experience.

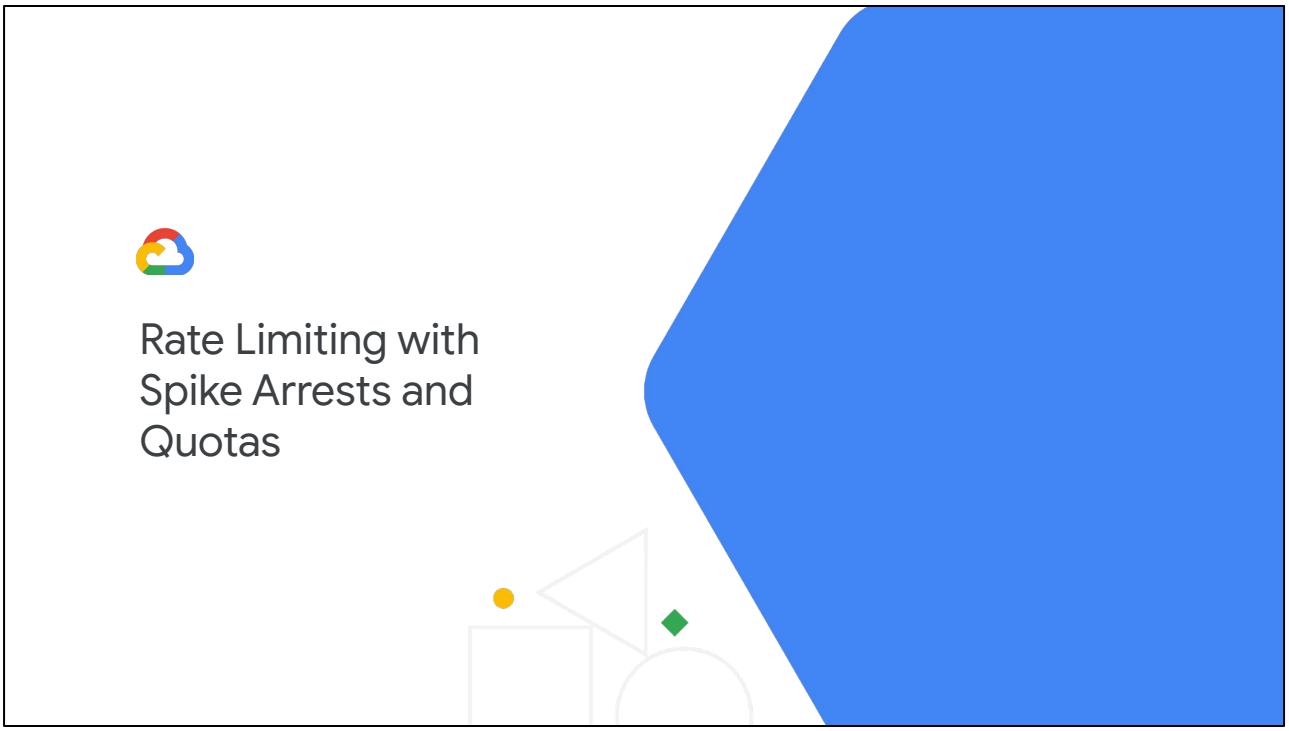


A hybrid deployment of Apigee provides a fully featured API management platform, with infrastructure management responsibilities shared between Google and the customer.

The management plane for hybrid is hosted in Google Cloud and is managed by Google.

The runtime plane is managed by the customer. Runtime plane components are containerized and can be run in private data centers, or in a private cloud in Google Cloud or other clouds.

From the perspective of an API developer, the hybrid deployment works the same way as the Google Cloud-managed deployment.



This lecture discusses methods for limiting traffic to your API proxies.

Rate limiting

- Rate limiting can be used to reject excess API traffic.
- Two built-in types of rate limiting:
  - Protecting against traffic spikes
  - Allowing a specific number of requests over a period of time

When you increase the usage of your API or make it available on the internet for the first time, your API traffic may significantly increase. Unless your backends can scale up to handle all the additional traffic, you may need to limit the rate of traffic to your backend services.

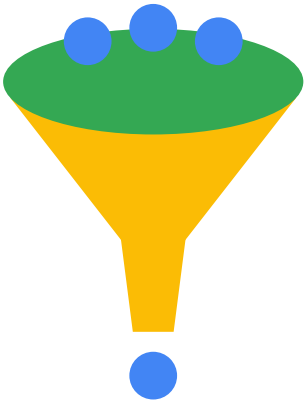
Rate limiting is the process of rejecting excess traffic to your API and your backend services. What "excess" means is up to you.

Apigee provides two built-in types of rate limiting.

The first type protects your backend services from spikes in traffic by limiting the allowed rate of API traffic.

The second type sets specific limits on the number of requests allowed over a period of time.

Traffic spikes



- Traffic spikes can overwhelm backends.
- Protecting against traffic spikes solves a technical problem.
- The SpikeArrest policy smooths traffic by allowing only a specified rate of requests.

Spikes of API traffic can overwhelm backends, thus causing services to become extremely slow or go completely out of service.

Protecting your backend against spikes in traffic is solving a technical problem, and is typically an IT concern.

We generally use load testing to determine how much sustained traffic can be handled by our backend services.

The SpikeArrest policy blocks traffic spikes by specifying a maximum rate of requests.

- Keeps track of when the last matching request was allowed:
  - Does not use a counter.
- Requests are grouped by *Identifier* value.
- Rejects request if the message processor allowed traffic for the same identifier too recently.
- Rejected traffic returns 429 Too Many Requests status code.

The most important feature of the SpikeArrest policy is that it does not use a counter. Instead, it keeps track of the time that the last matching request was allowed through the SpikeArrest policy.

The Identifier element specifies the group of traffic that will be rate-limited together. For example, if the identifier is the client ID of the application, any traffic for the same application will be considered when rate limiting using the SpikeArrest policy.

The SpikeArrest policy running in a proxy on a message processor will reject a request if that same message processor has allowed traffic for the same Identifier too recently.

We will discuss the Rate element in more detail.

When traffic is rejected, the policy will return the status code 429, Too Many Requests.

Let's take a look at how the identifier is used in the policy.

SpikeArrest policy

```
<SpikeArrest continueOnError="false" enabled="true"
 name="SA-GlobalSpikeArrest">
  <Rate>100ps</Rate>
  <!-- no identifier, so applies to all traffic -->
</SpikeArrest>
```

- Keeps track of when the last matching request was allowed:
  - Does not use a counter.
- Requests are grouped by *Identifier* value.
- Rejects request if the message processor allowed traffic for the same identifier too recently.
- Rejected traffic returns 429 Too Many Requests status code.

The first policy specifies a rate of 100 per second.

There is no Identifier configured for this policy, so this applies to all requests sent to this proxy.

This SpikeArrest policy would use all traffic for determining whether to allow a request through.

SpikeArrest policy

```
<SpikeArrest continueOnError="false" enabled="true"
 name="SA-GlobalSpikeArrest">
  <Rate>100ps</Rate>
  <!-- no identifier, so applies to all traffic -->
</SpikeArrest>
```

```
<SpikeArrest continueOnError="false" enabled="true"
 name="SA-AppSpikeArrest">
  <Rate>10pm</Rate>
  <Identifier ref="client_id"/>
</SpikeArrest>
```

- Keeps track of when the last matching request was allowed:
  - Does not use a counter.
- Requests are grouped by *Identifier* value.
- Rejects request if the message processor allowed traffic for the same identifier too recently.
- Rejected traffic returns 429 Too Many Requests status code.

The second policy uses an identifier with a reference to client underscore id.

The client_id variable is set when an API key or OAuth token is verified and contains the client ID for the application.

Therefore, the second policy would only check the last allowed traffic for the same application when determining whether to reject the traffic.

It is not unusual for a proxy to have two SpikeArrest policies like this, with one controlling the overall traffic rate and another controlling the rate allowed for each app.

SpikeArrest: Rate

- Can be specified per minute or per second.
- Specifies how often traffic is allowed.

```
<SpikeArrest continueOnError="false" enabled="true"
 name="SA-SpikeArrest">
  <Rate>10ps</Rate>
</SpikeArrest>
```

Let's look at the Rate configuration for SpikeArrest.

The rate for a SpikeArrest can be specified per minute or per second.

The rates are specified as a number followed by pm, indicating per minute, or ps, indicating per second.

The SpikeArrest policy does not use counters. Instead, the Rate element indicates how often you are allowed to pass through traffic on a given message processor.

SpikeArrest: Rate

- Can be specified per minute or per second.
- Specifies how often traffic is allowed.

```
<SpikeArrest continueOnError="false" enabled="true"
 name="SA-SpikeArrest">
  <Rate>10ps</Rate>
</SpikeArrest>
```

Calculating time period between requests:

Rate = 10ps
10 req per 1000 ms
= 1 req per 100ms

Let's look at a couple of examples, and see how rate can be used to calculate the time period between requests.

When the rate element is specified per second, the rate is calculated as a number of full requests in intervals of milliseconds.

If the rate is specified as 10 per second, this is equivalent to 10 requests per 1000 milliseconds, which is equal to 1 request per 100 milliseconds.

This means that a request would be rejected if the previous allowed request to the same proxy in the same message processor with the same identifier was less than 100 milliseconds ago.

SpikeArrest: Rate

- Can be specified per minute or per second.
- Specifies how often traffic is allowed.

```
<SpikeArrest continueOnError="false" enabled="true"
 name="SA-SpikeArrest">
  <Rate>30pm</Rate>
</SpikeArrest>
```

Calculating time period between requests:

Rate = 10ps
10 req per 1000 ms
= 1 req per 100ms

Rate = 30pm
30 req per 60 sec
= 1 req per 2 sec

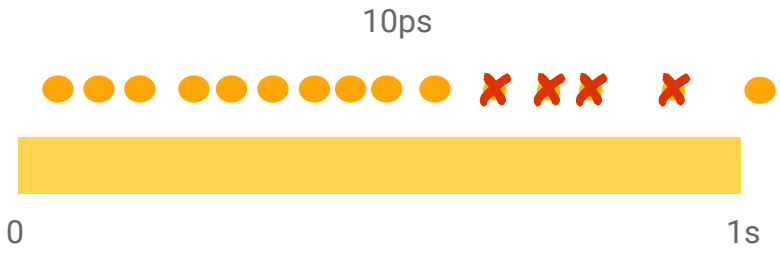
When the rate element is specified per minute, the rate is calculated as a number of full requests in intervals of seconds.

A rate of 30 per minute is equivalent to 30 requests per 60 seconds, so 30 per minute is one request every 2 seconds.

If a matching request was allowed less than 2 seconds ago, the traffic would be rejected.

How SpikeArrest does not work

- No counter



The SpikeArrest policy can be confusing. Let's try to get a deeper understanding of how the SpikeArrest policy works.

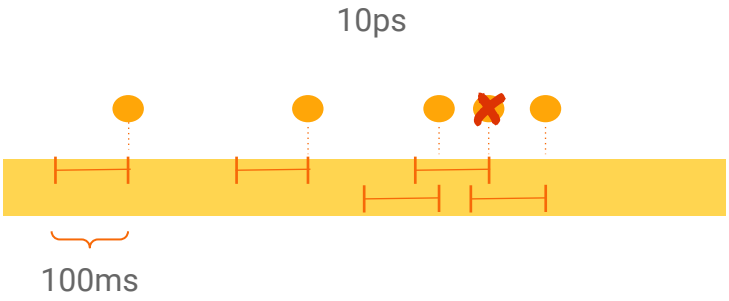
First, how SpikeArrest does NOT work.

SpikeArrest does not use a counter.

If the rate is 10 requests per second, the SpikeArrest policy does not keep track of the first 10 requests within the second, and reject the 11th.

How SpikeArrest works

- 10ps = 100ms between requests
- Rejects traffic if previous matching request came in less than 100ms ago.



Here is how SpikeArrest works.

Remember, a configured rate of 10 per second indicates 100 milliseconds between requests.

If UseEffectiveCount is true, remember to divide the time between requests by the number of message processors. In this case, we'll assume UseEffectiveCount is not true.

Each time the SpikeArrest policy is evaluated, it checks the last time that matching traffic was allowed through the SpikeArrest policy. In this case, if the previous matching request came in less than 100 milliseconds ago, the traffic will be rejected.

Looking at the example, the first three requests are each allowed because no traffic had been received in the previous 100 milliseconds.

The fourth request is rejected, because the third request had been allowed less than 100 milliseconds before the fourth.

The fifth request is allowed, because, even though the fourth request came in less than 100 milliseconds before the fifth, that fourth request was rejected, and therefore not counted as allowed traffic.

How SpikeArrest works

- Possible to send 2 total requests and have the second one be rejected.

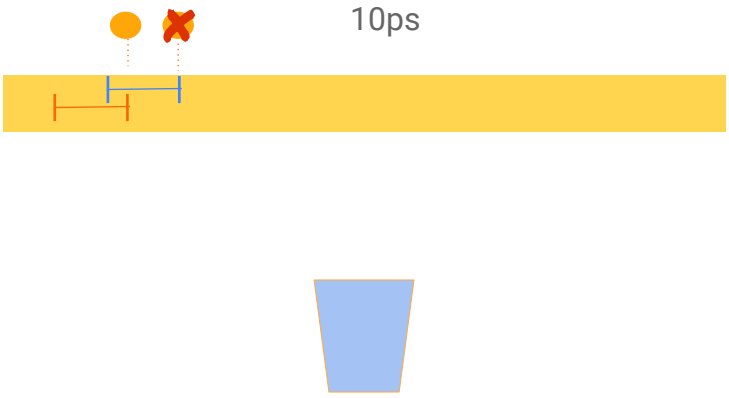
You may be wondering now whether it is possible to send 2 total requests, and have the second request be rejected. The answer is yes.

You might think that this doesn't constitute much of a spike, and you'd be right.

The SpikeArrest policy works very much like the examples I've been giving, but actually uses a slightly different algorithm.

How SpikeArrest works

- Possible to send 2 total requests and have the second one be rejected.
- Uses Token Bucket algorithm.



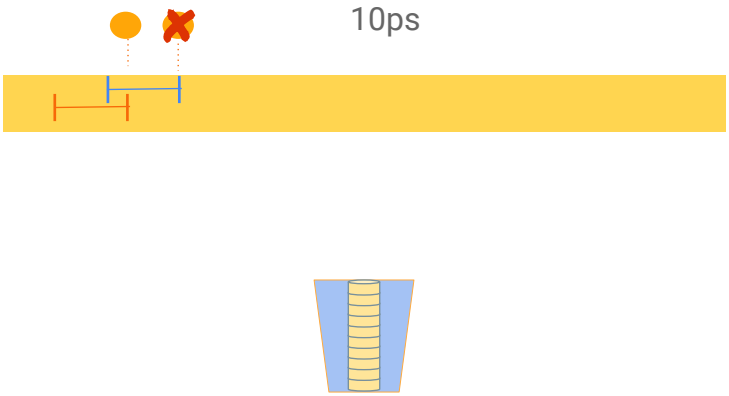
In order to allow for mini spikes without rejecting traffic, the SpikeArrest policy is implemented using the Token Bucket algorithm.

Here's how the Token Bucket algorithm works.

Imagine there is a bucket...

How SpikeArrest works

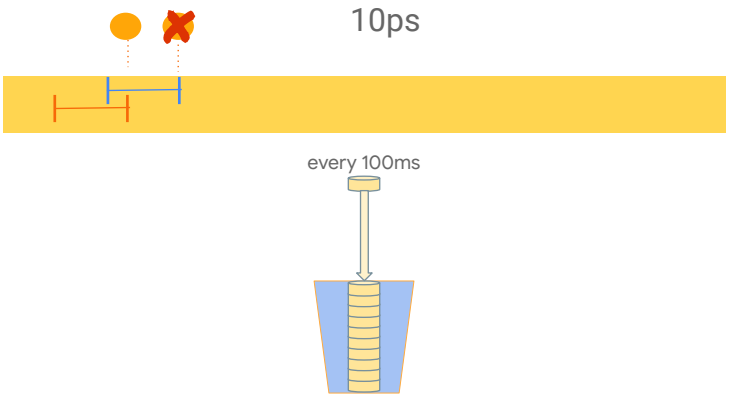
- Possible to send 2 total requests and have the second one be rejected.
- Uses Token Bucket algorithm.



...and it starts out filled with tokens.

How SpikeArrest works

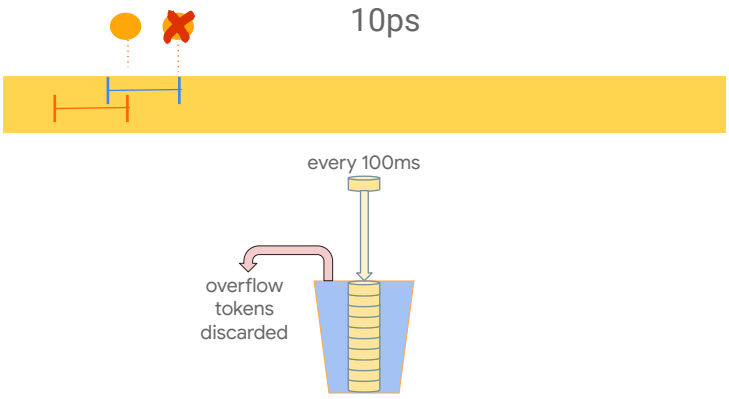
- Possible to send 2 total requests and have the second one be rejected.
- Uses Token Bucket algorithm.



If the configured rate is 10 per second, a token would be added to the bucket every 100 milliseconds.

How SpikeArrest works

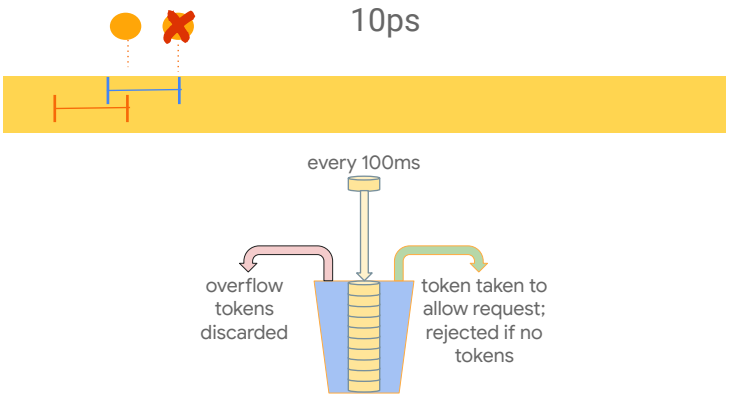
- Possible to send 2 total requests and have the second one be rejected.
- Uses Token Bucket algorithm.



If the bucket is already full, the token is discarded.

How SpikeArrest works

- Possible to send 2 total requests and have the second one be rejected.
- Uses Token Bucket algorithm.



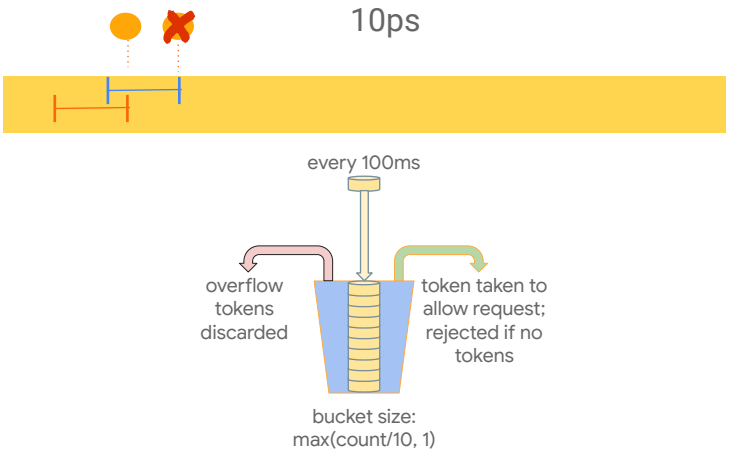
When a request is received, the bucket is checked for a token.

If there is at least one token in the bucket, the token is taken and the traffic is allowed. If there are no tokens in the bucket, the traffic is rejected.

You can see how this would allow bursts of traffic, but the overall rate of traffic would still be one request every 100 milliseconds.

How SpikeArrest works

- Possible to send 2 total requests and have the second one be rejected.
- Uses Token Bucket algorithm.

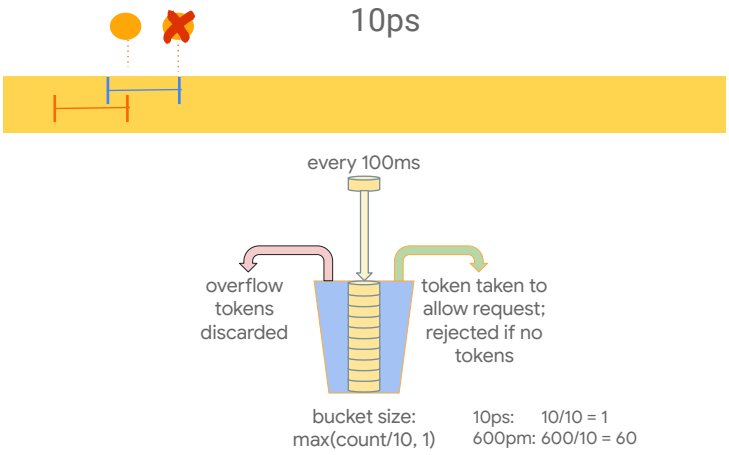


For the SpikeArrest policy, the size of the bucket depends on the number used in the configured rate.

The bucket size is equal to the count divided by 10, with a minimum bucket size of one.

How SpikeArrest works

- Possible to send 2 total requests and have the second one be rejected.
- Uses Token Bucket algorithm.



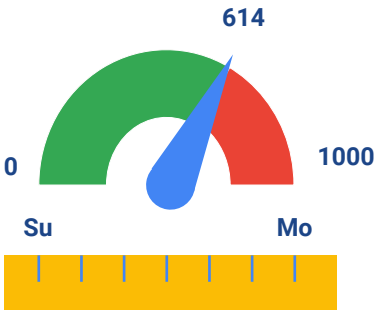
For 10 per second, ten divided by 10 is one. So the bucket size is 1, which means that this algorithm works very much like the earlier examples.

But for a configured rate of 600 per minute, which is equivalent to 10 per second, the bucket size would be 600 divided by 10, or 60.

This means that a burst of 60 simultaneous requests would be allowed, but then the bucket would be empty, and a token would be added to the bucket only once per 100 milliseconds.

When creating a SpikeArrest policy, you can choose the per minute or per second configuration to control this behavior.

Quotas



- A quota solves a business problem: how many requests should be allowed for a specific entity over a specified period of time?
- Once the limit is reached, policy rejects requests until the time period has elapsed.
- No maximum rate is specified.

Now, let's talk about quotas.

The SpikeArrest policy solves a technical problem: protecting your backends from excessive traffic spikes.

A Quota policy solves a business problem: how many requests should be allowed for a specific entity over a specified period of time?

The policy keeps track of the number of requests that have been accepted.

Each time a request is handled by the Quota policy, the counter is checked to see whether the maximum number of requests has been received. If the quota limit has not been reached, the quota counter is incremented and processing continues. If the quota limit has been reached, the quota policy raises a fault to reject the request. Requests will be rejected until the time period has elapsed, and the count is reset to zero.

A quota does not control the rate of requests, so quotas don't protect against traffic spikes. For example, if a quota allowed 1000 requests per month, it would be legal to use all 1000 requests during the first minute of the month.

The Quota and SpikeArrest policies solve two different problems, so it is very common to use both types of policies in the same proxy.

Quota policy

```
<Quota continueOnError="false" enabled="true"
 name="Q-AppQuota">
  <Allow count="1" countRef="allowedCount"/>
  <Interval ref="interval">1</Interval>
  <TimeUnit ref="timeUnit">month</TimeUnit>
  <Identifier ref="client_id"/>
</Quota>
```

- Counts matching requests over a specified time period.
- Count is typically shared among all message processors.
- Combination of proxy, policy, and *Identifier* uniquely identifies a quota counter:
  - If no Identifier is specified, all traffic through the policy is counted.
- Reject request returns with 429 Too Many Requests.

Unlike the SpikeArrest policy, the Quota policy uses a counter. The Quota policy keeps track of the number of matching requests received over a given time period.

The Quota policy count is typically shared among all of the message processors, with the counter value stored in the runtime data store.

A quota is scoped to a single proxy and policy. Counters cannot be shared between policies or proxies.

The combination of policy, proxy, and Identifier value uniquely specifies a Quota counter.

If no Identifier is specified, all traffic through the policy in the proxy is counted.

The Quota policy, like the SpikeArrest policy, returns 429 Too Many Requests when rejecting a request.

Quota: Allowed requests

- Number of allowed requests per time period specified using 3 configuration elements:
  - *Allow* (allowed requests per time period)
  - *TimeUnit* (minute/hour/day/week/month)
  - *Interval* (number of time units in the period)
- Each element can be set using a variable reference or hardcoded value.

```
<Quota continueOnError="false" enabled="true"
 name="Q-AppQuota">
  <Allow count="10" countRef="allowedCount"/>
  <Interval ref="interval">1</Interval>
  <TimeUnit ref="timeUnit">hour</TimeUnit>
  <Identifier ref="client_id"/>
</Quota>
```

The number of allowed requests per time period is specified using three configuration elements.

Allow is the number of allowed requests per time period.

TimeUnit is the unit of time to use: minute, hour, day, week, or month.

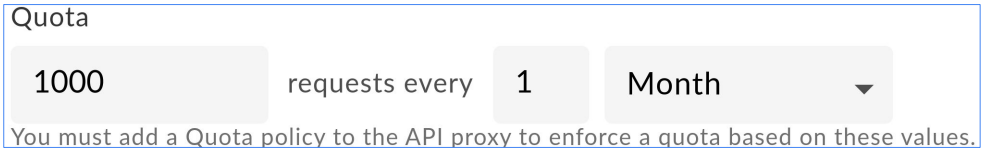
Interval is the number of time units in the period.

So if Allow is 10, Interval is 1, and TimeUnit is hour, the allowed quota is 10 requests per every 1 hour.

Each of these elements can be configured using a reference to a variable, a hardcoded value, or both. If the reference exists, and the variable has a value, that value is used. If there is no configured reference, or if the variable has a value of null, the hardcoded value is used.

Quota: API product quota settings

- The *Allow, Interval,* and *TimeUnit* variables can be set at the API Product level and accessed via variables created when an API key or token is verified.




```
<Quota continueOnError="false" enabled="true" name="Q-AppQuota">
  <Allow count="1" countRef="verifyapikey.VK-VerifyKey.apiproduct.developer.quota.limit"/>
  <Interval ref="verifyapikey.VK-VerifyKey.apiproduct.developer.quota.interval">1</Interval>
  <TimeUnit ref="verifyapikey.VK-VerifyKey.apiproduct.developer.quota.timeunit">month</TimeUnit>
  <Identifier ref="client_id"/>
</Quota>
```

The Allow, Interval, and TimeUnit variables are often configured using the API product quota settings.

When an API key or OAuth token is verified, the values of the quota settings for the API product are automatically populated into variables.

These variables can be used in the Quota policy to specify a quota for the application associated with the product.

The API product quota settings may be used to create different levels of service for applications.

For example, an API product named Basic might allow 100 requests per month, and an API product named Premium might allow 100 requests per day.

In the Quota policy shown here, variables populated by the VerifyAPIKey policy named VK-VerifyKey are used to set the quota values.

Quota: UseQuotaConfigInAPIProduct

- Simpler configuration for quotas using API products

```
<Quota continueOnError="false" enabled="true" name="Q-AppQuota">
  <UseQuotaConfigInAPIProduct stepName="VK-VerifyKey">
    <DefaultConfig>
      <Allow>1</Allow>
      <Interval>1</Interval>
      <TimeUnit>month</TimeUnit>
    </DefaultConfig>
  </UseQuotaConfigInAPIProduct>
  <Identifier ref="client_id"/>
</Quota>
```

The UseQuotaConfigInAPIProduct element is an alternative method for using the API product quota settings without needing to specify each of the very long variable names.

The stepName attribute specifies the policy that determines the API product for the calling application. This policy can be a VerifyAPIKey policy or an OAuth policy using the ValidateToken operation.

The DefaultConfig element specifies the default values used if the quota values are not set in the API product configured for the application.

Quota: Type

```
<Quota continueOnError="false" enabled="true"
 name="Q-AppQuota" type="rollingwindow">
  <Allow count="1" countRef="allowedCount"/>
  <Interval ref="interval">1</Interval>
  <TimeUnit ref="timeUnit">month</TimeUnit>
  <Identifier ref="client_id"/>
</Quota>
```

- The quota type specifies when the quota time period starts and when it resets.
- There are four quota types:
  - default (when type is not specified): begins on the next minute/hour/day/etc.
  - *calendar*: explicit start time
  - *flexi*: begins when first message is received
  - *rollingwindow:* counts requests over past time period

A quota is specified with a type. The quota type specifies when the quota time period starts and when it resets.

There are four different types of quotas.

When a type is not specified, the quota period starts at the beginning of the next GMT minute, hour, day, or month, depending on the time unit. It then resets every interval number of time units. For example, if the quota is 1000 requests per 1 day, the quota would reset at midnight each day.

Calendar has an explicit start time. The StartTime is specified as an element in the policy. Like the default quota type, a calendar type quota is reset each time an interval number of time units elapses.

The flexi type is like the calendar type, except that the time period begins when the first request is handled by the Quota policy.

The rolling window calendar type works differently from the other calendar types.

Requests are counted over the time period ending at the time of the request, and accepted or rejected based on this rolling window. So, for example, if the Quota is checking a request at 10 o'clock, and the quota allows 100 requests every two hours, the request would be rejected if 100 or more requests had been accepted between 8 o'clock and 10 o'clock.

Quota: Distributed, Synchronous

- By default, quota counter is separate for each message processor (*Distributed=false*).
- When *Distributed=true*, a shared counter is used for the message processors.
  - This is the more common choice.
- If *Distributed=true*, update of the counter can be specified as synchronous (*Synchronous=true*) or asynchronous (*Synchronous=false*).

```
<Quota continueOnError="false" enabled="true"
 name="Q-AppQuota">
  <Allow count="1" countRef="allowedCount"/>
  <Interval ref="interval">1</Interval>
  <TimeUnit ref="timeUnit">month</TimeUnit>
  <Identifier ref="client_id"/>
  <Distributed>true</Distributed>
  <Synchronous>false</Synchronous>
</Quota>
```

By default, quota counts are separately stored for each message processor. This is equivalent to setting the Distributed element to false.

When Distributed is set to true, a shared counter is used for all the message processors. This is typically the setting you will want to use.

Scaling up and down message processors should typically not affect the number of allowed requests for a quota.

When using a quota with distributed set to true, you can specify whether the update of the counter is synchronous or asynchronous by setting the Synchronous element.

Quota: AsynchronousConfiguration

```
<Quota continueOnError="false" enabled="true"
 name="Q-AppQuota">
  <Allow count="1" countRef="allowedCount"/>
  <Interval ref="interval">1</Interval>
  <TimeUnit ref="timeUnit">month</TimeUnit>
  <Identifier ref="client_id"/>
  <Distributed>true</Distributed>
  <Synchronous>false</Synchronous>
  <AsynchronousConfiguration>
    <SyncMessageCount>5</SyncMessageCount>
  </AsynchronousConfiguration>
</Quota>
```

- *SyncIntervalInSeconds* syncs with the central counter every n seconds.
  - 10 seconds is the minimum.
- *SyncMessageCount* syncs with the central counter every time the message processor has collected n requests (also syncs every 60 seconds if *SyncIntervalInSeconds* is not set).
- If *AsynchronousConfiguration* is not specified, default is to sync every 10 seconds.

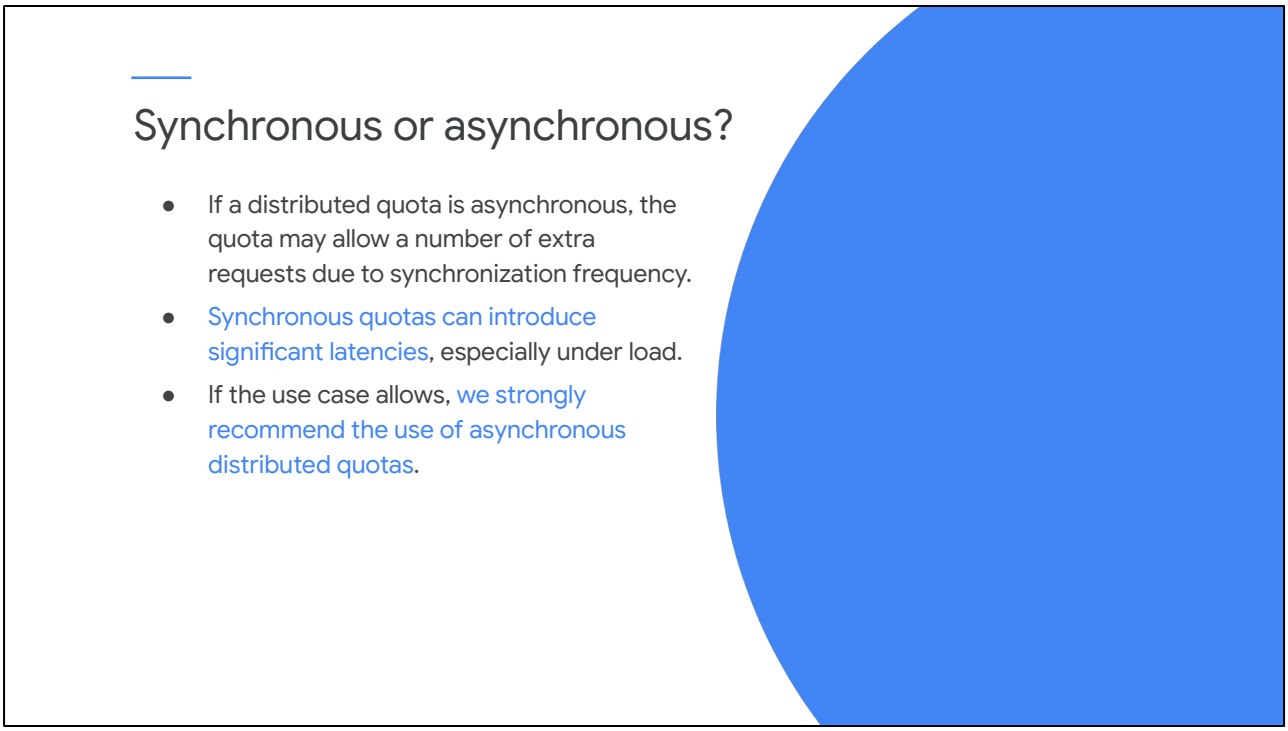
When Synchronous is set to false, you can configure how often the central counter is updated by using the AsynchronousConfiguration settings.

Setting SyncIntervalInSeconds specifies the frequency that the message processor synchronizes its count with the central counter. 10 seconds is the minimum interval you can set.

SyncMessageCount specifies that the counter be synchronized whenever the configured number of requests has been received.

If SyncMessageCount is set, but SyncIntervalInSeconds is not, the count will also be synchronized at least every 60 seconds.

If Synchronous is set to false, but the AsynchronousConfiguration settings are not set, the count will be synchronized every 10 seconds.



So how do you decide between synchronous and asynchronous for distributed quotas?

When a distributed quota is asynchronous, some extra requests may be accepted due to the synching frequency.

A synchronous quota is much less likely to allow extra requests.

However, a synchronous quota can introduce significant latencies, especially under load. Each quota policy must check the runtime data store before allowing the request through.

In most cases, a few extra allowed requests won't make a difference.

We strongly recommend that you use asynchronous quotas if your use case will tolerate the extra requests.

Quota: MessageWeight

- *MessageWeight* can be used to modify the impact of a request against the quota.
- Could be used to count each item within a batch request against the quota.

```
<Quota continueOnError="false" enabled="true"
 name="Q-AppQuota">
  <Allow count="1" countRef="allowedCount"/>
  <Interval ref="interval">2</Interval>
  <TimeUnit ref="timeUnit">hour</TimeUnit>
  <Identifier ref="client_id"/>
  <MessageWeight ref="batchSize"/>
</Quota>
```

The MessageWeight element is used to modify the impact of a request against the quota.

Using a MessageWeight of 2 would count that request the same as two requests that have MessageWeight of 1.

When MessageWeight is not specified, the default weight is 1.

MessageWeight can be used when certain messages use more resources than others. For example, an API that allows multiple items in a batch request might count 1 request per item in the batch.

ResetQuota policy

- ResetQuota decreases the number of counted requests for a quota counter.

```
<ResetQuota continueOnError="false"
 enabled="true" name="Q-ResetQuota">
  <Quota name="Q-AppQuota">
    <Identifier name="_default">
      <Allow ref="newEntitlement">100</Allow>
    </Identifier>
  </Quota>
</ResetQuota>
```

The ResetQuota policy is used to decrease the number of counted requests in a quota counter.

These requests are decreased without regard to the quota time period; the quota allowance will reset to the Allow value at the time the current quota period ends.

ResetQuota policy

```
<ResetQuota continueOnError="false"
 enabled="true" name="Q-ResetQuota">
  <Quota name="Q-AppQuota">
    <Identifier name="_default">
      <Allow ref="newEntitlement">100</Allow>
    </Identifier>
  </Quota>
</ResetQuota>
```

- ResetQuota decreases the number of counted requests for a quota counter.
- *Quota* name specifies the Quota policy name.

The Quota element specifies which quota policy is being referenced.

The quota policy name can also be set via reference.

ResetQuota policy

```
<ResetQuota continueOnError="false"
 enabled="true" name="Q-ResetQuota">
  <Quota name="Q-AppQuota">
    <Identifier name="_default">
      <Allow ref="newEntitlement">100</Allow>
    </Identifier>
  </Quota>
</ResetQuota>
```

- ResetQuota decreases the number of counted requests for a quota counter.
- *Quota* name specifies the Quota policy name.
- *Identifier* specifies the identifier value to be updated.

Identifier indicates the identifier value for the quota counter to be updated.

This value can also be specified by name or reference.

For example, if the Quota policy uses an identifier of the client id, a variable containing the same client id will generally be used for the ResetQuota policy.

ResetQuota policy

```
<ResetQuota continueOnError="false"
 enabled="true" name="Q-ResetQuota">
  <Quota name="Q-AppQuota">
    <Identifier name="_default">
      <Allow ref="newEntitlement">100</Allow>
    </Identifier>
  </Quota>
</ResetQuota>
```

- ResetQuota decreases the number of counted requests for a quota counter.
- *Quota* name specifies the Quota policy name.
- *Identifier* specifies the identifier value to be updated.
  - Use the name *_default* to affect the default quota counter.

If no identifier is used in the Quota policy, use the name underscore default to update the quota counter.

ResetQuota policy

```
<ResetQuota continueOnError="false"
 enabled="true" name="Q-ResetQuota">
  <Quota name="Q-AppQuota">
    <Identifier name="_default">
      <Allow ref="newEntitlement">100</Allow>
    </Identifier>
  </Quota>
</ResetQuota>
```

- ResetQuota decreases the number of counted requests for a quota counter.
- *Quota* name specifies the Quota policy name.
- *Identifier* specifies the identifier value to be updated.
  - Use the name *_default* to affect the default quota counter.
- *Allow* is a required configuration that specifies the number of counted requests to be removed.

Allow specifies the number of counted requests to be removed.

For example, if the current count is 500, and the maximum count is also 500, specifying an Allow value of 100 will reduce the count to 400 and allow 100 more requests to come in during the time period.

In this lab you add traffic management to your retail API. You add a spike arrest intended to protect against bursts of traffic, and add a quota that uses limits set in the API product.

This lecture will discuss caching within API proxies.

We will discuss how caches work in Apigee, how to use general-purpose caching to store information between calls, and how to use a response cache to easily eliminate unnecessary calls to backend services.

Caching

- Entry is persisted in a cache using a cache key.
- Entry can be retrieved from a cache by providing the cache key.
- Entry is stored with a time-to-live (TTL).
  - Entry is automatically removed from cache when TTL elapses.

Caches store entries by cache key, which is an index into the data store.

The cache key can be used to search for and retrieve an entry from the cache.

When an entry is stored, a time-to-live, or TTL, is also provided.

This provides a method for expiring cache entries. Once the time to live has elapsed, the entry is automatically removed from the cache.

Caching

- Persist state between calls.
- Eliminate unnecessary backend service calls.
  - Improve stability of backend services.
  - Improve call latency.
  - Reduce cost of metered services.

In an API proxy, you can persist state between calls using a cache.

You can also use caching to eliminate unnecessary backend service calls, by caching HTTP responses in the API proxy.

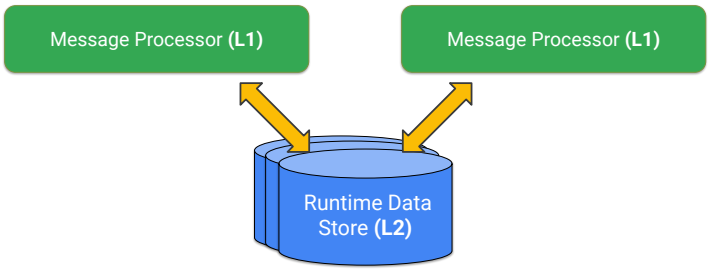
Reducing traffic to the backend can improve stability of your backend services and allow your APIs to handle more traffic.

Call latency is improved for calls that can be served from cache.

And caching responses can reduce the number of calls to a service, and thus reduce the cost of a metered service.

L1 cache

- L1 in-memory cache exists in each MP.
- L1 cache is checked first.
- L1 caches entries for a short period of time.



Before we discuss how caching policies can be used in your proxies, let's learn how caches work in Apigee gateways.

Apigee caches have two levels: L1 and L2.

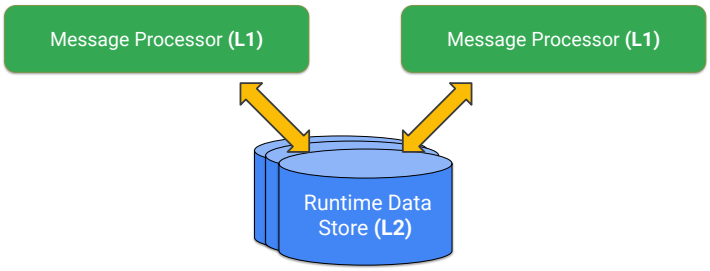
There is a level 1, or L1, in-memory cache in each message processor.

The L1 cache is checked first whenever a proxy is looking up an entry in the cache. If the entry is in the L1 cache, it is available very quickly.

The L1 cache automatically caches entries for a short period of time, currently 1 second. Also, if the cache fills up, least recently used entries would be removed from the L1 cache.

L2 cache

- L2 cache persisted in runtime data store.
- Entries are removed from L2 based on specified TTL only, unless the maximum number of cache entries is reached.
- L2 is slower than L1 but much faster than network calls.
- MP populates L1 cache when entry is read from L2.



When a message processor does not find an entry in L1 cache, it next checks L2 cache.

The entries for a level 2, or L2 cache are stored in a persistent database, the runtime data store inside a Cassandra database.

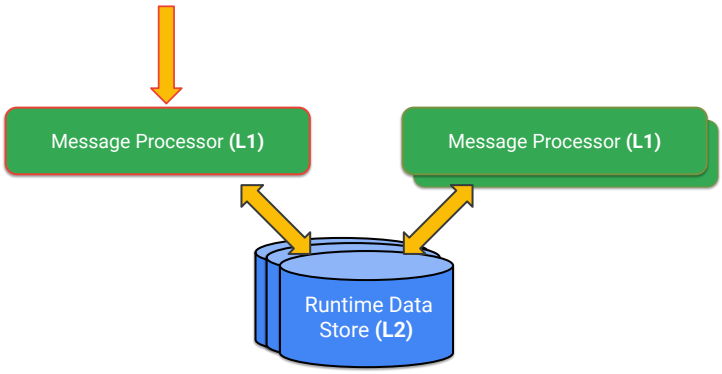
L2 cache entries are removed only when the cache entry's time to live has elapsed, unless the maximum number of cache entries is reached.

L2 cache is slower than L1 cache, but much faster than making a network call to retrieve the data.

When a message processor successfully retrieves a cache entry from L2 cache, the message processor stores the entry in L1 cache for a short time.

Entry updates and deletes

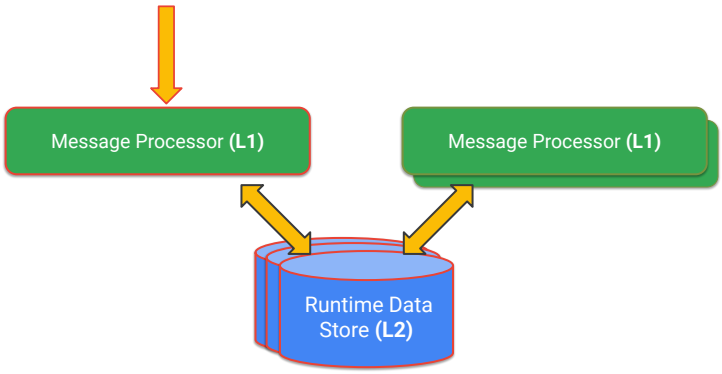
- When an entry is updated or deleted by an MP, it is updated/deleted in both the L1 cache and the L2 runtime data store.



When an entry is updated or deleted by a message processor, it is updated or deleted in the message processor's L1 cache...

Entry updates and deletes

- When an entry is updated or deleted by an MP, it is updated/deleted in both the L1 cache and the L2 runtime data store.

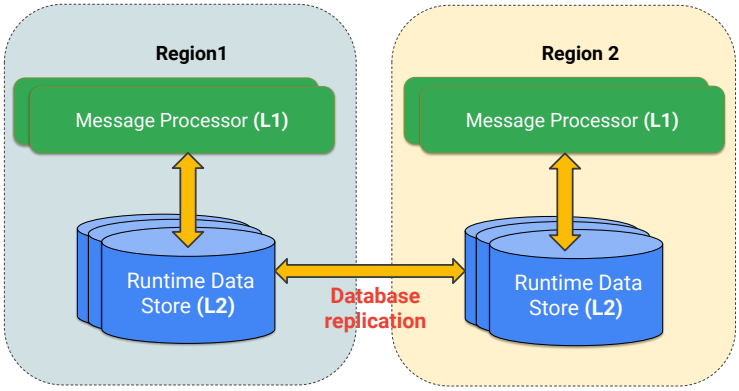


and the shared L2 cache in the runtime data store.

That entry would be unchanged in the L1 cache of other message processors. Any stale data would remain in the L1 cache until the short L1 time-to-live expired.

Multi-region

- L2 changes in the data store are propagated to other regions using database replication.
- Consider stale L1 data and L2 data store replication delays when architecting your proxies.



L2 changes in the runtime data store are propagated to other regions using database replication.

Data can be stale for short periods of time in the L1 cache, and L2 cache database replication may cause different regions to receive data at different times.

You should architect your proxies, and your use of caches, with this behavior in mind.

L1 automatic caching

- Apigee automatically caches the following entities in L1 cache for 180 seconds after access:
  - OAuth access tokens
  - Developers, developer apps, and API products
  - Custom attributes on access tokens, developers, developer apps, and API products

Apigee automatically caches several types of entities in the L1 cache for 180 seconds after access.

OAuth access tokens, developers, developer apps, and API products are all cached automatically for 180 seconds after access, as are any custom attributes attached to them.

General-purpose caching

- Store and retrieve strings
- Policies for population, look up, and invalidation of cache entries

The general purpose caching policies can be used to store and retrieve strings.

There are separate policies to populate, look up, and invalidate entries from the cache.

PpopulateCache policy

```
<PopulateCache continueOnError="false"
 enabled="true" name="PC-SaveBackendToken">
  <CacheResource>backendData</CacheResource>
  <CacheKey>
    <Prefix>BackendAuth</Prefix>
    <KeyFragment ref="userId"/>
    <KeyFragment ref="backendService"/>
  </CacheKey>
  <ExpirySettings>
    <TimeoutInSeconds ref="tknTTL">300</TimeoutInSeconds>
  </ExpirySettings>
  <Source>backendToken</Source>
</PopulateCache>
```

- Adds an entry to a cache.
- *CacheResource* specifies cache name.

The populate cache policy is used to add an entry to the cache.

The CacheResource element specifies a logical name of the cache. The cache resource is implicitly created when the proxy is deployed.

Caches are always scoped at the environment level.

If the CacheResource element is not specified, a default cache is used.

PpopulateCache policy

```
<PopulateCache continueOnError="false"
 enabled="true" name="PC-SaveBackendToken">
  <CacheResource>backendData</CacheResource>
  <CacheKey>
    <Prefix>BackendAuth</Prefix>
    <KeyFragment ref="userId"/>
    <KeyFragment ref="backendService"/>
  </CacheKey>
  <ExpirySettings>
    <TimeoutInSeconds ref="tknTTL">300</TimeoutInSeconds>
  </ExpirySettings>
  <Source>backendToken</Source>
</PopulateCache>
```

- Adds an entry to a cache.
- *CacheResource* specifies cache name.
- Expiration set in *ExpirySettings* by date (*ExpiryDate* - mm-dd-yyyy), time of day (*TimeOfDay* - HH:mm:ss), or number of seconds (*TimeoutInSeconds*).

Expiration for the entry is set using the ExpirySettings element.

You can expire an entry at a particular date, a particular time of day every day, or in a specified number of seconds.

Each of these types of expiration can be hardcoded or specified using a reference to a variable, with the variable reference taking precedence.

If the ExpirySettings element is not provided, the entry will expire based on the default time to live for the CacheResource.

PopulateCache policy

```
<PopulateCache continueOnError="false"
 enabled="true" name="PC-SaveBackendToken">
  <CacheResource>backendData</CacheResource>
  <CacheKey>
    <Prefix>BackendAuth</Prefix>
    <KeyFragment ref="userId"/>
    <KeyFragment ref="backendService"/>
  </CacheKey>
  <ExpirySettings>
    <TimeoutInSeconds ref="tknTTL">300</TimeoutInSeconds>
  </ExpirySettings>
  <Source>backendToken</Source>
</PopulateCache>
```

- Adds an entry to a cache.
- *CacheResource* specifies cache name.
- Expiration set in *ExpirySettings* by date (*ExpiryDate* - mm-dd-yyyy), time of day (*TimeOfDay* - HH:mm:ss), or number of seconds (*TimeoutInSeconds*).
- *Source* indicates the name of the variable containing the string value to be stored.

The Source element specifies the name of the variable that contains the string value to be stored.

Cache: Prefix, Scope

- *Prefix* specifies a prefix for cache keys.

```
<PopulateCache continueOnError="false"
 enabled="true" name="PC-SaveToken">
  <CacheResource>backendTokenCache</CacheResource>
  <CacheKey>
    <Prefix>BackendAuth</Prefix>
    <KeyFragment ref="userId"/>
    <KeyFragment ref="backendService"/>
  </CacheKey>
  <Source>backendToken</Source>
</PopulateCache>
```

The Prefix element can be used to specify a prefix for the CacheKey string.

The prefix will scope the entry so that it can only be retrieved using the same prefix.

If the prefix element is not set, ...

Cache: Prefix, Scope

- *Prefix* specifies a prefix for cache keys.
- If the *Prefix* element is not set, *Scope* is used to specify the prefix:
  - *Global*: all proxies in environment
  - *Application*: available for every endpoint in the proxy
  - *Exclusive*: only this proxy and endpoint

```
<PopulateCache continueOnError="false"
 enabled="true" name="PC-SaveToken">
  <CacheResource>backendTokenCache</CacheResource>
  <CacheKey>
    <KeyFragment ref="userId"/>
    <KeyFragment ref="backendService"/>
  </CacheKey>
  <Scope>Exclusive</Scope>
  <Source>backendToken</Source>
</PopulateCache>
```

org: apigee env: test proxy: orders-v1
attached in proxy endpoint: ep

Global: apigee__test
Application: apigee__test__orders-v1
Exclusive: apigee__test__orders-v1__ep

...the Scope element should contain the cache's scope. The prefix will be automatically generated based on the scope.

A Global scope makes the entry available for all proxies in the environment. The Global prefix includes the organization name and the environment name.

Application scope makes the entry available anywhere in the existing proxy. The prefix includes the organization, environment, and proxy name.

Exclusive scope adds the name of the endpoint in which the policy is attached.

The Global and Application scopes tend to be the most useful.

Cache keys

- Cache keys are generated from prefix and key fragments.
- Double underscores are used as separators.

```
<PopulateCache continueOnError="false"
 enabled="true" name="PC-SaveToken">
  <CacheResource>backendTokenCache</CacheResource>
  <CacheKey>
    <KeyFragment ref="userId"/>
    <KeyFragment ref="backendService"/>
  </CacheKey>
  <Scope>Application</Scope>
  <Source>backendToken</Source>
</PopulateCache>
```

apigee__test__orders-v1__joeapigeek__orders
(scope) (key fragments)

Cache keys are generated using the prefix and the key fragments.

Each key fragment can be specified as a hardcoded value or a reference.

Double underscores separate the concatenated parts of the cache key string.

In this example, the Application scope specifies the prefix, which includes the organization, environment, and proxy name.

The two key fragments come from the userId variable and the backendService variable.

The prefix and fragments are separated by double underscores.

LookupCache policy

- Looks for an entry in a cache.

```
<LookupCache continueOnError="false"
 enabled="true" name="LC-GetToken">
  <CacheResource>backendData</CacheResource>
  <CacheKey>
    <Prefix>BackendAuth</Prefix>
    <KeyFragment ref="userId"/>
    <KeyFragment ref="backendService"/>
  </CacheKey>
  <CacheLookupTimeoutInSeconds>5</CacheLookupTimeoutInSeconds>
  <AssignTo>backendToken</AssignTo>
</LookupCache>
```

The LookupCache policy looks for an entry in a cache.

The prefix, scope, and cache key work the same way as in the PopulateCache policy.

LookupCache policy

```
<LookupCache continueOnError="false"
 enabled="true" name="LC-GetToken">
  <CacheResource>backendData</CacheResource>
  <CacheKey>
    <Prefix>BackendAuth</Prefix>
    <KeyFragment ref="userId"/>
    <KeyFragment ref="backendService"/>
  </CacheKey>
  <CacheLookupTimeoutInSeconds>5</CacheLookupTimeoutInSeconds>
  <AssignTo>backendToken</AssignTo>
</LookupCache>
```

- Looks for an entry in a cache.
- *AssignTo* indicates the name of the output variable.

The AssignTo element specifies the name of the output variable to be populated with the value of the entry.

LookupCache policy

```
<LookupCache continueOnError="false"
 enabled="true" name="LC-GetToken">
  <CacheResource>backendData</CacheResource>
  <CacheKey>
    <Prefix>BackendAuth</Prefix>
    <KeyFragment ref="userId"/>
    <KeyFragment ref="backendService"/>
  </CacheKey>
  <CacheLookupTimeoutInSeconds>5</CacheLookupTimeoutInSeconds>
  <AssignTo>backendToken</AssignTo>
</LookupCache>
```

- Looks for an entry in a cache.
- *AssignTo* indicates the name of the output variable.
- *CacheLookupTimeoutInSeconds* specifies the timeout for cache lookup:
  - Default is 30 seconds; setting it shorter is a good idea.

CacheLookupTimeoutInSeconds specifies the timeout for the cache lookup.

The default timeout is 30 seconds, which is a very long timeout, so you should generally set a shorter timeout.

LookupCache policy

```
<LookupCache continueOnError="false"
 enabled="true" name="LC-GetToken">
  <CacheResource>backendData</CacheResource>
  <CacheKey>
    <Prefix>BackendAuth</Prefix>
    <KeyFragment ref="userId"/>
    <KeyFragment ref="backendService"/>
  </CacheKey>
  <CacheLookupTimeoutInSeconds>5</CacheLookupTimeoutInSeconds>
  <AssignTo>backendToken</AssignTo>
</LookupCache>
```

lookupcache.LC-GetToken.cachehit = true/false

- Looks for an entry in a cache.
- *AssignTo* indicates the name of the output variable.
- *CacheLookupTimeoutInSeconds* specifies the timeout for cache lookup:
  - Default is 30 seconds; setting it shorter is a good idea.
- Cachehit variable is populated.

When the policy has completed, it sets a cachehit variable to true or false, indicating whether the entry was found and returned.

InvalidateCache policy

- Purge entries from a cache.

```
<InvalidateCache continueOnError="false"
 enabled="true" name="IC-RemoveToken">
  <CacheResource>backendData</CacheResource>
  <CacheKey>
    <KeyFragment ref="userId"/>
  </CacheKey>
  <Scope>Exclusive</Scope>
  <CacheContext>
    <APIProxyName>orders-v1</APIProxyName>
  </CacheContext>
</InvalidateCache>
```

The InvalidateCache policy purges entries from a cache.

Entries that match the supplied CacheKey will be purged.

InvalidateCache policy

- Purge entries from a cache.
- *CacheContext* is used to overwrite the scoped API proxy name (*APIProxyName*), Proxy Endpoint name (*ProxyName*), and Target Endpoint name (*TargetName*) when building the prefix.
    - Invalidation may be in different proxy.

```
<InvalidateCache continueOnError="false"
 enabled="true" name="IC-RemoveToken">
  <CacheResource>backendData</CacheResource>
  <CacheKey>
    <KeyFragment ref="userId"/>
  </CacheKey>
  <Scope>Exclusive</Scope>
  <CacheContext>
    <APIProxyName>orders-v1</APIProxyName>
  </CacheContext>
</InvalidateCache>
```

The CacheContext element is used to override the API proxy name, ProxyEndpoint name, and TargetEndpoint name when the policy builds a prefix using the scope.

This may be necessary if cache invalidation is being done in a different proxy from where cache population occurred.

Pattern: Caching expensive calls

- API calls can be expensive:
  - Calls cost money.
  - Calls can be slow or resource-intensive.
- Can cache those calls and serve from cache where possible.

Sometimes API calls can be expensive.

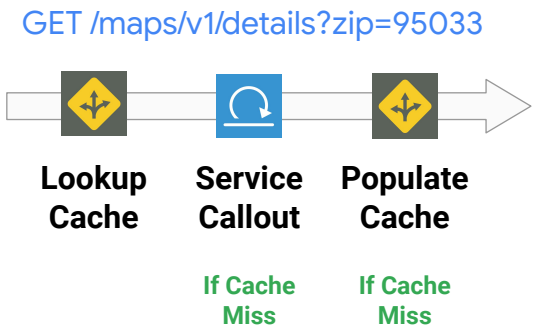
They can be financially expensive, where you are paying for access to a service.

Or they can be slow or resource-intensive, where repeatedly calling the service can add significant latency.

If the API's data is cacheable, the LookupCache and PopulateCache policies may be used to store the information in a cache and reduce the number of calls to the expensive service.

Pattern: Caching expensive calls

- API calls can be expensive:
  - Calls cost money.
  - Calls can be slow or resource-intensive.
- Can cache those calls and serve from cache where possible.
- Example: Calling a third-party mapping API that returns details about a ZIP code.



An example might be a third-party mapping API that returns details about a ZIP code. The data from this API is probably cacheable, because there are a relatively small number of ZIP codes, and the mapping data is probably relatively static.

The pattern is relatively simple.

The proxy first attempts to look up the entry in the cache by ZIP code. If the entry is not in the cache, the proxy calls out to the service and populates the cache with the result.

If the entry is in the cache, the ServiceCallout and PopulateCache policies are skipped.

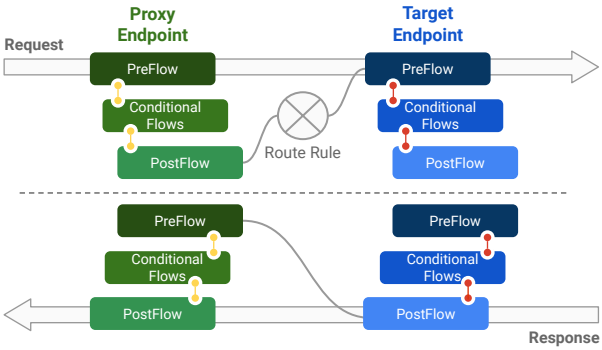
Response caching

- Cache entire HTTP responses.

The second type of caching used in Apigee proxies is response caching.

Entire HTTP responses are cached, which eliminates calls to the backend when the response would be the same as the cached response.

ResponseCache policy



- A ResponseCache policy streamlines the process of caching HTTP responses.
- The policy handles both the lookup and population of the cache.
- The policy is attached in exactly two places: a request flow and a response flow.

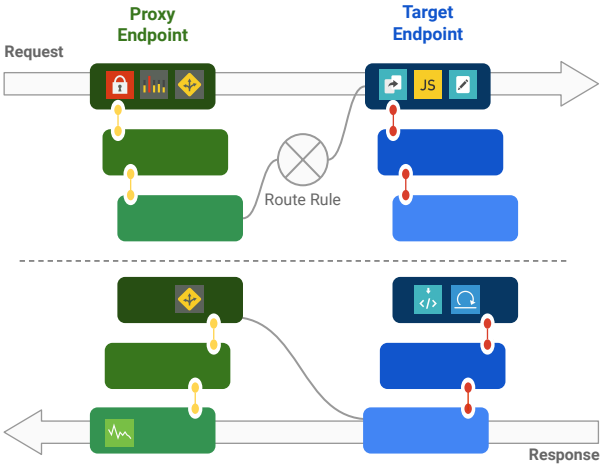
A ResponseCache policy is used to streamline the process of caching HTTP responses.

The ResponseCache policy handles both the lookup and population of the cache.

The policy is attached in the request flow, where it checks the cache, and in the response flow, where it populates the cache.

How it works (attachment)

- The ResponseCache policy is attached in both the request flow and the response flow.

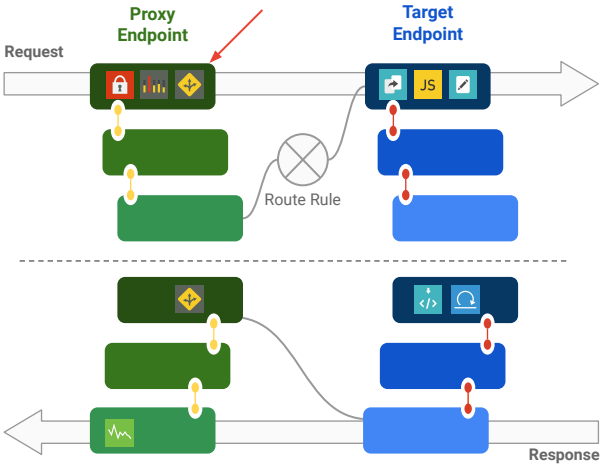


Let's look at how the ResponseCache policy works.

The ResponseCache policy is attached in the request flow and the response flow.

How it works (attachment)

- The ResponseCache policy is attached in both the request flow and the response flow.
- The policy is attached after the security and input validation policies in the Proxy Request PreFlow.

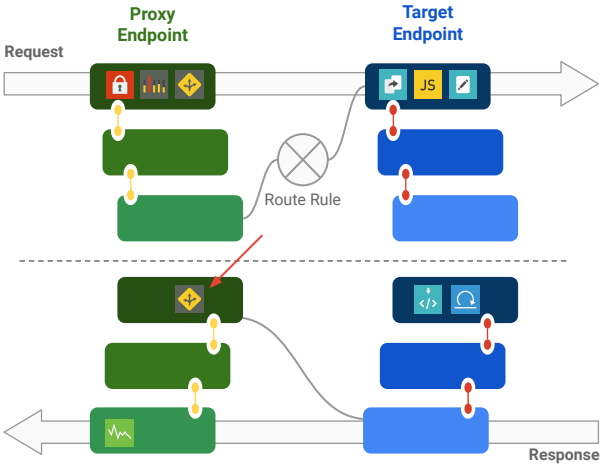


In the request flow, the response cache policy should be attached after the security and input validation policies.

If the request is not valid, the response cache policy should not be used.

How it works (attachment)

- The ResponseCache policy is attached in both the request flow and the response flow.
- The policy is attached after the security and input validation policies in the Proxy Request PreFlow.
- The response flow attachment is generally as far toward the end of the response as possible.

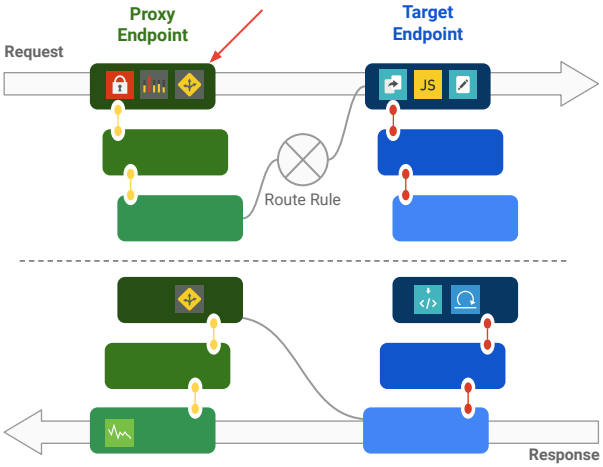


In the response flow, the policy should be attached toward the end of the response handling.

Some operations, like logging, would need to happen after the ResponseCache step in the response flow.

How it works (cache miss)

- At the request flow attachment, the cache is checked for a match to the cache key.

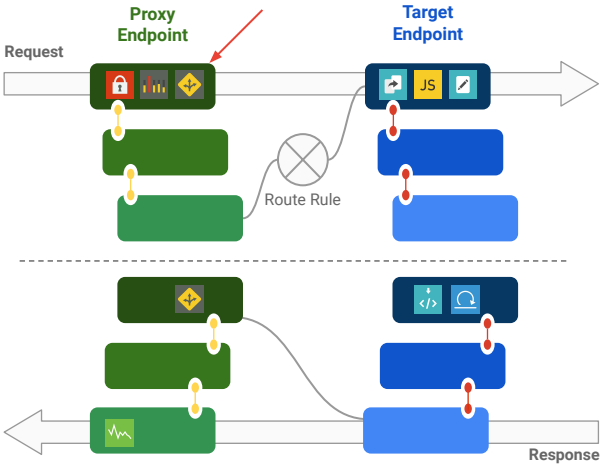


When a request comes in, policies are executed up to the ResponseCache policy in the request flow.

The ResponseCache policy uses the cache key to check for a matching entry.

How it works (cache miss)

- At the request flow attachment, the cache is checked for a match to the cache key.
- There is no match, so the rest of the policies are executed, as is the call to the target.

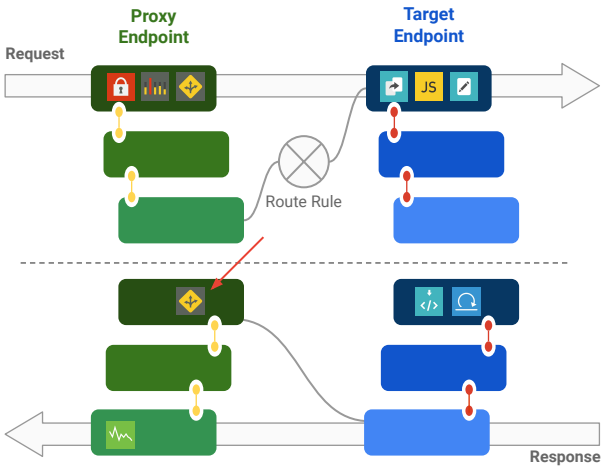


When there is no match, which is called a cache miss, processing continues in the request flow as usual.

The rest of the policies in the request flow are executed, and then the backend target is called.

How it works (cache miss)

- At the request flow attachment, the cache is checked for a match to the cache key.
- There is no match, so the rest of the policies are executed, as is the call to the target.
- When execution reaches the response flow attachment, the cache is populated with the response message, and then the rest of the execution is completed.

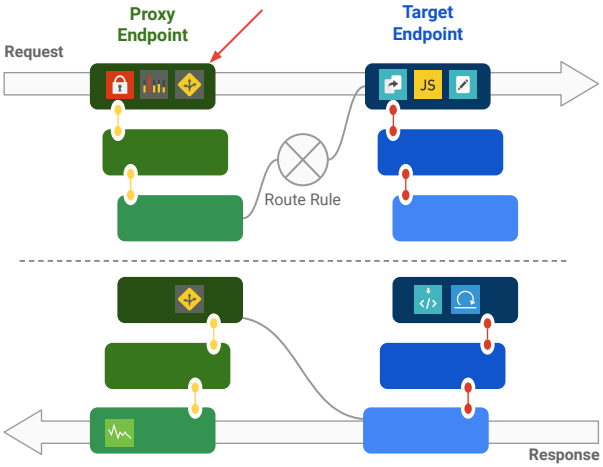


In the response flow, when execution reaches the second attachment of the ResponseCache policy, the cache is populated with the response message.

The rest of the policies following the ResponseCache are then executed.

How it works (cache hit)

- At the request flow attachment, the cache is checked for a match to the cache key.



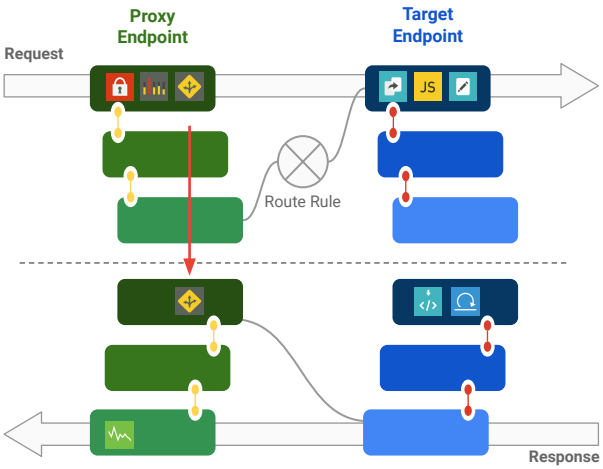
Later when a request comes in, policies are again executed up to the ResponseCache policy in the request flow.

The ResponseCache policy uses the cache key to check for a matching entry.

In this case there is a match, a cache hit.

How it works (cache hit)

- At the request flow attachment, the cache is checked for a match to the cache key.
- Match! Execution continues in the response cache attached to the response flow, and the response is loaded from cache.

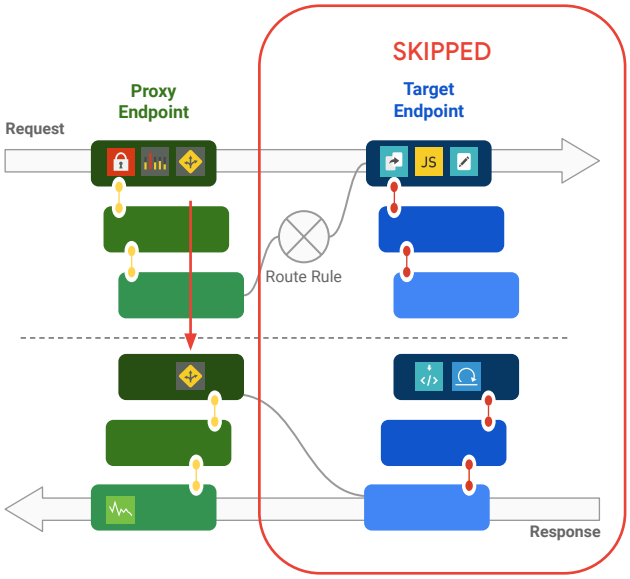


Execution continues immediately with the ResponseCache policy in the response flow.

The response message is loaded from cache, and the remaining policies are executed.

How it works (cache hit)

- At the request flow attachment, the cache is checked for a match to the cache key.
- Match! Execution continues in the response cache attached to the response flow, and the response is loaded from cache.
- All of the processing between the two attachments, including the call to the target and policies (including ServiceCallout), is skipped!



When there is a cache hit, all of the processing between the two attachments is skipped. This includes the call to the backend service and any other services called.

This can result in a much faster response to the client, and less API traffic to the backend.

ResponseCache policy

```
<ResponseCache continueOnError="false" enabled="true"
 name="RC-CacheResponse">
  <CacheResource>backendData</CacheResource>
  <CacheKey>
    <KeyFragment ref="request.queryparam.zip"/>
    <KeyFragment ref="proxy.pathsuffix"/>
  </CacheKey>
  <ExpirySettings>
    <TimeoutInSeconds>600</TimeoutInSeconds>
  </ExpirySettings>
  <UseAcceptHeader>true</UseAcceptHeader>
  <ExcludeErrorResponse>true</ExcludeErrorResponse>
</ResponseCache>
```

The ResponseCache policy uses cache keys and scope the same way that they are used in the general purpose caching policies.

ResponseCache policy

- When *UseAcceptHeader=true*, the incoming Accept header is added as a cache key fragment.

```
<ResponseCache continueOnError="false" enabled="true"
 name="RC-CacheResponse">
  <CacheResource>backendData</CacheResource>
  <CacheKey>
    <KeyFragment ref="request.queryparam.zip"/>
    <KeyFragment ref="proxy.pathsuffix"/>
  </CacheKey>
  <ExpirySettings>
    <TimeoutInSeconds>600</TimeoutInSeconds>
  </ExpirySettings>
  <UseAcceptHeader>true</UseAcceptHeader>
  <ExcludeErrorResponse>true</ExcludeErrorResponse>
</ResponseCache>
```

Setting UseAcceptHeader to true will automatically add the accept header as a cache key fragment.

The Accept header is used by the client to request a certain content type for the response.

If your API allows more than one content type format for responses, UseAcceptHeader can make sure that the client is not served the incorrect format from cache.

ResponseCache policy

- When *UseAcceptHeader=true*, the incoming Accept header is added as a cache key fragment.
- When *ExcludeErrorResponse=true*, the response will only be cached if response.status.code is between 200 and 205.

```
<ResponseCache continueOnError="false" enabled="true"
 name="RC-CacheResponse">
  <CacheResource>backendData</CacheResource>
  <CacheKey>
    <KeyFragment ref="request.queryparam.zip"/>
    <KeyFragment ref="proxy.pathsuffix"/>
  </CacheKey>
  <ExpirySettings>
    <TimeoutInSeconds>600</TimeoutInSeconds>
  </ExpirySettings>
  <UseAcceptHeader>true</UseAcceptHeader>
  <ExcludeErrorResponse>true</ExcludeErrorResponse>
</ResponseCache>
```

Setting ExcludeErrorResponse to true ensures that a response will only be cached if the status code is between 200 and 205.

Status codes in the 200s indicate success.

ResponseCache: SkipCacheLookup

```
<ResponseCache continueOnError="false" enabled="true"
 name="RC-CacheResponse">
  <CacheResource>backendData</CacheResource>
  <CacheKey>
    <KeyFragment ref="request.queryparam.zip"/>
    <KeyFragment ref="proxy.pathsuffix"/>
  </CacheKey>
  <ExpirySettings>
    <TimeoutInSeconds>600</TimeoutInSeconds>
  </ExpirySettings>
  <UseAcceptHeader>true</UseAcceptHeader>
  <ExcludeErrorResponse>true</ExcludeErrorResponse>
  <SkipCacheLookup>request.verb != "GET"</SkipCacheLookup>
</ResponseCache>
```

- If *SkipCacheLookup* evaluates to true, the cache will not be checked during request flow.
- This can be used to force a call to the backend, which can force a refresh of the cache.
- Cache lookup should also be skipped if the request is not safe.

SkipCacheLookup can be configured with a conditional expression. If the SkipCacheLookup condition evaluates to true, the cache will not be searched for a matching entry during the execution of the ResponseCache policy in the request flow.

SkipCacheLookup can be used to force an update to the cache. The backend target would be called, even if there was already a matching cache entry. The response from the backend could then be stored in the cache.

Cache lookup should always be skipped if the request is not safe. Generally, when verbs other than GET are used, there are modifications to the state of the backend. Retrieving the response from cache would cause those modifications to be skipped.

ResponseCache: SkipCachePopulation

- If the SkipCachePopulation condition is true, the cache will not be updated during the response flow.
- This can be used if a response is unlikely to be used again.
- Cache population should also be skipped if the request is not safe.

```
<ResponseCache continueOnError="false" enabled="true"
 name="RC-CacheResponse">
  <CacheResource>backendData</CacheResource>
  <CacheKey>
    <KeyFragment ref="request.queryparam.zip"/>
    <KeyFragment ref="proxy.pathsuffix"/>
  </CacheKey>
  <ExpirySettings>
    <TimeoutInSeconds>600</TimeoutInSeconds>
  </ExpirySettings>
  <UseAcceptHeader>true</UseAcceptHeader>
  <ExcludeErrorResponse>true</ExcludeErrorResponse>
  <SkipCacheLookup>request.verb != "GET"</SkipCacheLookup>
  <SkipCachePopulation>request.verb!="GET"</SkipCachePopulation>
</ResponseCache>
```

When the SkipCachePopulation condition is true, the cache is not updated during the response flow.

Population should be skipped if a response is unlikely to be used again, or the response is an error response.

Like cache lookup, cache population should not be done for unsafe requests.

ResponseCache: UseResponseCacheHeaders

- When *UseResponseCacheHeaders is true*, caching-related headers in the response will be used with this precedence:
  1. Cache-Control s-maxage
  2. Cache-Control max-age
  3. Expires

```
<ResponseCache continueOnError="false" enabled="true"
 name="RC-CacheResponse">
  <CacheResource>backendData</CacheResource>
  <CacheKey>
    <KeyFragment ref="request.queryparam.zip"/>
    <KeyFragment ref="proxy.pathsuffix"/>
  </CacheKey>
  <ExpirySettings>
    <TimeoutInSeconds>600</TimeoutInSeconds>
  </ExpirySettings>
  <UseAcceptHeader>true</UseAcceptHeader>
  <ExcludeErrorResponse>true</ExcludeErrorResponse>
  <SkipCacheLookup>request.verb != "GET"</SkipCacheLookup>
  <SkipCachePopulation>request.verb!="GET"</SkipCachePopulation>
  <UseResponseCacheHeaders>true</UseResponseCacheHeaders>
</ResponseCache>
```

When UseResponseCacheHeaders is set to true, any cache-related headers in the response will be honored in this order:

First is a Cache-Control header specifying s-maxage. s-maxage specifies the number of seconds to cache the response in a shared cache. A cache in an API proxy would be considered a shared cache.

Second is a Cache-Control header specifying max-age. This indicates the number of seconds to cache the response in a non-shared cache.

Third is an Expires header specifying the date and time of expiration.

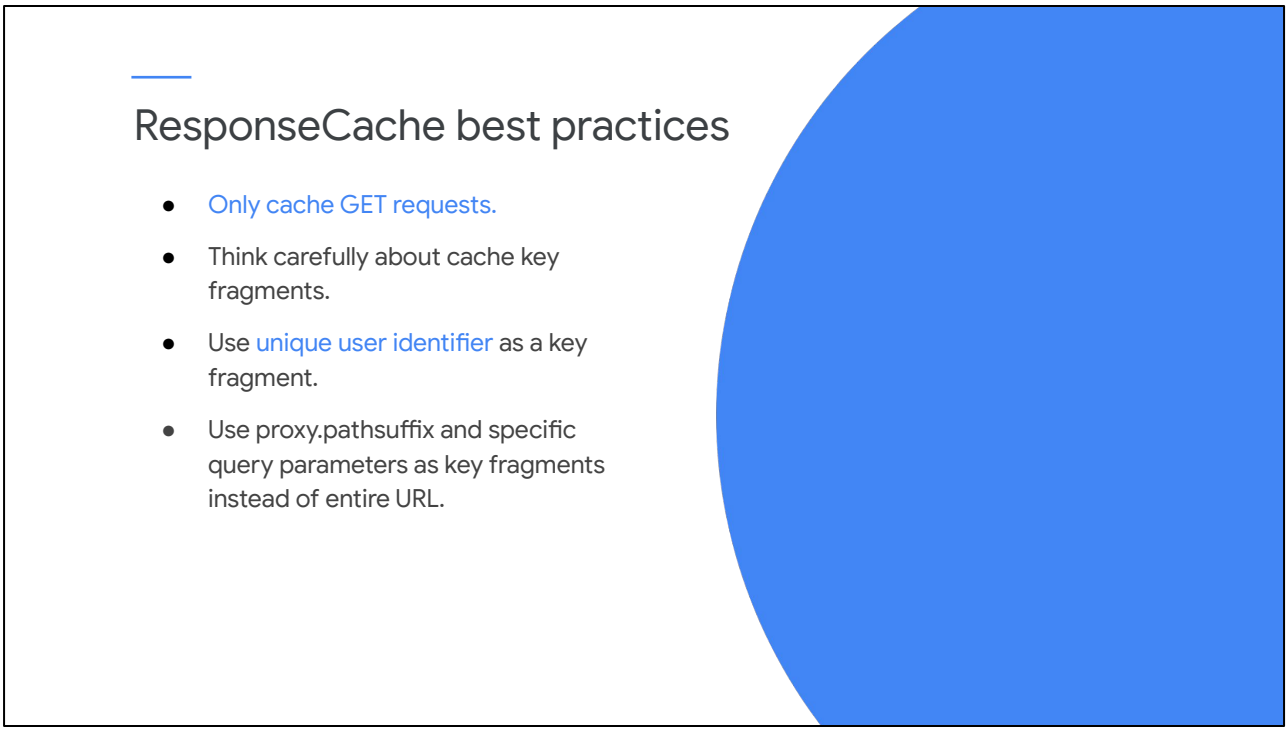
ResponseCache: UseResponseCacheHeaders

- When *UseResponseCacheHeaders is true*, caching-related headers in the response will be used with this precedence:
  1. Cache-Control s-maxage
  2. Cache-Control max-age
  3. Expires
- If one of those headers is used, its expiration value is compared to the *ExpirySettings* value, and the lower expiration time is used.

```
<ResponseCache continueOnError="false" enabled="true"
 name="RC-CacheResponse">
  <CacheResource>backendData</CacheResource>
  <CacheKey>
    <KeyFragment ref="request.queryparam.zip"/>
    <KeyFragment ref="proxy.pathsuffix"/>
  </CacheKey>
  <ExpirySettings>
    <TimeoutInSeconds>600</TimeoutInSeconds>
  </ExpirySettings>
  <UseAcceptHeader>true</UseAcceptHeader>
  <ExcludeErrorResponse>true</ExcludeErrorResponse>
  <SkipCacheLookup>request.verb != "GET"</SkipCacheLookup>
  <SkipCachePopulation>request.verb!="GET"</SkipCachePopulation>
  <UseResponseCacheHeaders>true</UseResponseCacheHeaders>
</ResponseCache>
```

If one of the cache headers is used, the indicated expiration is compared to the ExpirySettings element.

Whichever would result in the lower expiration time will be used.



There are several best practices you should consider when adding a response cache to your proxy.

First, only GET requests should be cached. Other requests have side effects, and you wouldn't want to skip the side effects by skipping the call to the backend.

Second, you should always think carefully about which cache key fragments to use, and when to skip cache lookup and population. Any variable that causes the response to change should be used in the cache key.

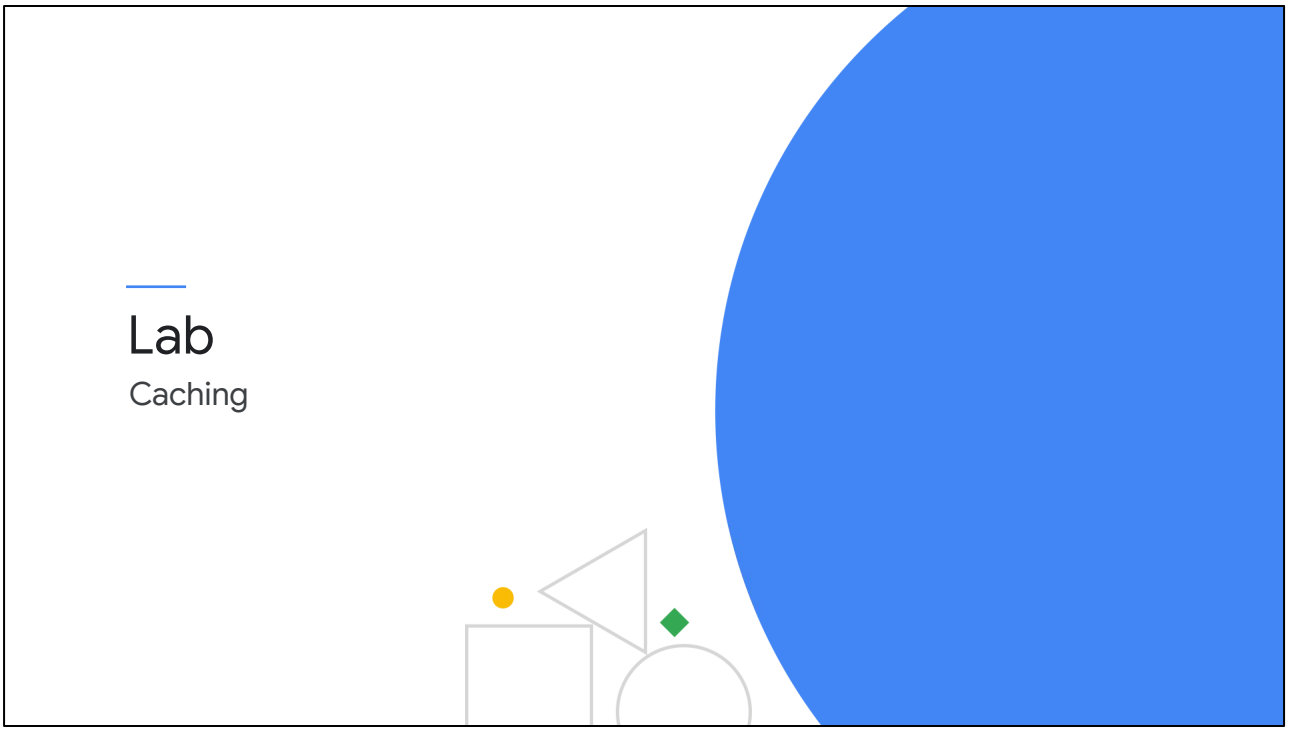
Third, whenever your API returns user data, you must include a unique user identifier as a key fragment.

This is extremely important. One of the worst things you can do in an API is return one user's data to a different user.

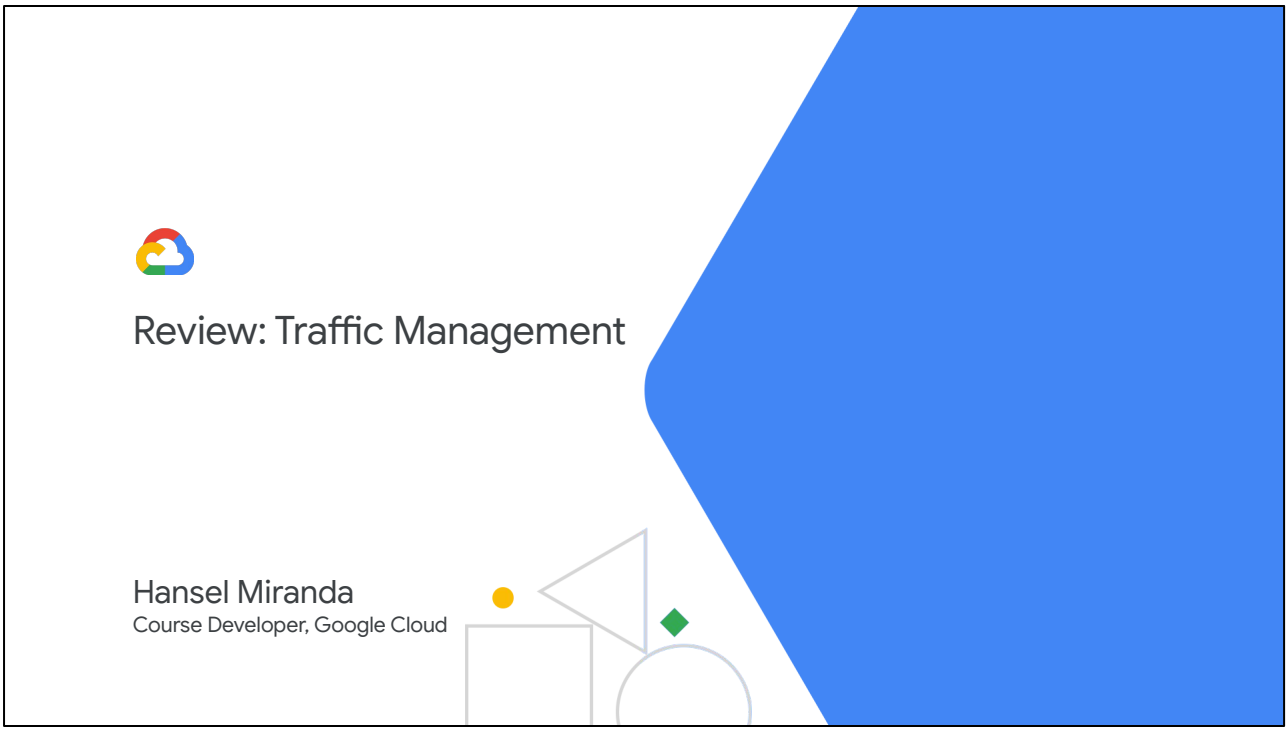
Fourth, it is best to use the proxy.pathsuffix variable and specific query parameters as cache key fragments.

If you use the full URL in your cache key, you are likely to get fewer cache hits.

Clients can send query parameters in any order, or send ignored query parameters, causing unnecessary cache misses.



In this lab you add a response cache to your retail API. You will reduce repetitive requests to the backend by caching the entire response when retrieving products by ID.



This module taught you about spike arrests, which is how we control the rate of traffic through our API proxies.

You learned about quotas, which are used to control the number of requests that can be made to an API.

And you learned about the caching policies that can be used in API proxies to reduce unnecessary calls to backend services.

You also completed labs that added a spike arrest, quota, and response cache to your retail API proxy.