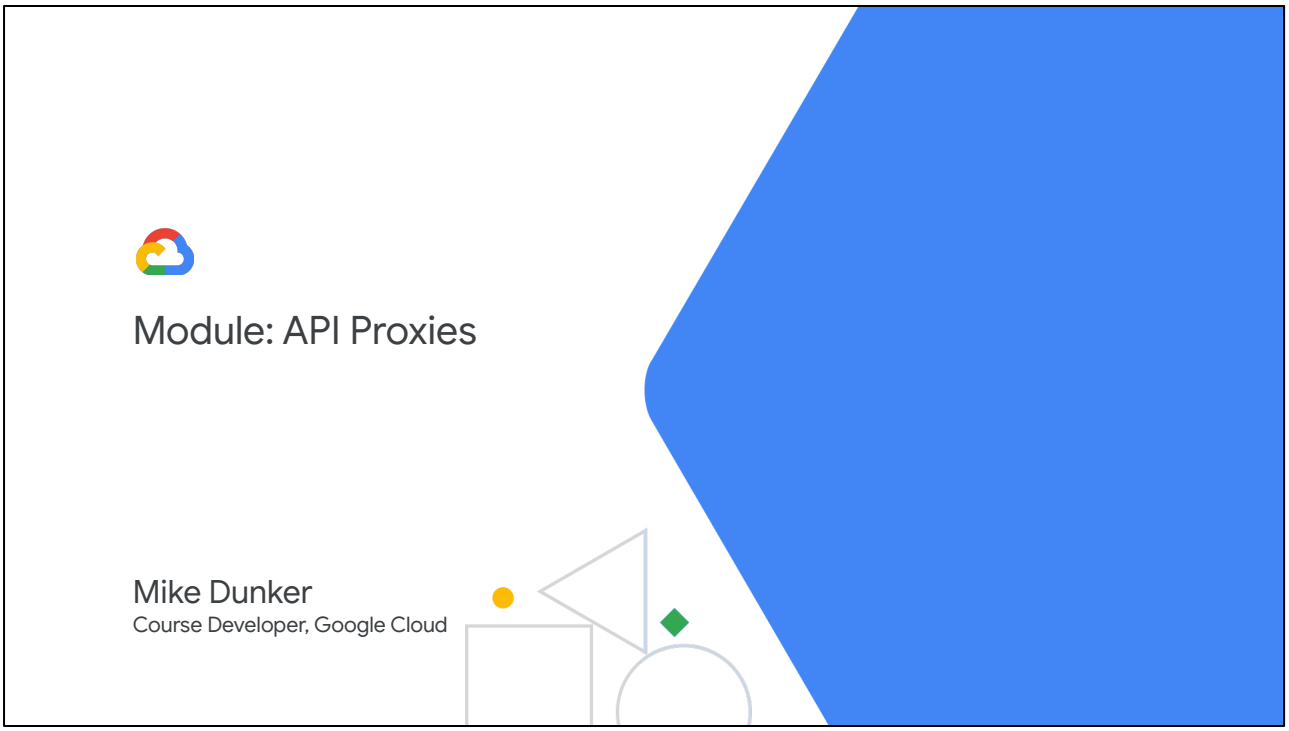


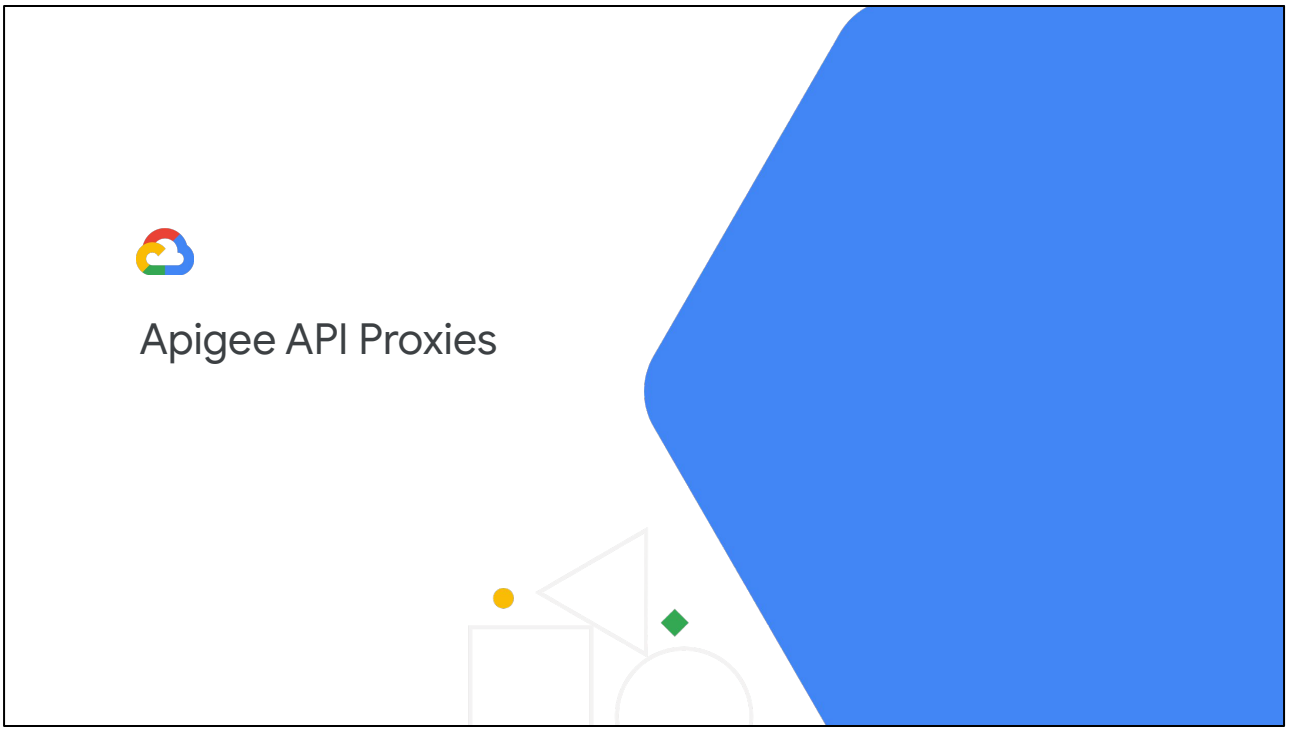
In this module, we'll learn about API proxies.

We'll learn how API calls are routed to a specific API proxy in an organization and environment, how backend services are called from your API proxies, and how we use flows, variables, conditions, and policies to control the functionality of our API proxies.

You'll also create a retail API proxy during this course using a series of labs.

You will use an OpenAPI specification to generate an API proxy stub, and use a target server to configure the backend service for your proxy.

You'll also use labs to explore the trace tool and return an error when an insecure request is made to an API proxy.



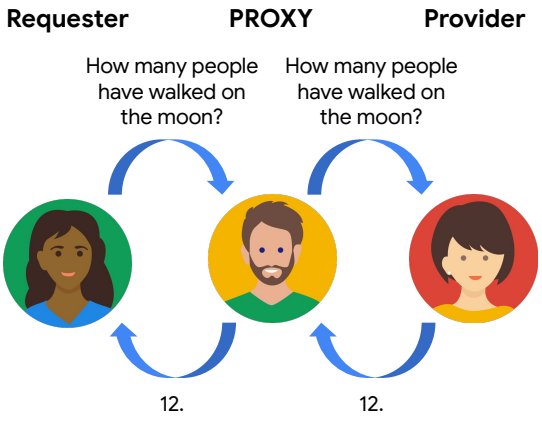
This lecture discusses Apigee API proxies, explains why we need them, and introduces proxy and target endpoints.

What is a proxy?

The first step in understanding API proxies is to recognize what a proxy does.

What is a proxy?

- An intermediary acting on behalf of something or someone else
- The requester does not need to know anything about the eventual provider
- The provider does not need to know or communicate directly with the requester



A proxy is an intermediary that sits between requesters and providers and acts on behalf of the provider. A request is forwarded, or "proxied," to one or more providers. When the proxy has the response from the provider, the request can be fulfilled.

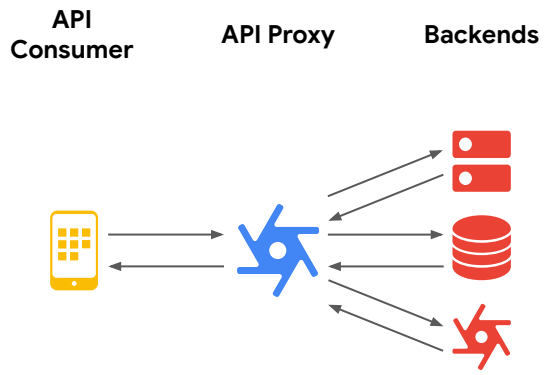
Let's look at this from the requester's point of view. As far as the requester is concerned, she got the information from the proxy. The requester does not know, or need to know, that the proxy got the information from someone else.

From the provider's perspective, the request came from the proxy. The provider does not need to know that someone other than the proxy originally requested the information.

We'll see that this concept of a proxy has benefits for APIs.

What is an API proxy?

- Decouples API consumers from backend services
- Implements consumer-facing APIs designed specifically for app developers
- Shields app developers from complexity of backend systems, thus allowing changes to backends without breaking apps



So, what is an API proxy?

An API proxy provides a facade layer that separates API consumers, typically applications, from backend services and resources.

The API consumer does not connect directly to those backend services. Instead, the API consumer connects to the API proxy, and the API proxy provides the required functionality by calling the backend services.

The proxy provides consumer-facing APIs. These APIs can be designed specifically to address the needs of app developers and their apps.

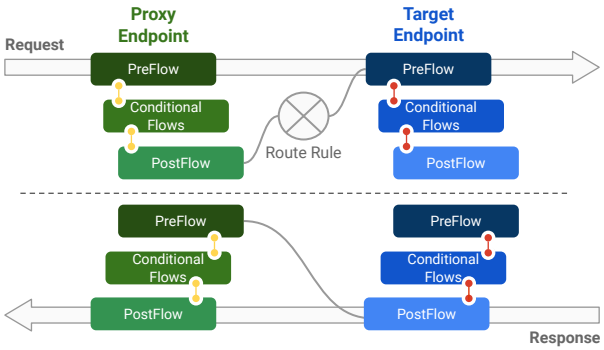
The advantage of this decoupling is that app developers do not need to worry about the complexity or usage patterns of backend services.

The API proxy itself can deal with authentication, authorization, data manipulation, removing sensitive data, and other related concerns.

Your backends can all use different API patterns or methods of access, and the API proxy can avoid passing that complexity on to app developers.

This pattern also allows us to make changes in our backends, such as modernizing backend services or moving services from a data center to the cloud, all without having any effect on the apps that are using the proxy APIs.

Apigee API proxy



- Receives API requests based on specified hostname, scheme, and URL path combinations
- Adds security, scalability, and rate limiting to APIs by attaching policies (pre-built functions) to flows
- Can call multiple services and combine and transform their responses during an API call
- Captures analytics for all calls

This is a diagram of Apigee's version of an API proxy. At a high level, what does an Apigee API proxy do?

The proxy receives API requests from the app consuming the API. A particular proxy receives that request based on the scheme, hostname, and URL.

An API developer can add features like security, scalability, and rate limiting to API proxies by attaching policies to flows. We will learn more about policies and flows later.

Proxies can call multiple backend services during an API call. The API response can be built by combining and transforming backend responses.

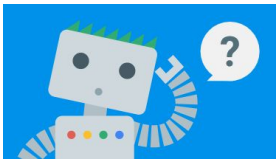
Analytics data is also captured for each call. This helps you gain visibility into all of the requests, responses, and the backend services called, even when those services are not capturing analytics data themselves.

We will learn much more about each of these features of Apigee API proxies later.

But what about the extra network hop?

When you add this layer:

- You gain control and insight.
  - Verify security tokens and enforce security policies
  - Perform traffic management and serve requests from cache
  - Collect analytics and gain insight into API usage and performance
- You decouple API consumers from backend services.
  - Create APIs that combine backend and third-party services
  - Innovate in consumer-facing APIs without affecting backend services
  - Modernize and move backends without breaking consumer apps



A common objection is "But what about the extra network hop for the API proxy? Doesn't that make everything slower?"

The answer is that the proxy layer does add incremental latency to some calls, but the benefits of using the API proxy typically outweigh the costs.

We've discussed the high-level benefits of API proxies in previous slides, but let's dig a bit deeper.

By using an API proxy layer, you gain control and insight over your API calls. This is extremely important. When you expose an API, you are allowing indirect access to its backend services.

You need to secure incoming requests, especially if your API is accessible from the internet. Your API should only allow access to information that is appropriate for the consumer. You may need to add security to the API proxy for backend services that don't have enough security built in or remove sensitive fields before returning responses.

You may need to limit the rate of requests to your backend so it doesn't receive more traffic than it can handle, or handle certain errors at the proxy layer to reduce the load to the backend.

You can also use an API proxy to serve requests from cache, which can reduce

unnecessary traffic to backend services.

With Apigee's built-in analytics, you gain insight into API calls and backend service performance. Control and insight can be difficult to manage at the individual service level.

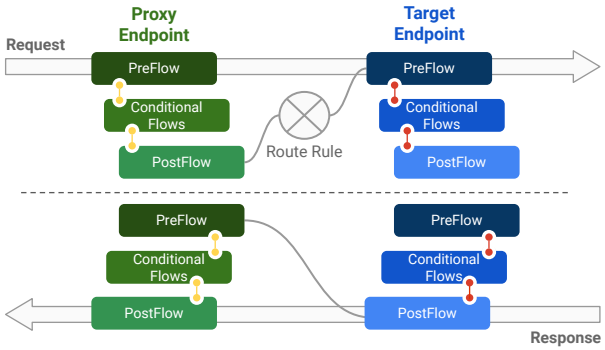
A second benefit you get from adding an API proxy is the decoupling of apps from backend services. This allows you to create APIs tailored to your app developers, as opposed to directly exposing the APIs of backend services.

You can create APIs that combine multiple services, including those from third parties.

Features can be added to APIs, and API interfaces can be improved, without having to modify backend services.

You can also move services between data centers or completely change backend services and how they are used without changing the calls being made by apps. When required changes are handled in the API proxy, apps consuming your API will continue to work as expected.

Apigee API proxy



An Apigee API Proxy is split into two parts: the proxy endpoint and the target endpoint.

The proxy endpoint is on the left of the diagram, closer to the API consumer.

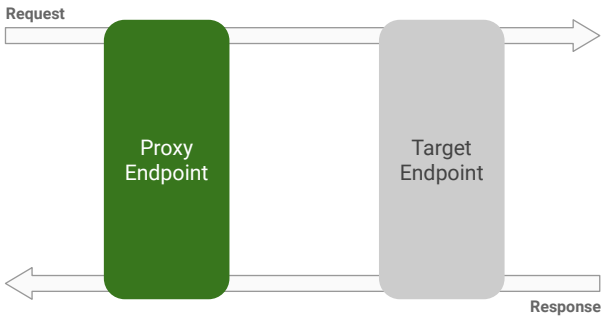
The target endpoint is on the right of the diagram, closer to the backend service, which is also called the target.

The proxy developer attaches policies to flows to build the API proxy's functionality.

We'll discuss the specifics of policies and flows in later lectures.

Proxy endpoint

- Traffic with matching protocol, host alias, and base URL is routed to the endpoint
- Parse and verify inbound request
- Attach policies to rate-limit traffic and protect against invalid and malicious requests
- Build API response



The concerns of the API consumer are generally handled by the proxy endpoint.

An incoming request is sent to a specific proxy based on the protocol, host alias, and base URL of the request.

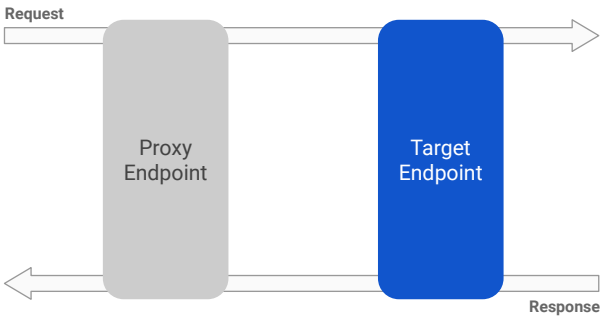
The proxy endpoint parses and validates the request.

Policies are usually attached in the proxy endpoint request to rate-limit traffic and protect against invalid and malicious requests, blocking the request from reaching the backend service.

The proxy endpoint also typically builds the API response to be sent back to the caller.

Target endpoint

- Specify backend service to call
- Build API requests for a backend service
- Specify connection and I/O timeouts for backend service calls
- Validate status codes and payloads of backend responses



The target endpoint handles the calling of the backend service.

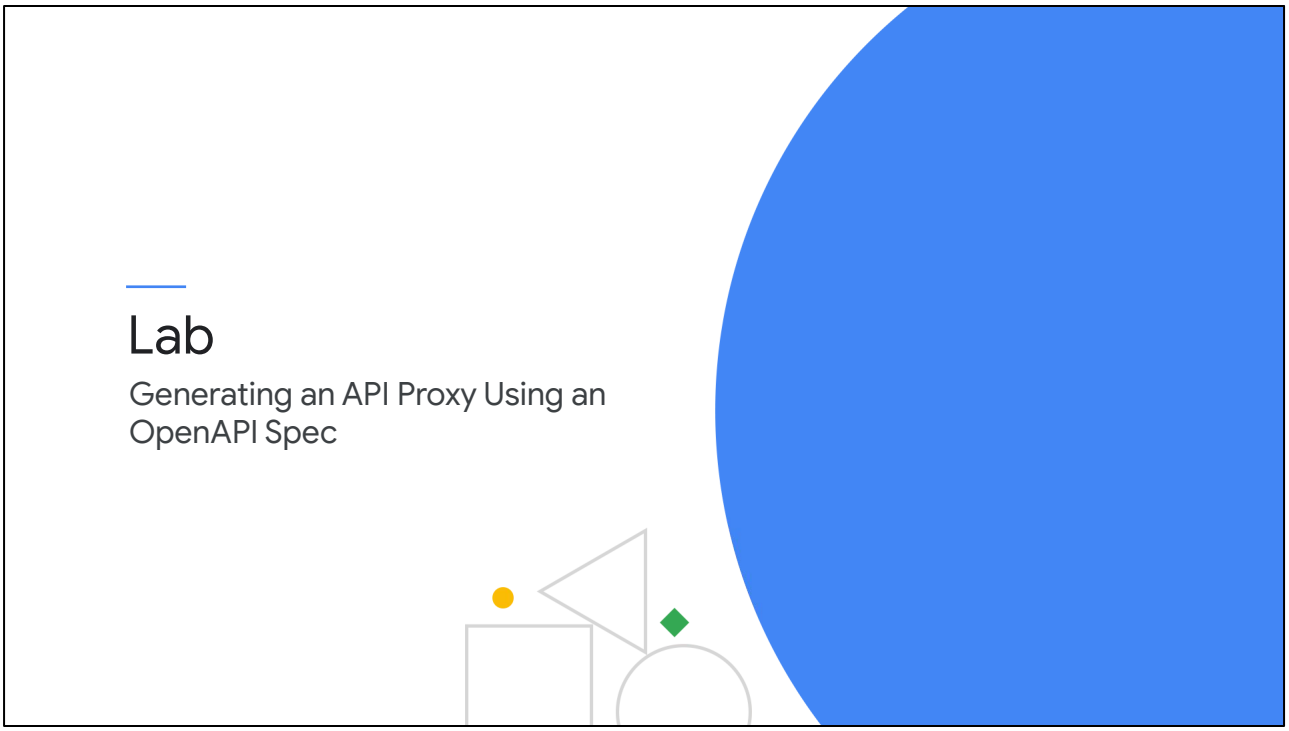
The URL of the backend service is specified in the target endpoint.

The target endpoint generally builds the request for calling the backend service.

Parameters like connection and I/O timeouts are specified in the target endpoint so that the API call won't hang indefinitely when the backend is not responding.

The response from the backend is also generally parsed and validated in the target endpoint.

We will learn much more about the flow of API calls during the next few lectures.



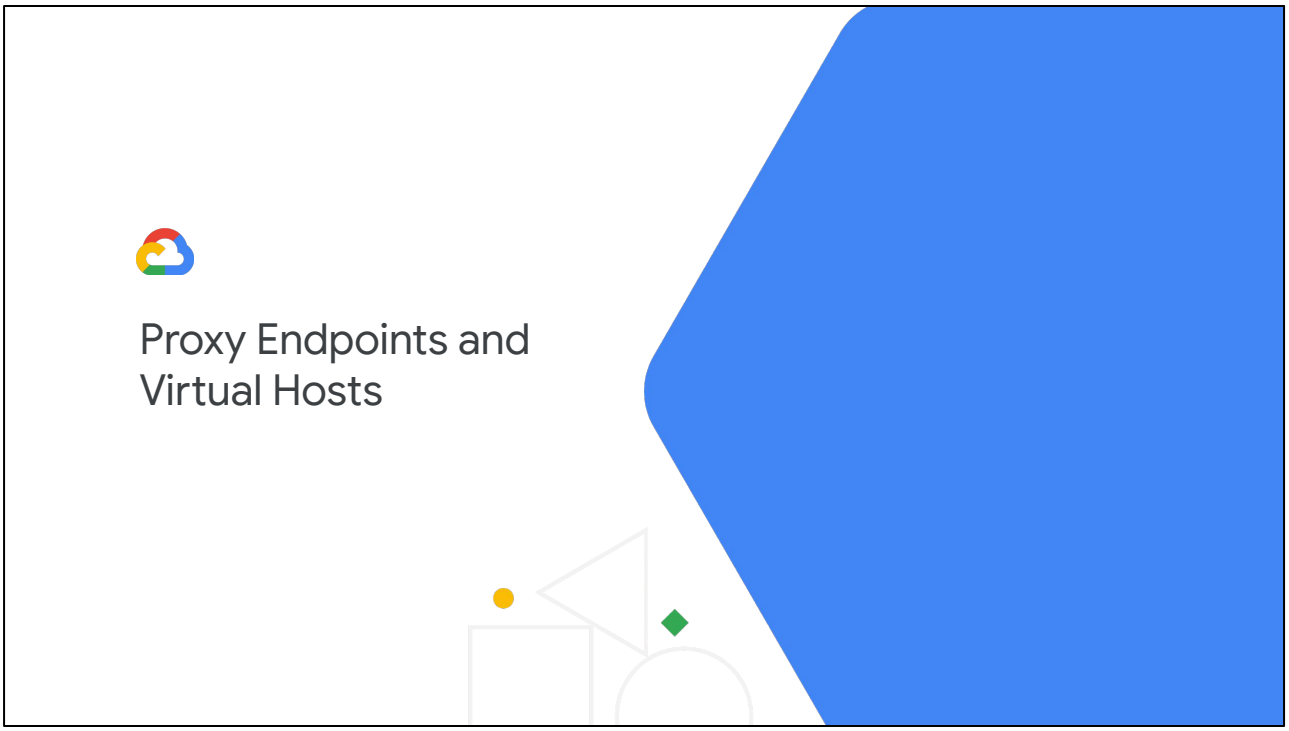
In this lab, you create an API proxy for a retail API. Many of the labs in these courses will add features to this same proxy, like security, traffic management, and caching. Later, you will also create API products and deploy them to a developer portal to allow application developers to use your API.

In this first lab, you import the OpenAPI specification for a backend retail service into your Apigee organization. You will use this backend spec to create a new API proxy using the Apigee proxy wizard. You will explore the configuration for the new proxy and trace calls you make to it. Finally, you will modify the OpenAPI specification to use to your proxy instead of the backend service.

Try to use the same proxy throughout these courses. When you start building your own proxies, you will likely follow a similar incremental approach as you add functionality to your APIs.

If you are having problems getting your proxy to work, try to use the trace tool to debug and fix the issue.

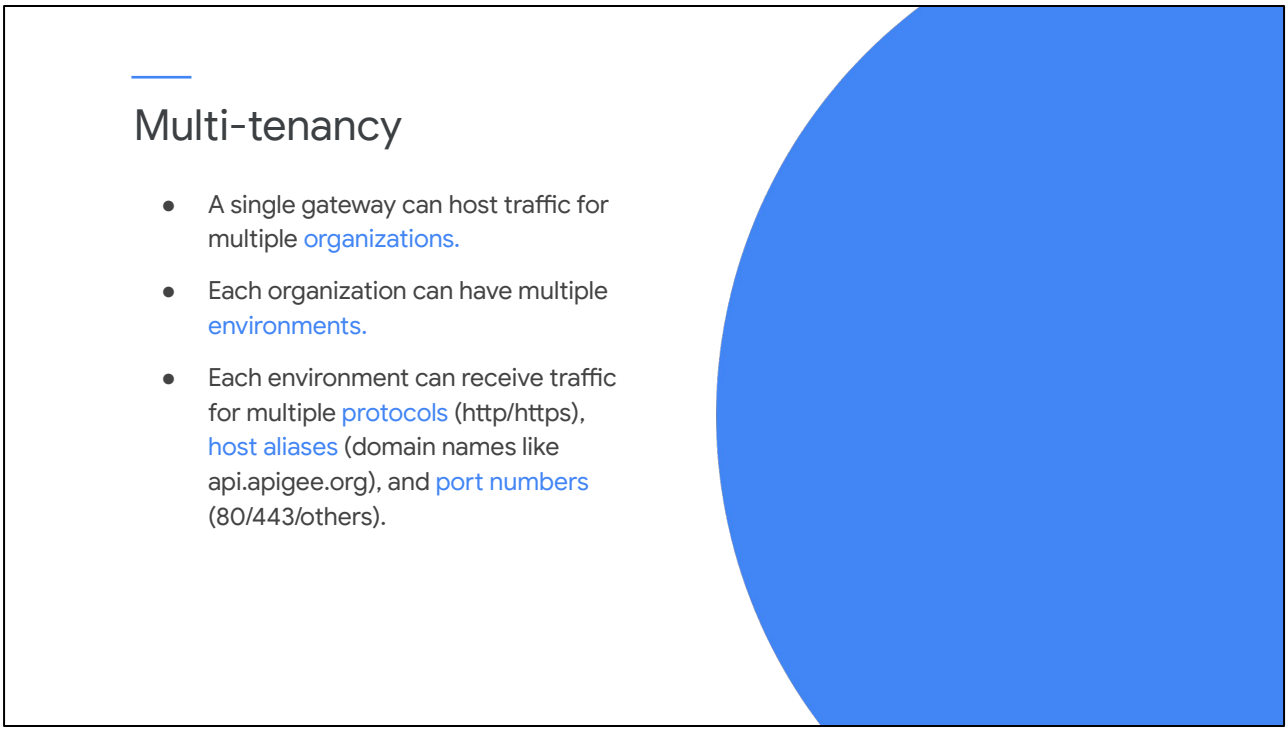
If you are unable to make the proxy work, labs have instructions on how to download a solution to the previous lab to use as a starting point.



In this lecture we explore the proxy endpoint side of the API proxy.

We'll discuss multi-tenancy, and how virtual hosts allow multiple host aliases to be handled on a single API gateway.

We'll also see how virtual hosts and base paths cause a request to be delivered to the correct proxy endpoint.



Apigee API gateways handle your runtime API traffic.

There are typically many API proxies running on a single Apigee gateway, and Apigee must be able to route traffic to the correct proxy.

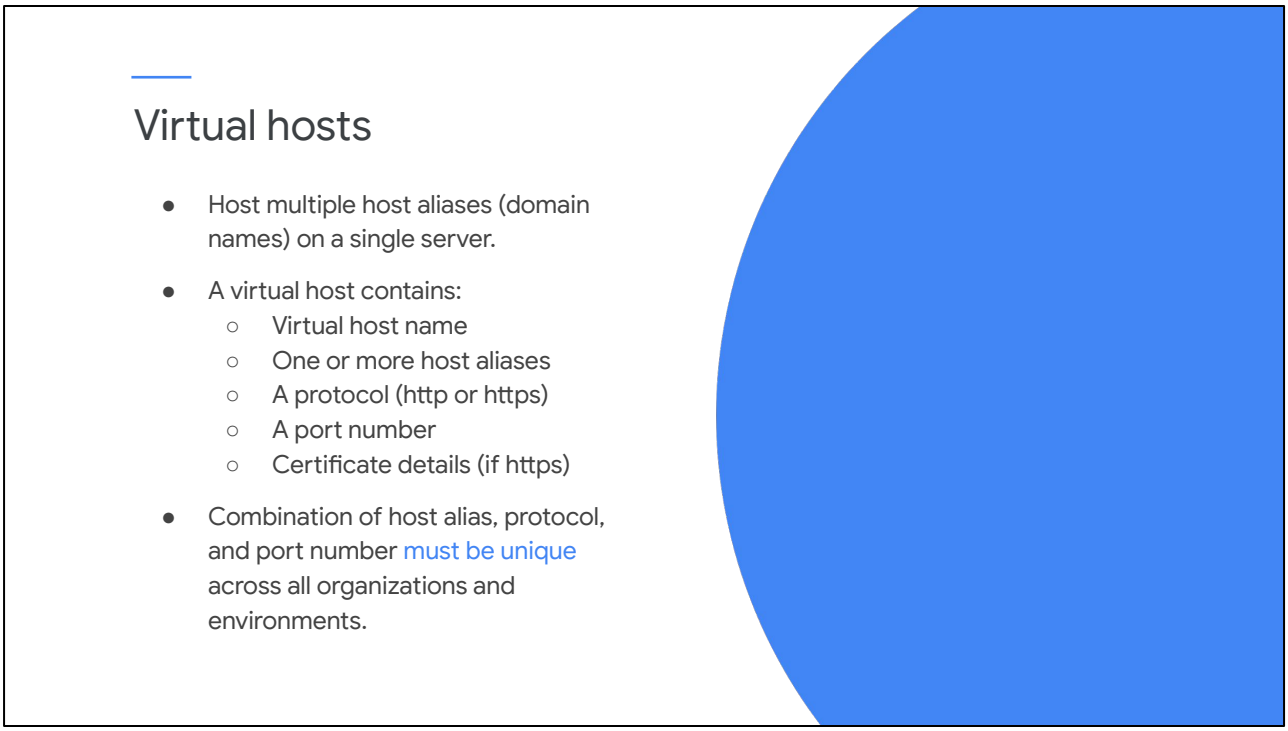
An Apigee gateway can host traffic for multiple organizations. As discussed before, an organization is the top-level entity for Apigee. Multiple organizations may be owned by the same customer, but nothing inside an organization is accessible from within other organizations.

Each organization has multiple environments. Environments are typically used to control the lifecycle of an API. For example, an API revision may start out in the development environment, be promoted to the test environment, and eventually be promoted to the production environment.

An API gateway generally handles traffic for multiple environments within an organization. Each environment can receive http or https traffic for multiple host aliases on multiple ports.

With all the traffic for multiple organizations, environments, and host aliases being received on the same gateway, how does traffic get routed to the correct proxy?

The first part of the answer involves virtual hosts.



Virtual hosts allow multiple host aliases, or domain names, to be hosted on a single Apigee gateway.

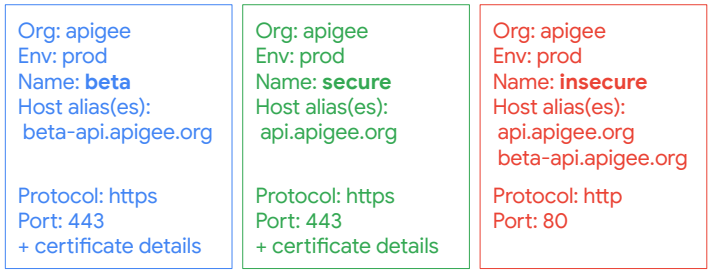
When Apigee receives an API request, it will map the request to a virtual host to determine which organization and environment should receive the request.

Virtual hosts used by a proxy are configured by name. Virtual host names must be unique within an environment.

A virtual host has one or more host aliases, a specific protocol (http or https), a port number, and certificate details if the protocol is https.

Only the host alias, protocol, and port number from a URL are used to route an API request to a specific environment within a specific organization. Therefore, each combination of host alias, protocol, and port number must be unique across all organizations and environments in an API gateway.

Virtual hosts example



Let's look at three example virtual hosts and see how a request might be routed to a specific one.

Within the organization named apigee, and the environment named prod, the first virtual host is named beta. There is a single host alias mapped to this virtual host: beta-api.apigee.org. This beta host alias might be used to send requests to the prerelease version of an API.

The protocol is https, and it uses the default port of 443. A TLS certificate for this host alias, and its private key, must be associated with this virtual host.

We will learn more about certificates in a later lecture.

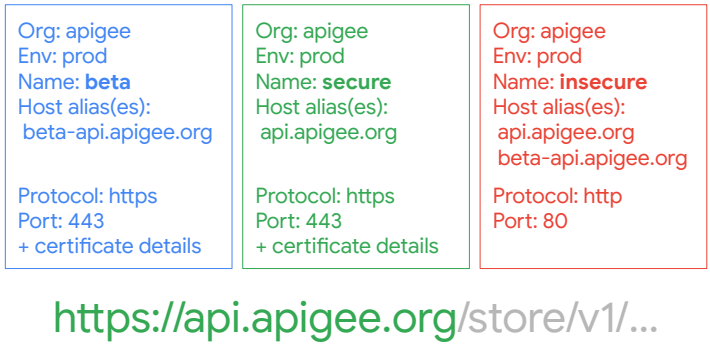
The second virtual host in the organization and environment is named secure. This virtual host also has a single host alias: api.apigee.org. Like the beta virtual host, it is also for secure traffic and requires a certificate.

The third virtual host is called insecure. Both of the previous host aliases are defined for this virtual host. However, this virtual host is only used for non-secure http traffic, which comes in on port 80. No certificate is required for this virtual host.

When your APIs should only allow secure traffic, it is a good practice to create a similar virtual host so that you can return custom error messages to app developers, instructing them to send requests securely.

Along with these three virtual hosts, we might have many more that map to other organizations and/or environments. So how does an API request get mapped to a particular organization and environment?

Virtual hosts example



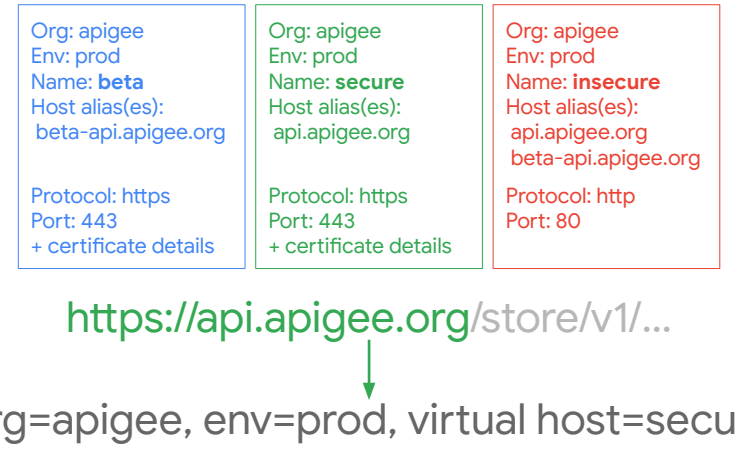
What if this API request is received?

The first thing to realize is that no information after the domain name is used in routing to the organization and environment.

The request is being received securely, using the default port 443, with the host alias of api.apigee.org.

This is a match for the second virtual host.

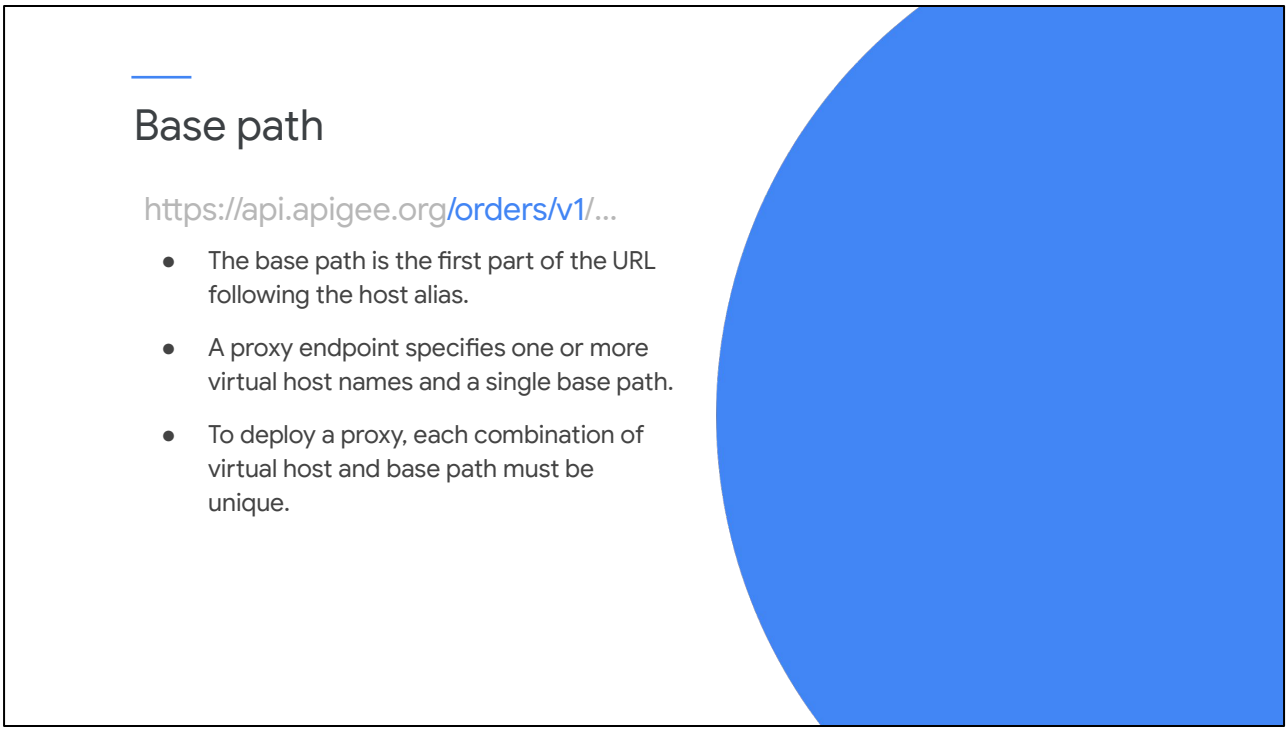
Virtual hosts example



So this is how the prod environment in the apigee organization receives the traffic via the secure virtual host.

Requests that map to one of the virtual hosts will always be routed to the associated organization and environment, because no other organization and environment can use the same combination of host alias, protocol and port number.

Note that requests that do not map to one of the virtual hosts within this environment will never be routed to the environment.



The virtual host determines the organization and environment to receive the traffic.

A base path is configured in a proxy endpoint to indicate that requests starting with the base path should be routed to that endpoint.

The base path is the first part of the URL following the host alias. In this example, /orders/v1 is the base path, which corresponds to version 1 of the orders API.

A proxy endpoint specifies one or more virtual host names and a single base path. API requests that match a specified virtual host and the specified base path will be routed to that proxy endpoint.

Note that you could have multiple proxy endpoints that match an API request, with each endpoint having a different base path. For example, you could also have proxy endpoints with base paths of /orders, or even just /. If there are multiple base paths that match a request, the longer base path match will be chosen.

A proxy cannot be deployed unless the combination of virtual host and base path for each proxy endpoint is unique within the environment.

How is a call sent to the correct proxy endpoint?

- The proxy endpoint's HTTPProxyConnection specifies a single base path and one or more virtual hosts.
- Proxies deployed to the environment with matching virtual host and base path will handle requests for the URL.

```
<ProxyEndpoint name="store">
  ⋮
  <HTTPProxyConnection>
    <BasePath>/stores/v1</BasePath>
    <VirtualHost>secure</VirtualHost>
    <VirtualHost>beta</VirtualHost>
  </HTTPProxyConnection>
  ⋮
</ProxyEndpoint>
```

https://beta-api.apigee.org/stores/v1/regions

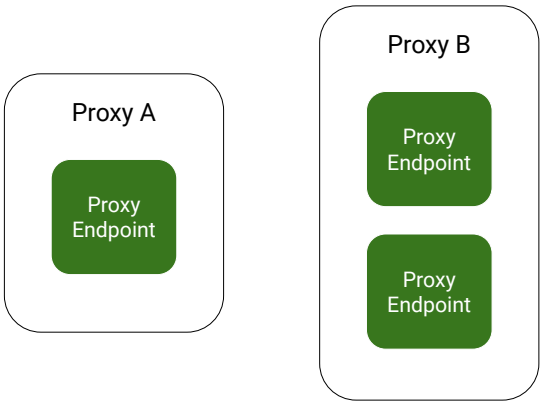
A proxy endpoint is configured using XML.

The http proxy connection tag in the proxy endpoint contains a single base path and one or more virtual hosts. Any requests matching one of the virtual hosts and the base path will be routed to this proxy endpoint.

So, in this example, https://beta-api.apigee.org/stores/v1/regions matches both the beta virtual host and the base path /stores/v1, so the store proxy endpoint would receive the traffic.

Proxy endpoints

- Organizations typically have many deployed proxies for a given environment.
- Proxies typically have a single proxy endpoint, but may have multiple.
- A proxy endpoint is the entry point for an API call.

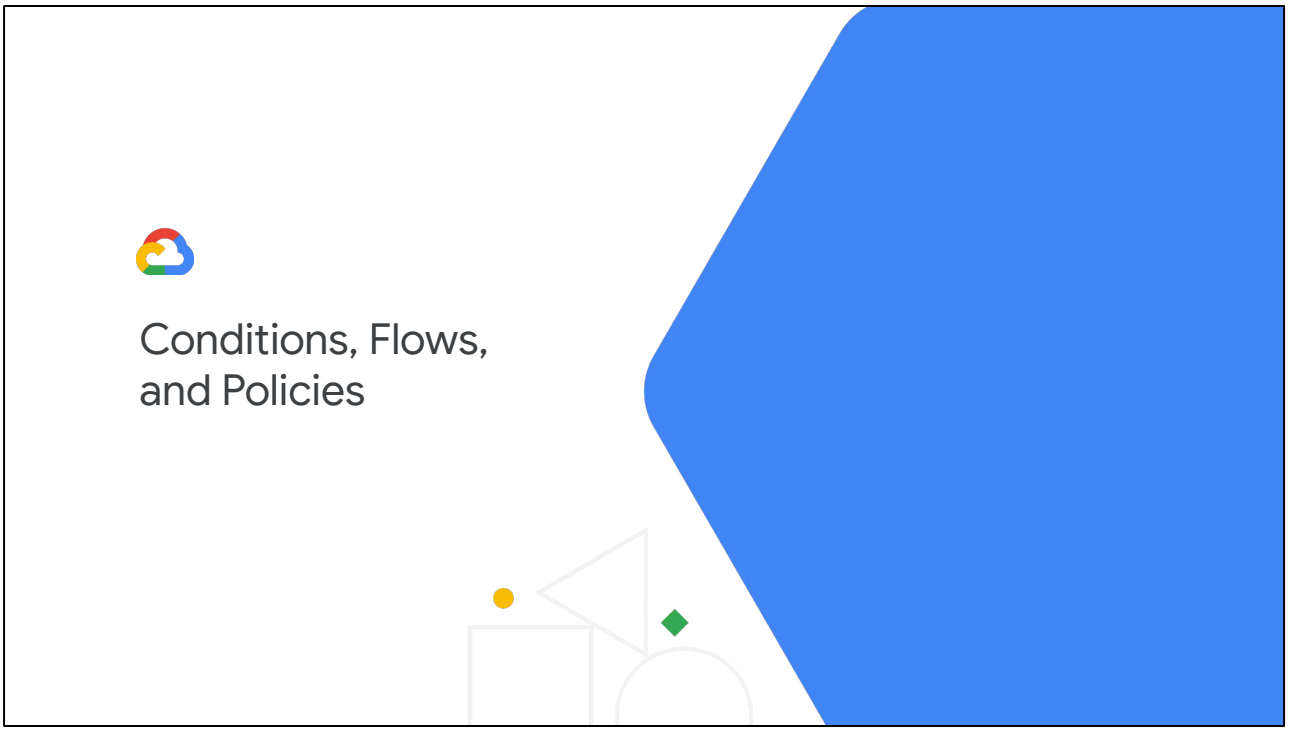


Within a specific organization, many proxies are deployed to a given environment.

Proxies typically have just one proxy endpoint, but they may have more than one.

A proxy endpoint is the entry point into a proxy for an API call.

Now you know how traffic is routed to a specific proxy endpoint in an organization and environment.



We have learned how traffic is routed to a proxy endpoint in an Apigee API proxy.

In this lecture we will learn about the features of proxies.

We will learn about variables, operators, and conditions, which you can use to dynamically control the execution of your proxies.

And we'll learn more about flows and policies.

Variables

- Predefined variables are automatically set during an API call.

Predefined Variable Examples

```
request.header.*
request.queryparam.*
request.verb
request.content
response.header.*
response.status.code
system.time
target.url
```

Apigee API proxies allow the use of variables. Variables can be used to control the functionality of your proxies.

Apigee has predefined variables that are set for each API call.

Some variables give us information about the incoming request. For example, all of the incoming request headers and query parameters are populated as variables. If a request contains a query parameter called action, the variable request.queryparam.action is available for use in your proxy, containing the value of the request's action query parameter. Request.verb contains the verb, and request.content contains the payload.

There are variables for the API response too. Setting these variables will modify the API response. For example, by setting the value of the variable response.header.Foo, you can create a header called Foo in the response with the value you have specified.

There are many other predefined variables. system.time returns the time the variable was read. target.url can be modified to change the URL used to call the target.

Variables

- Predefined variables are automatically set during an API call.
- Custom variables can also be created and referenced.
- Policies use and create variables.

Predefined Variable Examples

`request.header.*`
`request.queryparam.*`
`request.verb`
`request.content`
`response.header.*`
`response.status.code`
`system.time`
`target.url`

Custom Variable Examples

`flow.businessUnit`
`myCustomVariable`

You can also create your own custom variables. Your custom variables can contain any information you need to build requests and responses, log to analytics, or control the processing flow.

Policies will also create and use variables. The variables set by policies will be used to customize the flow and behavior of your APIs.

Proxy path suffix

- Variable name is proxy.pathsuffix.
- proxy.pathsuffix is set to the part of the incoming URL following the base path.
- The verb and proxy path suffix typically indicate the operation being performed.

An example of a variable that you will use quite often is proxy.pathsuffix

proxy.pathsuffix is automatically set to the part of the incoming URL following the base path, not including the query parameters.

For a RESTful API, the request's proxy path suffix and verb indicate the operation being performed.

Proxy path suffix

- Variable name is proxy.pathsuffix.
- proxy.pathsuffix is set to the part of the incoming URL following the base path.
- The verb and proxy path suffix typically indicate the operation being performed.

Request:

GET https://api.apigee.org/store/v1/orders/1234

org=apigee, env=prod, virtual host=secure

base path=/store/v1,
proxy=store-v1, proxy endpoint=store

proxy.pathsuffix=/orders/1234

request.verb=GET

Let's look at an example request. The verb, GET, and the URL are shown here.

The virtual host is specified with the host alias, protocol, and port number. In this case, https://api.apigee.org specifies the virtual host named secure in the prod environment of the apigee organization.

The base path is /store/v1 which is routed to the proxy endpoint named store inside the store-v1 proxy.

The proxy.pathsuffix variable contains the rest of the URL following the base path. In this case, that is /orders/1234.

The request.verb variable contains GET. From the verb and pathsuffix, GET /orders/1234, the request should retrieve the order with id 1234.

Conditions

- Like "if statements" in other languages.
- Must always evaluate to true or false.
- Predefined and custom variables can be used.
- String literals, numeric literals, true, false, and null may also be used.
- Can be chained using AND and OR.

Conditions allow dynamic handling during API calls.

Conditions are like if statements in other programming languages, and must always evaluate to true or false.

Variables and literals may be used in conditions, and conditions may be chained using the AND and OR operators.

Let's look at some examples.

Conditions

- Like "if statements" in other languages.
- Must always evaluate to true or false.
- Predefined and custom variables can be used.
- String literals, numeric literals, true, false, and null may also be used.
- Can be chained using AND and OR.

```
<Condition>request.queryparam.action != null</Condition>
```

This condition is true if the query parameter named action exists.

Conditions

- Like "if statements" in other languages.
- Must always evaluate to true or false.
- Predefined and custom variables can be used.
- String literals, numeric literals, true, false, and null may also be used.
- Can be chained using AND and OR.

```
<Condition>request.queryparam.action != null</Condition>

<Condition>response.status.code >= 500</Condition>
```

The second condition is true when the variable response.status.code is greater than or equal to 500.

Conditions

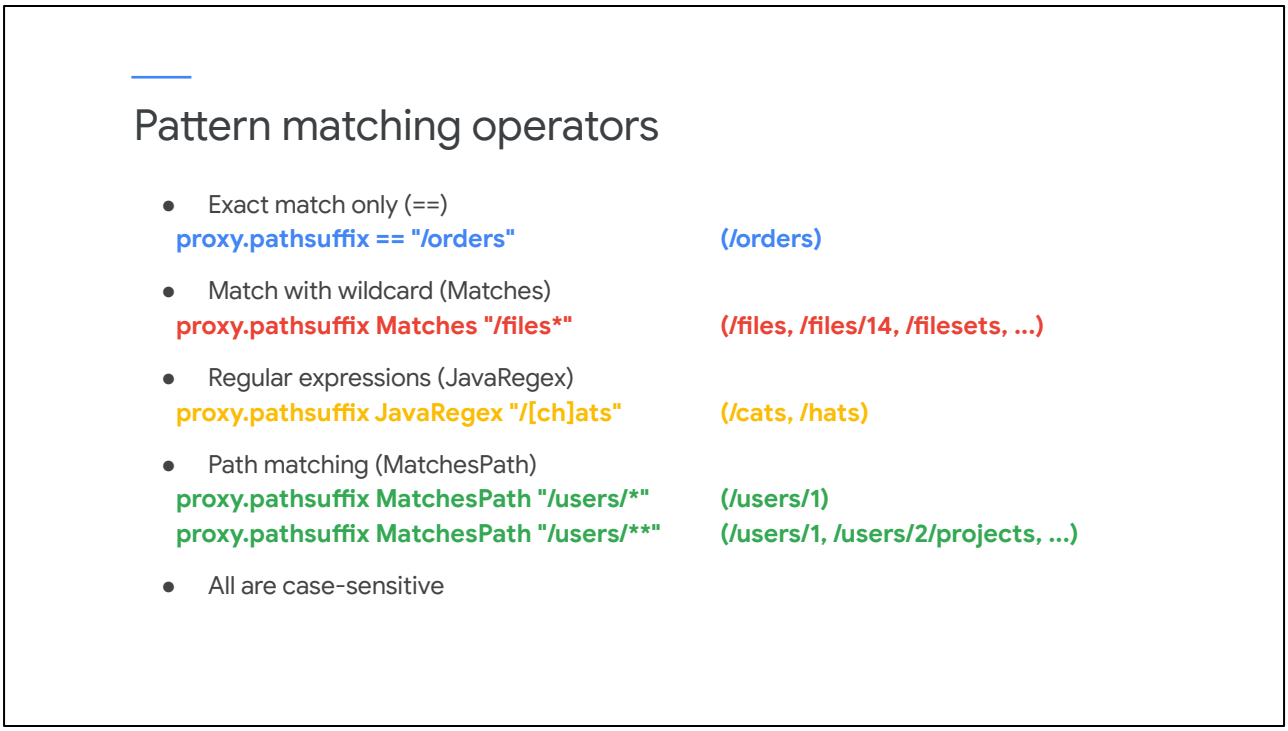
- Like "if statements" in other languages.
- Must always evaluate to true or false.
- Predefined and custom variables can be used.
- String literals, numeric literals, true, false, and null may also be used.
- Can be chained using AND and OR.

```
<Condition>request.queryparam.operation != null</Condition>

<Condition>response.status.code >= 500</Condition>

<Condition>request.verb == "GET" AND proxy.pathsuffix MatchesPath "/orders/*"</Condition>
```

The third condition is true when the incoming request is a GET, and the proxy path suffix matches /orders/star, which is true for a path suffix like /orders/1234.



Conditions always have an operator. Most operators are similar to the operators you'll recognize from other programming languages, like equals, not equals, or greater than.

Apigee also has multiple pattern matching operators.

The first pattern matching operator is equals. It is represented by a single or double equals sign. Note that conditions can never assign values to the variables being used, so using the single equals sign would never result in a value being assigned. In this example, proxy.pathsuffix would have to be exactly equal to /orders for this condition to be true.

The Matches operator adds a wildcard to the comparison, indicated by an asterisk. This condition would match any string that started with /files.

The JavaRegex operator allows you to use regular expressions with the same semantics as are used in the Java programming language. In this example, the square brackets section of the search pattern matches any single character inside the brackets. So this JavaRegex would match cats and hats, but not chats.

Regular expressions are very powerful; it is worth your time to learn them if you do not already use them.

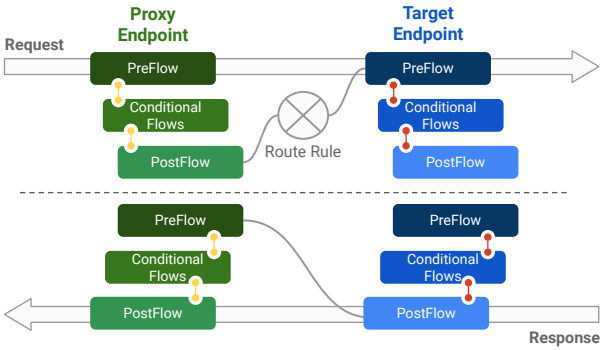
The MatchesPath operator is used often when trying to match URL patterns. This

operator is typically used with the proxy.pathsuffix variable. /* will match a single segment of a URL, and /** will match one or more segments of a URL.

Note that all of these pattern matching operators are case sensitive.

API proxy flows

- As API call is processed, it travels through flows and executes policies attached to those flows



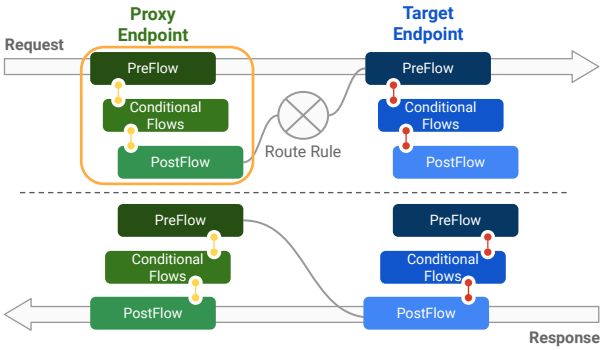
Let's look at the flows in an API proxy.

API call processing proceeds through the flows shown in the diagram.

Any policies attached to the evaluated flows are executed in order.

API proxy flows

- As API call is processed, it travels through flows and executes policies attached to those flows
- Evaluation order:
  - Proxy Endpoint Request
  - Route Rules
  - Target Endpoint Request
  - Request sent to target, and response received
  - Target Endpoint Response
  - Proxy Endpoint Response



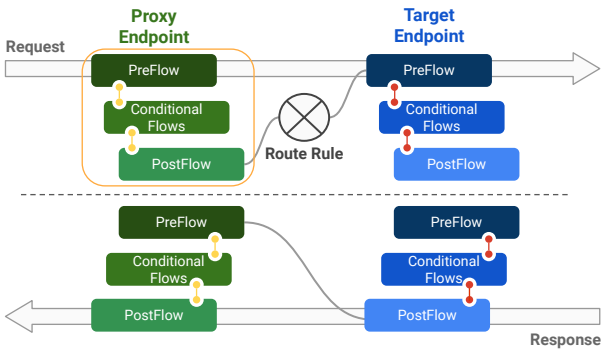
The evaluation order of flows is as follows:

When the API request is received by the proxy endpoint, it is first handled by the proxy endpoint request flows.

It is here that the request is typically validated and parsed.

API proxy flows

- As API call is processed, it travels through flows and executes policies attached to those flows
- Evaluation order:
  - Proxy Endpoint Request
  - Route Rules
  - Target Endpoint Request
  - Request sent to target, and response received
  - Target Endpoint Response
  - Proxy Endpoint Response

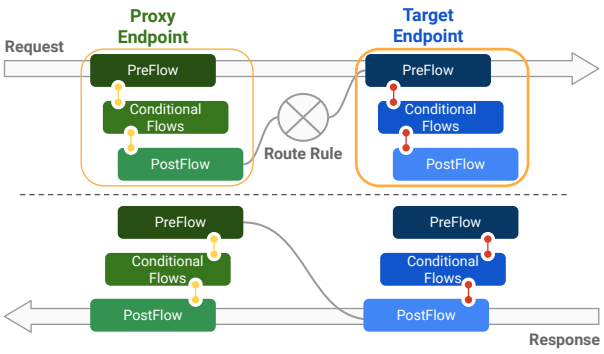


Next, route rules are evaluated to determine which target, if any, should continue with the processing.

Route rules will be discussed in the next lecture.

API proxy flows

- As API call is processed, it travels through flows and executes policies attached to those flows
- Evaluation order:
  - Proxy Endpoint Request
  - Route Rules
  - Target Endpoint Request
  - Request sent to target, and response received
  - Target Endpoint Response
  - Proxy Endpoint Response

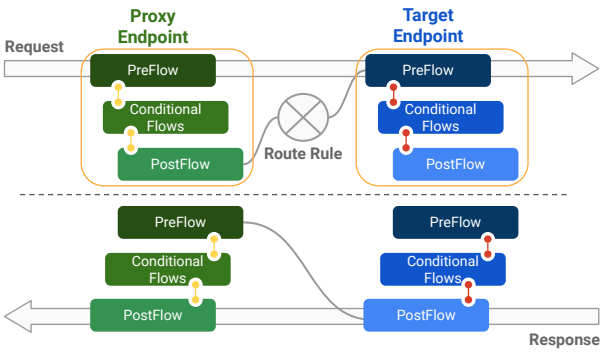


Next, the target endpoint request flows are processed.

The request to be sent to the backend is generally built in these flows.

API proxy flows

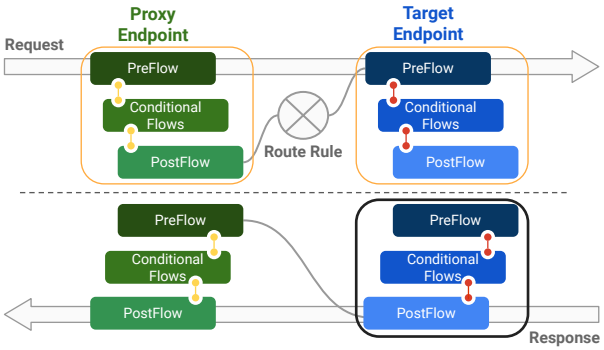
- As API call is processed, it travels through flows and executes policies attached to those flows
- Evaluation order:
  - Proxy Endpoint Request
  - Route Rules
  - Target Endpoint Request
  - Request sent to target, and response received
  - Target Endpoint Response
  - Proxy Endpoint Response



Next, the request is sent to the backend target service, which returns a response.

API proxy flows

- As API call is processed, it travels through flows and executes policies attached to those flows
- Evaluation order:
  - Proxy Endpoint Request
  - Route Rules
  - Target Endpoint Request
  - Request sent to target, and response received
  - Target Endpoint Response
  - Proxy Endpoint Response

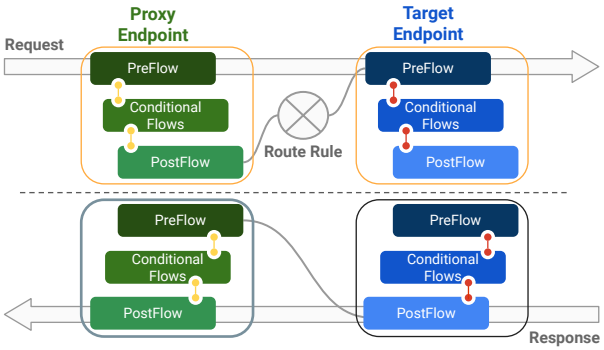


The next set of flows to handle the processing is the target endpoint response.

We generally parse the backend response here.

API proxy flows

- As API call is processed, it travels through flows and executes policies attached to those flows
- Evaluation order:
  - Proxy Endpoint Request
  - Route Rules
  - Target Endpoint Request
  - Request sent to target, and response received
  - Target Endpoint Response
  - Proxy Endpoint Response



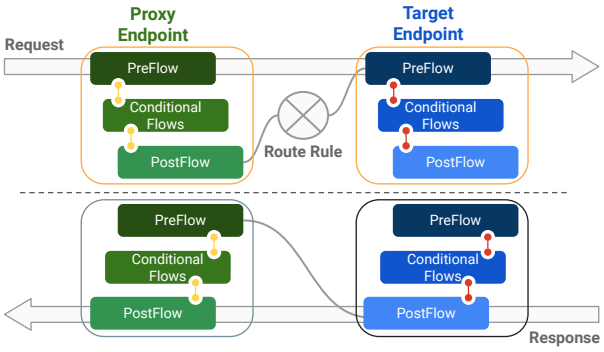
Finally, processing continues to the proxy endpoint response.

It is here we typically build the API response and log any necessary information.

After these flows, the response is returned to the API consumer.

API proxy flows

- As API call is processed, it travels through flows and executes policies attached to those flows.
- Evaluation order:
  - Proxy Endpoint Request
  - Route Rules
  - Target Endpoint Request
  - Request sent to target, and response received
  - Target Endpoint Response
  - Proxy Endpoint Response

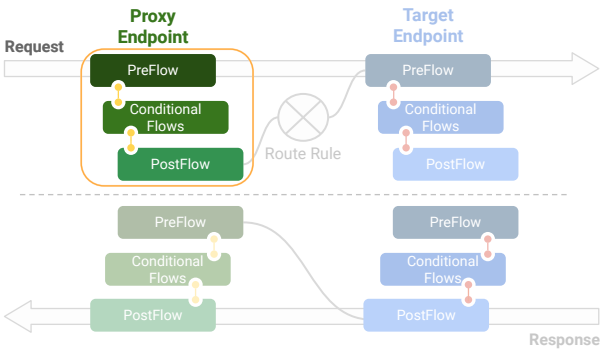


You can see that all of the four sections here look very similar.

Let's explore how the flows work within each section.

Flows

- For each section, policies in PreFlow are evaluated first.
- Next, Conditional Flows are evaluated in order.
  - Only policies in the first matching conditional flow are evaluated.
  - If no conditions are true, no flows are executed.
- Finally, policies in PostFlow are evaluated.



Within each section, policies in the preflow are executed first. Policies in the preflow are always evaluated.

After the preflow are the conditional flows. The conditional flow section contains zero or more flows, each with an optional condition. These flows are evaluated in order, and only the policies in the first conditional flow with a condition that evaluates to true are executed. If no flow conditions are true, none of them are executed.

After at most one conditional flow is evaluated, policies in the postflow are executed. Like the preflow, policies in the postflow are always evaluated.

Policies

- Policies are pre-built modules with specific, limited functions.
- Policy functions include rate limiting, transformation, mediation, and security.
- Policies can run custom code.
- Policies can be conditionally executed.

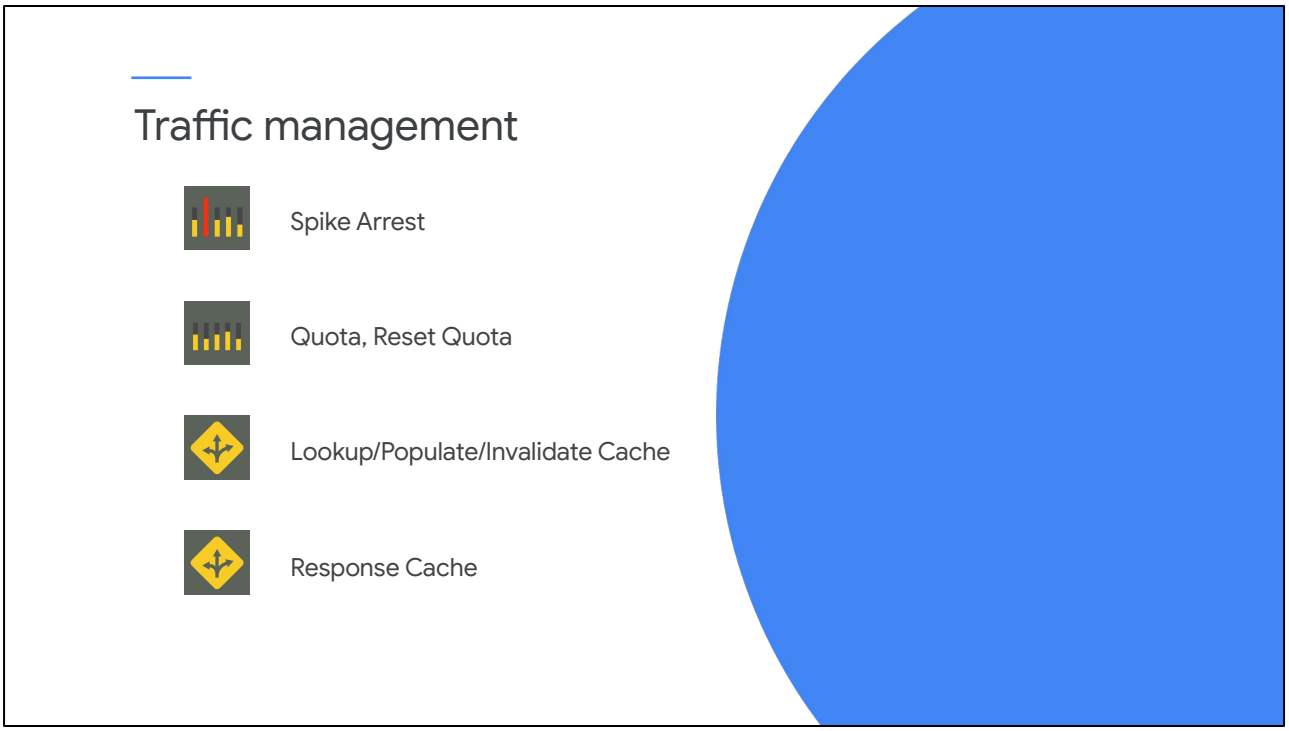
We've been talking about policies quite a bit, but haven't really explained what they are.

Policies are pre-built modules that implement specific functionality.

They allow features like security, rate limiting, and payload manipulation. These features are controlled using simple XML configuration, and without writing any code.

However, when you need code for a complex use case, there are also policies that allow you to run custom code.

A policy can also be conditionally executed. If the condition for a policy evaluates to false, policy execution is skipped.



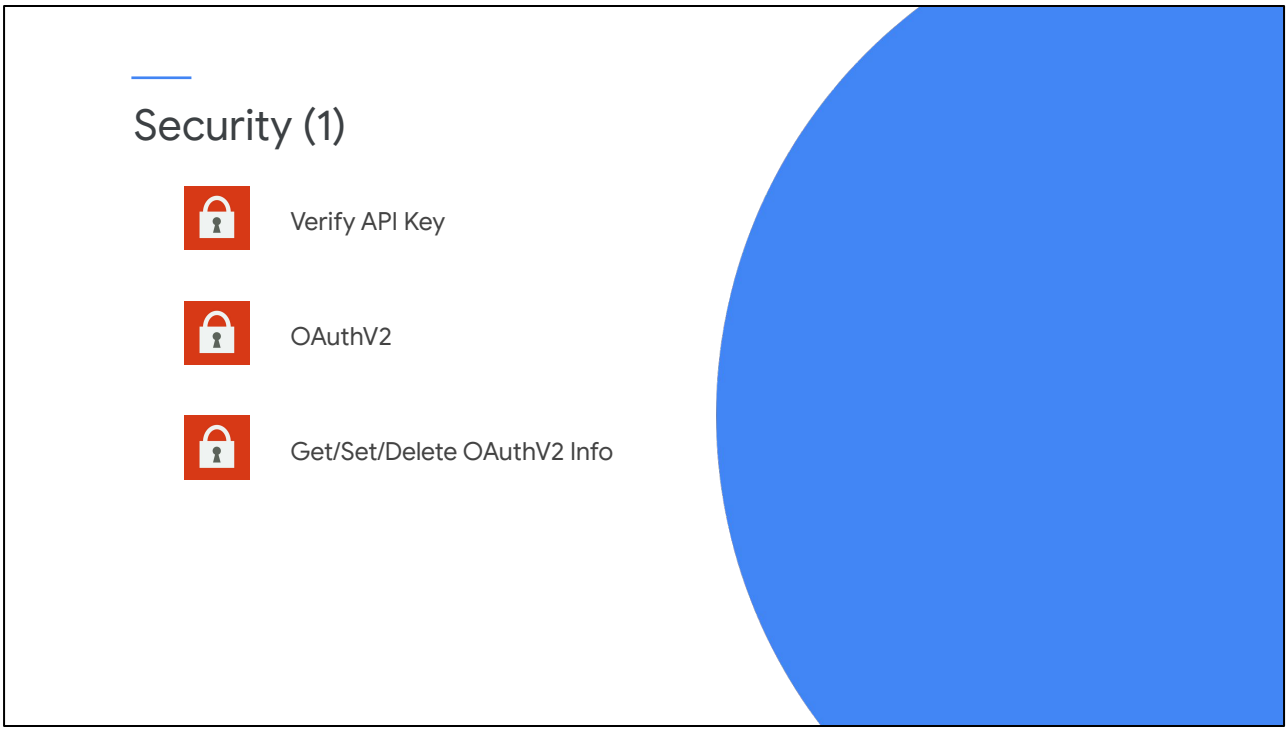
Let's take a quick tour of the policy types available in your toolbox, starting with the traffic management policies. Don't worry if some of these policies are confusing now; we will discuss the policies in much greater detail in later lectures.

The SpikeArrest policy protects your proxy and backends against bursts of traffic.

The Quota policy limits the number of calls allowed over a specified period of time.

Policies for standard caches allow look up, population, and invalidation of cache entries.

A special cache policy called ResponseCache caches entire responses and can eliminate unnecessary traffic to your backends.

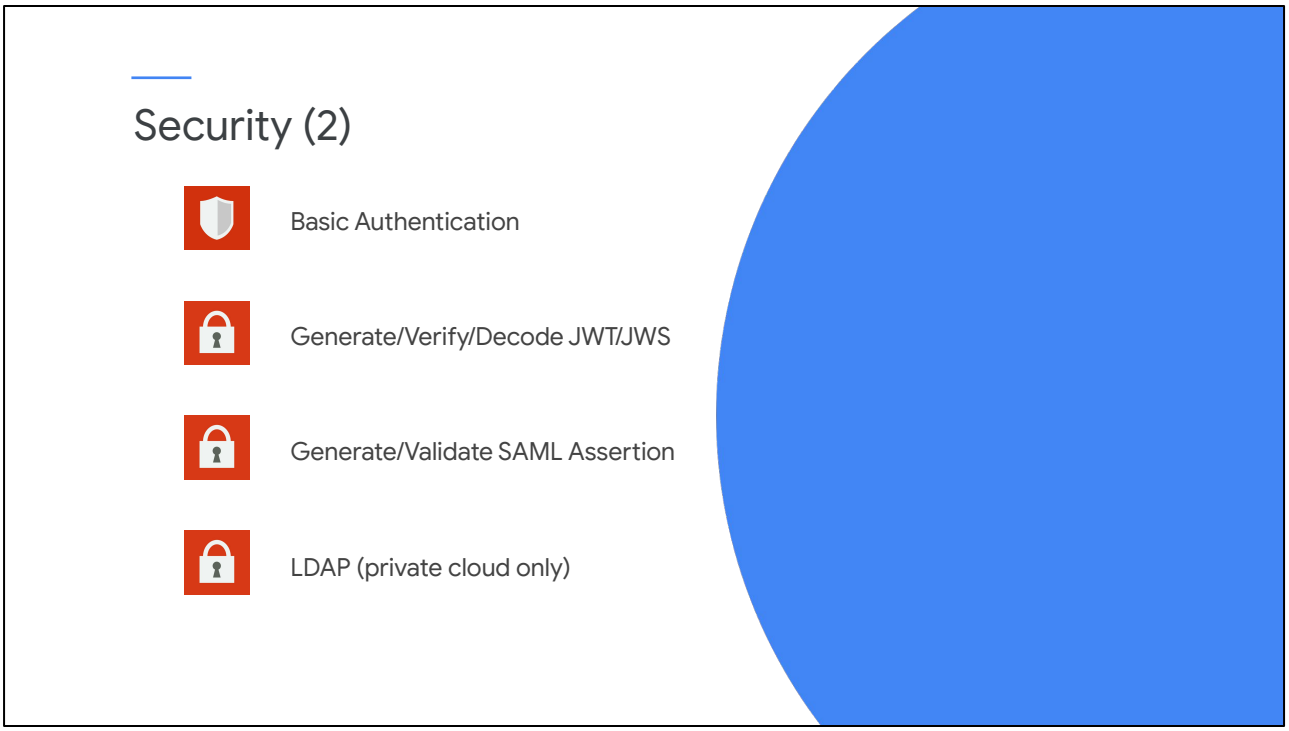


Several security policies are available for securing your proxies.

VerifyAPIKey identifies and validates the application making an API request.

The OAuthV2 policy enables OAuth version 2.0 functionality, providing the ability to generate, verify, refresh, and invalidate OAuth tokens.

There are also policies to set, delete, and retrieve custom attributes associated with OAuth tokens.



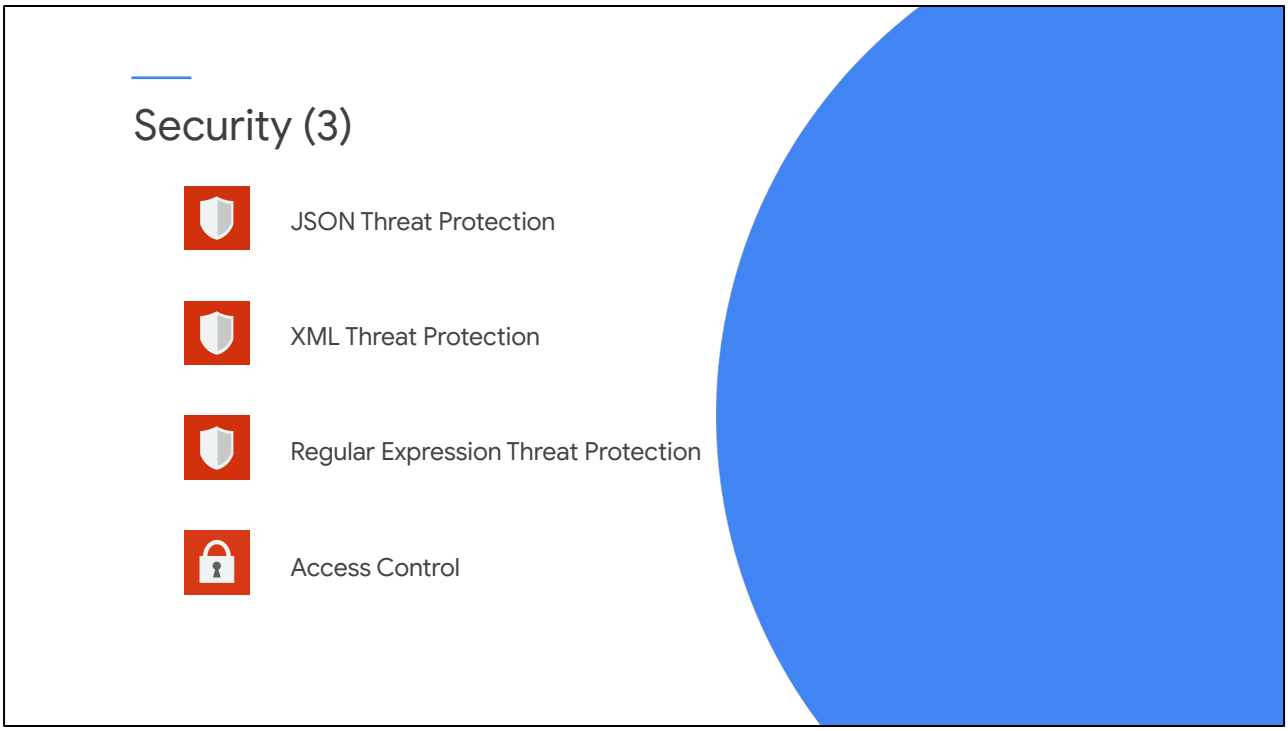
The BasicAuthentication policy builds and parses Authorization headers when used for basic authentication.

There are policies to support creation, decoding, and verifying of JSON Web Tokens, also known as JWT or "jot" tokens.

You can also perform the same operations on JSON Web Signatures.

You can support SAML assertions in your APIs by using the Generate and ValidateSAMLAssertion policies.

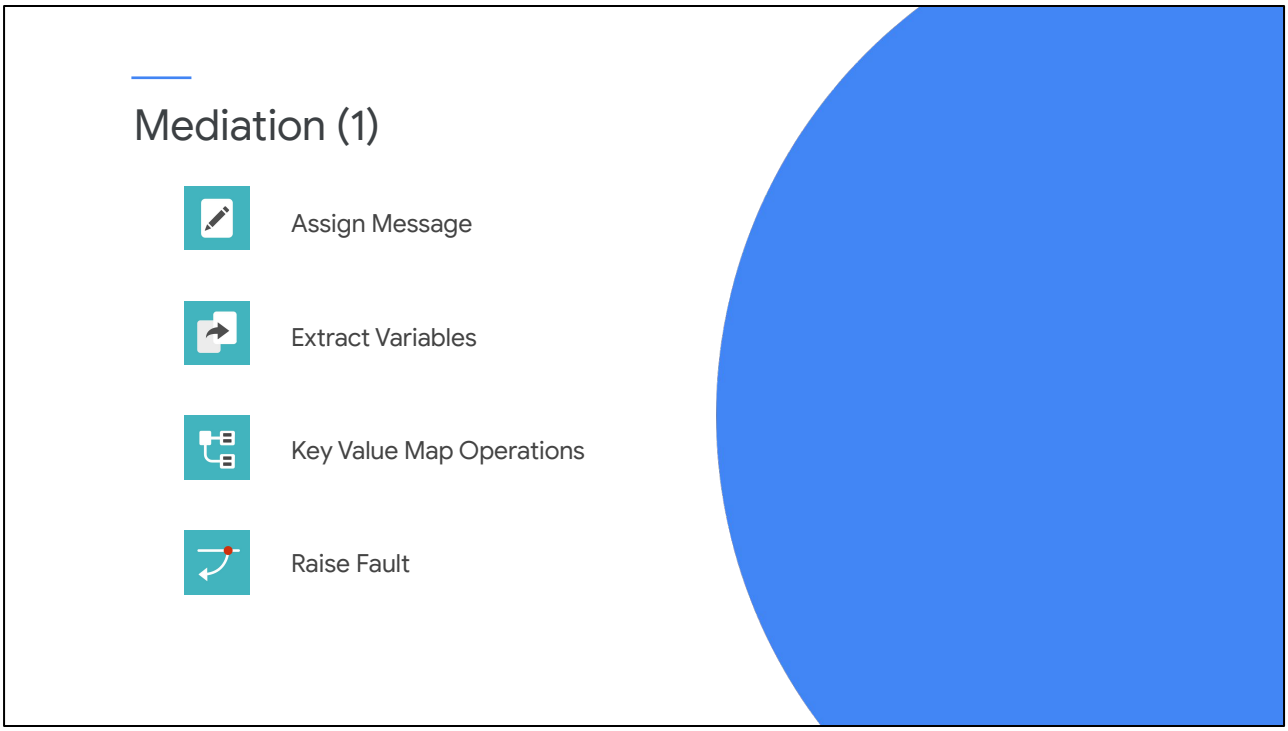
For private cloud deployments, the LDAP policy can verify the users of an app against an LDAP provider.



Apigee provides JSON and XMLThreatProtection policies to protect against malicious JSON and XML payloads.

The RegularExpressionThreatProtection policy can detect malicious patterns in incoming requests.

The AccessControl policy allows or blocks incoming IP addresses and ranges.



Mediation policies allow you to manipulate data, usually in your payloads.

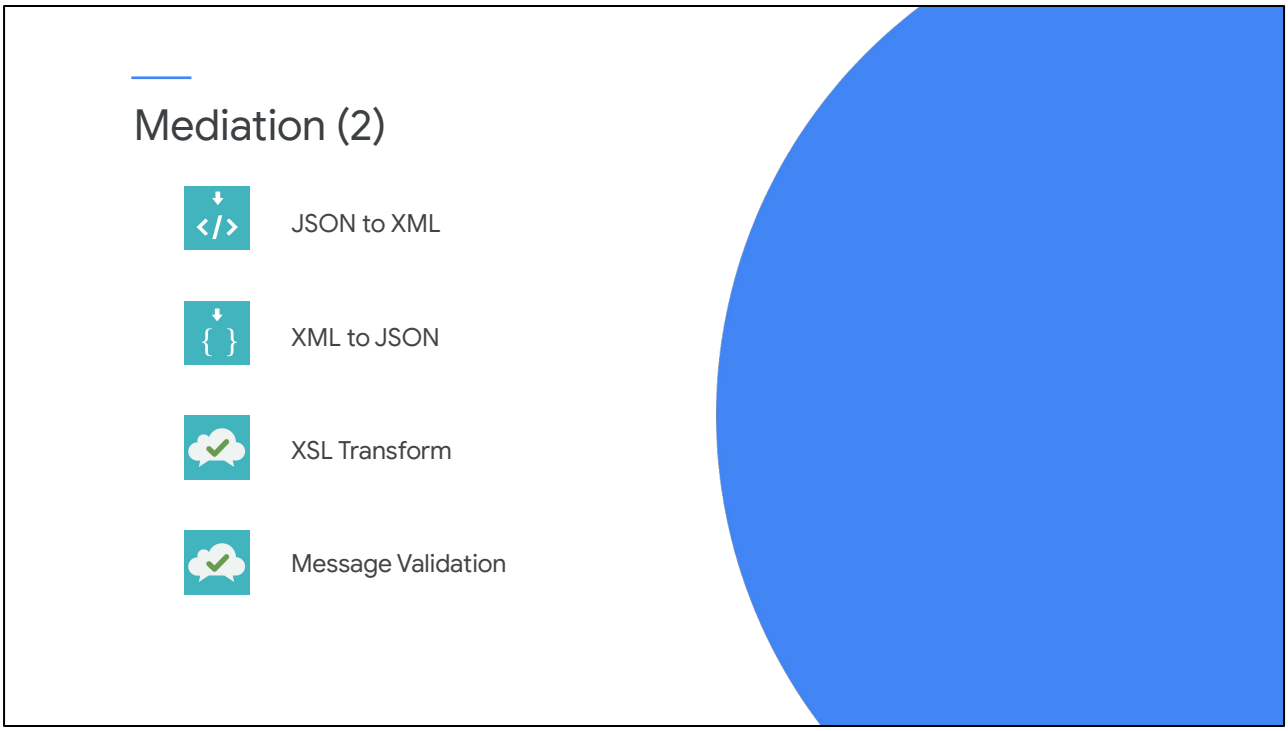
AssignMessage is a very common policy used to create, modify, and remove HTTP message elements and variables.

ExtractVariables extracts information from message elements and variables into new variables.

Key value maps, or KVMs, are usually used to store configuration data.

The KeyValueMapOperations policy reads, writes, and deletes key value map entries.

Using the RaiseFault policy is similar to throwing an exception in many programming languages.

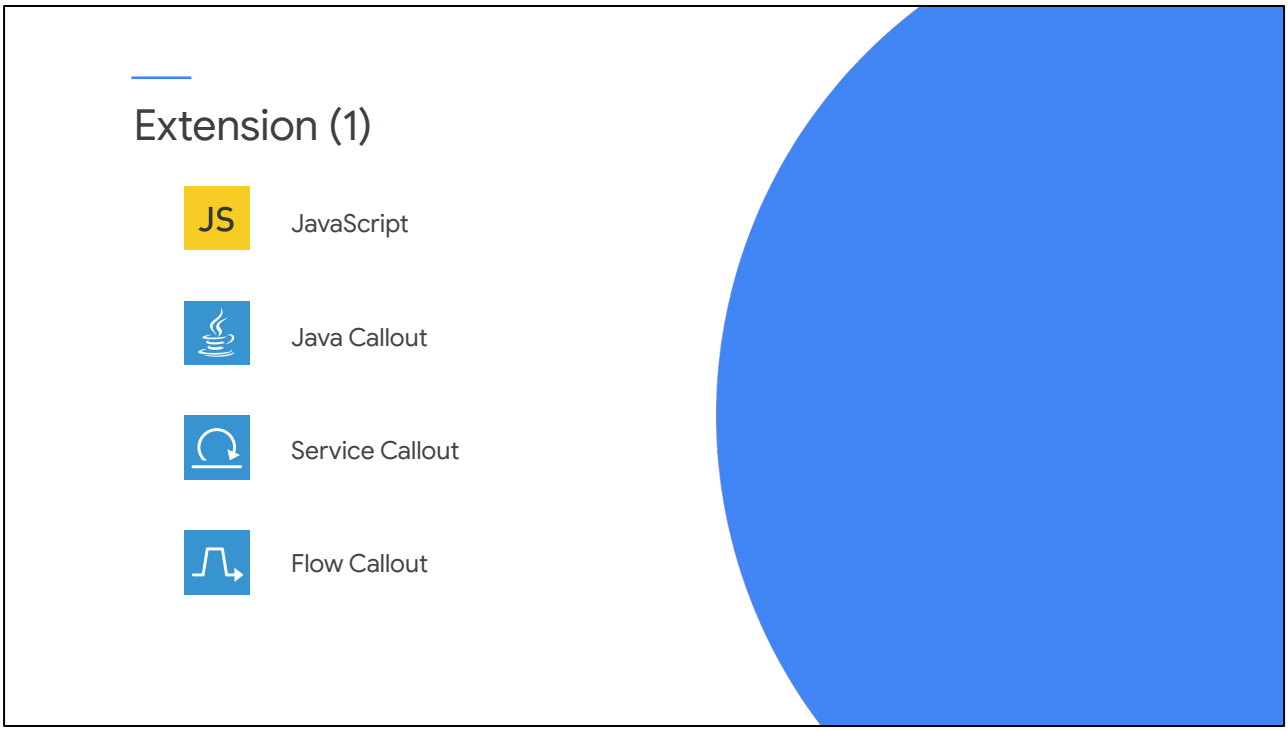


The JSONToXML and XMLToJSON policies convert payloads between XML and JSON formats.

These policies can be especially useful when you are creating a JSON RESTful API that uses a SOAP service backend.

The XSL transform policy transforms XML using XSLT, or Extensible Stylesheet Language Transformations.

The MessageValidation policy validates XML against XSD schemas or SOAP messages against WSDL definitions.



Extension policies provide other specialized functionality for your proxies.

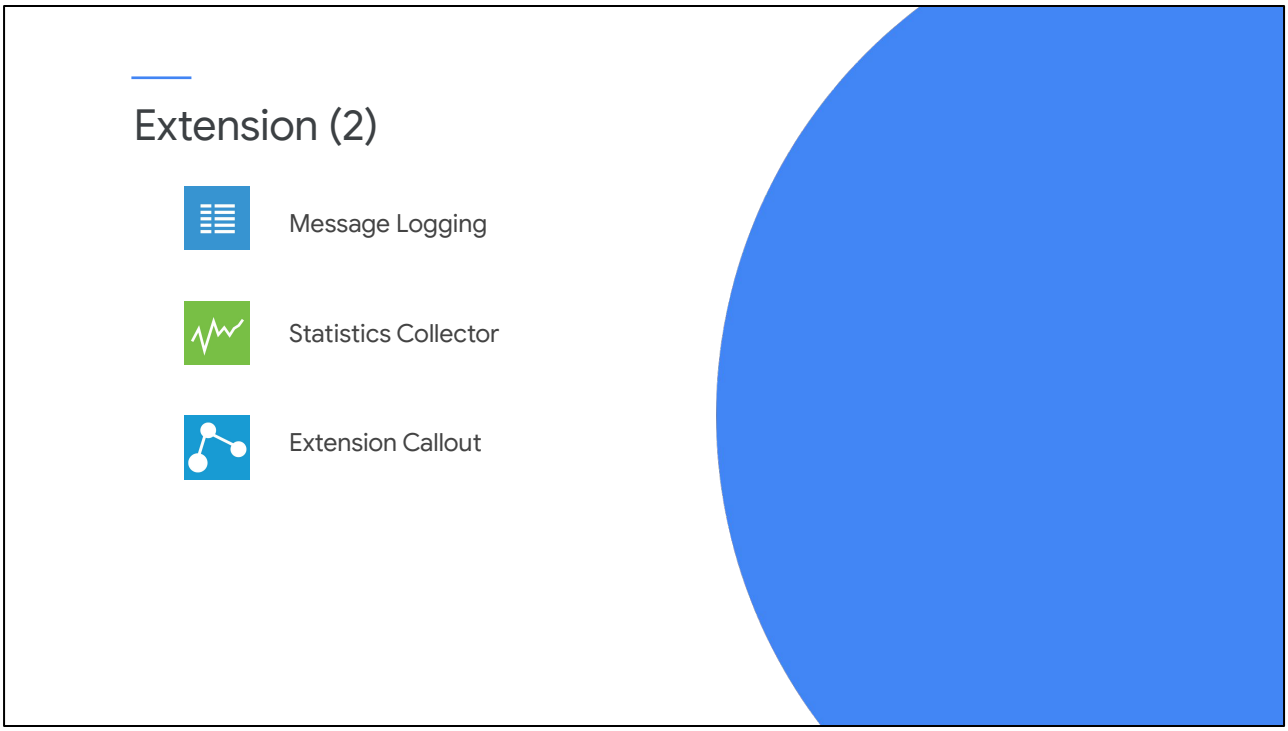
The JavaScript and Java Callout policies allow execution of custom JavaScript and Java code.

ServiceCallout calls an external service or third party API, or calls into another proxy in your environment.

This allows you to call more than one service during an API call.

Shared flows specify a sequence of conditional policies that can be used within multiple proxies.

The flow callout policy calls shared flows.



The MessageLogging policy logs custom information to a syslog server.

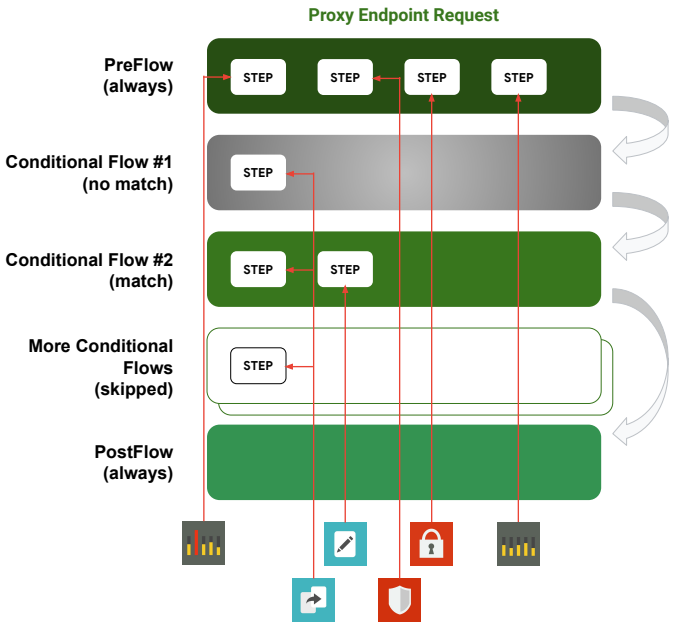
StatisticsCollector allows you to add custom data to the analytics record that is captured for each call.

And finally, the ExtensionCallout policy can be used to call Google Cloud services from within your proxies.

As you can see, there are many out-of-the-box policies that can be used when building your proxies. These policies will help you build powerful and efficient proxies using much less effort than if you had to code it all yourself.

Attaching policies

- Policies can be attached within one or more flows as steps.
- A step can be associated with a condition, executing only if the condition resolves to true.



Policies are created within a proxy. Each policy contains its own configuration written in XML.

Proxies can contain multiple policies of the same policy type, and a policy can be attached to one or more flows within a proxy.

When a policy is attached to a flow, it is called a step.

Each step can be associated with a condition. If the condition evaluates to true, the policy is executed during the step.

When a flow is run, steps are evaluated in order.

The diagram shows the proxy endpoint request flows. The policies are shown at the bottom. They are attached to flows as steps.

When an API call comes in, the steps in the preflow would be run, in order, followed by steps in the first matching conditional flow, if any, and ending with any steps in the postflow.

Example flow config

```
<ProxyEndpoint name="store">
  ⋮
  <PreFlow name="PreFlow">
    <Request>
      <Step>
        <Name>VAK-VerifyKey</Name>
      </Step>
    </Request>
    <Response/>
  </PreFlow>
  <Flows>
    <Flow name="GetOrders">
      <Condition>proxy.pathsuffix="/orders" AND request.verb="GET"</Condition>
      <Request>
        <Step>
          <Condition>request.queryparam.filter != null</Condition>
          <Name>EV-ExtractFilter</Name>
        </Step>
      </Request>
      <Response/>
    </Flow>
    ⋮
  </Flows>
  <PostFlow name="PostFlow"/>
  ⋮
</ProxyEndpoint>
```

Let's look at an example of flows within a proxy endpoint.

Within the request preflow for the proxy endpoint, there is a step with the name VAK-VerifyKey. This is the name of a policy that is defined in a separate policy configuration XML file.

There is no condition for this step, so the verify key policy is always executed.

That is the only step in the proxy endpoint request preflow, so now we look at the request conditional flows.

Example flow config

```
<ProxyEndpoint name="store">
  ⋮
  <PreFlow name="PreFlow">
    <Request>
      <Step>
        <Name>VAK-VerifyKey</Name>
      </Step>
    </Request>
    <Response/>
  </PreFlow>
  <Flows>
    <Flow name="GetOrders">
      <Condition>proxy.pathsuffix="/orders" AND request.verb="GET"</Condition>
      <Request>
        <Step>
          <Condition>request.queryparam.filter != null</Condition>
          <Name>EV-ExtractFilter</Name>
        </Step>
      </Request>
      <Response/>
    </Flow>
    ⋮
  </Flows>
  <PostFlow name="PostFlow"/>
  ⋮
</ProxyEndpoint>
```

The first conditional flow has a name of GetOrders. It executes if the proxy pathsuffix is /orders, and the request verb is GET.

Assuming that compound condition is true, we look at the first step containing the policy EV-ExtractFilter.

This step contains a condition. The policy only runs if there is a query parameter named filter.

Other conditional flow and postflow steps are not shown.

Choose a policy naming convention

- Use a prefix to specify the type of policy.
- Policy's name should explain what the policy does (the default name is usually not descriptive).
- The naming convention we'll use is:

  `{policy type abbreviation}-{name}`

Name *(policy type)*

VAK-VerifyKey *(VerifyAPIKey)*

AM-RemoveHeaders *(AssignMessage)*
AM-BuildRequest *(AssignMessage)*

O2-VerifyToken *(OAuthV2)*
O2-GenerateAuthCode *(OAuthV2)*

JS-CleanResponse *(JavaScript)*

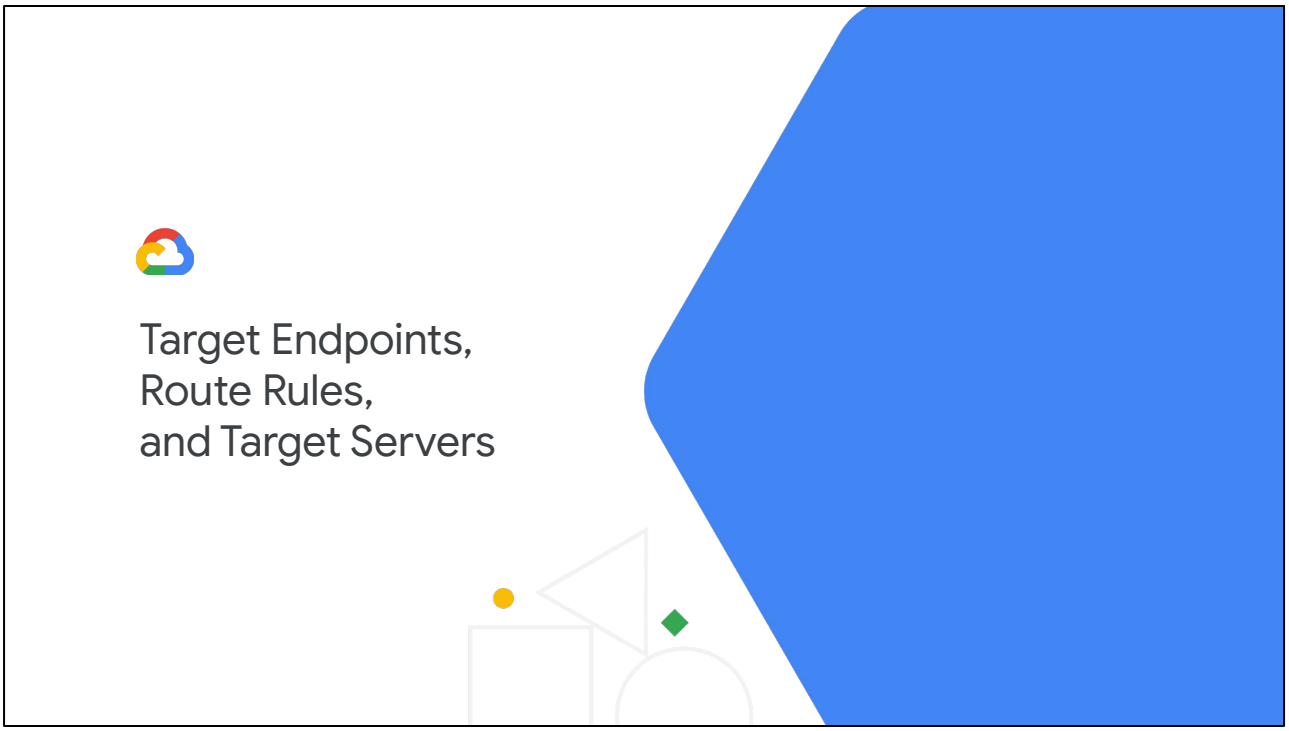
As we saw from the previous example, it might not be easy to determine what policy type is specified in a step without some sort of naming convention. The policy type is shown in the policy definition, not the flow.

One common convention for a policy name is to use an abbreviation for the policy type, followed by a descriptive name, separated by a hyphen.

The policy type is usually not enough information to understand what the policy is meant to do. For example, we might have one assign message policy that is meant to remove headers from a message, and another that builds a request to send to a backend service. Naming the policies assign message 1 and assign message 2 would not tell you what they do.

However, if you name the first policy A M hyphen remove headers, and the second policy A M hyphen build request, it is clear that they are both assign message policies with specific purposes.

We will use this policy naming convention in the labs.



We have learned how traffic is routed to a proxy, and we have explored proxy features like variables, conditions, flows, policies, and steps.

In this lecture we will learn about target endpoints.

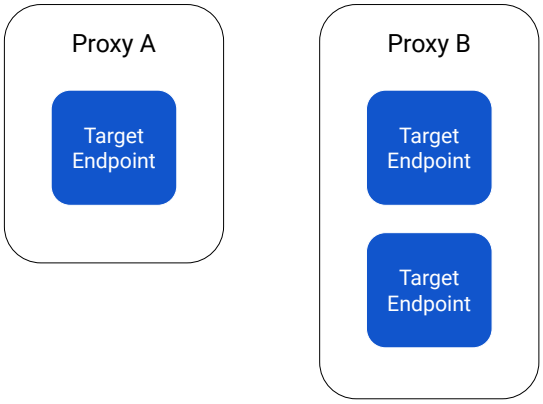
We'll see how we select a target endpoint using route rules.

And we'll also look at how we invoke backend targets and how can we use an Apigee feature called target servers to support our API development lifecycle.

Finally, we'll look at endpoint properties, which can help us control our communication with remote servers.

Target endpoints

- Target endpoints live inside a proxy, and proxies sometimes have multiple target endpoints.

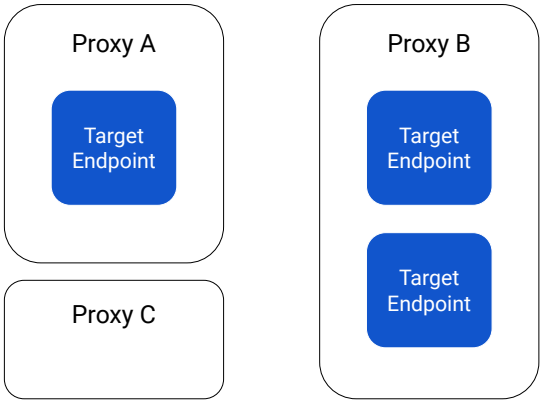


Just like proxy endpoints, multiple target endpoints can exist within a single proxy.

And just like proxy endpoints, only a single target endpoint can be called during an API request.

Target endpoints

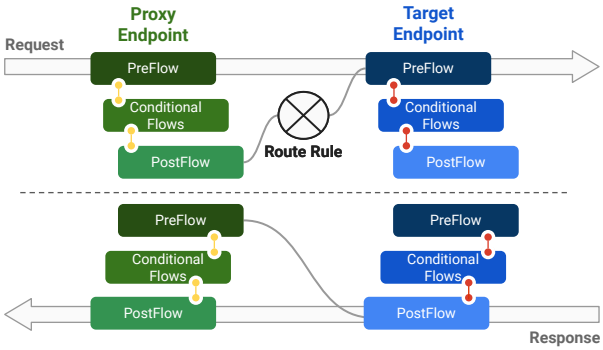
- Target endpoints live inside a proxy, and proxies sometimes have multiple target endpoints.
- Sometimes no target endpoint should be called (when the proxy handles the entire request/response).
- How is the correct target endpoint (if any) determined?



We can also choose to call no target endpoint, if the processing of the API call does not require calling a backend target. Proxies sometimes have no target endpoints.

We learned that routing to proxy endpoints is determined by a combination of virtual host and base path. If there can be multiple target endpoints, how do we determine which one is used?

Route rules



- After proxy endpoint request flows, the configured route rules are evaluated, in order, to determine the target endpoint.
- The first matching route rule indicates the target endpoint.
- Response returns through same proxy and target endpoints.

Target endpoints are chosen based on a proxy's route rules.

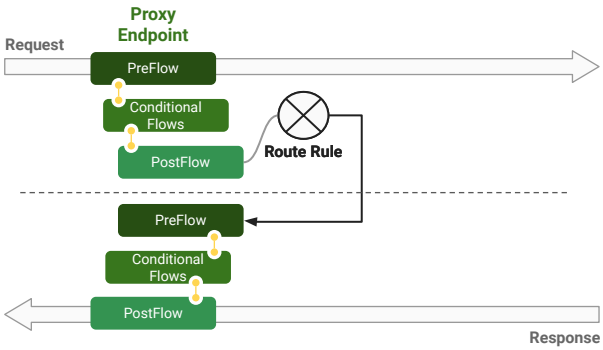
After the postflow for the proxy endpoint request completes, the proxy's configured route rules are evaluated. As with conditional flows, the route rule conditions are evaluated in order to determine which target endpoint to use, if any.

The first matching route rule indicates which target endpoint should be used.

The request processing would continue with the specified target endpoint request flows, the request would be sent to the backend target, and then processing would continue with the target endpoint response flow, followed by the proxy endpoint response flow.

The same target and proxy endpoints that handled the request path would also handle the response path.

Route rules



- After proxy endpoint request flows, the configured route rules are evaluated, in order, to determine the target endpoint.
- The first matching route rule indicates the target endpoint.
- Response returns through same proxy and target endpoints.
- Route rules can indicate that no target endpoint should be used; execution continues with ProxyEndpoint Response.

A matched route rule might specify that no target endpoint should be used.

In this case, the processing would continue with the proxy endpoint response.

It would be up to the proxy endpoint response flows to build the correct response for the API call.

Route rules

- Has an optional condition.
  - If no condition, rule automatically matches.
- Has an optional target endpoint.
  - If no target endpoint specified for a matching route rule, no target endpoint is run.
- Best practice: Last route rule should have no condition.

```
<ProxyEndpoint name="store">
  ⋮
  <RouteRule name="auth-notarget">
    <Condition>proxy.pathsuffix == "/auth"</Condition>
  </RouteRule>
  <RouteRule name="test-store-server">
    <Condition>request.queryparam.test != null</Condition>
    <TargetEndpoint>test-store-backend</TargetEndpoint>
  </RouteRule>
  <RouteRule name="default">
    <TargetEndpoint>store-backend</TargetEndpoint>
  </RouteRule>
</ProxyEndpoint>
```

Each route rule has an optional condition. The route rule is a match if the condition is true. If there is no condition on a route rule, the route rule is always a match.

The target endpoint for a route rule specifies which target to use if the route rule is a match. If there is no target endpoint specified, no target endpoint will be used, and processing will continue in the proxy endpoint response.

For your ordered list of route rules, it is a best practice to have the last route rule with no condition, so it always matches if none of the other route rules do.

Let's look at the route rule example here.

The first route rule, auth-notarget, has a condition but no target endpoint. If the path suffix is /auth, no target endpoint will be used, and processing will continue with the proxy endpoint response flows.

The second route rule, test-store-server, matches if there is a test query parameter in the request. If there is, the test-store-backend target endpoint is selected.

The third route rule, default, has no condition. If the first two route rules are not a match, this route rule is an automatic match and will select the store-backend target endpoint.

HTTPTargetConnection

```
<TargetEndpoint name="store-backend">
  ⋮
  <HTTPTargetConnection>
    <URL>https://internal-test.apigee.org/store</URL>
  </HTTPTargetConnection>
</TargetEndpoint>
```

- Target endpoint specifies the backend target to call in the HTTPTargetConnection.

The backend destination is specified in the HTTPTargetConnection element.

In this example, the target URL for the backend is hardcoded.

HTTPTargetConnection

```
<TargetEndpoint name="store-backend">
  ⋮
  <HTTPTargetConnection>
    <URL>https://internal-test.apigee.org/store</URL>
  </HTTPTargetConnection>
</TargetEndpoint>
```

API Request:
GET
https://test-api.apigee.org/stores/v1/orders/12

Backend Request:
GET
https://internal-test.apigee.org/store/orders/12

- Target endpoint specifies the backend target to call in the HTTPTargetConnection.
- By default, proxy.pathsuffix is appended to backend URL.

The proxy pathsufffix is typically appended to the configured target URL to determine the full backend URL.

In this example, the original request was to the stores-v1 proxy.

The proxy pathsuffix is /orders/12. The proxy pathsuffix is appended to the URL configured in the HTTPTargetConnection.

The original verb, GET, would also be used for the backend request unless you change it.

Pitfall of hardcoded targets

test:

```
<TargetEndpoint name="store-backend">
  ⋮
  <HTTPTargetConnection>
    <URL>https://internal-test.apigee.org/store</URL>
  </HTTPTargetConnection>
</TargetEndpoint>
```

- What happens when we want to promote this revision of the API proxy from test into production?

prod:

Say this proxy code is deployed to our test environment.

Assuming our test and prod backends are different, how would we promote the code to the production environment?

Pitfall of hardcoded targets

test:

```
<TargetEndpoint name="store-backend">
  ⋮
  <HTTPTargetConnection>
    <URL>https://internal-test.apigee.org/store</URL>
  </HTTPTargetConnection>
</TargetEndpoint>
```

prod:

```
<TargetEndpoint name="store-backend">
  ⋮
  <HTTPTargetConnection>
    <URL>https://internal-prod.apigee.org/store</URL>
  </HTTPTargetConnection>
</TargetEndpoint>
```

- What happens when we want to promote this revision of the API proxy from test into production?
- Answer: We'd have to change the URL!
- This is an anti-pattern!

We'd need to change the hardcoded URL in the code. The code validated in the test environment would need to be changed before putting it into production.

This is an anti-pattern. We shouldn't put untested code into production.

However, there is a solution for this.

Target server configuration

- Create a target server configuration with the same name in each environment.
- The host name and port are specified in the target server configuration, along with any certificate information needed for client or server validation.

Target Server Configuration

test environment:
name=store-backend,
host=internal-test.apigee.org, port=443,
certificate info

prod environment:
name=store-backend,
host=internal-prod.apigee.org, port=443,
certificate info

We can create a target server in all environments, each with the same name.

The hostname, port, and any required certificate information in the target server configuration can be different for each environment.

We can use the same target server name in each environment, and the environment-specific target servers can have different hostnames.

In the example, the target server named store-backend routes to the test backend for the test environment, and to the prod backend for the prod environment.

Target servers

- Reference the target server name in a LoadBalancer section.
- Path is specified in the proxy.
- The same code can now be used in each environment!

test:

```
<TargetEndpoint name="store-backend">
  ⋮
  <HTTPTargetConnection>
    <URL>https://internal-test.apigee.org/store</URL>
  </HTTPTargetConnection>
</TargetEndpoint>
```


test & prod:

```
<TargetEndpoint name="store-backend">
  ⋮
  <HTTPTargetConnection>
    <LoadBalancer>
      <Server name="store-backend"/>
    </LoadBalancer>
    <Path>/store</Path>
  </HTTPTargetConnection>
</TargetEndpoint>
```

Now we can use the same proxy code, regardless of the environment. We replace the URL with the target server name.

The hostname and port is taken from the target server configuration for the current environment in which the proxy is deployed.

The part of the URL following the hostname is configured with the Path element.

Now the HTTPTargetConnection configuration does not mention test or prod. No change to the code is required when promoting the proxy into production, because the target server contains the environment-specific information.

Notice the LoadBalancer element in the new code. We aren't doing any load balancing here, because we only have one backend, but target servers do have to be specified inside a LoadBalancer element.

Load balancing your backend servers

- Distribute traffic among backend servers using algorithms like least connections or round robin.

```
<TargetEndpoint name="my-backend">
  ⋮
  <HTTPTargetConnection>
    <LoadBalancer>
      <Algorithm>RoundRobin</Algorithm>
      <Server name="backend-1"/>
      <Server name="backend-2"/>
      <MaxFailures>5</MaxFailures>
    </LoadBalancer>
    <HealthMonitor>
      <IsEnabled>true</IsEnabled>
      <IntervalInSec>10</IntervalInSec>
      <TCPMonitor>
        <ConnectTimeoutInSec>5</ConnectTimeoutInSec>
        <Port>80</Port>
      </TCPMonitor>
    </HealthMonitor>
    <Path>/v1</Path>
  </HTTPTargetConnection>
</TargetEndpoint>
```

Apigee provides a load balancing feature for distributing traffic among multiple backend servers.

Backends need to be configured as target servers.

Load balancing algorithms like round robin and least connections can be used to distribute traffic between the backends.

Load balancing your backend servers

- Distribute traffic among backend servers using algorithms like least connections or round robin.
- Can automatically take failing backends out of service.

```
<TargetEndpoint name="my-backend">
  ⋮
  <HTTPTargetConnection>
    <LoadBalancer>
      <Algorithm>RoundRobin</Algorithm>
      <Server name="backend-1"/>
      <Server name="backend-2"/>
      <MaxFailures>5</MaxFailures>
    </LoadBalancer>
    <HealthMonitor>
      <IsEnabled>true</IsEnabled>
      <IntervalInSec>10</IntervalInSec>
      <TCPMonitor>
        <ConnectTimeoutInSec>5</ConnectTimeoutInSec>
        <Port>80</Port>
      </TCPMonitor>
    </HealthMonitor>
    <Path>/v1</Path>
  </HTTPTargetConnection>
</TargetEndpoint>
```

You can use the MaxFailures element to specify the maximum number of consecutive failures that are allowed before a backend is automatically taken out of service.

Load balancing your backend servers

- Distribute traffic among backend servers using algorithms like least connections or round robin.
- Can automatically take failing backends out of service.
- TCP and HTTP health monitoring can detect when backends are healthy and bring them back into service.

```
<TargetEndpoint name="my-backend">
  ⋮
  <HTTPTargetConnection>
    <LoadBalancer>
      <Algorithm>RoundRobin</Algorithm>
      <Server name="backend-1"/>
      <Server name="backend-2"/>
      <MaxFailures>5</MaxFailures>
    </LoadBalancer>
    <HealthMonitor>
      <IsEnabled>true</IsEnabled>
      <IntervalInSec>10</IntervalInSec>
      <TCPMonitor>
        <ConnectTimeoutInSec>5</ConnectTimeoutInSec>
        <Port>80</Port>
      </TCPMonitor>
    </HealthMonitor>
    <Path>/v1</Path>
  </HTTPTargetConnection>
</TargetEndpoint>
```

Configured TCP or HTTP health checks can detect when a backend is healthy again and bring the backend back into rotation.

When you use the MaxFailures element, you should always include a health check; if you don't, backends removed from service will never be put back into service.

Load balancing your backend servers

- Distribute traffic among backend servers using algorithms like least connections or round robin.
- Can automatically take failing backends out of service.
- TCP and HTTP health monitoring can detect when backends are healthy and bring them back into service.

```
<TargetEndpoint name="my-backend">
  ⋮
  <HTTPTargetConnection>
    <LoadBalancer>
      <Algorithm>RoundRobin</Algorithm>
      <Server name="backend-1"/>
      <Server name="backend-2"/>
      <MaxFailures>5</MaxFailures>
    </LoadBalancer>
    <HealthMonitor>
      <IsEnabled>true</IsEnabled>
      <IntervalInSec>10</IntervalInSec>
      <TCPMonitor>
        <ConnectTimeoutInSec>5</ConnectTimeoutInSec>
        <Port>80</Port>
      </TCPMonitor>
    </HealthMonitor>
    <Path>/v1</Path>
  </HTTPTargetConnection>
</TargetEndpoint>
```

Apigee's load balancing feature is not intended to replace the use of load balancers in backend data centers.

Endpoint properties

- Important target endpoint properties include:
  - connect.timeout.millis—connection timeout
  - Io.timeout.millis—socket read/write timeout

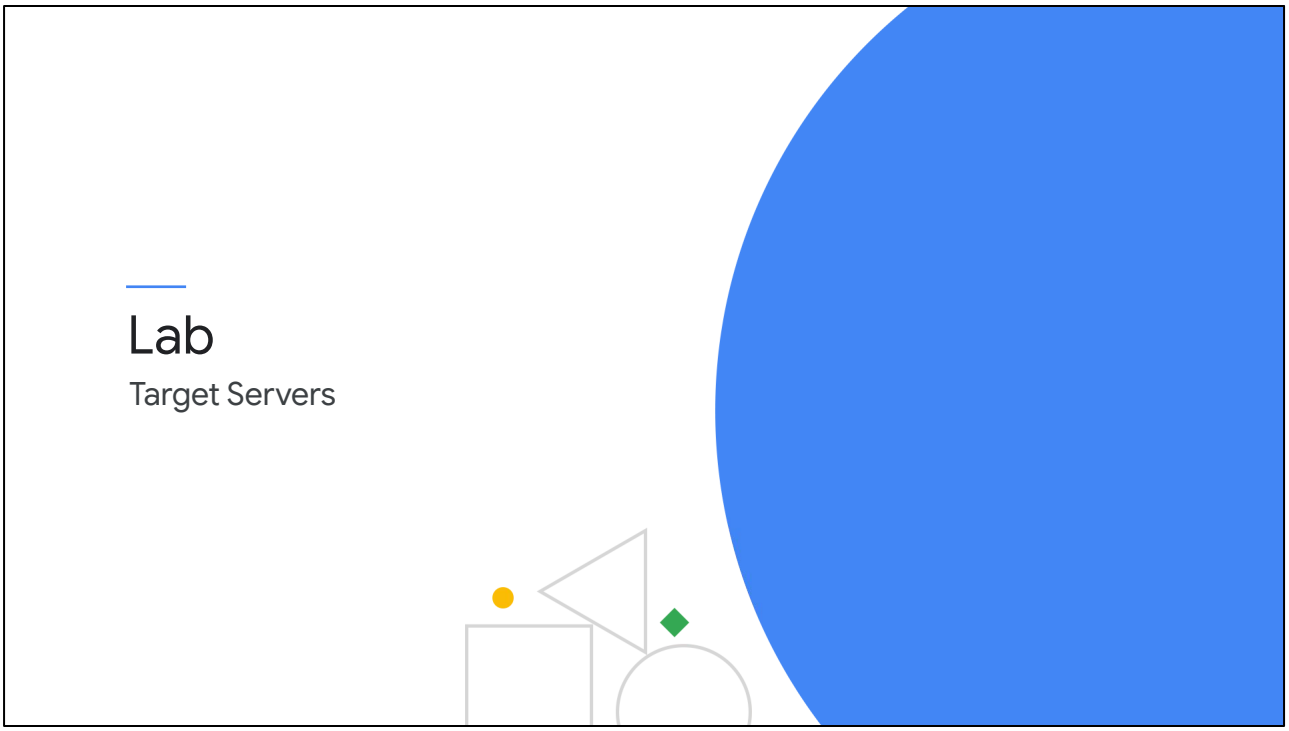
```
<TargetEndpoint name="my-backend">
  ⋮
  <HTTPTargetConnection>
    <URL>https://internal-test.apigee.org/store</URL>
    <Properties>
      <Property name="connect.timeout.millis">2000</Property>
      <Property name="io.timeout.millis">15000</Property>
      <Property name="compression.algorithm">gzip</Property>
    </Properties>
  </HTTPTargetConnection>
</TargetEndpoint>
```

You can use endpoint properties to change many transport settings for your connections to the backend.

In most cases you should reduce the default connection and socket read/write timeouts.

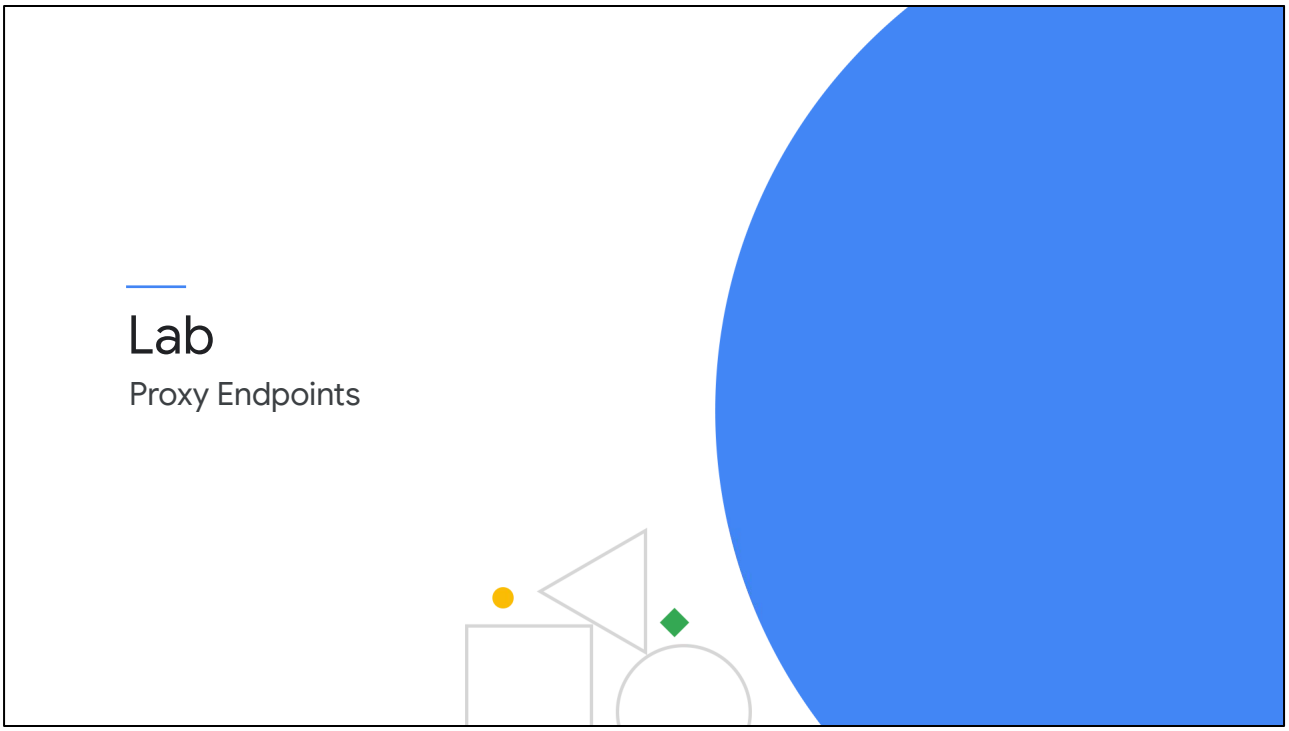
At the time this course is being recorded, the default socket I/O timeout for target endpoints is 55 seconds, and the default connection timeout is 3 seconds.

These default timeouts could result in unreasonable latencies when the backend is not responding.



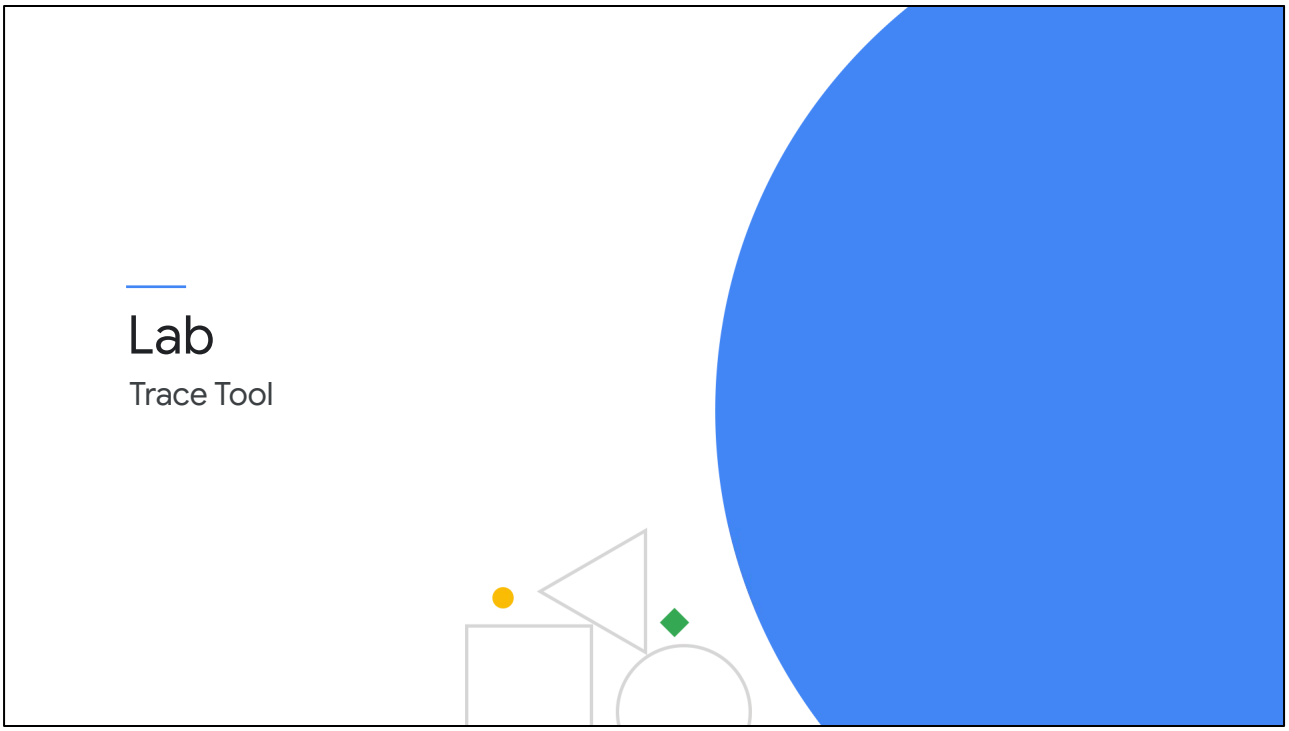
In this lab you build upon your retail proxy, replacing the hardcoded target URL with a reference to a target server. This allows you to deploy your proxy to different environments without making any code changes, even though the URL for the backend service is different for each environment.

You will create and configure the target server in the test environment and update your retail proxy to use the target server instead of the hardcoded URL. After you complete this change, you will test your proxy to make sure that requests are still being proxied successfully to the backend.



In this lab you create a new proxy that has two proxy endpoints. Your first endpoint uses the secure virtual host, and proxies traffic through to the backend. Your second endpoint uses the default virtual host, and returns an error whenever traffic is received by this endpoint.

This is a useful pattern for your proxies. It is generally recommended that API requests be accepted only when sent securely. If you choose to support only secure traffic in your proxy, any insecure request would, by default, get a generic Apigee error message indicating that a proxy matching the request was not found. It is more user friendly to respond to insecure requests in your API, returning an error message indicating how the app developer can fix the error.



The Apigee trace tool is used to troubleshoot and debug your proxies. In this lab you explore the features of the trace tool and trace some API calls through a proxy. You also filter requests using the trace tool, allowing you to capture specific requests when your API is receiving other live traffic. Finally, you learn how to save trace sessions for future investigation.

You have learned about API proxies, proxy and target endpoints, route rules, virtual hosts, target servers, flows, variables, and conditions.

We also introduced policies, pre-built functions that are used within your API proxies.

We will learn more about the different policies in later lectures.

You also completed four labs, including two labs in which you started building your retail API proxy.

You will add more features to your API proxy during this series of courses.