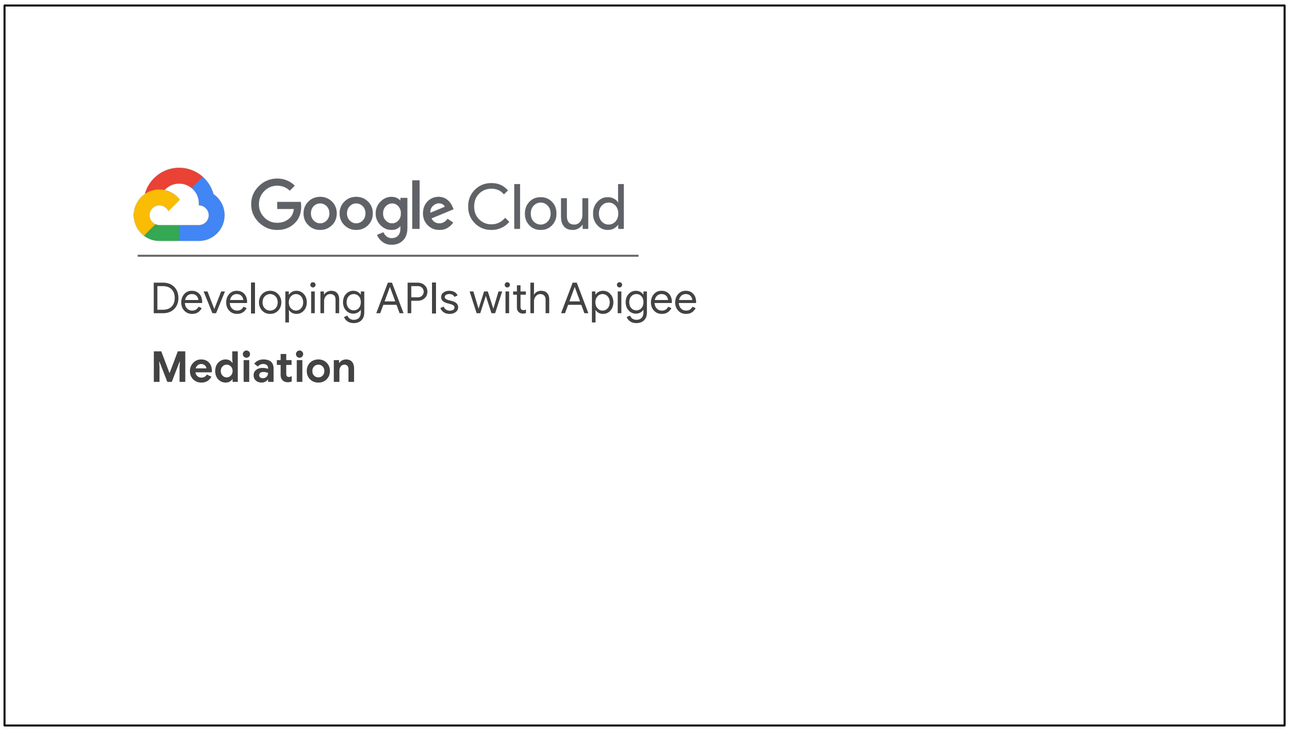


Google Cloud

Developing APIs with Apigee

Mediation

In this module, you'll learn about API mediation.

We will learn about JSON, XML, and SOAP, and the related Apigee policies.

You'll learn how to call multiple services during an API call, and mash up responses.

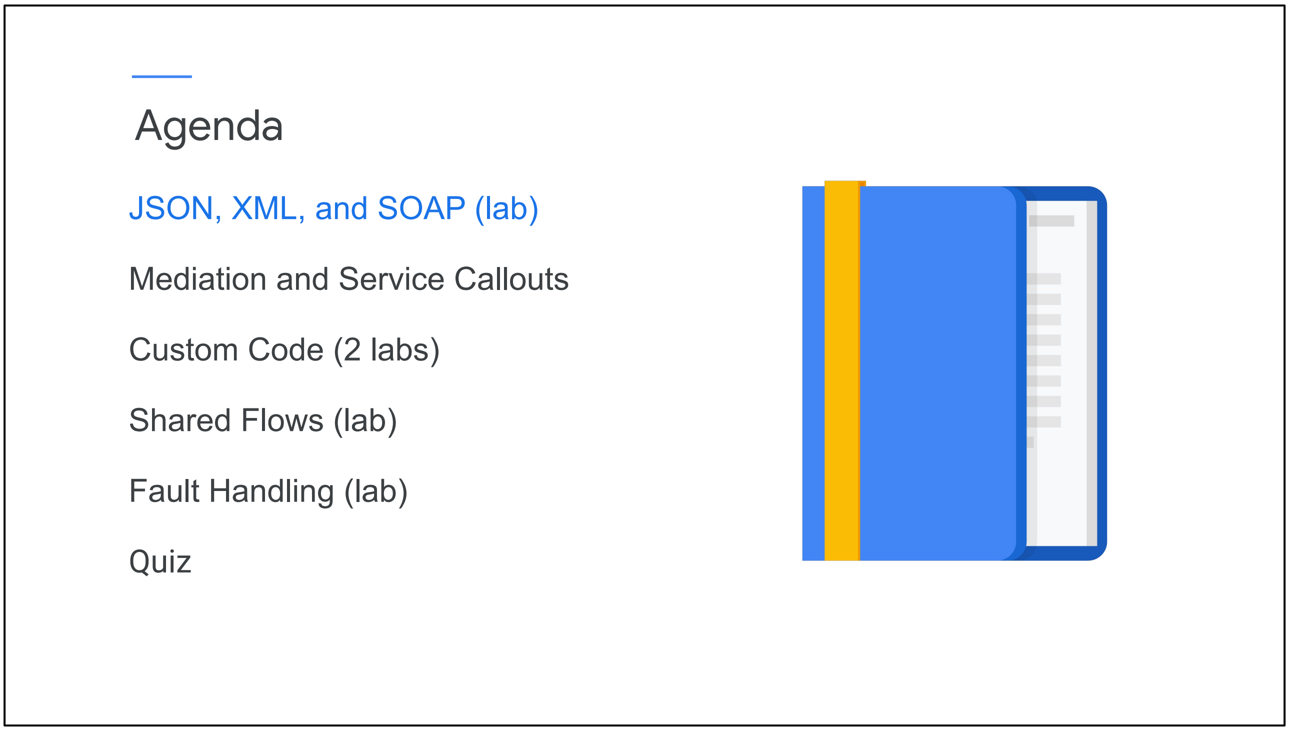
You will also learn about using custom code, sharing logic between API proxies, and how errors are handled in Apigee.

You'll also explore mediation using several labs.

You will add XML support to your retail API, and call a Google geocoding API to add a feature to your API.

You will also add fault handling to your API, and use a shared flow to implement proxy code to securely call the backend service.

Another lab will explore how you can call services in parallel in your API proxies.



Agenda

JSON, XML, and SOAP (lab)

Mediation and Service Callouts

Custom Code (2 labs)

Shared Flows (lab)

Fault Handling (lab)

Quiz

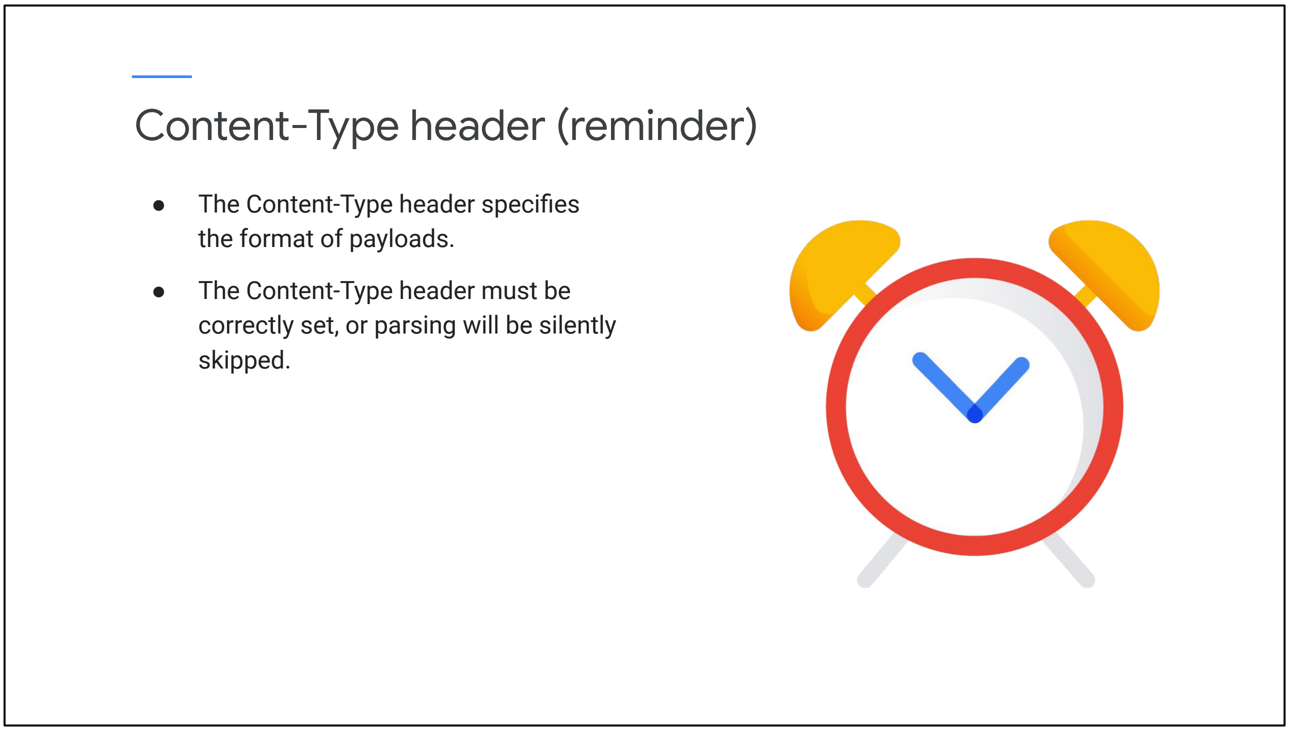
The majority of new APIs being created these days are JSON-based.

However, in many industries, XML and SOAP are still the dominant formats.

Apigee can help you support all three formats for your APIs.

This lecture will discuss the differences between JSON and XML and introduce some new XML and JSON-focused policies.

We will finish the section with a lab.



Content-Type header (reminder)

- The Content-Type header specifies the format of payloads.
- The Content-Type header must be correctly set, or parsing will be silently skipped.

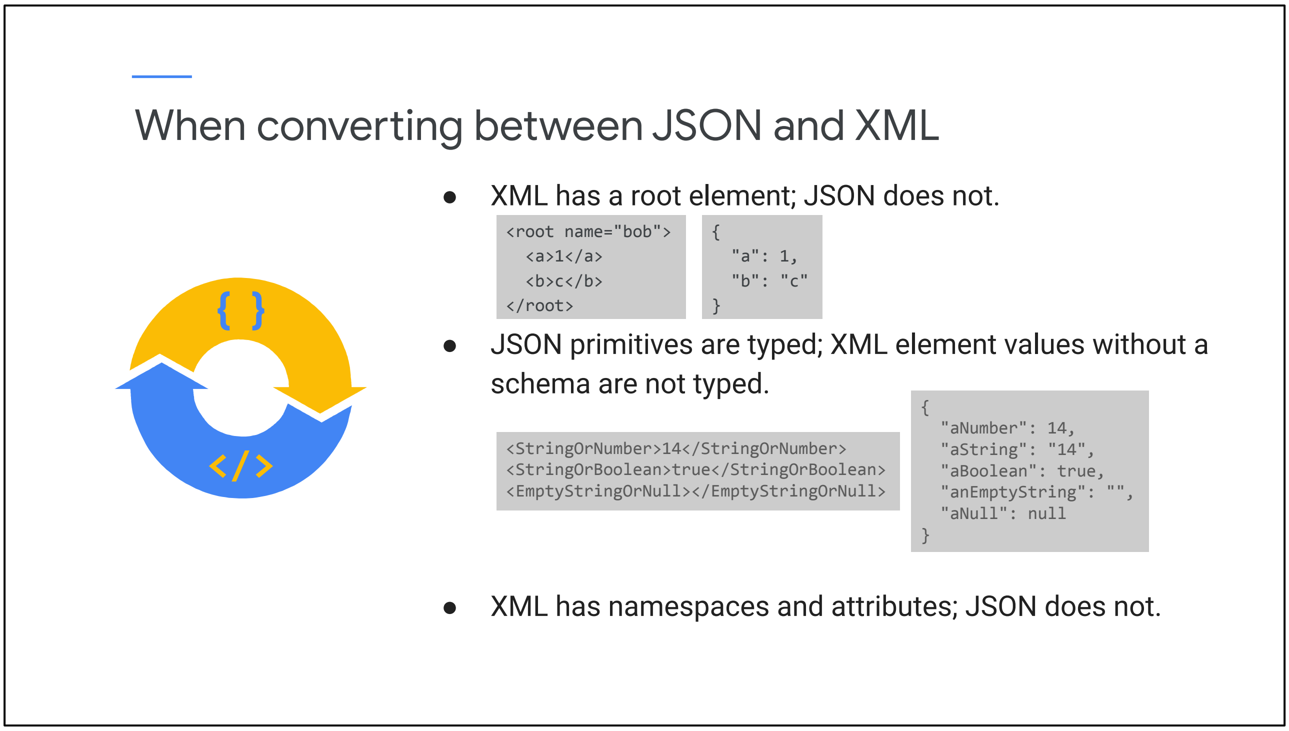
Before we get started, just a reminder.

The format of a request or response is specified by the Content-Type header.

Even if the payload looks like JSON, it isn't JSON if the Content-Type header isn't application/json.

If you are trying to use a policy that acts on JSON, but the Content-Type header is not application/json, the JSON operation in the policy will be skipped.

So remember to validate or set the Content-Type header for your APIs.



When converting between JSON and XML

- XML has a root element; JSON does not.

```
<root name="bob">
  <a>1</a>
  <b>c</b>
</root>
```

```
{
  "a": 1,
  "b": "c"
}
```

- JSON primitives are typed; XML element values without a schema are not typed.

```
<StringOrNumber>14</StringOrNumber>
<StringOrBoolean>true</StringOrBoolean>
<EmptyStringOrNull></EmptyStringOrNull>
```

```
{
  "aNumber": 14,
  "aString": "14",
  "aBoolean": true,
  "anEmptyString": "",
  "aNull": null
}
```

- XML has namespaces and attributes; JSON does not.

You will probably need to convert between JSON and XML in your API proxies. For example, you might want to support both formats or use SOAP service backends for your REST API. Apigee can help you easily convert between the formats, but there are some differences to keep in mind.

First, XML has a required root element, but JSON does not. When converting between the two formats, you'll need to think about how to handle the root element.

Second, XML element values are not typed unless an XML schema is used. Therefore, converting from XML to JSON may require guessing about a data type.

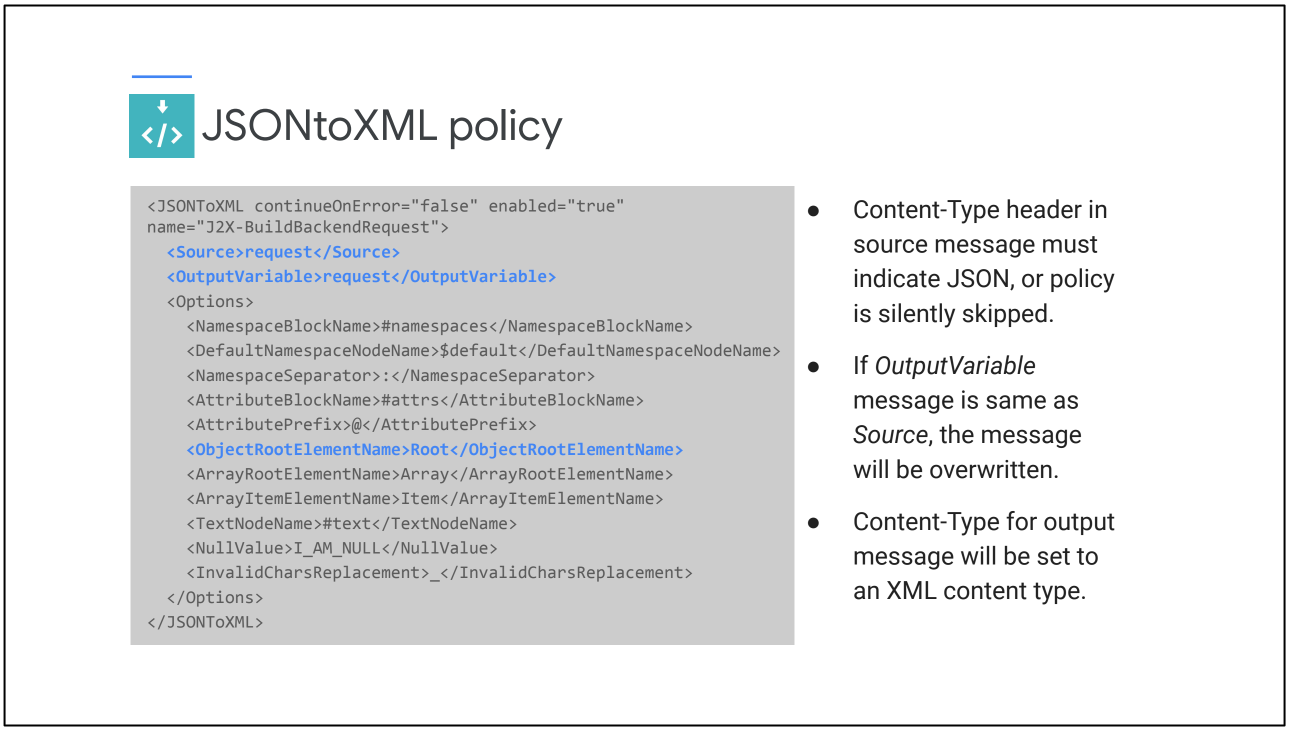
As an example, in the United States, a ZIP code is a postal code used for routing mail. A ZIP code often looks like a 5-digit number, but the long form of a ZIP code adds a hyphen and 4 more digits to the 5-digit number. A ZIP code should typically be stored as a string so that either format can be used.

An XML element value that looks like a number may need to be a string in JSON, depending on the context.

Third, XML has namespaces and attributes, and JSON does not. It may not make sense to strip out attributes when converting from XML to JSON, because attributes are often a key part of the data. You'll need to figure out how to map the attributes into JSON.

When converting from JSON to XML, if some of the JSON data should be represented as attributes in XML, the XML might require some manipulation.

For all these reasons, you may need to manipulate your XML or JSON when trying to convert between the formats.



JSONtoXML policy

```
<JSONToXML continueOnError="false" enabled="true"
name="J2X-BuildBackendRequest">
  <Source>request</Source>
  <OutputVariable>request</OutputVariable>
  <Options>
    <NamespaceBlockName>#namespaces</NamespaceBlockName>
    <DefaultNamespaceNodeName>$default</DefaultNamespaceNodeName>
    <NamespaceSeparator>:</NamespaceSeparator>
    <AttributeBlockName>#attrs</AttributeBlockName>
    <AttributePrefix>@</AttributePrefix>
    <ObjectRootElementName>Root</ObjectRootElementName>
    <ArrayRootElementName>Array</ArrayRootElementName>
    <ArrayItemElementName>Item</ArrayItemElementName>
    <TextNodeName>#text</TextNodeName>
    <NullValue>I_AM_NULL</NullValue>
    <InvalidCharsReplacement>_</InvalidCharsReplacement>
  </Options>
</JSONToXML>
```

- Content-Type header in source message must indicate JSON, or policy is silently skipped.
- If *OutputVariable* message is same as *Source*, the message will be overwritten.
- Content-Type for output message will be set to an XML content type.

The JSONtoXML policy can be used to convert a JSON payload to XML.

The source must be in the payload of an HTTP message; the policy cannot be used to convert a value stored in a normal flow variable.

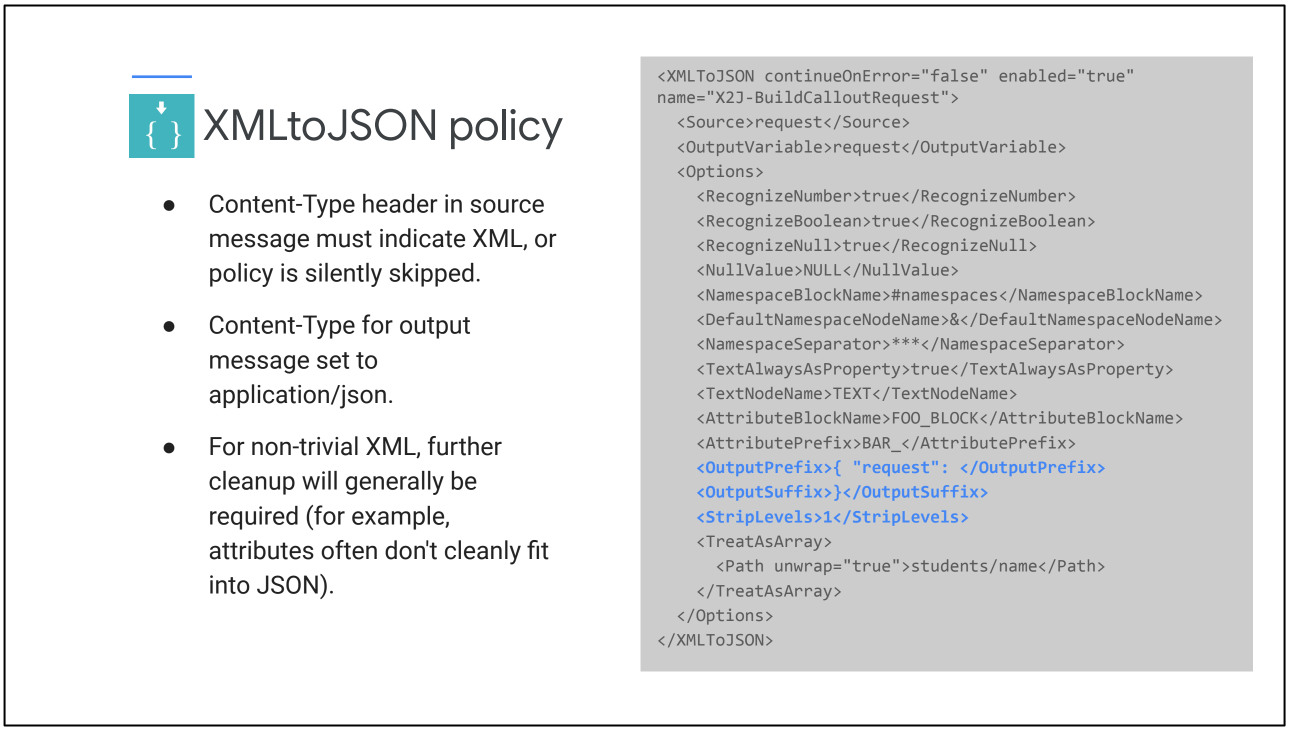
Like all JSON and XML policies, the Content-Type header for the source must be set correctly or the policy will be silently skipped.

The output variable is another message, not a string.

The Content-Type header will be set to an XML type when conversion is successful.

If the source and output variable are in the same location, the message payload and Content-Type header will be overwritten.

One of the many configurable elements for the policy is the ObjectElementRootName. Because JSON does not have a root element, the object element root name for the resulting XML should be specified here.



XMLtoJSON policy

- Content-Type header in source message must indicate XML, or policy is silently skipped.
- Content-Type for output message set to application/json.
- For non-trivial XML, further cleanup will generally be required (for example, attributes often don't cleanly fit into JSON).

```
<XMLToJSON continueOnError="false" enabled="true"
name="X2J-BuildCalloutRequest">
  <Source>request</Source>
  <OutputVariable>request</OutputVariable>
  <Options>
    <RecognizeNumber>true</RecognizeNumber>
    <RecognizeBoolean>true</RecognizeBoolean>
    <RecognizeNull>true</RecognizeNull>
    <NullValue>NULL</NullValue>
    <NamespaceBlockName>#namespaces</NamespaceBlockName>
    <DefaultNamespaceNodeName>&</DefaultNamespaceNodeName>
    <NamespaceSeparator>***</NamespaceSeparator>
    <TextAlwaysAsProperty>true</TextAlwaysAsProperty>
    <TextNodeName>TEXT</TextNodeName>
    <AttributeBlockName>FOO_BLOCK</AttributeBlockName>
    <AttributePrefix>BAR_</AttributePrefix>
    <OutputPrefix>{ "request": </OutputPrefix>
    <OutputSuffix>}</OutputSuffix>
    <StripLevels>1</StripLevels>
    <TreatAsArray>
      <Path unwrap="true">students/name</Path>
    </TreatAsArray>
  </Options>
</XMLToJSON>
```

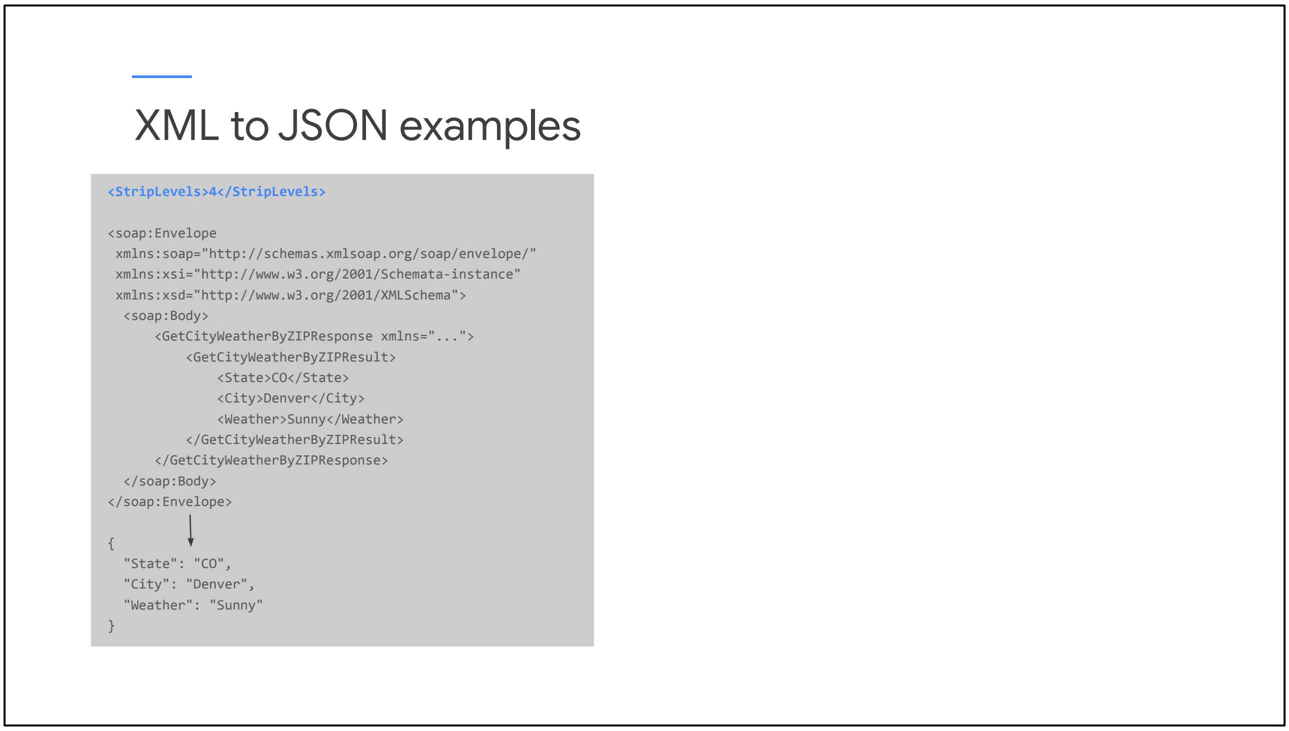
The XMLtoJSON policy converts an XML payload to JSON.

This policy operates much like the JSONtoXML policy, acting on HTTP messages and using the content type header.

The OutputPrefix and OutputSuffix elements can be used to control the root level of the resulting JSON.

The StripLevels element can be used to remove the outermost levels of the XML before converting. Stripping the first level off can be used to remove the XML root element.

When converting XML to JSON, note that you may need to do further cleanup on the JSON to convert data types or handle the conversion of attributes into JSON.



XML to JSON examples

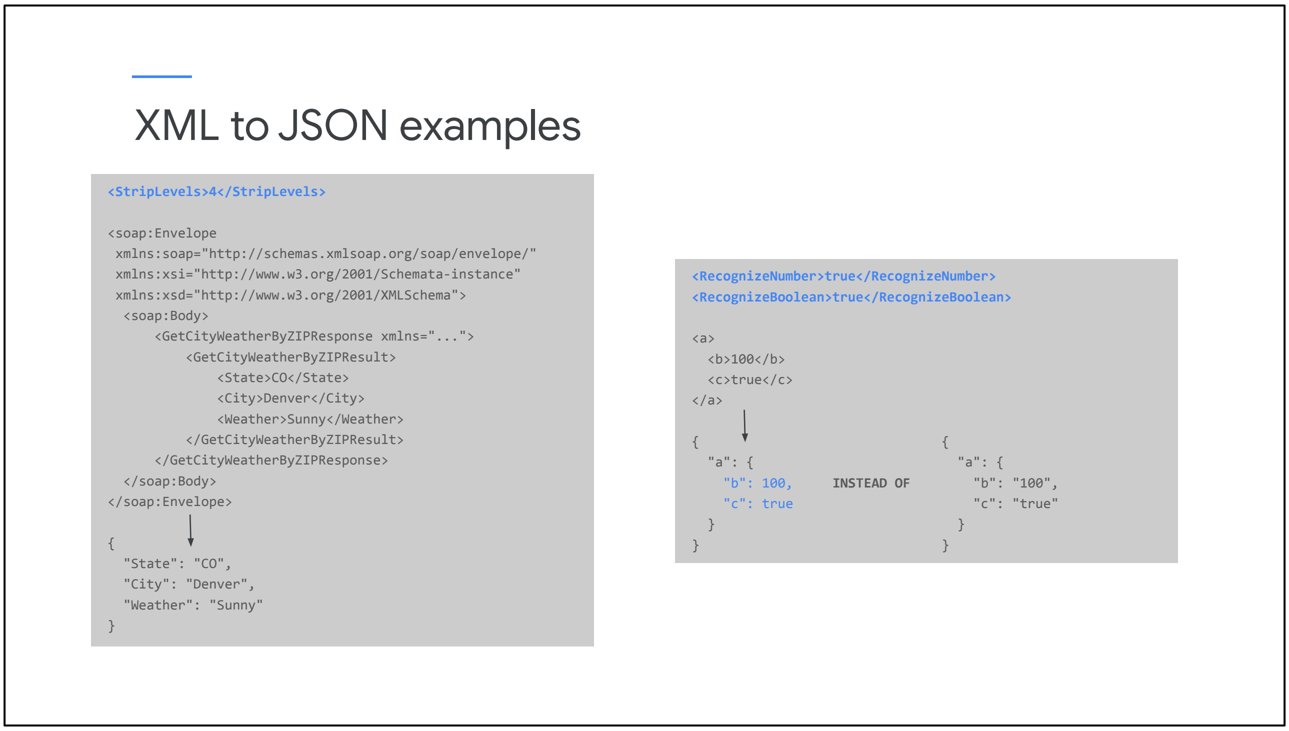
```
<StripLevels>4</StripLevels>

<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsi="http://www.w3.org/2001/Schemata-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soap:Body>
      <GetCityWeatherByZIPResponse xmlns="...">
          <GetCityWeatherByZIPResult>
              <State>CO</State>
              <City>Denver</City>
              <Weather>Sunny</Weather>
          </GetCityWeatherByZIPResult>
      </GetCityWeatherByZIPResponse>
  </soap:Body>
</soap:Envelope>

{
  "State": "CO",
  "City": "Denver",
  "Weather": "Sunny"
}
```

XML to JSON tends to be the more difficult conversion, due to lack of data types and because XML tends to be more verbose. Let's look at a couple of examples.

When you have a SOAP message, specifically a SOAP envelope, there are many layers of XML that serve no purpose when converting to a RESTful JSON structure.

The StripLevels element can be used to remove the SOAP envelope from the message. In this example, the outer 4 levels are removed, from the SOAP envelope root element to the GetCityWeatherByZIPResult element.

The resulting JSON is very clean.



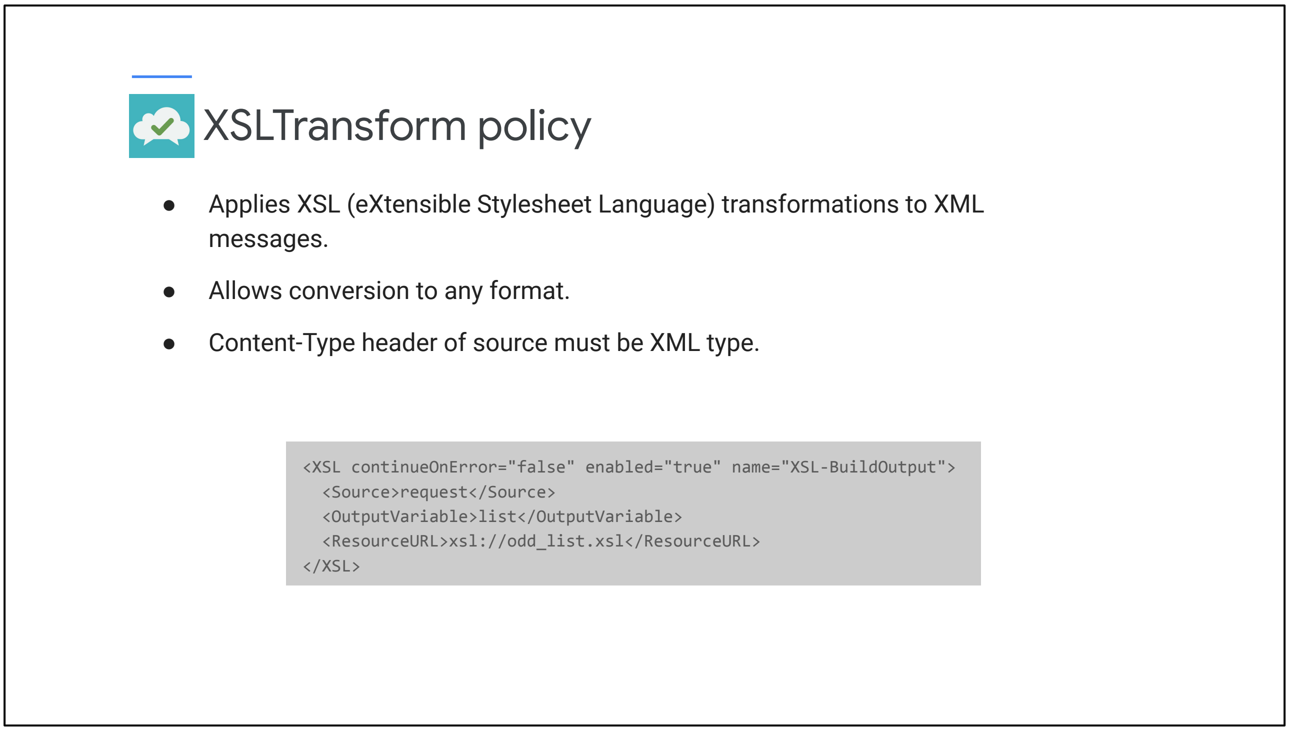
XML to JSON examples

```
<StripLevels>4</StripLevels>

<soap:Envelope
 xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsi="http://www.w3.org/2001/Schemata-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <soap:Body>
      <GetCityWeatherByZIPResponse xmlns="...">
          <GetCityWeatherByZIPResult>
              <State>CO</State>
              <City>Denver</City>
              <Weather>Sunny</Weather>
          </GetCityWeatherByZIPResult>
      </GetCityWeatherByZIPResponse>
  </soap:Body>
</soap:Envelope>
          ↓
{
  "State": "CO",
  "City": "Denver",
  "Weather": "Sunny"
}
```

```
<RecognizeNumber>true</RecognizeNumber>
<RecognizeBoolean>true</RecognizeBoolean>

<a>
  <b>100</b>
  <c>true</c>
</a>
      ↓
{                                {
  "a": {                           "a": {
    "b": 100,      INSTEAD OF        "b": "100",
    "c": true                        "c": "true"
  }                                }
}                                }
```

The second example uses the recognize number and recognize boolean elements to convert into those types when the data looks like a number or boolean.

By default, the XMLToJSON policy will always convert values to strings. In this case, telling the policy to look for fields that look like numbers or booleans will result in cleaner JSON. You may still need to clean up data types, but this can help set the correct type in most cases.



XSLTransform policy

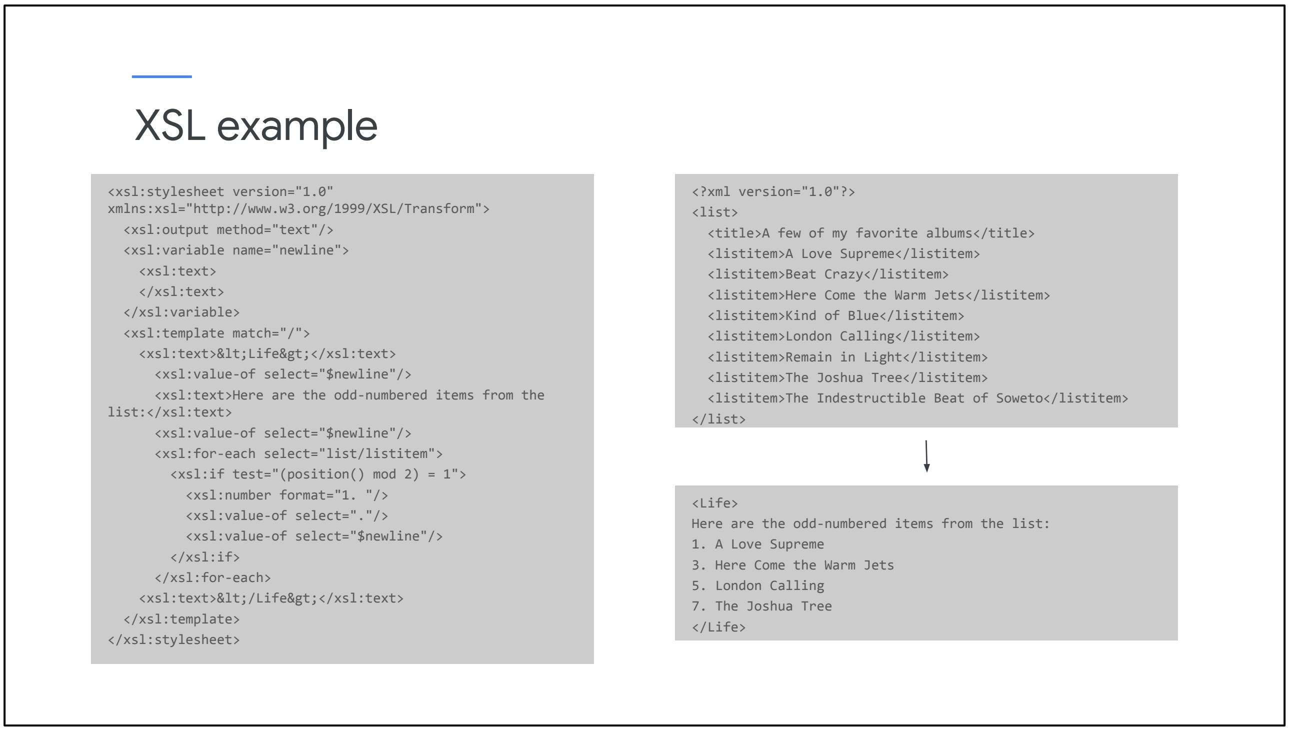
- Applies XSL (eXtensible Stylesheet Language) transformations to XML messages.
- Allows conversion to any format.
- Content-Type header of source must be XML type.

```
<XSL continueOnError="false" enabled="true" name="XSL-BuildOutput">
  <Source>request</Source>
  <OutputVariable>list</OutputVariable>
  <ResourceURL>xsl://odd_list.xsl</ResourceURL>
</XSL>
```

The XSLTransform policy can be used to convert XML documents into any format.

XSL stands for Extensible Stylesheet Language, which describes a set of rules for converting an XML document into another document.

The ResourceURL is a reference to the XSL stylesheet stored in the same proxy. The stylesheet must be stored as code in the proxy.



XSL example

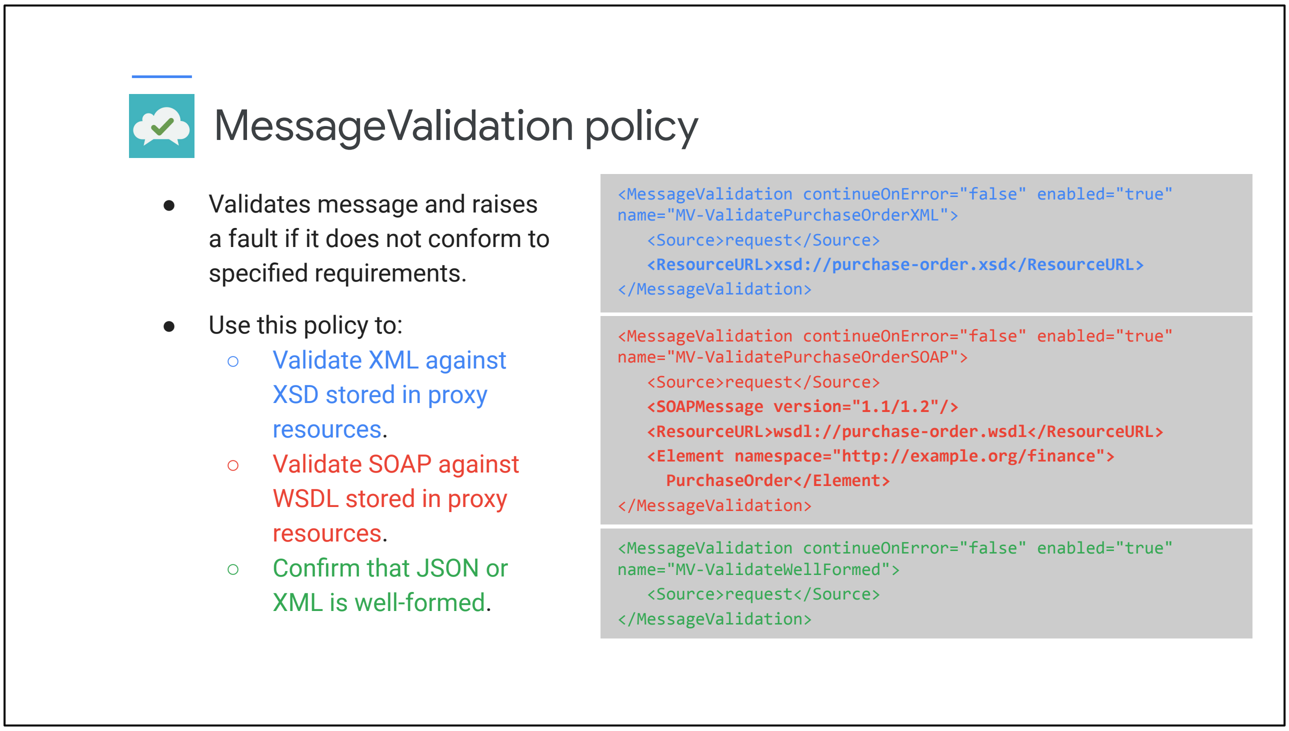
```
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
  <xsl:variable name="newline">
    <xsl:text>
    </xsl:text>
  </xsl:variable>
  <xsl:template match="/">
    <xsl:text>&lt;Life&gt;</xsl:text>
      <xsl:value-of select="$newline"/>
      <xsl:text>Here are the odd-numbered items from the
list:</xsl:text>
      <xsl:value-of select="$newline"/>
      <xsl:for-each select="list/listitem">
        <xsl:if test="(position() mod 2) = 1">
          <xsl:number format="1. "/>
          <xsl:value-of select="."/>
          <xsl:value-of select="$newline"/>
        </xsl:if>
      </xsl:for-each>
    <xsl:text>&lt;/Life&gt;</xsl:text>
  </xsl:template>
</xsl:stylesheet>
```

```
<?xml version="1.0"?>
<list>
  <title>A few of my favorite albums</title>
  <listitem>A Love Supreme</listitem>
  <listitem>Beat Crazy</listitem>
  <listitem>Here Come the Warm Jets</listitem>
  <listitem>Kind of Blue</listitem>
  <listitem>London Calling</listitem>
  <listitem>Remain in Light</listitem>
  <listitem>The Joshua Tree</listitem>
  <listitem>The Indestructible Beat of Soweto</listitem>
</list>
```

```
<Life>
Here are the odd-numbered items from the list:
1. A Love Supreme
3. Here Come the Warm Jets
5. London Calling
7. The Joshua Tree
</Life>
```

Here is an example of an XSL stylesheet on the left, the source XML document on the upper right, and some text that has been created as the output on the lower right.

XSL can be a powerful tool for reformatting complex XML.



MessageValidation policy

- Validates message and raises a fault if it does not conform to specified requirements.
- Use this policy to:
  - Validate XML against XSD stored in proxy resources.
  - Validate SOAP against WSDL stored in proxy resources.
  - Confirm that JSON or XML is well-formed.

```
<MessageValidation continueOnError="false" enabled="true"
name="MV-ValidatePurchaseOrderXML">
   <Source>request</Source>
   <ResourceURL>xsd://purchase-order.xsd</ResourceURL>
</MessageValidation>
```

```
<MessageValidation continueOnError="false" enabled="true"
name="MV-ValidatePurchaseOrderSOAP">
   <Source>request</Source>
   <SOAPMessage version="1.1/1.2"/>
   <ResourceURL>wsdl://purchase-order.wsdl</ResourceURL>
   <Element namespace="http://example.org/finance">
     PurchaseOrder</Element>
</MessageValidation>
```

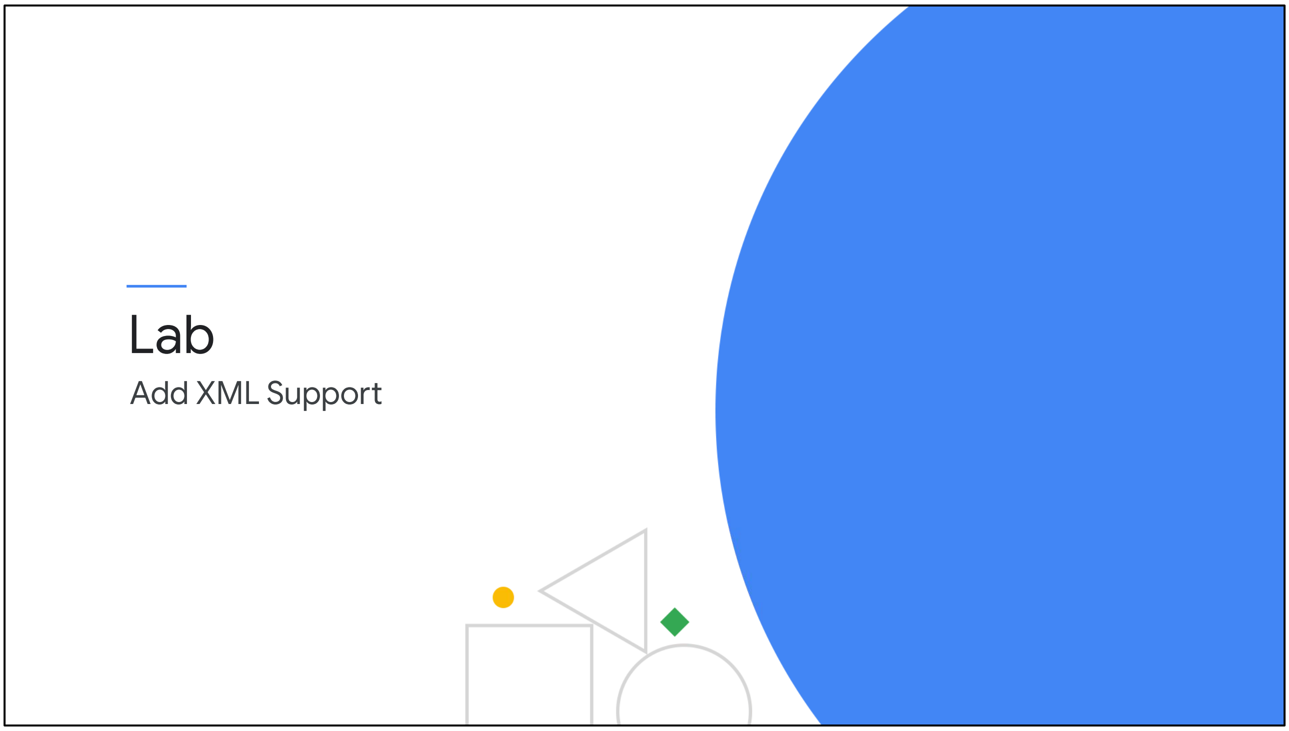
```
<MessageValidation continueOnError="false" enabled="true"
name="MV-ValidateWellFormed">
   <Source>request</Source>
</MessageValidation>
```

The MessageValidation policy, sometimes known as the SOAP message validation policy, can validate a message and raise a fault if the message does not match the specified configuration.

The policy can be used to validate XML against XML Schema Definition, or XSD, files.

It can also validate SOAP against a WSDL file, or simply confirm that the JSON or XML is well-formed.

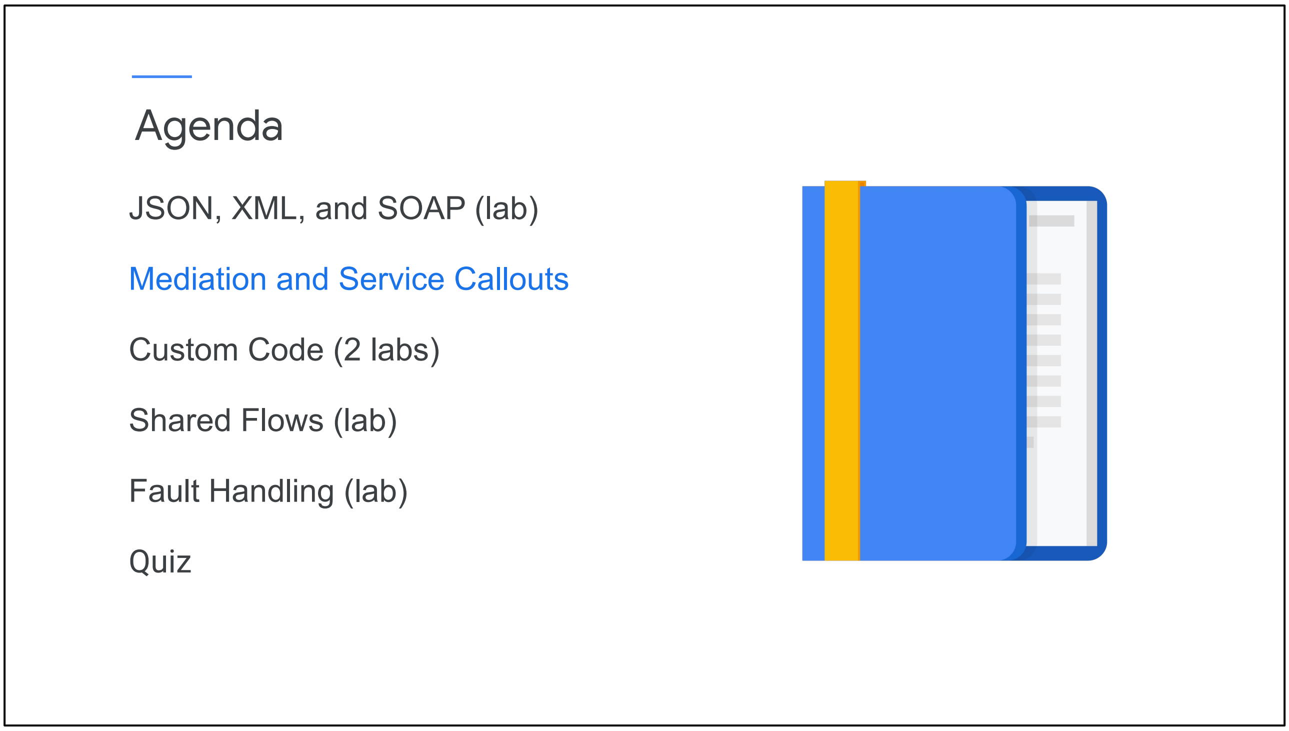
The Content-Type header must be set to the correct type for validation to occur, and, like the XSL policy, XSD and WSDL files must be stored in the proxy code.



Lab

Add XML Support

In this lab you add XML support to your retail API. You will use the JSON to XML policy to convert the response to XML when the request's accept header is used to request an XML response.



Agenda

JSON, XML, and SOAP (lab)

Mediation and Service Callouts

Custom Code (2 labs)

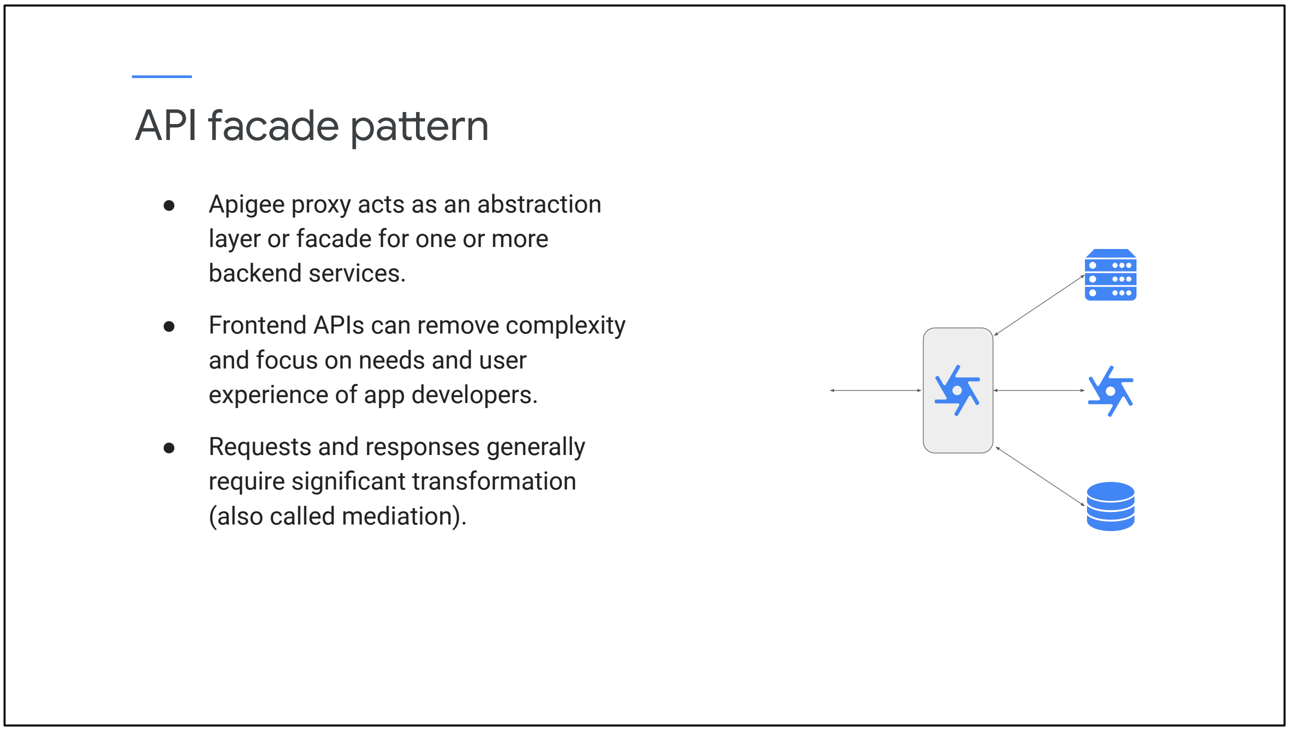
Shared Flows (lab)

Fault Handling (lab)

Quiz

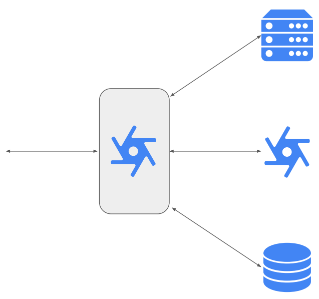
This section will discuss patterns of transformation and mediation used in API proxies.

We'll also learn about policies that can help us transform payloads and make calls to external services.



API facade pattern

- Apigee proxy acts as an abstraction layer or facade for one or more backend services.
- Frontend APIs can remove complexity and focus on needs and user experience of app developers.
- Requests and responses generally require significant transformation (also called mediation).



API First development specifies that we should design our APIs with our app developers in mind.

We often talk about an Apigee API proxy being an abstraction or facade layer for your backend services.

This facade layer allows us to build APIs that allow app developers to leverage backend services in a much simpler way than if they were called directly. The app developer can use the clean API that you've created in your proxy and allow the proxy to deal with any backend complexity.

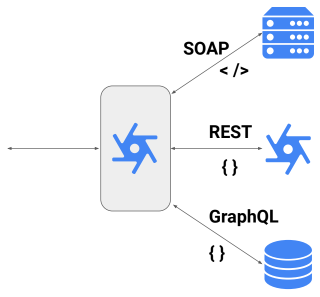
You can even create multiple APIs for the same backend services, if your app developers have different needs.

To accomplish this, you need to be able to transform requests and responses and often call multiple services during a single API call.

We call this ability to mash up API calls and payloads "mediation."

Mediation pattern: Format conversion

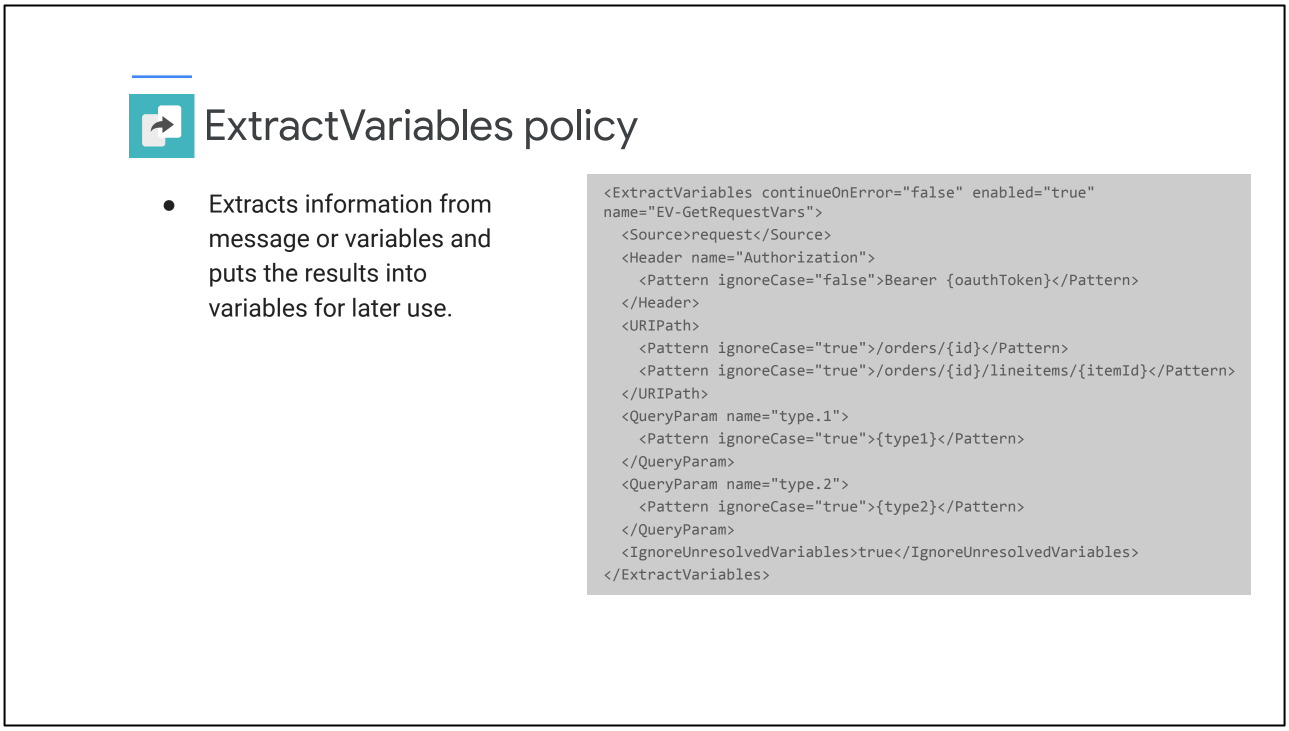
- Backend services often use different content types or payload formats from those in your API.
- ExtractVariables and AssignMessage policies can extract data from original message and build a new message.



One mediation pattern is format conversion.

Backend or third-party services usually use content types or payload formats that differ from those in your API. Even if your API and your backend services are using JSON, for example, the payload itself often must be rewritten using a different structure.

The ExtractVariables policy can extract data from your message, and the AssignMessage policy can be used to build a new message using the extracted values.



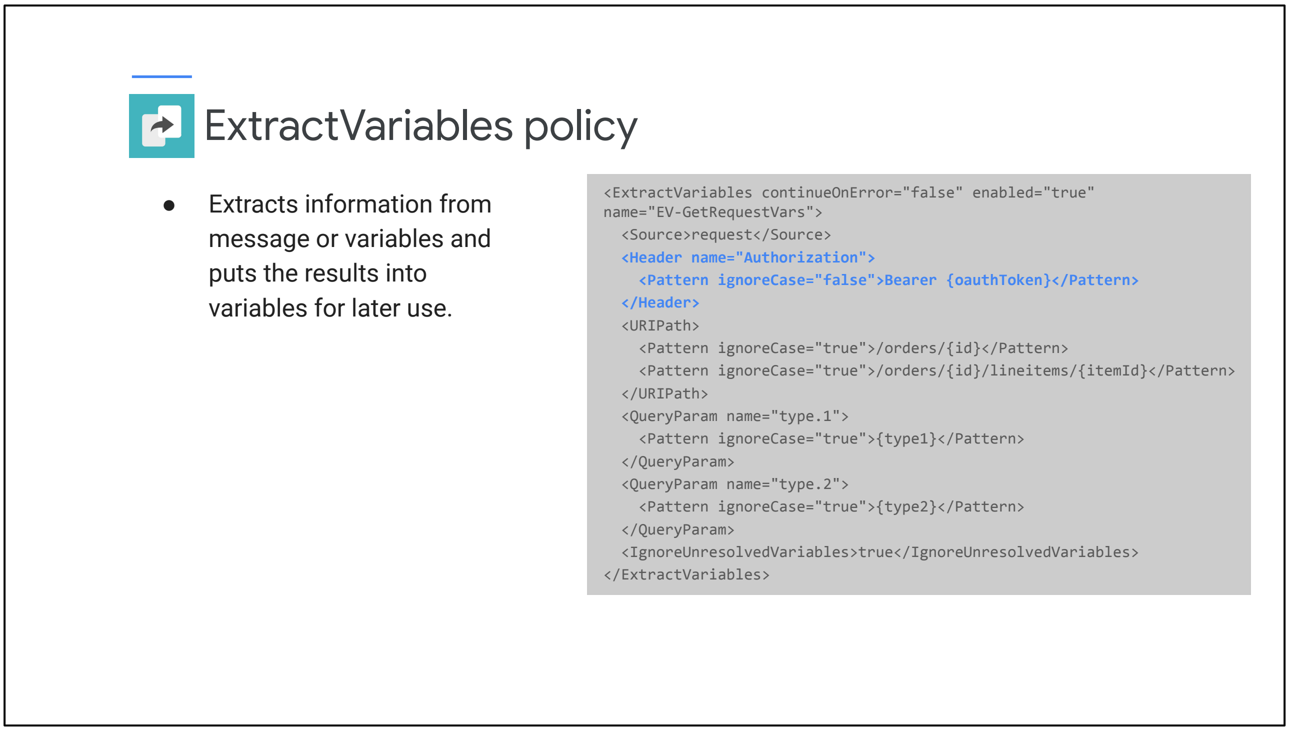
ExtractVariables policy

- Extracts information from message or variables and puts the results into variables for later use.

```
<ExtractVariables continueOnError="false" enabled="true"
name="EV-GetRequestVars">
  <Source>request</Source>
  <Header name="Authorization">
    <Pattern ignoreCase="false">Bearer {oauthToken}</Pattern>
  </Header>
  <URIPath>
    <Pattern ignoreCase="true">/orders/{id}</Pattern>
    <Pattern ignoreCase="true">/orders/{id}/lineitems/{itemId}</Pattern>
  </URIPath>
  <QueryParam name="type.1">
    <Pattern ignoreCase="true">{type1}</Pattern>
  </QueryParam>
  <QueryParam name="type.2">
    <Pattern ignoreCase="true">{type2}</Pattern>
  </QueryParam>
  <IgnoreUnresolvedVariables>true</IgnoreUnresolvedVariables>
</ExtractVariables>
```

The ExtractVariables policy extracts information from messages or variables and stores the results in variables for use in your proxy.

Variables can be extracted from headers, query parameters, the URI path, or any variable.



ExtractVariables policy

- Extracts information from message or variables and puts the results into variables for later use.

```
<ExtractVariables continueOnError="false" enabled="true"
name="EV-GetRequestVars">
  <Source>request</Source>
  <Header name="Authorization">
    <Pattern ignoreCase="false">Bearer {oauthToken}</Pattern>
  </Header>
  <URIPath>
    <Pattern ignoreCase="true">/orders/{id}</Pattern>
    <Pattern ignoreCase="true">/orders/{id}/lineitems/{itemId}</Pattern>
  </URIPath>
  <QueryParam name="type.1">
    <Pattern ignoreCase="true">{type1}</Pattern>
  </QueryParam>
  <QueryParam name="type.2">
    <Pattern ignoreCase="true">{type2}</Pattern>
  </QueryParam>
  <IgnoreUnresolvedVariables>true</IgnoreUnresolvedVariables>
</ExtractVariables>
```

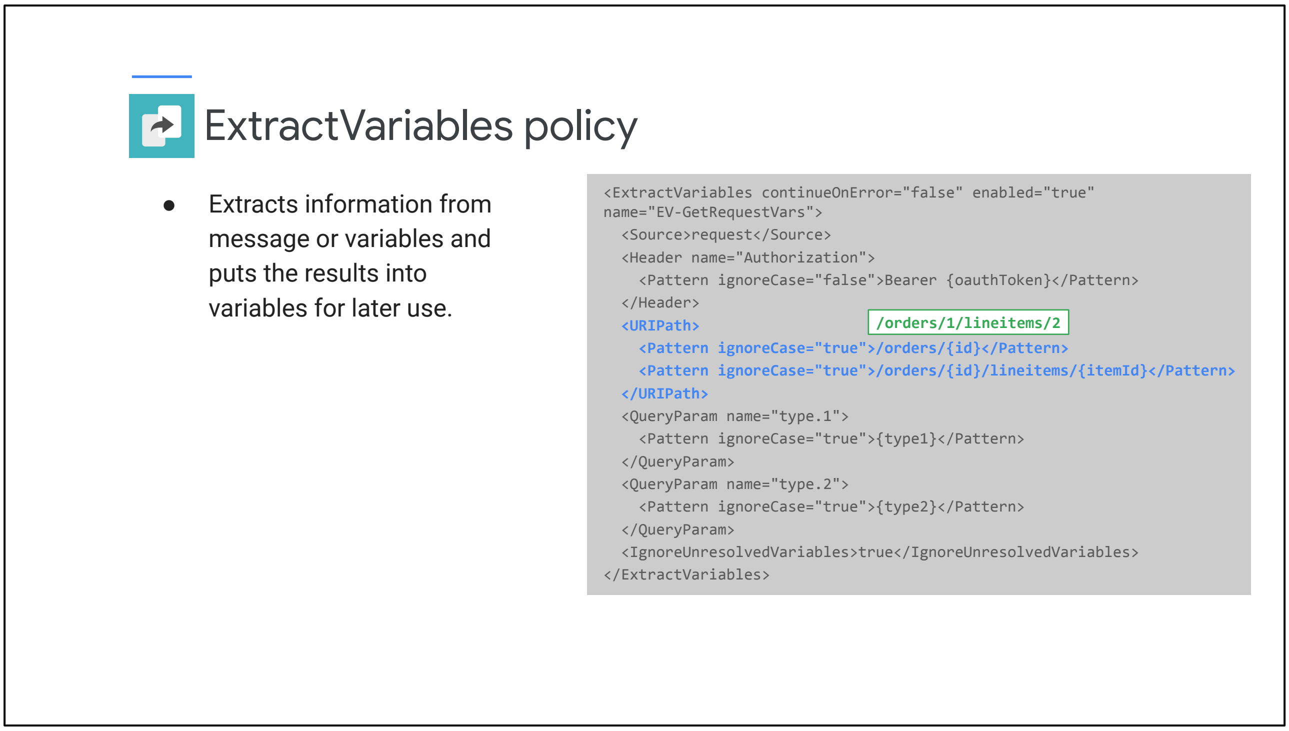
In this example, we are extracting information from the Authorization header.

We look for a pattern starting with Bearer and a space.

The ignore case attribute is set to false, so Bearer must be spelled with a capital B.

The curly braces indicate that OAuth token is the variable name, and the resulting variable will contain all of the header value that follows Bearer and the space.

If there is no Authorization header, or the value of the header does not start with Bearer space, the OAuth token variable will not be set.



ExtractVariables policy

- Extracts information from message or variables and puts the results into variables for later use.

```
<ExtractVariables continueOnError="false" enabled="true"
name="EV-GetRequestVars">
  <Source>request</Source>
  <Header name="Authorization">
    <Pattern ignoreCase="false">Bearer {oauthToken}</Pattern>
  </Header>
  <URIPath>                     /orders/1/lineitems/2
    <Pattern ignoreCase="true">/orders/{id}</Pattern>
    <Pattern ignoreCase="true">/orders/{id}/lineitems/{itemId}</Pattern>
  </URIPath>
  <QueryParam name="type.1">
    <Pattern ignoreCase="true">{type1}</Pattern>
  </QueryParam>
  <QueryParam name="type.2">
    <Pattern ignoreCase="true">{type2}</Pattern>
  </QueryParam>
  <IgnoreUnresolvedVariables>true</IgnoreUnresolvedVariables>
</ExtractVariables>
```

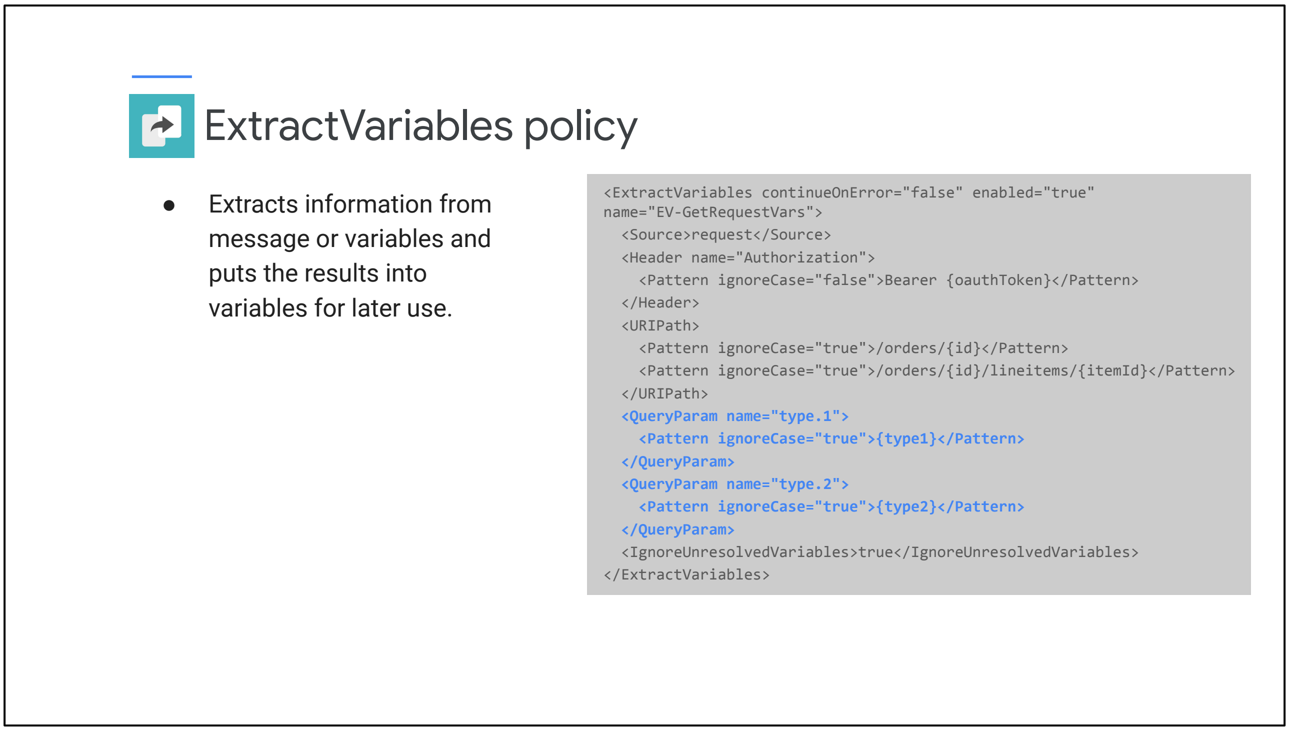
When extracting information from a particular location, you can also use multiple patterns to support multiple formats.

They are evaluated in order, and the first match is used.

URI paths are a special case: variable matching cannot span multiple path segments.

In the example, if the proxy pathsuffix for the request is /orders/1/lineitems/2, the id variable will contain 1 and the itemId variable will contain 2.

The first pattern wouldn't be a match, because the id variable could not match three path segments, 1/lineitems/2.



ExtractVariables policy

- Extracts information from message or variables and puts the results into variables for later use.

```
<ExtractVariables continueOnError="false" enabled="true"
name="EV-GetRequestVars">
  <Source>request</Source>
  <Header name="Authorization">
    <Pattern ignoreCase="false">Bearer {oauthToken}</Pattern>
  </Header>
  <URIPath>
    <Pattern ignoreCase="true">/orders/{id}</Pattern>
    <Pattern ignoreCase="true">/orders/{id}/lineitems/{itemId}</Pattern>
  </URIPath>
  <QueryParam name="type.1">
    <Pattern ignoreCase="true">{type1}</Pattern>
  </QueryParam>
  <QueryParam name="type.2">
    <Pattern ignoreCase="true">{type2}</Pattern>
  </QueryParam>
  <IgnoreUnresolvedVariables>true</IgnoreUnresolvedVariables>
</ExtractVariables>
```

You can even match multiple headers or query parameters that have the same name.

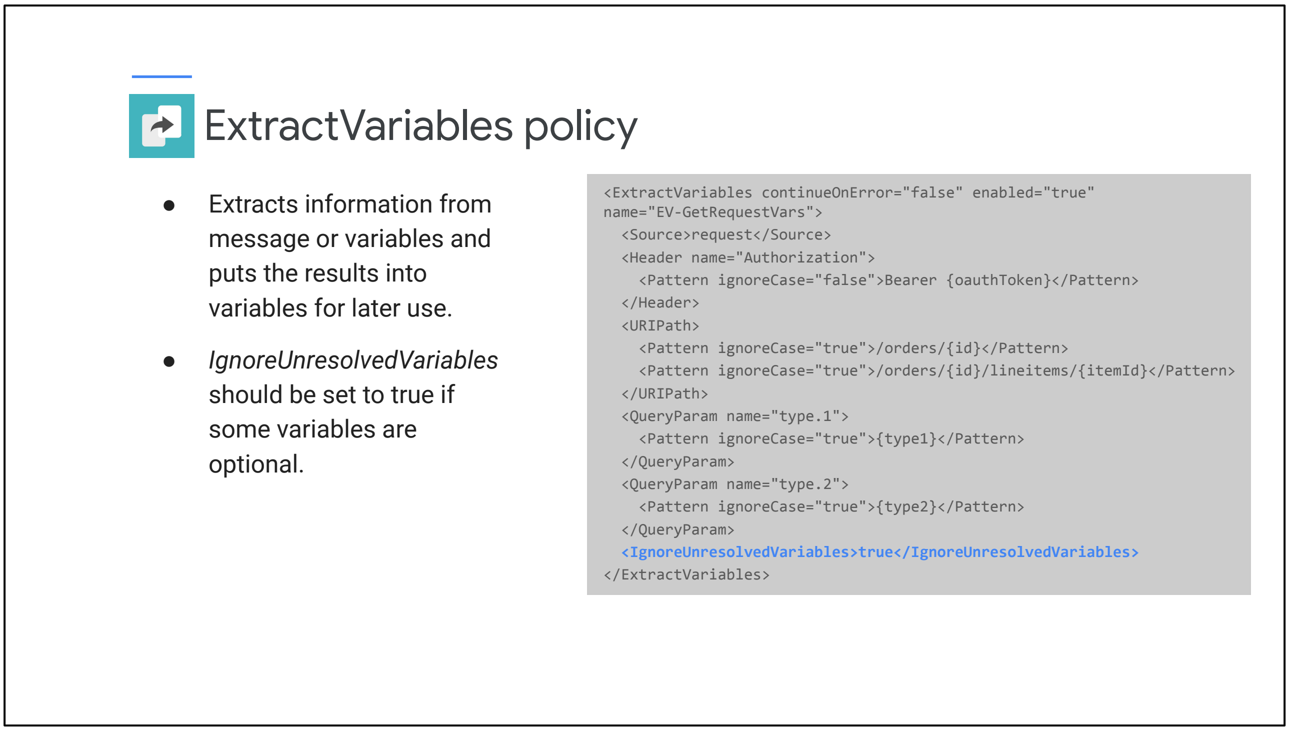
By default, if you searched for a query parameter named type, you could only find the first occurrence of that query parameter.

You can use indexing to select a particular occurrence of the header.

Indexes are specified using the dot followed by the one-based index number.

In the example, searching for a query parameter named type.1, or just type, would retrieve the first matching query parameter into the type1 variable.

The second one, using type.2, would be extracted into the type2 variable.



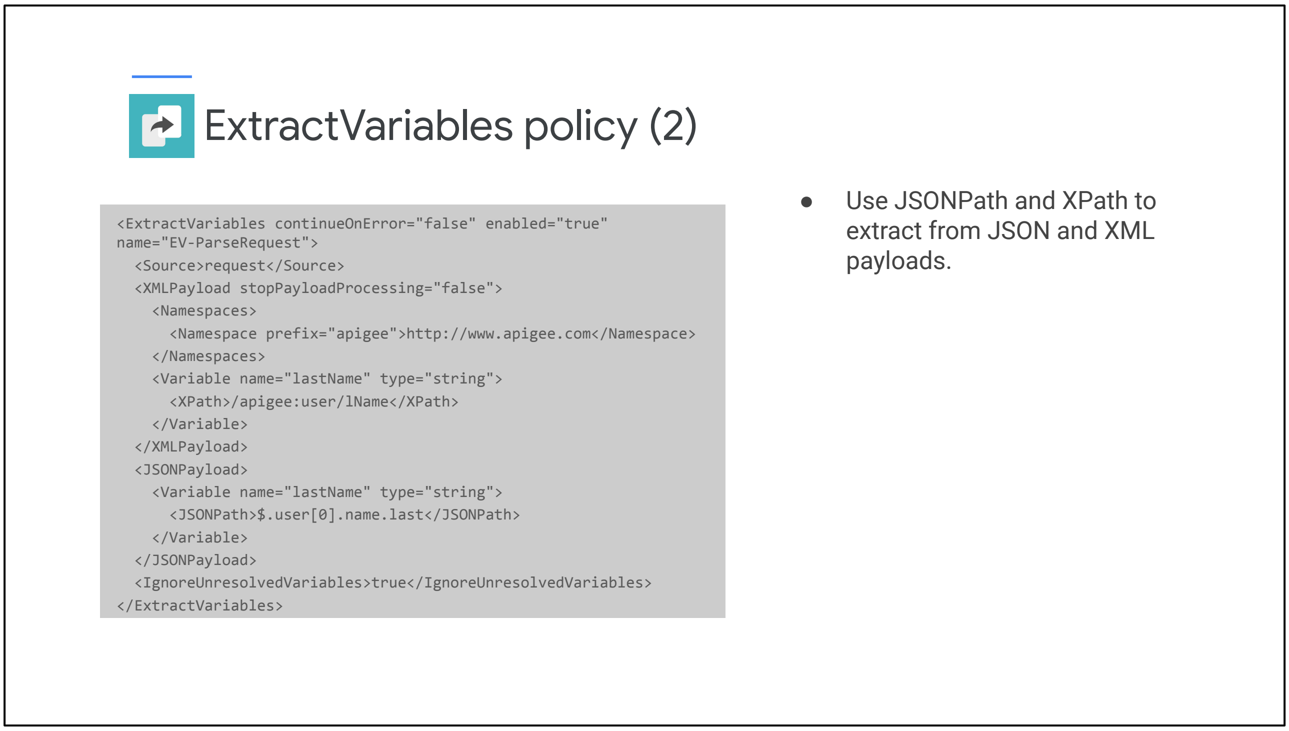
ExtractVariables policy

- Extracts information from message or variables and puts the results into variables for later use.
- *IgnoreUnresolvedVariables* should be set to true if some variables are optional.

```
<ExtractVariables continueOnError="false" enabled="true"
name="EV-GetRequestVars">
  <Source>request</Source>
  <Header name="Authorization">
    <Pattern ignoreCase="false">Bearer {oauthToken}</Pattern>
  </Header>
  <URIPath>
    <Pattern ignoreCase="true">/orders/{id}</Pattern>
    <Pattern ignoreCase="true">/orders/{id}/lineitems/{itemId}</Pattern>
  </URIPath>
  <QueryParam name="type.1">
    <Pattern ignoreCase="true">{type1}</Pattern>
  </QueryParam>
  <QueryParam name="type.2">
    <Pattern ignoreCase="true">{type2}</Pattern>
  </QueryParam>
  <IgnoreUnresolvedVariables>true</IgnoreUnresolvedVariables>
</ExtractVariables>
```

If any of the variables are optional, you should set IgnoreResolvedVariables to true.

Otherwise, if one or more matches are not found, the ExtractVariables policy will raise an error.

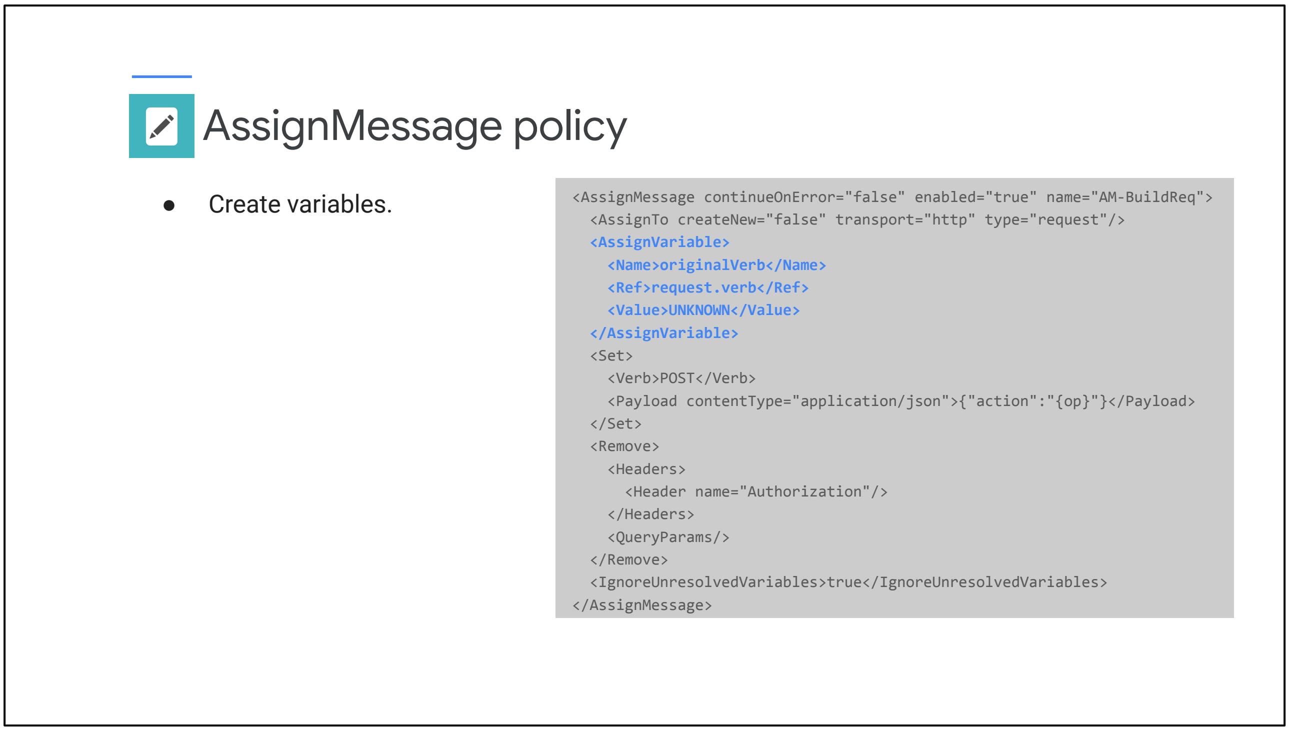


ExtractVariables policy (2)

```
<ExtractVariables continueOnError="false" enabled="true"
name="EV-ParseRequest">
  <Source>request</Source>
  <XMLPayload stopPayloadProcessing="false">
    <Namespaces>
      <Namespace prefix="apigee">http://www.apigee.com</Namespace>
    </Namespaces>
    <Variable name="lastName" type="string">
      <XPath>/apigee:user/lName</XPath>
    </Variable>
  </XMLPayload>
  <JSONPayload>
    <Variable name="lastName" type="string">
      <JSONPath>$.user[0].name.last</JSONPath>
    </Variable>
  </JSONPayload>
  <IgnoreUnresolvedVariables>true</IgnoreUnresolvedVariables>
</ExtractVariables>
```

- Use JSONPath and XPath to extract from JSON and XML payloads.

The ExtractVariables policy can also extract data from a JSON or XML payload into variables using JSONPath or XPath.



AssignMessage policy

- Create variables.

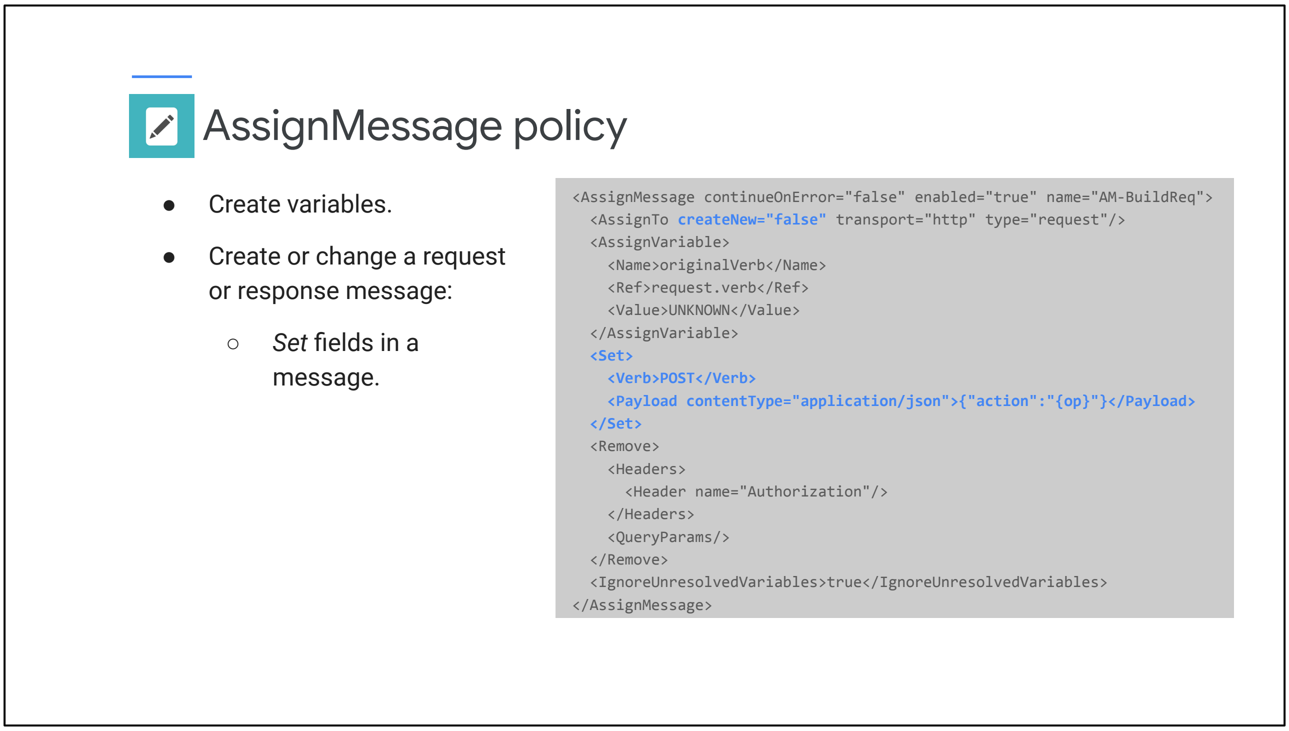
```
<AssignMessage continueOnError="false" enabled="true" name="AM-BuildReq">
  <AssignTo createNew="false" transport="http" type="request"/>
  <AssignVariable>
    <Name>originalVerb</Name>
    <Ref>request.verb</Ref>
    <Value>UNKNOWN</Value>
  </AssignVariable>
  <Set>
    <Verb>POST</Verb>
    <Payload contentType="application/json">{"action":"{op}"}</Payload>
  </Set>
  <Remove>
    <Headers>
      <Header name="Authorization"/>
    </Headers>
    <QueryParams/>
  </Remove>
  <IgnoreUnresolvedVariables>true</IgnoreUnresolvedVariables>
</AssignMessage>
```

The AssignMessage policy creates variables and HTTP messages.

Variables can be created by reference or by value.

In the example, the name of the variable is originalVerb. It takes the value of request.verb, if it exists.

Request.verb would typically exist, but if it did not, or if the value of the request.verb was null, the value "UNKNOWN" would be assigned.



AssignMessage policy

- Create variables.
- Create or change a request or response message:
    - *Set* fields in a message.

```
<AssignMessage continueOnError="false" enabled="true" name="AM-BuildReq">
  <AssignTo createNew="false" transport="http" type="request"/>
  <AssignVariable>
    <Name>originalVerb</Name>
    <Ref>request.verb</Ref>
    <Value>UNKNOWN</Value>
  </AssignVariable>
  <Set>
    <Verb>POST</Verb>
    <Payload contentType="application/json">{"action":"{op}"}</Payload>
  </Set>
  <Remove>
    <Headers>
      <Header name="Authorization"/>
    </Headers>
    <QueryParams/>
  </Remove>
  <IgnoreUnresolvedVariables>true</IgnoreUnresolvedVariables>
</AssignMessage>
```

The policy can also modify existing messages or create a new one.

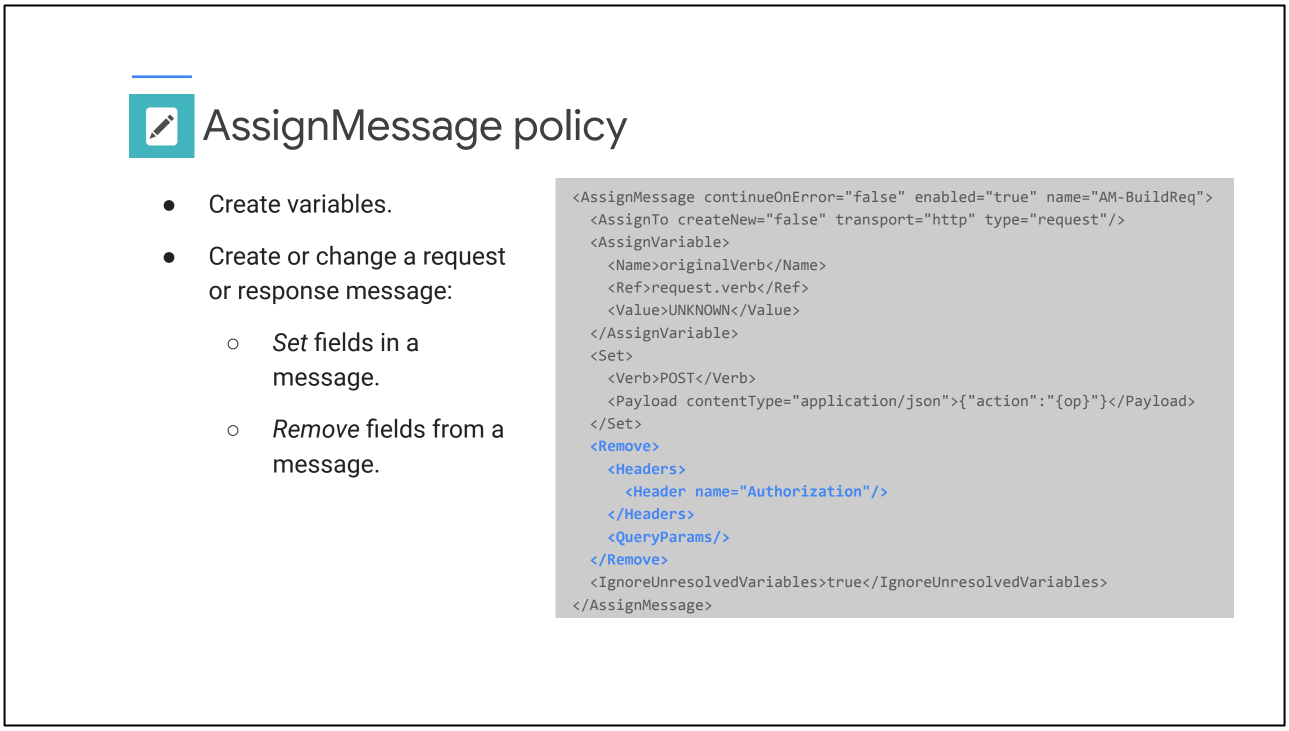
The createNew attribute of the AssignTo element specifies whether to update an existing variable or create a new one. In this case, createNew is false, so we are editing in place.

The Set element allows you to set fields in a message. In the example, we are setting the verb to POST and creating a JSON payload that uses the op variable as the action.

Note that the operations in the policy are executed in order.

The AssignVariable element saved the value of the request verb, and then the request verb was overwritten by the Set element.

You can also set headers, query and form parameters, status codes, and the request path.



AssignMessage policy

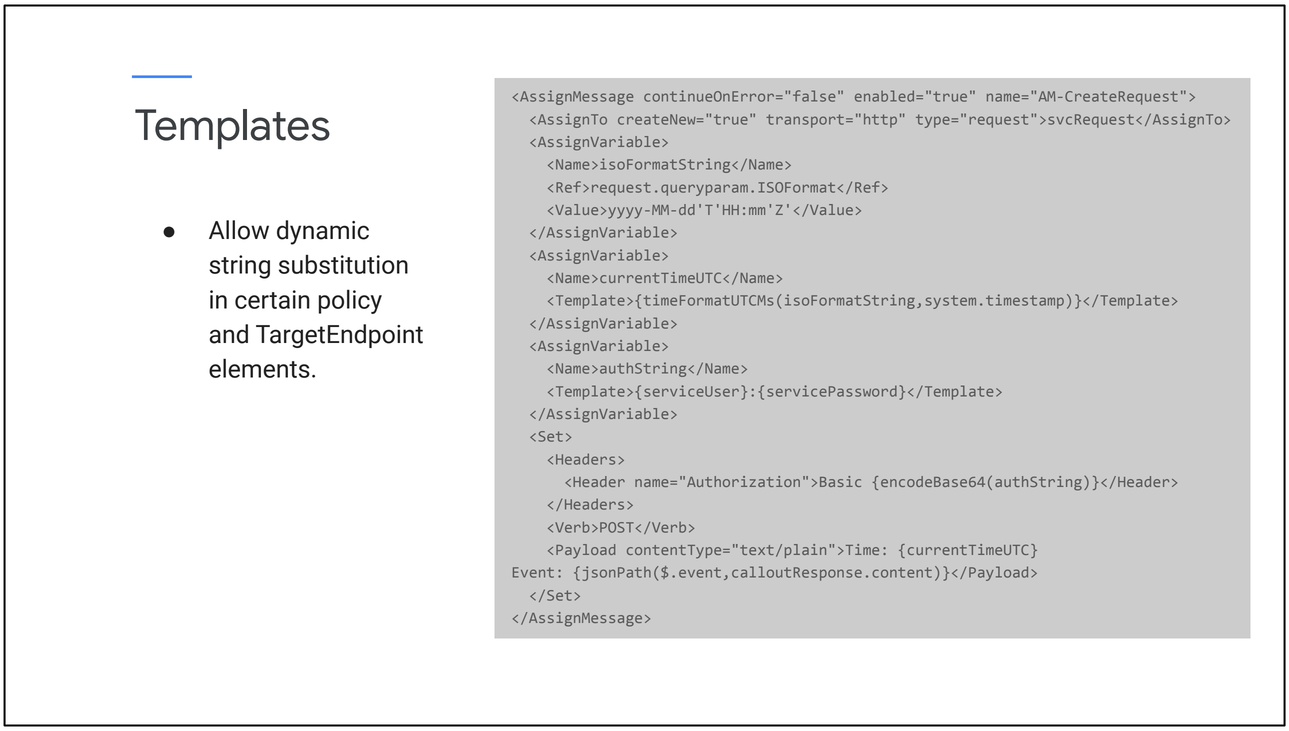
- Create variables.
- Create or change a request or response message:
  - *Set* fields in a message.
  - *Remove* fields from a message.

```
<AssignMessage continueOnError="false" enabled="true" name="AM-BuildReq">
  <AssignTo createNew="false" transport="http" type="request"/>
  <AssignVariable>
    <Name>originalVerb</Name>
    <Ref>request.verb</Ref>
    <Value>UNKNOWN</Value>
  </AssignVariable>
  <Set>
    <Verb>POST</Verb>
    <Payload contentType="application/json">{"action":"{op}"}</Payload>
  </Set>
  <Remove>
    <Headers>
      <Header name="Authorization"/>
    </Headers>
    <QueryParams/>
  </Remove>
  <IgnoreUnresolvedVariables>true</IgnoreUnresolvedVariables>
</AssignMessage>
```

The policy will also remove fields from the message.

The specified header, Authorization, is the only header that will be removed.

The query params element has no specified query parameter, so all query parameters are removed.



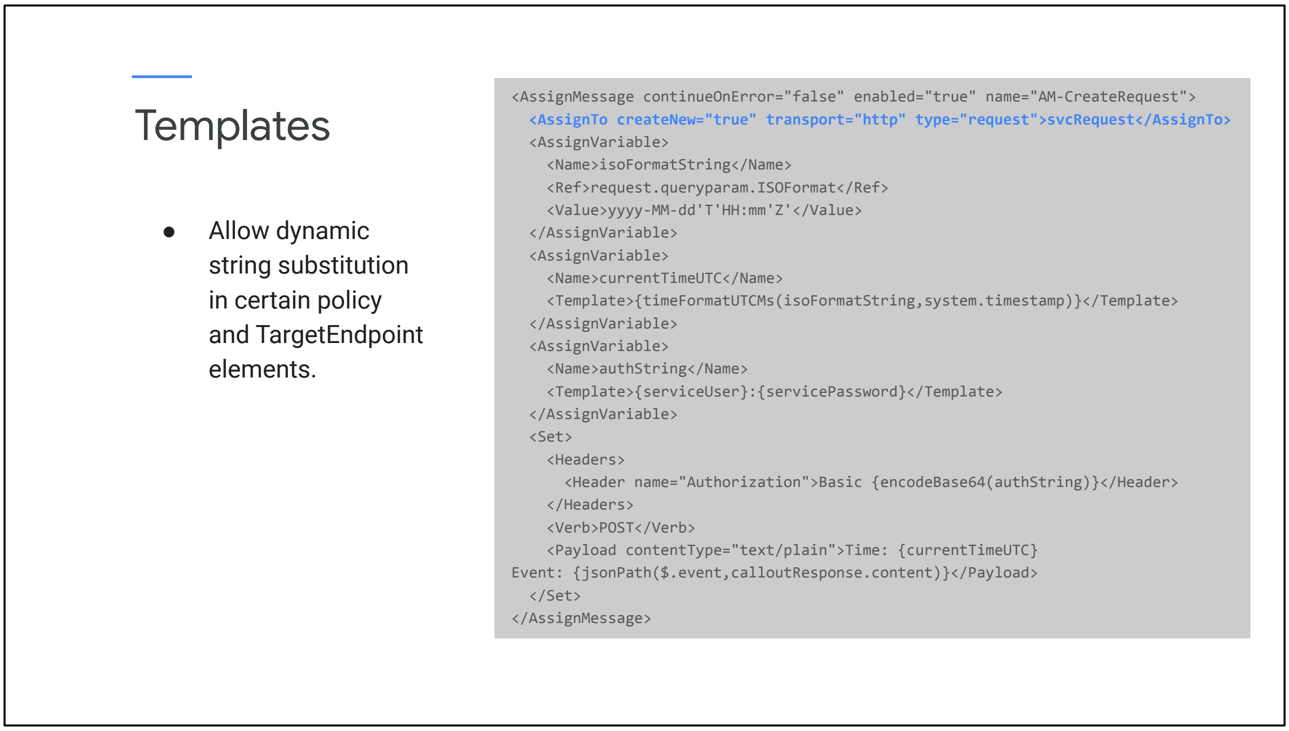
Templates

- Allow dynamic string substitution in certain policy and TargetEndpoint elements.

```
<AssignMessage continueOnError="false" enabled="true" name="AM-CreateRequest">
  <AssignTo createNew="true" transport="http" type="request">svcRequest</AssignTo>
  <AssignVariable>
    <Name>isoFormatString</Name>
    <Ref>request.queryparam.ISOFormat</Ref>
    <Value>yyyy-MM-dd'T'HH:mm'Z'</Value>
  </AssignVariable>
  <AssignVariable>
    <Name>currentTimeUTC</Name>
    <Template>{timeFormatUTCMs(isoFormatString,system.timestamp)}</Template>
  </AssignVariable>
  <AssignVariable>
    <Name>authString</Name>
    <Template>{serviceUser}:{servicePassword}</Template>
  </AssignVariable>
  <Set>
    <Headers>
      <Header name="Authorization">Basic {encodeBase64(authString)}</Header>
    </Headers>
    <Verb>POST</Verb>
    <Payload contentType="text/plain">Time: {currentTimeUTC}
Event: {jsonPath($.event,calloutResponse.content)}</Payload>
  </Set>
</AssignMessage>
```

Message templates allow dynamic variable substitution using functions and variables.

Let's look at an example of using message templates in a single AssignMessage policy to build a request message.

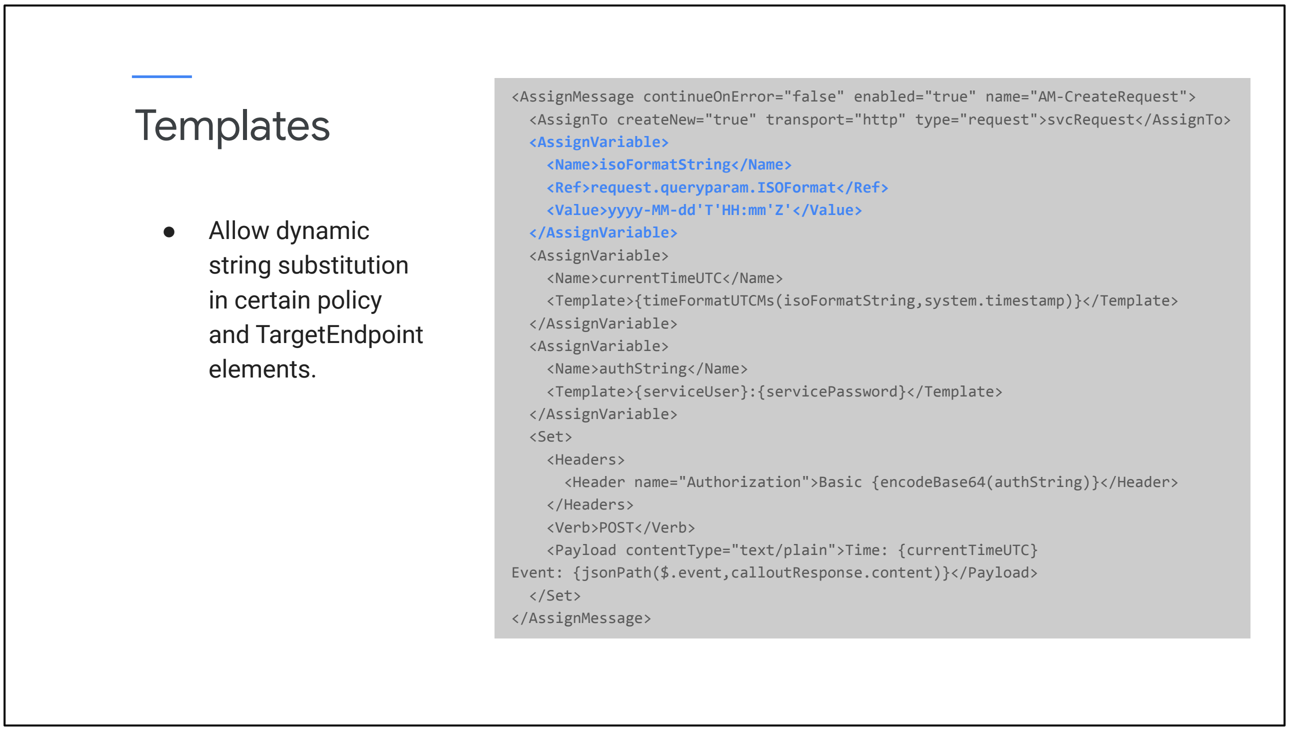


Templates

- Allow dynamic string substitution in certain policy and TargetEndpoint elements.

```
<AssignMessage continueOnError="false" enabled="true" name="AM-CreateRequest">
  <AssignTo createNew="true" transport="http" type="request">svcRequest</AssignTo>
  <AssignVariable>
    <Name>isoFormatString</Name>
    <Ref>request.queryparam.ISOFormat</Ref>
    <Value>yyyy-MM-dd'T'HH:mm'Z'</Value>
  </AssignVariable>
  <AssignVariable>
    <Name>currentTimeUTC</Name>
    <Template>{timeFormatUTCMs(isoFormatString,system.timestamp)}</Template>
  </AssignVariable>
  <AssignVariable>
    <Name>authString</Name>
    <Template>{serviceUser}:{servicePassword}</Template>
  </AssignVariable>
  <Set>
    <Headers>
      <Header name="Authorization">Basic {encodeBase64(authString)}</Header>
    </Headers>
    <Verb>POST</Verb>
    <Payload contentType="text/plain">Time: {currentTimeUTC}
Event: {jsonPath($.event,calloutResponse.content)}</Payload>
  </Set>
</AssignMessage>
```

This policy specifies that a new request message should be created and that it should be called service Request.



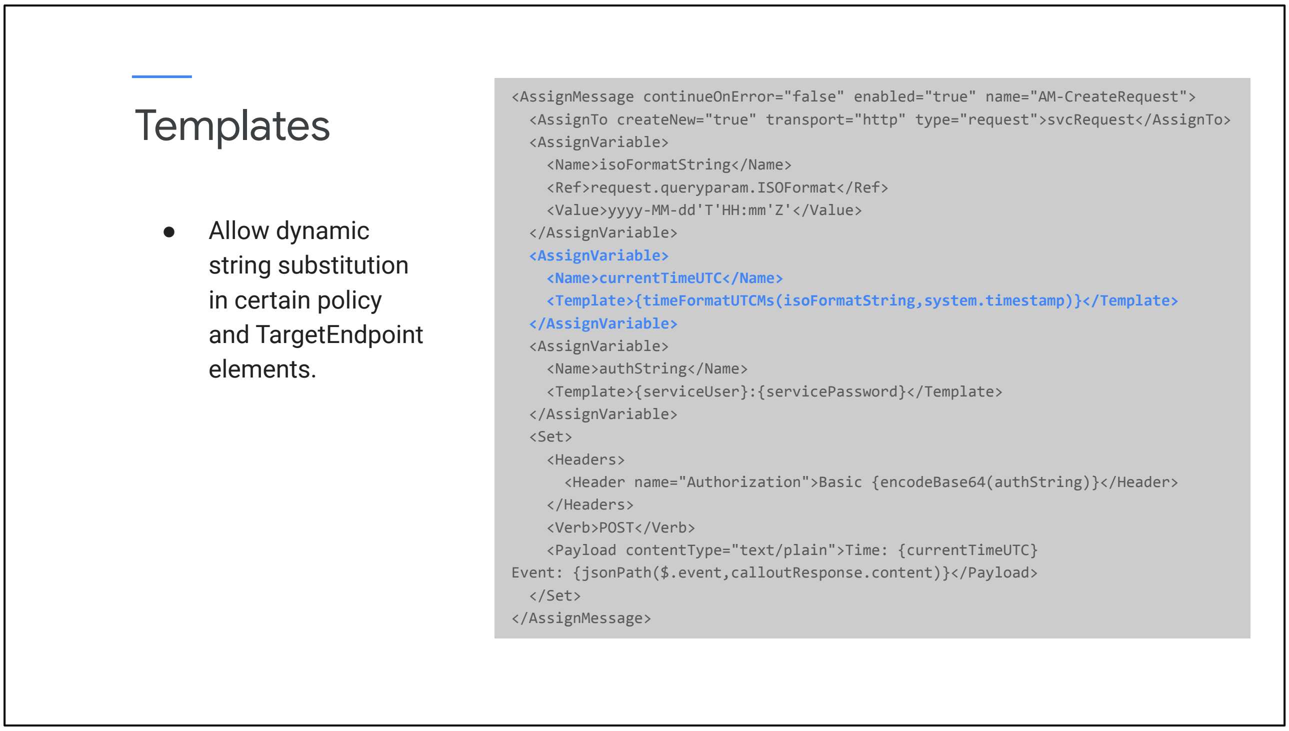
Templates

- Allow dynamic string substitution in certain policy and TargetEndpoint elements.

```
<AssignMessage continueOnError="false" enabled="true" name="AM-CreateRequest">
  <AssignTo createNew="true" transport="http" type="request">svcRequest</AssignTo>
  <AssignVariable>
    <Name>isoFormatString</Name>
    <Ref>request.queryparam.ISOFormat</Ref>
    <Value>yyyy-MM-dd'T'HH:mm'Z'</Value>
  </AssignVariable>
  <AssignVariable>
    <Name>currentTimeUTC</Name>
    <Template>{timeFormatUTCMs(isoFormatString,system.timestamp)}</Template>
  </AssignVariable>
  <AssignVariable>
    <Name>authString</Name>
    <Template>{serviceUser}:{servicePassword}</Template>
  </AssignVariable>
  <Set>
    <Headers>
      <Header name="Authorization">Basic {encodeBase64(authString)}</Header>
    </Headers>
    <Verb>POST</Verb>
    <Payload contentType="text/plain">Time: {currentTimeUTC}
Event: {jsonPath($.event,calloutResponse.content)}</Payload>
  </Set>
</AssignMessage>
```

The first variable created is isoFormatString, which has the value of the ISOFormat query parameter if it exists.

If it does not exist, the default format is specified in the Value element.



Templates

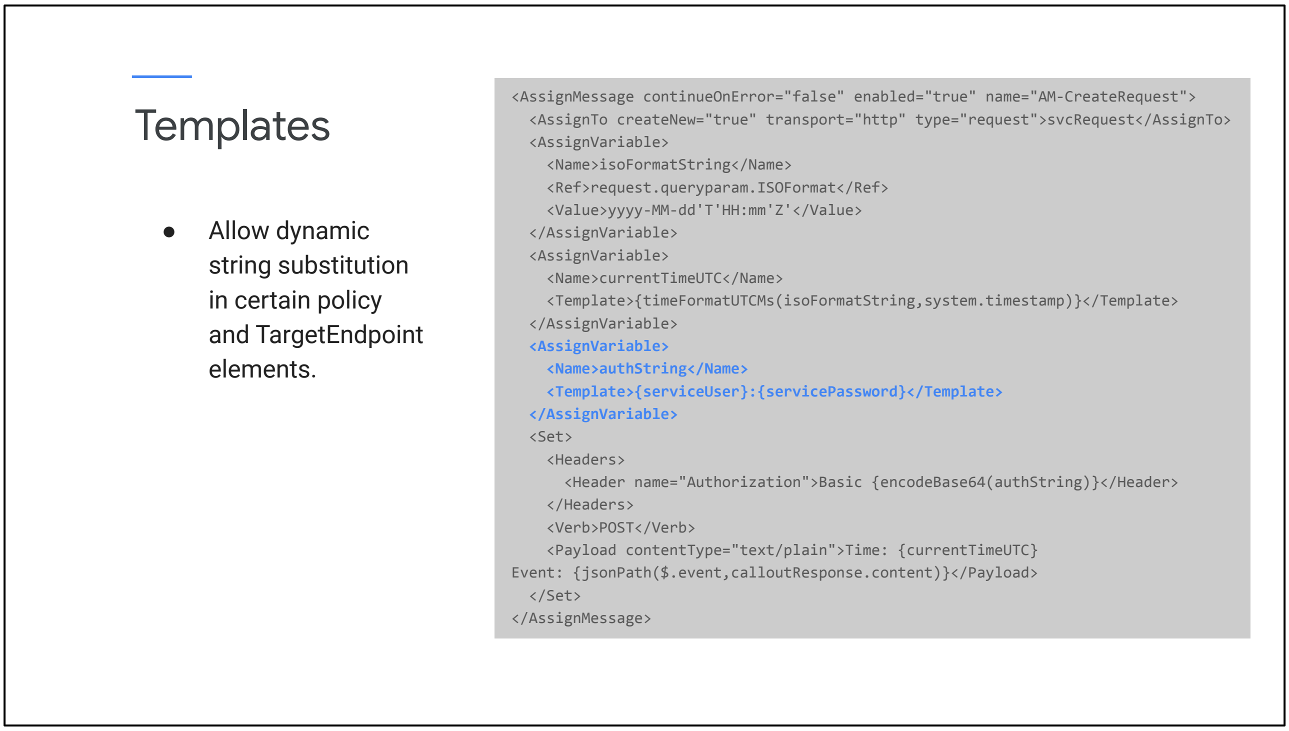
- Allow dynamic string substitution in certain policy and TargetEndpoint elements.

```
<AssignMessage continueOnError="false" enabled="true" name="AM-CreateRequest">
  <AssignTo createNew="true" transport="http" type="request">svcRequest</AssignTo>
  <AssignVariable>
    <Name>isoFormatString</Name>
    <Ref>request.queryparam.ISOFormat</Ref>
    <Value>yyyy-MM-dd'T'HH:mm'Z'</Value>
  </AssignVariable>
  <AssignVariable>
    <Name>currentTimeUTC</Name>
    <Template>{timeFormatUTCMs(isoFormatString,system.timestamp)}</Template>
  </AssignVariable>
  <AssignVariable>
    <Name>authString</Name>
    <Template>{serviceUser}:{servicePassword}</Template>
  </AssignVariable>
  <Set>
    <Headers>
      <Header name="Authorization">Basic {encodeBase64(authString)}</Header>
    </Headers>
    <Verb>POST</Verb>
    <Payload contentType="text/plain">Time: {currentTimeUTC}
Event: {jsonPath($.event,calloutResponse.content)}</Payload>
  </Set>
</AssignMessage>
```

The second variable, currentTimeUTC, uses a message template function that returns a formatted UTC time using a timestamp specified in milliseconds.

It takes two parameters: the format of the string to be created, and a timestamp in milliseconds.

system.timestamp is a predefined variable with a timestamp in milliseconds that represents the time the variable is read.

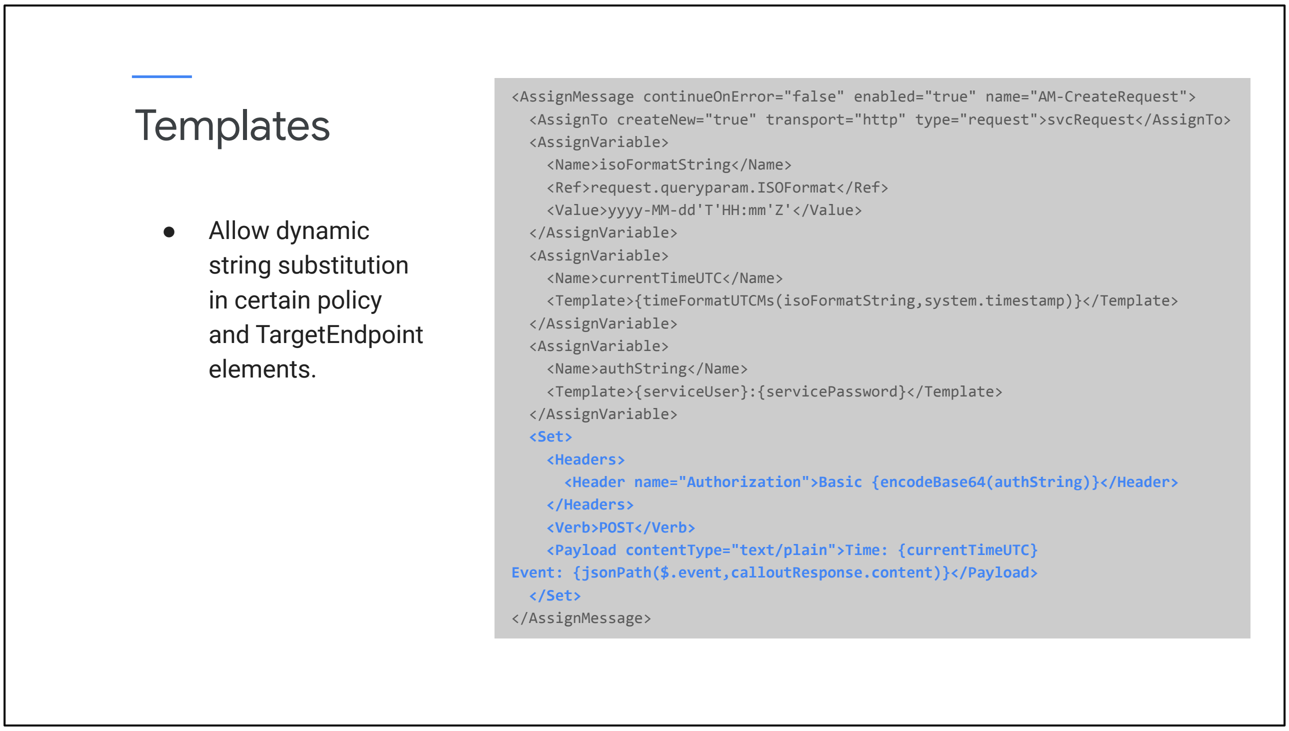


Templates

- Allow dynamic string substitution in certain policy and TargetEndpoint elements.

```
<AssignMessage continueOnError="false" enabled="true" name="AM-CreateRequest">
  <AssignTo createNew="true" transport="http" type="request">svcRequest</AssignTo>
  <AssignVariable>
    <Name>isoFormatString</Name>
    <Ref>request.queryparam.ISOFormat</Ref>
    <Value>yyyy-MM-dd'T'HH:mm'Z'</Value>
  </AssignVariable>
  <AssignVariable>
    <Name>currentTimeUTC</Name>
    <Template>{timeFormatUTCMs(isoFormatString,system.timestamp)}</Template>
  </AssignVariable>
  <AssignVariable>
    <Name>authString</Name>
    <Template>{serviceUser}:{servicePassword}</Template>
  </AssignVariable>
  <Set>
    <Headers>
      <Header name="Authorization">Basic {encodeBase64(authString)}</Header>
    </Headers>
    <Verb>POST</Verb>
    <Payload contentType="text/plain">Time: {currentTimeUTC}
Event: {jsonPath($.event,calloutResponse.content)}</Payload>
  </Set>
</AssignMessage>
```

The third variable, authString, uses a message template to combine two variables, serviceUser and servicePassword, separated by a colon.



Templates

- Allow dynamic string substitution in certain policy and TargetEndpoint elements.

```
<AssignMessage continueOnError="false" enabled="true" name="AM-CreateRequest">
  <AssignTo createNew="true" transport="http" type="request">svcRequest</AssignTo>
  <AssignVariable>
    <Name>isoFormatString</Name>
    <Ref>request.queryparam.ISOFormat</Ref>
    <Value>yyyy-MM-dd'T'HH:mm'Z'</Value>
  </AssignVariable>
  <AssignVariable>
    <Name>currentTimeUTC</Name>
    <Template>{timeFormatUTCMs(isoFormatString,system.timestamp)}</Template>
  </AssignVariable>
  <AssignVariable>
    <Name>authString</Name>
    <Template>{serviceUser}:{servicePassword}</Template>
  </AssignVariable>
  <Set>
    <Headers>
      <Header name="Authorization">Basic {encodeBase64(authString)}</Header>
    </Headers>
    <Verb>POST</Verb>
    <Payload contentType="text/plain">Time: {currentTimeUTC}
Event: {jsonPath($.event,calloutResponse.content)}</Payload>
  </Set>
</AssignMessage>
```

Finally, we set values in the new message.

We create the Authorization header using the encodeBase64 message template function, encoding the authString variable.

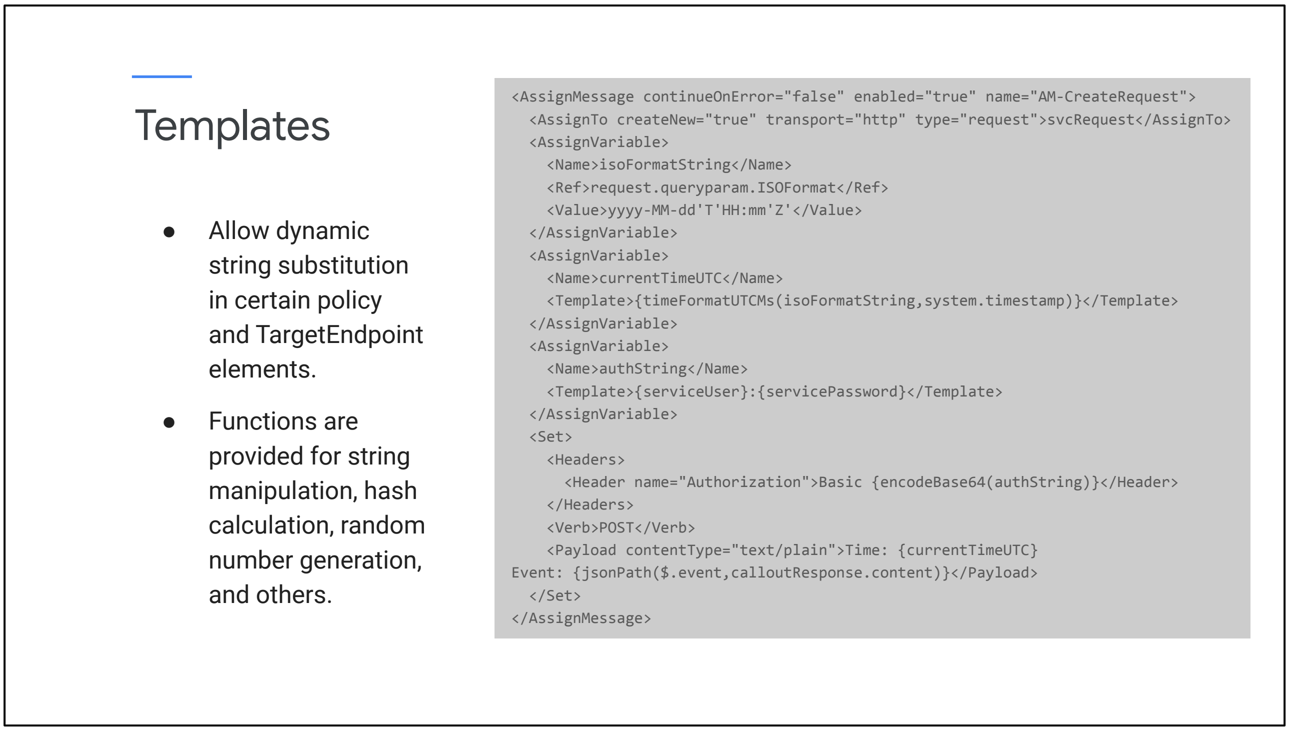
The verb is set to POST.

The payload has type text/plain.

Setting the Content-Type in the payload element also sets the Content-Type header.

The currentTimeUTC variable we created earlier is used in the payload.

We also use another message template function, jsonPath, to extract a JSON field from a variable named calloutResponse.content.



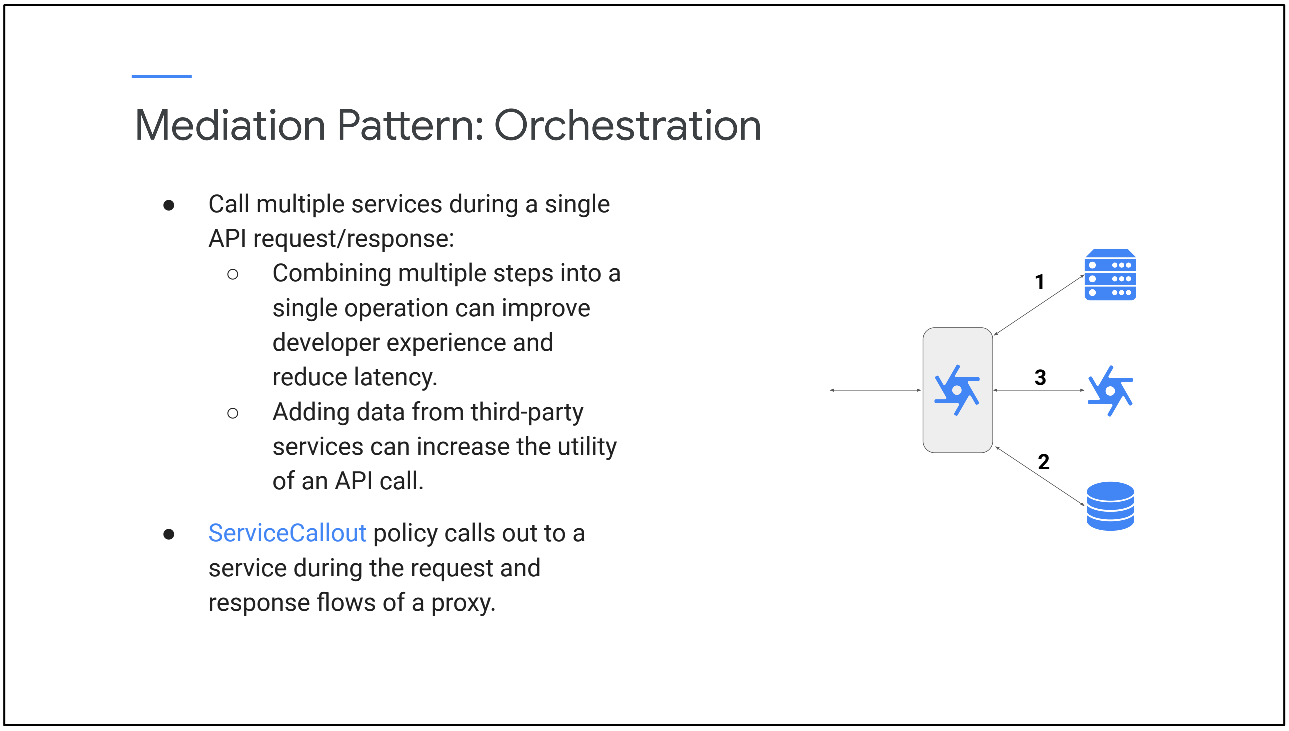
Templates

- Allow dynamic string substitution in certain policy and TargetEndpoint elements.
- Functions are provided for string manipulation, hash calculation, random number generation, and others.

```
<AssignMessage continueOnError="false" enabled="true" name="AM-CreateRequest">
  <AssignTo createNew="true" transport="http" type="request">svcRequest</AssignTo>
  <AssignVariable>
    <Name>isoFormatString</Name>
    <Ref>request.queryparam.ISOFormat</Ref>
    <Value>yyyy-MM-dd'T'HH:mm'Z'</Value>
  </AssignVariable>
  <AssignVariable>
    <Name>currentTimeUTC</Name>
    <Template>{timeFormatUTCMs(isoFormatString,system.timestamp)}</Template>
  </AssignVariable>
  <AssignVariable>
    <Name>authString</Name>
    <Template>{serviceUser}:{servicePassword}</Template>
  </AssignVariable>
  <Set>
    <Headers>
      <Header name="Authorization">Basic {encodeBase64(authString)}</Header>
    </Headers>
    <Verb>POST</Verb>
    <Payload contentType="text/plain">Time: {currentTimeUTC}
Event: {jsonPath($.event,calloutResponse.content)}</Payload>
  </Set>
</AssignMessage>
```

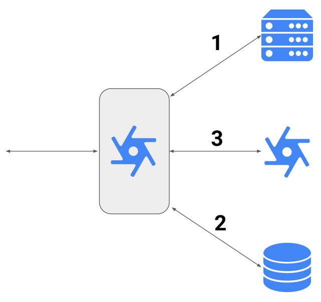
Many different message template functions are available for use in your proxies.

The Apigee documentation contains a complete list.



Mediation Pattern: Orchestration

- Call multiple services during a single API request/response:
  - Combining multiple steps into a single operation can improve developer experience and reduce latency.
  - Adding data from third-party services can increase the utility of an API call.
- ServiceCallout policy calls out to a service during the request and response flows of a proxy.



The second mediation pattern we'll discuss is orchestration.

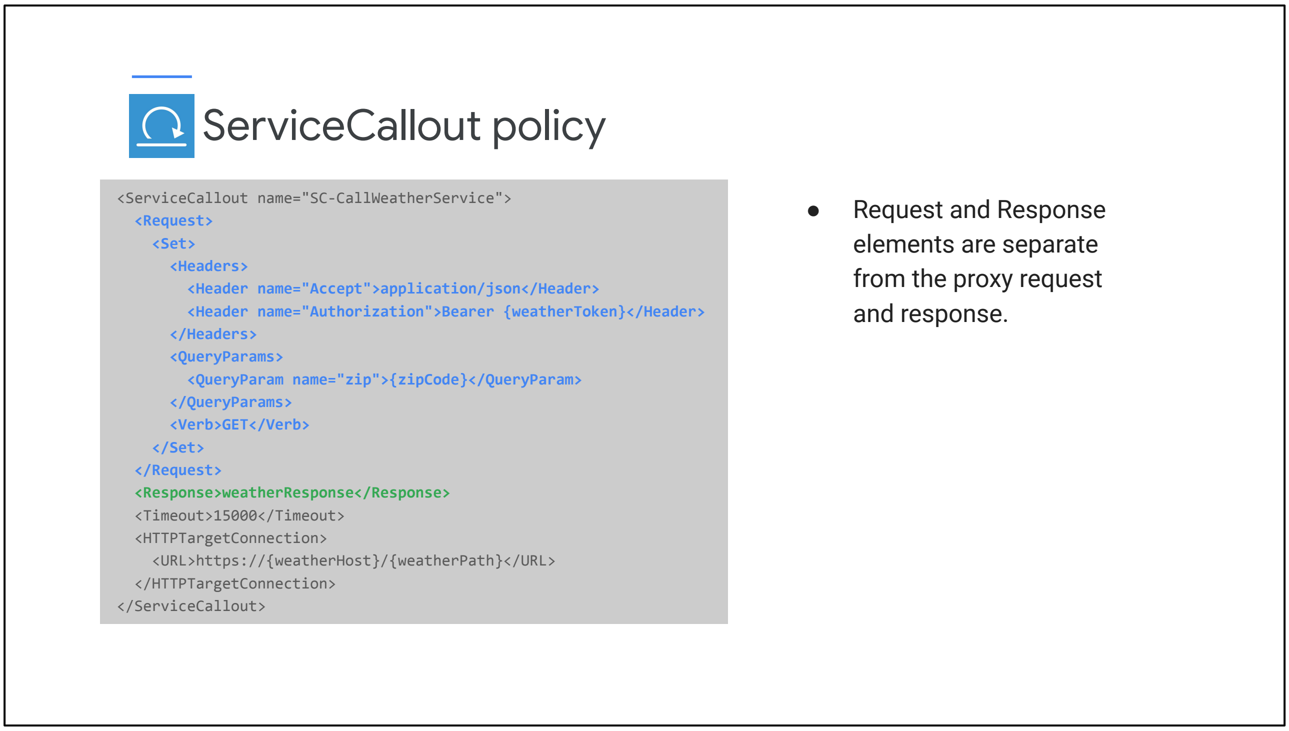
Calling multiple services during a single API request and response is often useful.

You can create a single call that performs multiple steps of an operation, which reduces latency and simplifies app development.

Also, you can use third-party services to add functionality to an existing API call.

However, you are only allowed a single target for an API call.

You can use the ServiceCallout policy to call additional services during the request and response flows of a proxy.



ServiceCallout policy

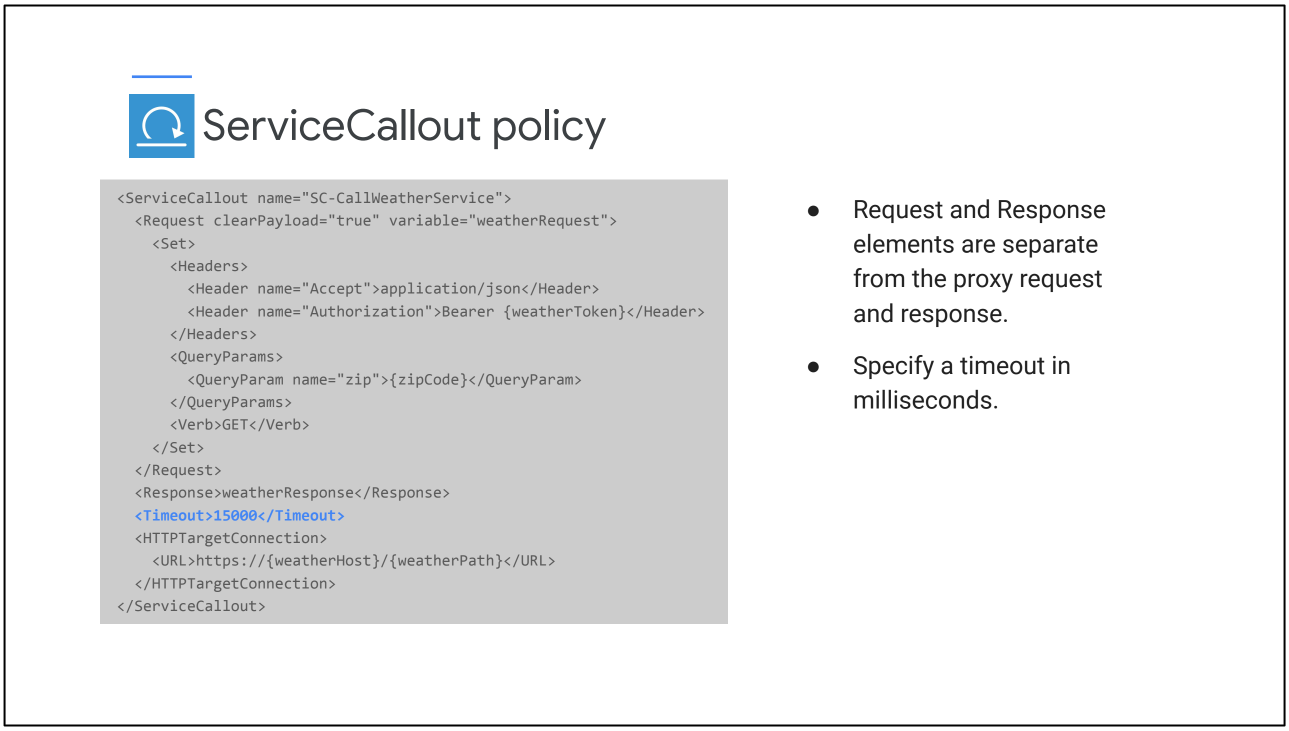
```xml
<ServiceCallout name="SC-CallWeatherService">
  <Request>
    <Set>
      <Headers>
        <Header name="Accept">application/json</Header>
        <Header name="Authorization">Bearer {weatherToken}</Header>
      </Headers>
      <QueryParams>
        <QueryParam name="zip">{zipCode}</QueryParam>
      </QueryParams>
      <Verb>GET</Verb>
    </Set>
  </Request>
  <Response>weatherResponse</Response>
  <Timeout>15000</Timeout>
  <HTTPTargetConnection>
    <URL>https://{weatherHost}/{weatherPath}</URL>
  </HTTPTargetConnection>
</ServiceCallout>
```

- Request and Response elements are separate from the proxy request and response.

The ServiceCallout policy uses separate request and response message variables from those of the proxy request and response.

The request can be specified by variable or be built in the policy, as shown here.

The response is specified as a variable, which will be populated after the call has been made to the service.



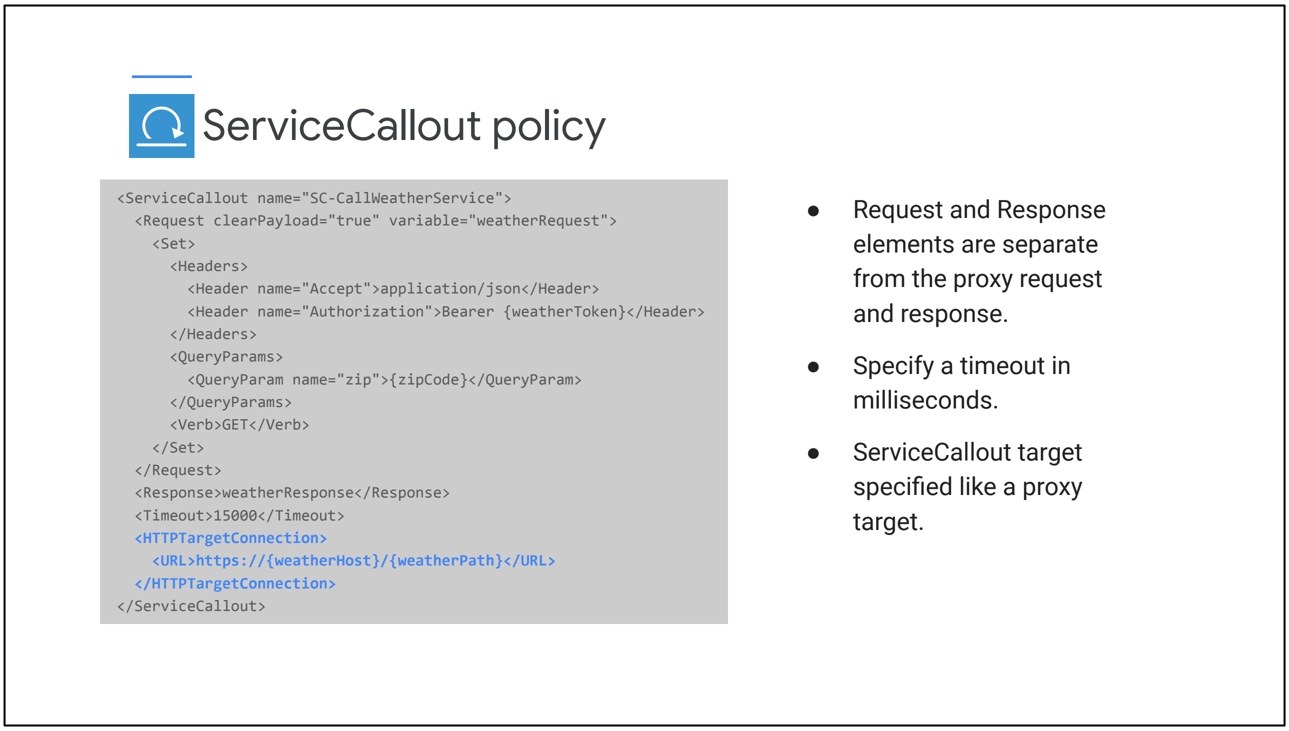
ServiceCallout policy

```
<ServiceCallout name="SC-CallWeatherService">
  <Request clearPayload="true" variable="weatherRequest">
    <Set>
      <Headers>
        <Header name="Accept">application/json</Header>
        <Header name="Authorization">Bearer {weatherToken}</Header>
      </Headers>
      <QueryParams>
        <QueryParam name="zip">{zipCode}</QueryParam>
      </QueryParams>
      <Verb>GET</Verb>
    </Set>
  </Request>
  <Response>weatherResponse</Response>
  <Timeout>15000</Timeout>
  <HTTPTargetConnection>
    <URL>https://{weatherHost}/{weatherPath}</URL>
  </HTTPTargetConnection>
</ServiceCallout>
```

- Request and Response elements are separate from the proxy request and response.
- Specify a timeout in milliseconds.

You can also specify a timeout in milliseconds.

If the timeout is reached, the policy throws an error.



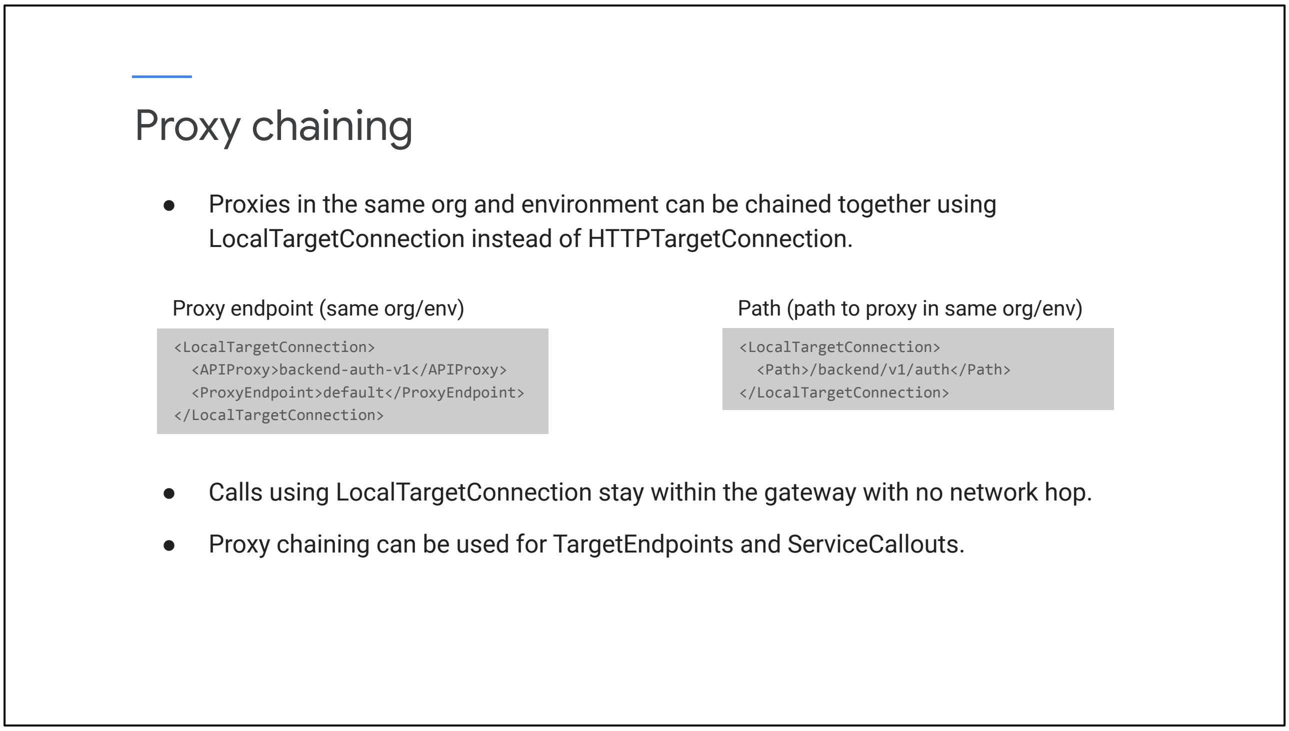
ServiceCallout policy

```
<ServiceCallout name="SC-CallWeatherService">
  <Request clearPayload="true" variable="weatherRequest">
    <Set>
      <Headers>
        <Header name="Accept">application/json</Header>
        <Header name="Authorization">Bearer {weatherToken}</Header>
      </Headers>
      <QueryParams>
        <QueryParam name="zip">{zipCode}</QueryParam>
      </QueryParams>
      <Verb>GET</Verb>
    </Set>
  </Request>
  <Response>weatherResponse</Response>
  <Timeout>15000</Timeout>
  <HTTPTargetConnection>
    <URL>https://{weatherHost}/{weatherPath}</URL>
  </HTTPTargetConnection>
</ServiceCallout>
```

- Request and Response elements are separate from the proxy request and response.
- Specify a timeout in milliseconds.
- ServiceCallout target specified like a proxy target.

The target for a service callout is configured in a similar way to the TargetEndpoint target.

In this case, the hostname and path for the call are being populated using variables in message templates.



Proxy chaining

- Proxies in the same org and environment can be chained together using LocalTargetConnection instead of HTTPTargetConnection.

Proxy endpoint (same org/env)

```
<LocalTargetConnection>
  <APIProxy>backend-auth-v1</APIProxy>
  <ProxyEndpoint>default</ProxyEndpoint>
</LocalTargetConnection>
```

Path (path to proxy in same org/env)

```
<LocalTargetConnection>
  <Path>/backend/v1/auth</Path>
</LocalTargetConnection>
```

- Calls using LocalTargetConnection stay within the gateway with no network hop.
- Proxy chaining can be used for TargetEndpoints and ServiceCallouts.

Proxies deployed in the same organization and environment can be called using a process called proxy chaining.

Instead of using the HTTPTargetConnection element, you use the LocalTargetConnection element.

The proxy to be called can be specified in two different ways: by specifying the proxy name and the proxy endpoint inside the proxy, or by specifying the path to the proxy without the hostname.

One of the benefits to proxy chaining is that the call to the local proxy stays within the gateway. There is no network hop and no additional network connection.

A LocalTargetConnection can be used in TargetEndpoints and ServiceCallouts.

Proxy chaining is one strategy for avoiding duplicate code in your proxies. In a later lecture we will discuss a more powerful method of code reuse: shared flows.

Agenda

JSON, XML, and SOAP (lab)

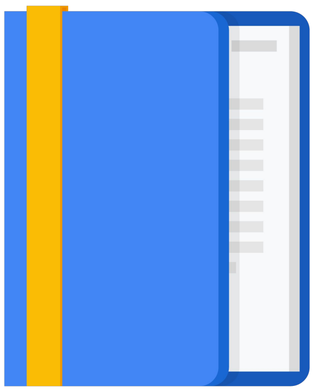
Mediation and Service Callouts

Custom Code (2 labs)

Shared Flows (lab)

Fault Handling (lab)

Quiz

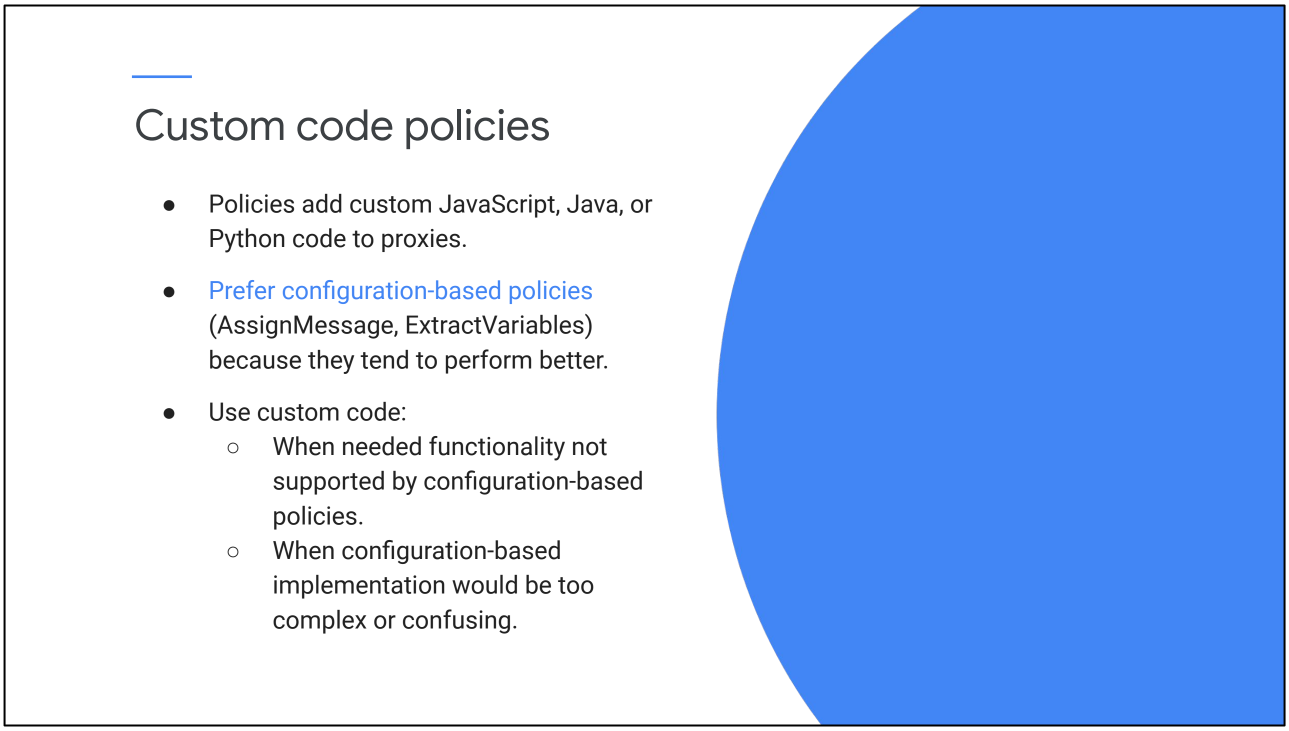


So far we've discussed how we use configuration-based policies to build our proxies.

Sometimes, you just want to write code.

This section will discuss the options for using code in your API proxies.

This section will end with 2 labs.



Custom code policies

- Policies add custom JavaScript, Java, or Python code to proxies.
- Prefer configuration-based policies (AssignMessage, ExtractVariables) because they tend to perform better.
- Use custom code:
    - When needed functionality not supported by configuration-based policies.
    - When configuration-based implementation would be too complex or confusing.

Apigee provides policies that allow you to write your own code within your proxies.

JavaScript, Java, and Python are the languages supported.

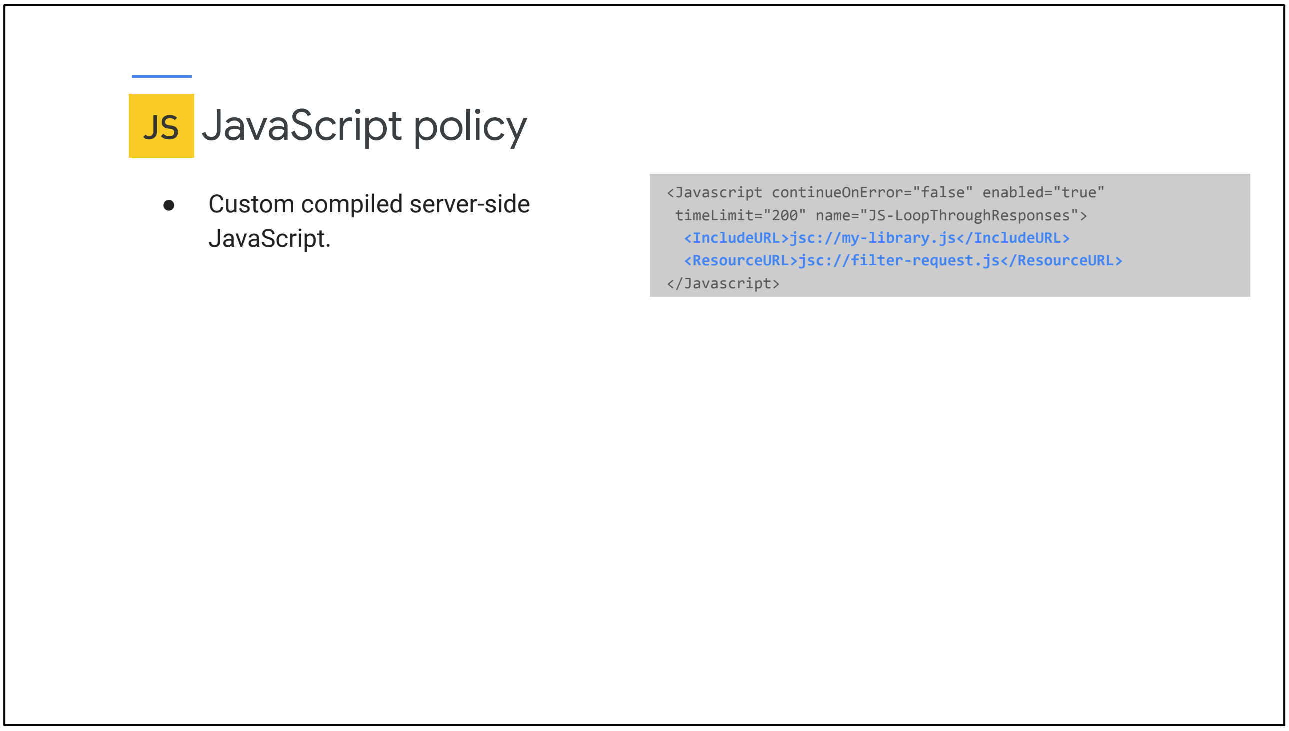
Even though these policy types are supported, you should use the configuration-based policies when possible. The configuration-based policies like AssignMessage and ExtractVariables have been optimized to run very efficiently on Apigee. In most cases, when you can reasonably implement your functionality using the configuration-based policies, they will perform better and more predictably.

So when should you use custom code? Sometimes the configuration-based policies don't provide the functionality required to solve the problem.

For example, if you need to iterate through a complex payload and make changes to it, using ExtractVariables and AssignMessage might not be possible.

Sometimes you can use the configuration-based policies, but the implementation would be too complex or too confusing to maintain.

In general, use the configuration-based policies when you can, and use the code policies when you must.



JS JavaScript policy

- Custom compiled server-side JavaScript.

```
<Javascript continueOnError="false" enabled="true"
 timeLimit="200" name="JS-LoopThroughResponses">
  <IncludeURL>jsc://my-library.js</IncludeURL>
  <ResourceURL>jsc://filter-request.js</ResourceURL>
</Javascript>
```

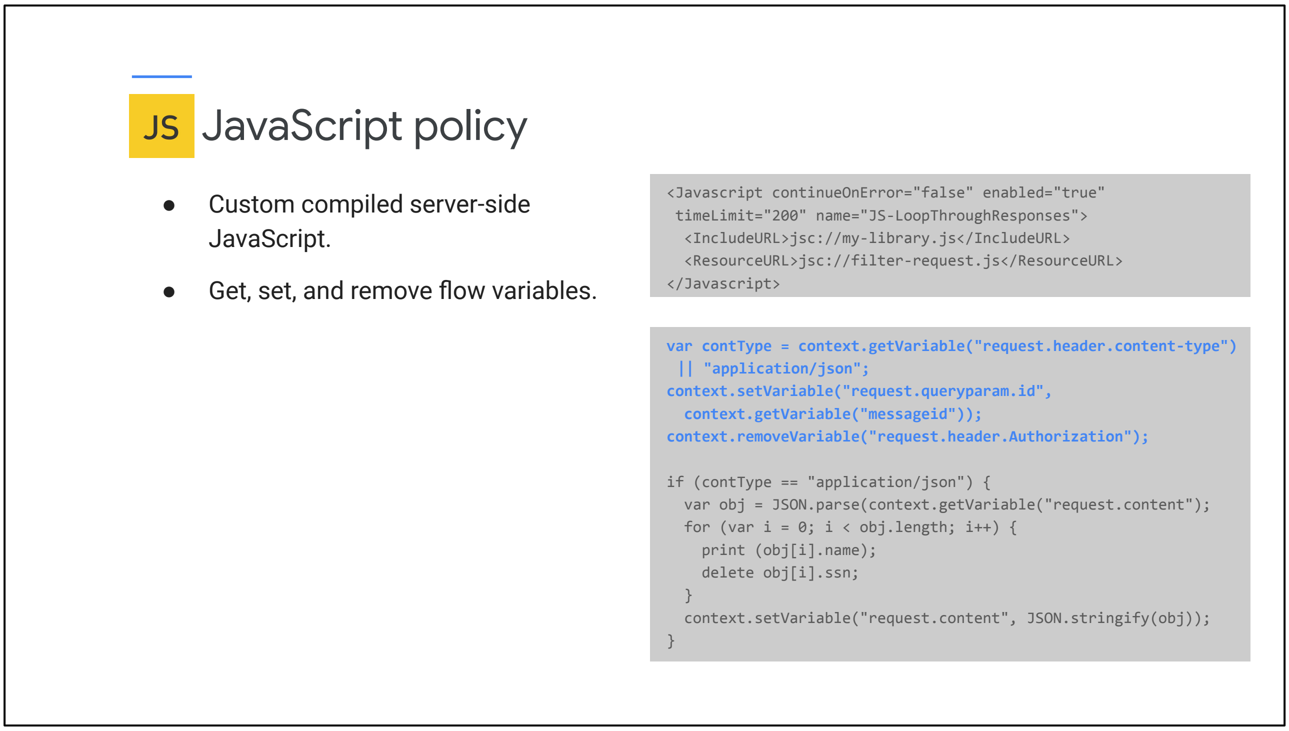
The XML policy configuration looks similar to the other policies we've seen.

The policy contains references to JavaScript code files that are stored in the resources section of the proxy.

The IncludeURL element specifies a JavaScript library file that is a dependency for the main JavaScript file, which is specified with the ResourceURL element. Multiple IncludeURL elements may be specified, and they are evaluated in the order listed in the proxy.

Each library file in an IncludeURL element must be stored in the resources section of the proxy or stored as an organization or environment scoped resource.

The Resource URL file is loaded last.



JS JavaScript policy

- Custom compiled server-side JavaScript.
- Get, set, and remove flow variables.

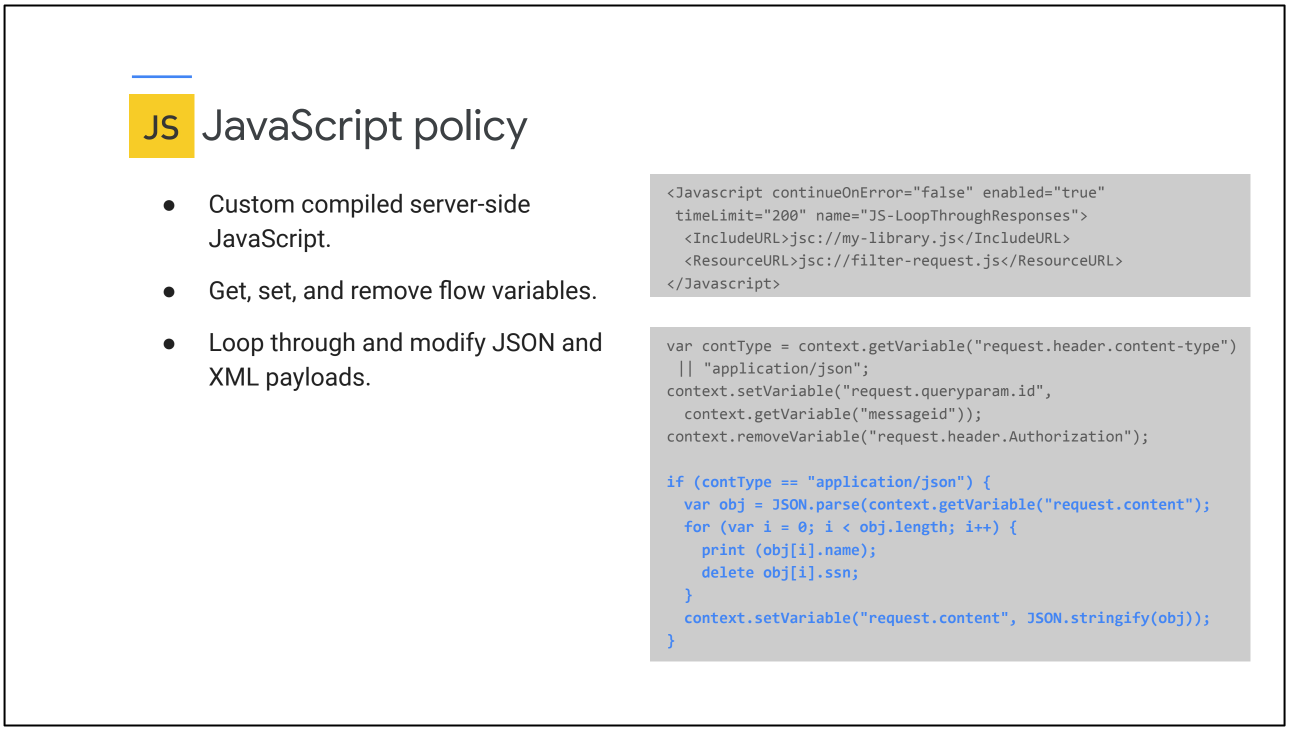
```
<Javascript continueOnError="false" enabled="true"
 timeLimit="200" name="JS-LoopThroughResponses">
  <IncludeURL>jsc://my-library.js</IncludeURL>
  <ResourceURL>jsc://filter-request.js</ResourceURL>
</Javascript>
```

```
var contType = context.getVariable("request.header.content-type")
 || "application/json";
context.setVariable("request.queryparam.id",
  context.getVariable("messageid"));
context.removeVariable("request.header.Authorization");

if (contType == "application/json") {
  var obj = JSON.parse(context.getVariable("request.content");
  for (var i = 0; i < obj.length; i++) {
    print (obj[i].name);
    delete obj[i].ssn;
  }
  context.setVariable("request.content", JSON.stringify(obj));
}
```

JavaScript code is written in a resource file and stored within the proxy.

You can get, modify, and delete flow variables from within the JavaScript code.

context.getVariable retrieves a flow variable by name, context.setVariable updates or creates a flow variable, and context.removeVariable deletes the flow variable.



JS JavaScript policy

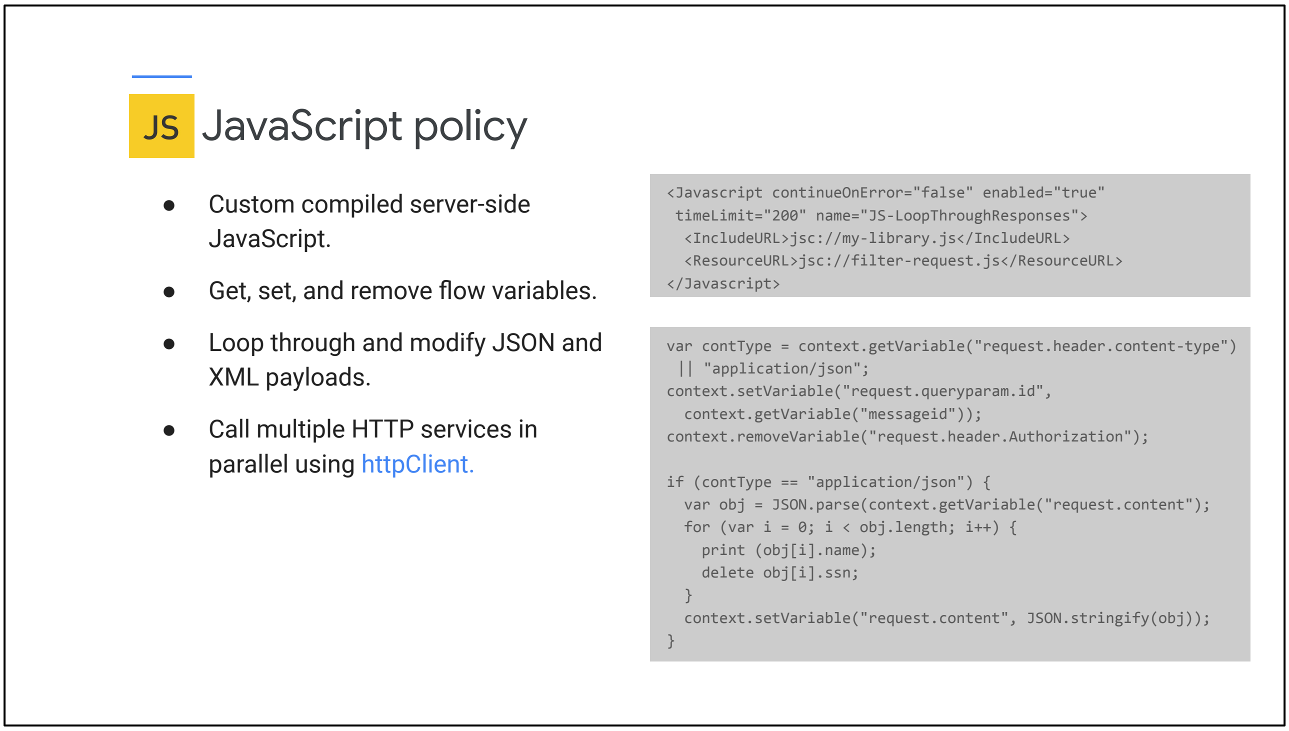
- Custom compiled server-side JavaScript.
- Get, set, and remove flow variables.
- Loop through and modify JSON and XML payloads.

```
<Javascript continueOnError="false" enabled="true"
 timeLimit="200" name="JS-LoopThroughResponses">
  <IncludeURL>jsc://my-library.js</IncludeURL>
  <ResourceURL>jsc://filter-request.js</ResourceURL>
</Javascript>
```

```
var contType = context.getVariable("request.header.content-type")
 || "application/json";
context.setVariable("request.queryparam.id",
  context.getVariable("messageid"));
context.removeVariable("request.header.Authorization");

if (contType == "application/json") {
  var obj = JSON.parse(context.getVariable("request.content");
  for (var i = 0; i < obj.length; i++) {
    print (obj[i].name);
    delete obj[i].ssn;
  }
  context.setVariable("request.content", JSON.stringify(obj));
}
```

JavaScript is also useful for looping through and modifying complex payloads.

In this case, we are taking the request payload, accessible by the variable request.content, and parsing it into a JavaScript object. We then manipulate the object, and put the results back into the request payload using JSON.stringify.



JS JavaScript policy

- Custom compiled server-side JavaScript.
- Get, set, and remove flow variables.
- Loop through and modify JSON and XML payloads.
- Call multiple HTTP services in parallel using httpClient.

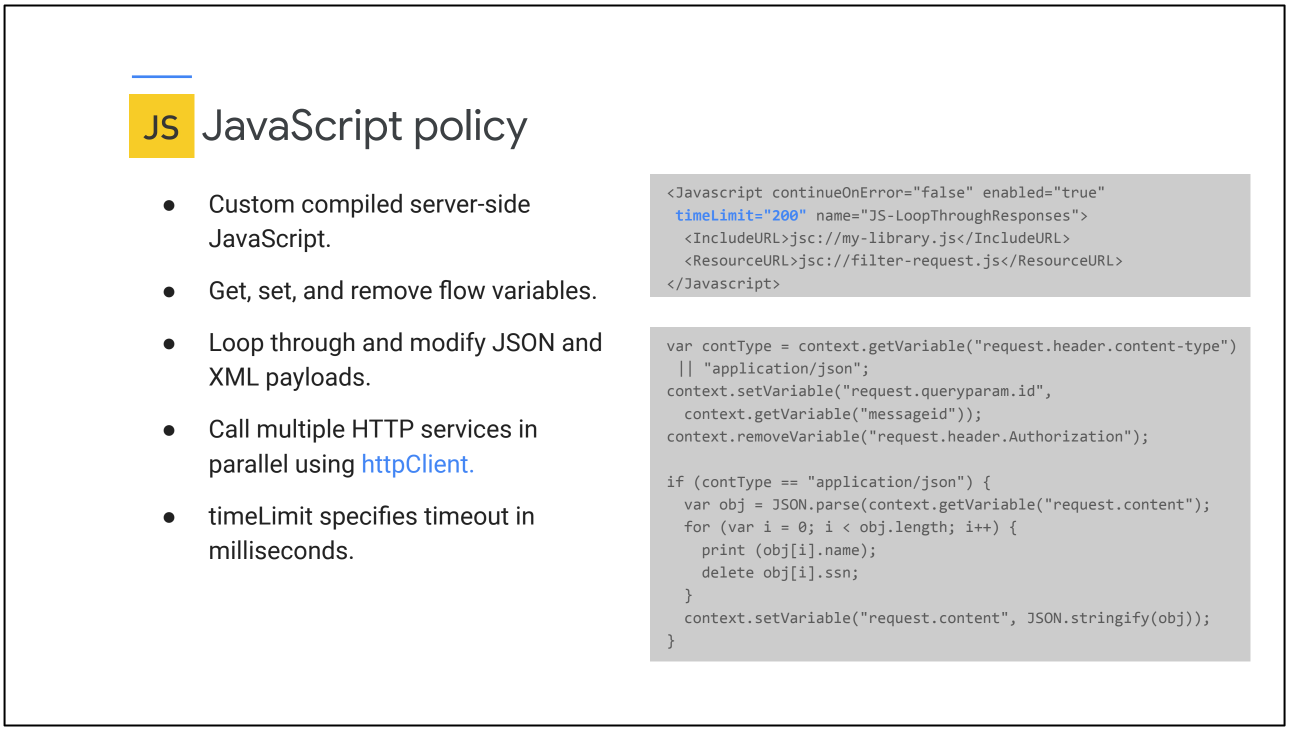
```
<Javascript continueOnError="false" enabled="true"
 timeLimit="200" name="JS-LoopThroughResponses">
  <IncludeURL>jsc://my-library.js</IncludeURL>
  <ResourceURL>jsc://filter-request.js</ResourceURL>
</Javascript>
```

```
var contType = context.getVariable("request.header.content-type")
  || "application/json";
context.setVariable("request.queryparam.id",
  context.getVariable("messageid"));
context.removeVariable("request.header.Authorization");

if (contType == "application/json") {
  var obj = JSON.parse(context.getVariable("request.content");
  for (var i = 0; i < obj.length; i++) {
    print (obj[i].name);
    delete obj[i].ssn;
  }
  context.setVariable("request.content", JSON.stringify(obj));
}
```

Another task that can be done with JavaScript is calling multiple HTTP services in parallel.

The ServiceCallout policy waits for a response from a called service before continuing with the proxy flow.

Using the httpclient object in JavaScript, you can call services in parallel and reduce the latency of the API proxy.



JS JavaScript policy

- Custom compiled server-side JavaScript.
- Get, set, and remove flow variables.
- Loop through and modify JSON and XML payloads.
- Call multiple HTTP services in parallel using httpClient.
- timeLimit specifies timeout in milliseconds.

```
<Javascript continueOnError="false" enabled="true"
 timeLimit="200" name="JS-LoopThroughResponses">
  <IncludeURL>jsc://my-library.js</IncludeURL>
  <ResourceURL>jsc://filter-request.js</ResourceURL>
</Javascript>
```

```
var contType = context.getVariable("request.header.content-type")
 || "application/json";
context.setVariable("request.queryparam.id",
  context.getVariable("messageid"));
context.removeVariable("request.header.Authorization");

if (contType == "application/json") {
  var obj = JSON.parse(context.getVariable("request.content");
  for (var i = 0; i < obj.length; i++) {
    print (obj[i].name);
    delete obj[i].ssn;
  }
  context.setVariable("request.content", JSON.stringify(obj));
}
```

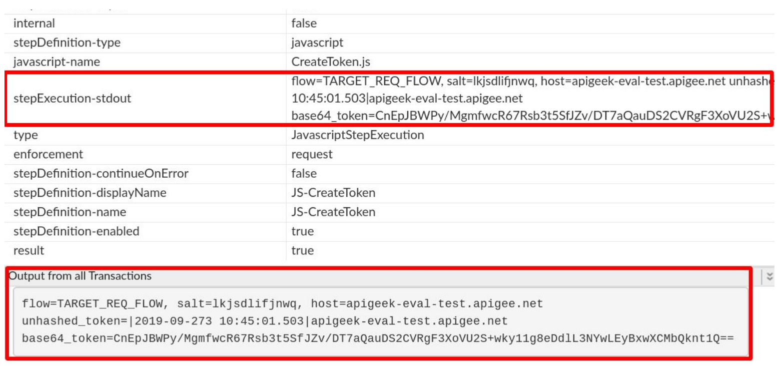
There is a time limit attribute for JavaScript policies.

This sets the timeout for a JavaScript policy in milliseconds.

If the policy is still executing when the timeout is reached, the policy will stop execution and raise a fault.

JavaScript logging to console

- The print() function logs to the console in the Trace Tool.



internal	false
stepDefinition-type	javascript
javascript-name	CreateToken.js

stepExecution-stdout	flow=TARGET_REQ_FLOW, salt=lkjsdlifjnwq, host=apigeek-eval-test.apigee.net unhash
	10:45:01.503|apigeek-eval-test.apigee.net
	base64_token=CnEpJBWPy/MgmfwcR67Rsb3t5SfJZv/DT7aQauDS2CVRgF3XoVU2S+

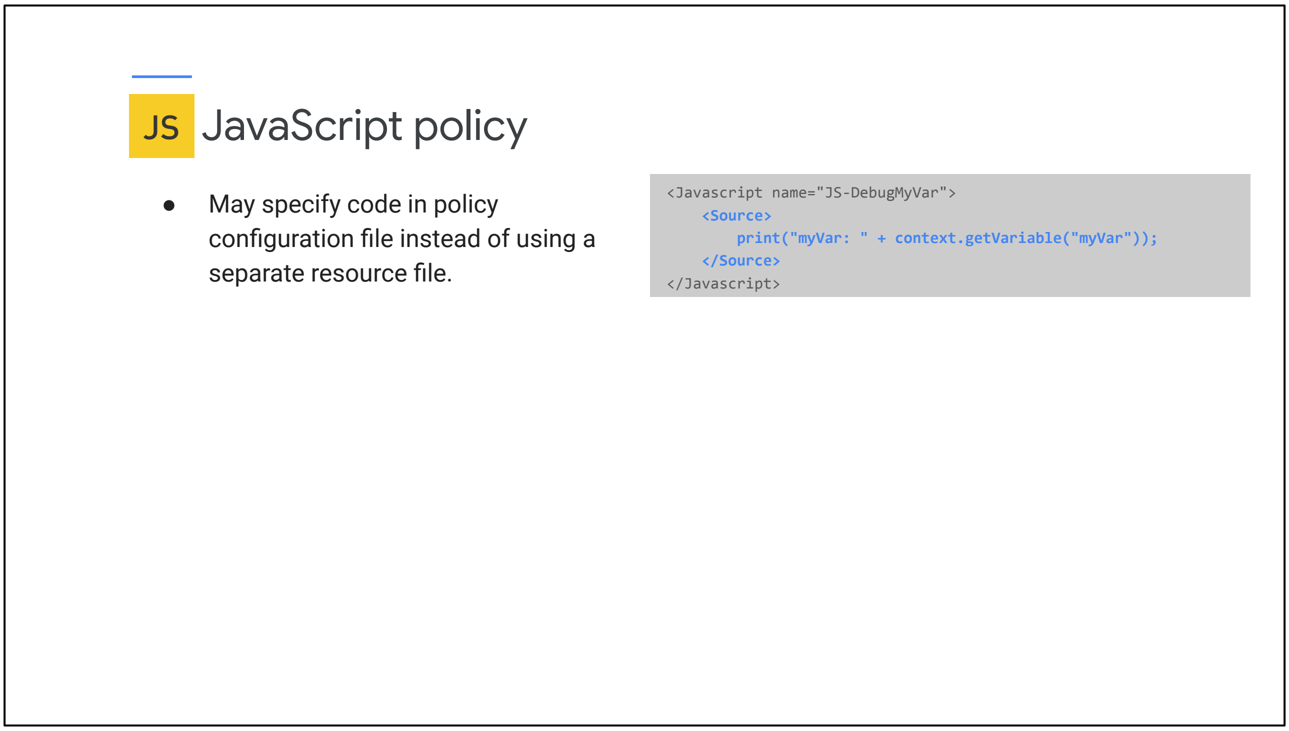
type	JavascriptStepExecution
enforcement	request
stepDefinition-continueOnError	false
stepDefinition-displayName	JS-CreateToken
stepDefinition-name	JS-CreateToken
stepDefinition-enabled	true
result	true

Output from all Transactions

```
flow=TARGET_REQ_FLOW, salt=lkjsdlifjnwq, host=apigeek-eval-test.apigee.net
unhashed_token=|2019-09-273 10:45:01.503|apigeek-eval-test.apigee.net
base64_token=CnEpJBWPy/MgmfwcR67Rsb3t5SfJZv/DT7aQauDS2CVRgF3XoVU2S+wky11g8eDdlL3NYwLEyBxwXCMbQknt1Q==
```

The print function in JavaScript logs a line to the console in the trace tool.

This can help you debug your JavaScript policies or provide useful information while tracing the proxy.



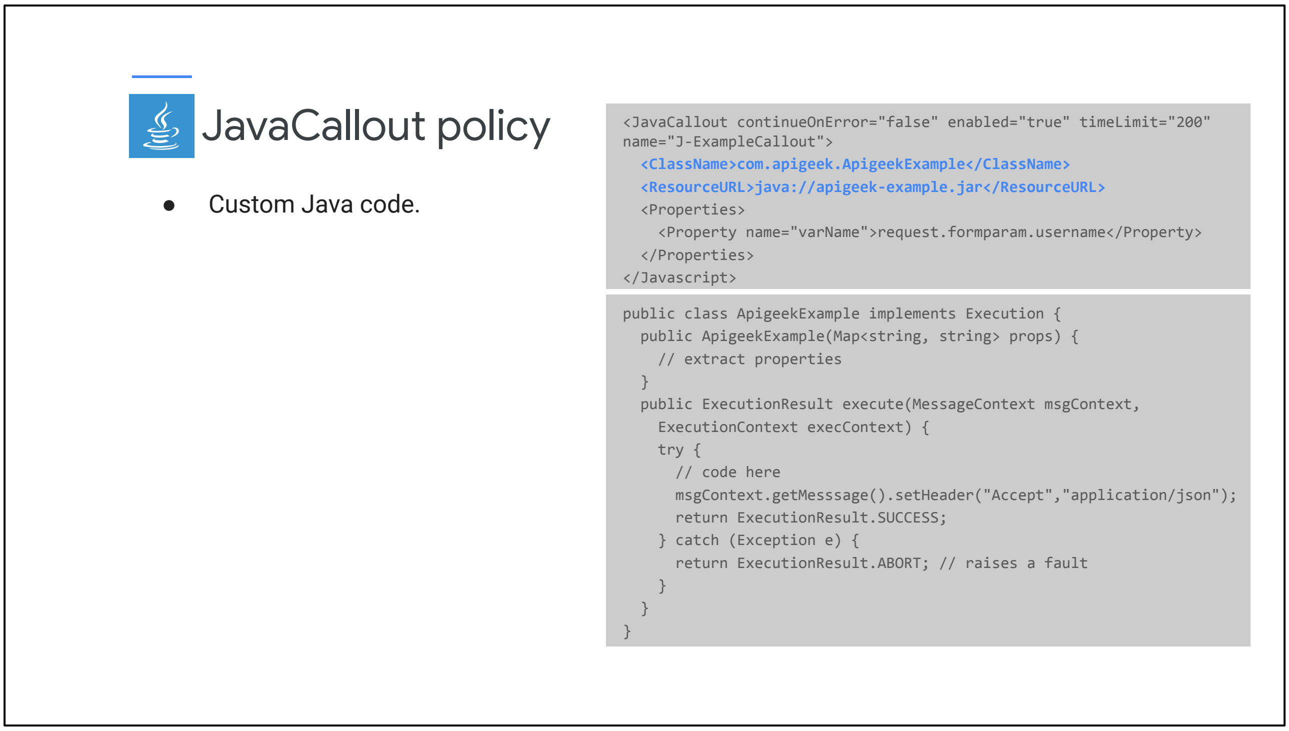
JS JavaScript policy

- May specify code in policy configuration file instead of using a separate resource file.

```
<Javascript name="JS-DebugMyVar">
    <Source>
        print("myVar: " + context.getVariable("myVar"));
    </Source>
</Javascript>
```

For simple use cases, JavaScript code may also be inserted directly into the Source element of the policy configuration file, eliminating the need for a separate resource file.

Code specified in the Source element executes identically to code in resource files.



JavaCallout policy

- Custom Java code.

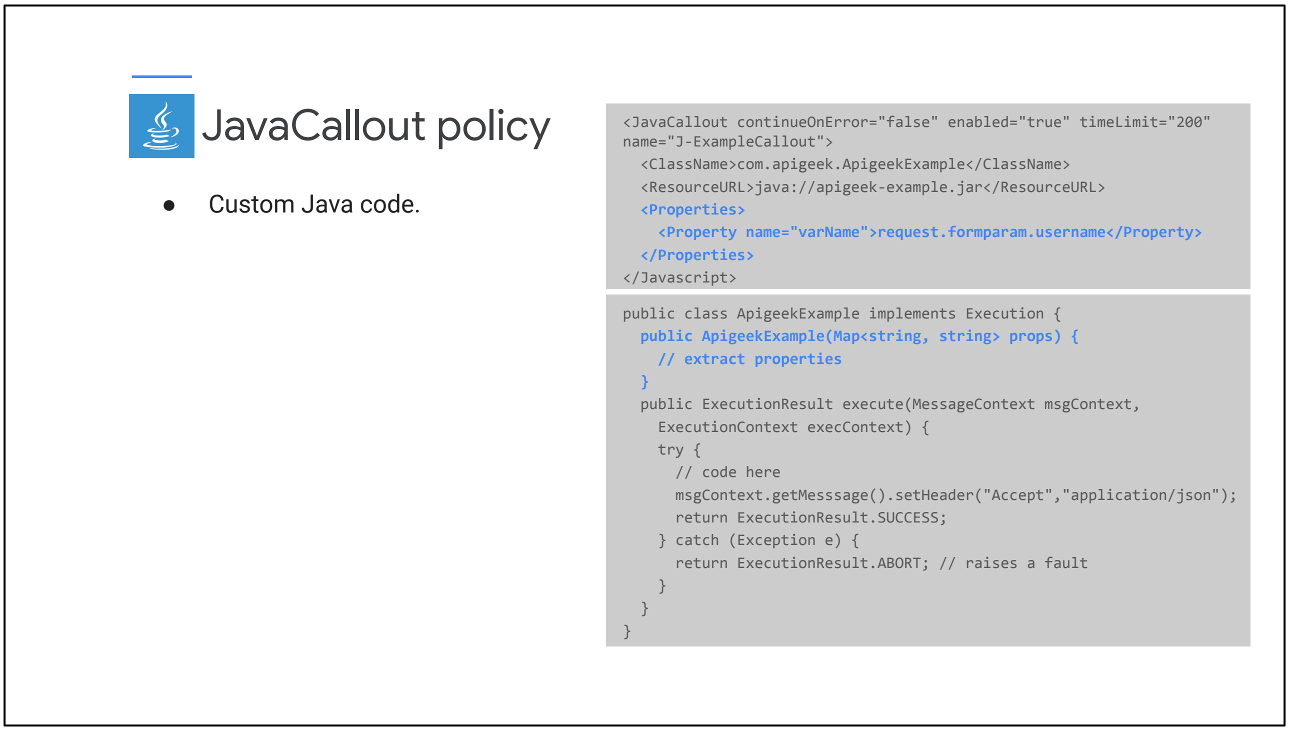
```
<JavaCallout continueOnError="false" enabled="true" timeLimit="200"
name="J-ExampleCallout">
  <ClassName>com.apigeek.ApigeekExample</ClassName>
  <ResourceURL>java://apigeek-example.jar</ResourceURL>
  <Properties>
    <Property name="varName">request.formparam.username</Property>
  </Properties>
</Javascript>
```

```
public class ApigeekExample implements Execution {
  public ApigeekExample(Map<string, string> props) {
    // extract properties
  }
  public ExecutionResult execute(MessageContext msgContext,
    ExecutionContext execContext) {
    try {
      // code here
      msgContext.getMesssage().setHeader("Accept","application/json");
      return ExecutionResult.SUCCESS;
    } catch (Exception e) {
      return ExecutionResult.ABORT; // raises a fault
    }
  }
}
```

For writing code in Java, Apigee provides a JavaCallout policy.

The JavaCallout policy allows you to use custom Java code in your proxies. Java code running in a JavaCallout will perform very well.

The elements in the JavaCallout policy are similar to those in the JavaScript policy, except that the class to run within the configured Java JAR file must be specified.



JavaCallout policy

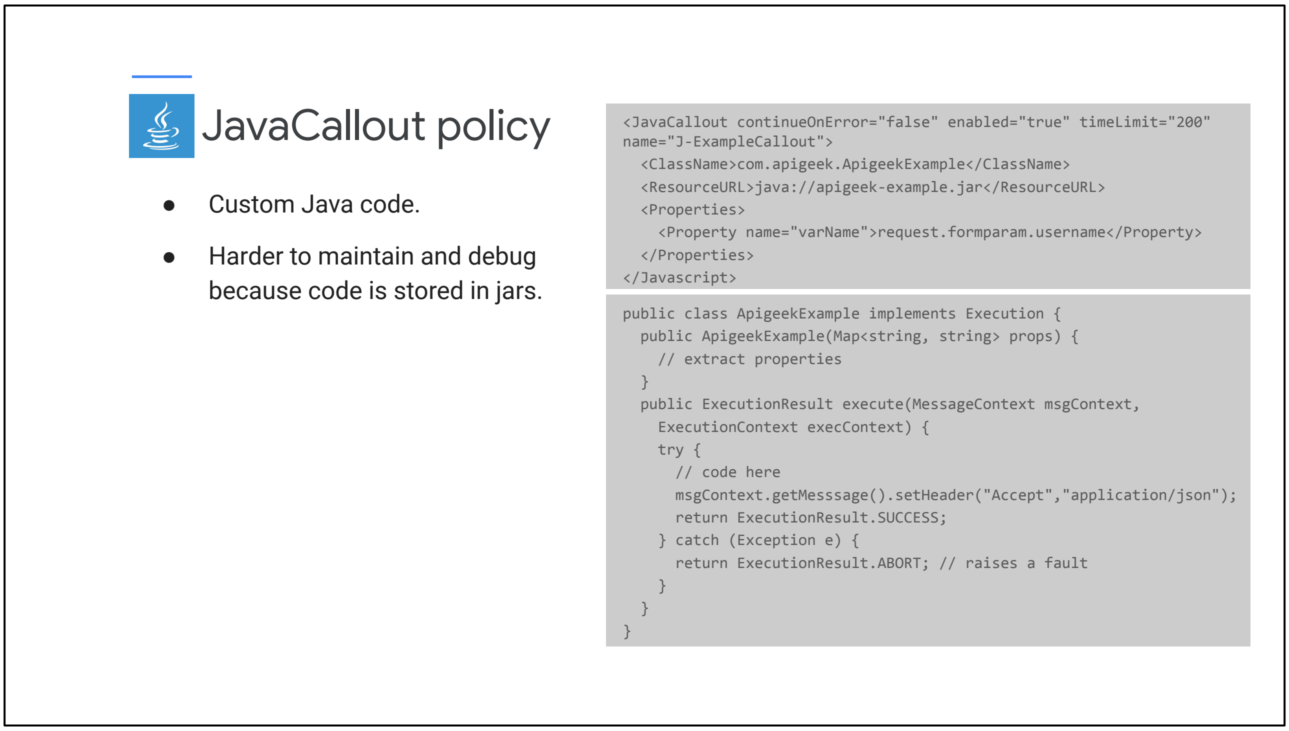
- Custom Java code.

```
<JavaCallout continueOnError="false" enabled="true" timeLimit="200"
name="J-ExampleCallout">
  <ClassName>com.apigeek.ApigeekExample</ClassName>
  <ResourceURL>java://apigeek-example.jar</ResourceURL>
  <Properties>
    <Property name="varName">request.formparam.username</Property>
  </Properties>
</Javascript>
```

```
public class ApigeekExample implements Execution {
  public ApigeekExample(Map<string, string> props) {
    // extract properties
  }
  public ExecutionResult execute(MessageContext msgContext,
    ExecutionContext execContext) {
    try {
      // code here
      msgContext.getMesssage().setHeader("Accept","application/json");
      return ExecutionResult.SUCCESS;
    } catch (Exception e) {
      return ExecutionResult.ABORT; // raises a fault
    }
  }
}
```

You can also specify a list of name value properties.

The values are plain strings, and the properties can be used within the Java code.



JavaCallout policy

- Custom Java code.
- Harder to maintain and debug because code is stored in jars.

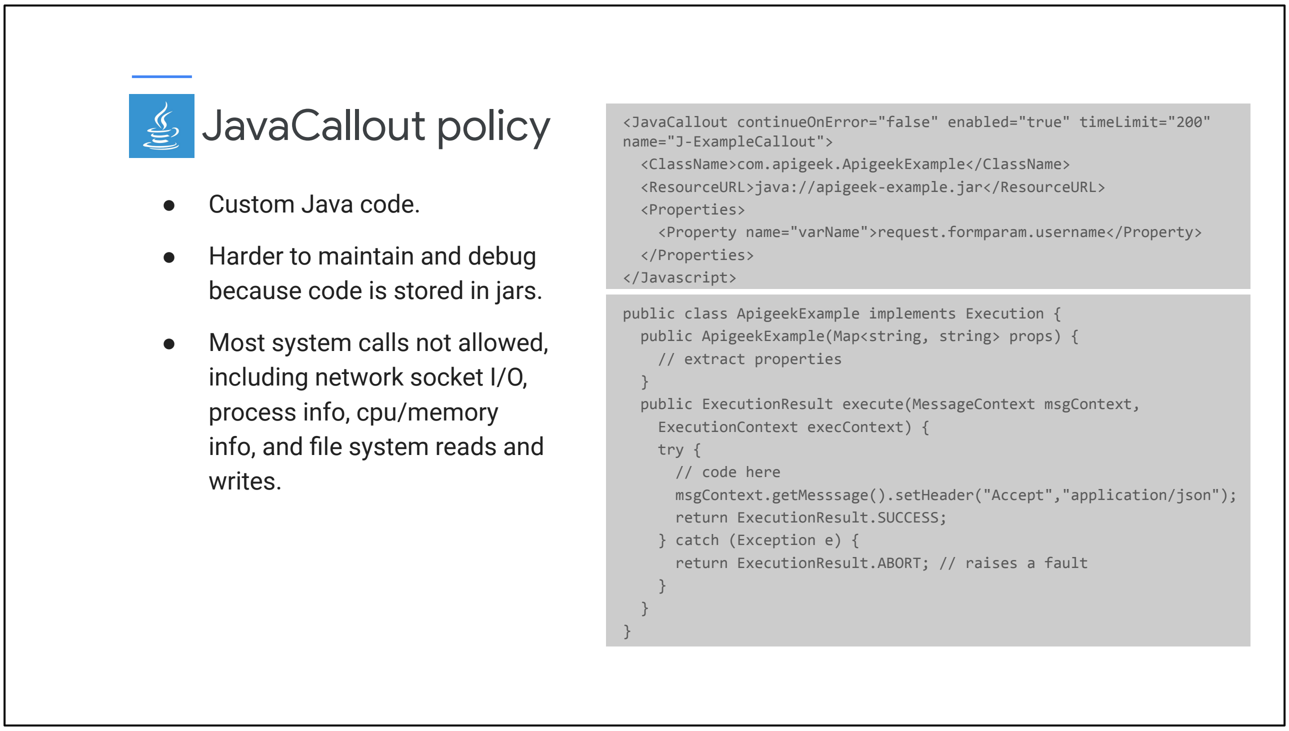
```
<JavaCallout continueOnError="false" enabled="true" timeLimit="200"
name="J-ExampleCallout">
  <ClassName>com.apigeek.ApigeekExample</ClassName>
  <ResourceURL>java://apigeek-example.jar</ResourceURL>
  <Properties>
    <Property name="varName">request.formparam.username</Property>
  </Properties>
</Javascript>
```

```
public class ApigeekExample implements Execution {
  public ApigeekExample(Map<string, string> props) {
    // extract properties
  }
  public ExecutionResult execute(MessageContext msgContext,
    ExecutionContext execContext) {
    try {
      // code here
      msgContext.getMesssage().setHeader("Accept","application/json");
      return ExecutionResult.SUCCESS;
    } catch (Exception e) {
      return ExecutionResult.ABORT; // raises a fault
    }
  }
}
```

JavaCallout code is harder to maintain and debug than the JavaScript code used with the JavaScript policy.

The Java code is stored in the proxy as a JAR file, which needs to be compiled outside of Apigee.

The Java code is not visible in trace and cannot log to the trace tool console.



JavaCallout policy

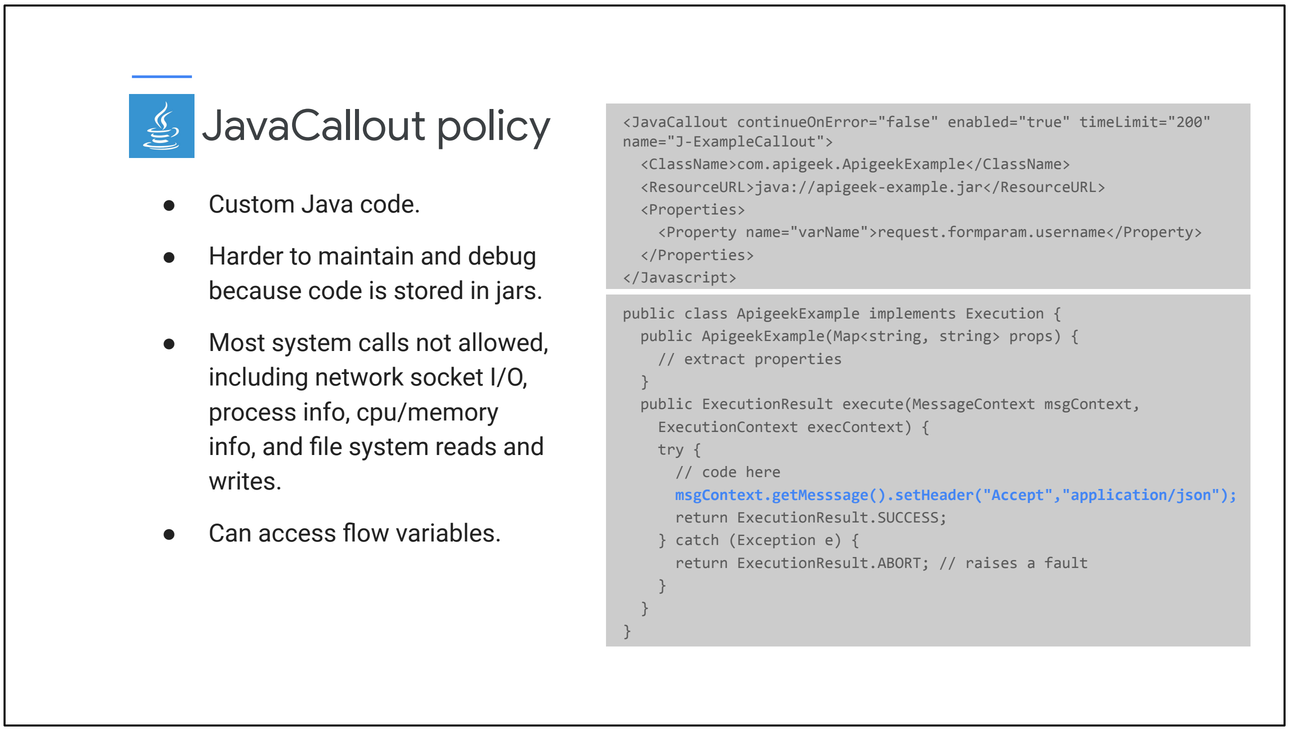
- Custom Java code.
- Harder to maintain and debug because code is stored in jars.
- Most system calls not allowed, including network socket I/O, process info, cpu/memory info, and file system reads and writes.

```
<JavaCallout continueOnError="false" enabled="true" timeLimit="200"
name="J-ExampleCallout">
  <ClassName>com.apigeek.ApigeekExample</ClassName>
  <ResourceURL>java://apigeek-example.jar</ResourceURL>
  <Properties>
    <Property name="varName">request.formparam.username</Property>
  </Properties>
</Javascript>
```

```
public class ApigeekExample implements Execution {
  public ApigeekExample(Map<string, string> props) {
    // extract properties
  }
  public ExecutionResult execute(MessageContext msgContext,
    ExecutionContext execContext) {
    try {
      // code here
      msgContext.getMesssage().setHeader("Accept","application/json");
      return ExecutionResult.SUCCESS;
    } catch (Exception e) {
      return ExecutionResult.ABORT; // raises a fault
    }
  }
}
```

Because this is running within the context of the gateway, most system calls are not allowed.

Your code cannot perform network socket i/o, get information about the process or CPU or memory usage, or write to the filesystem.



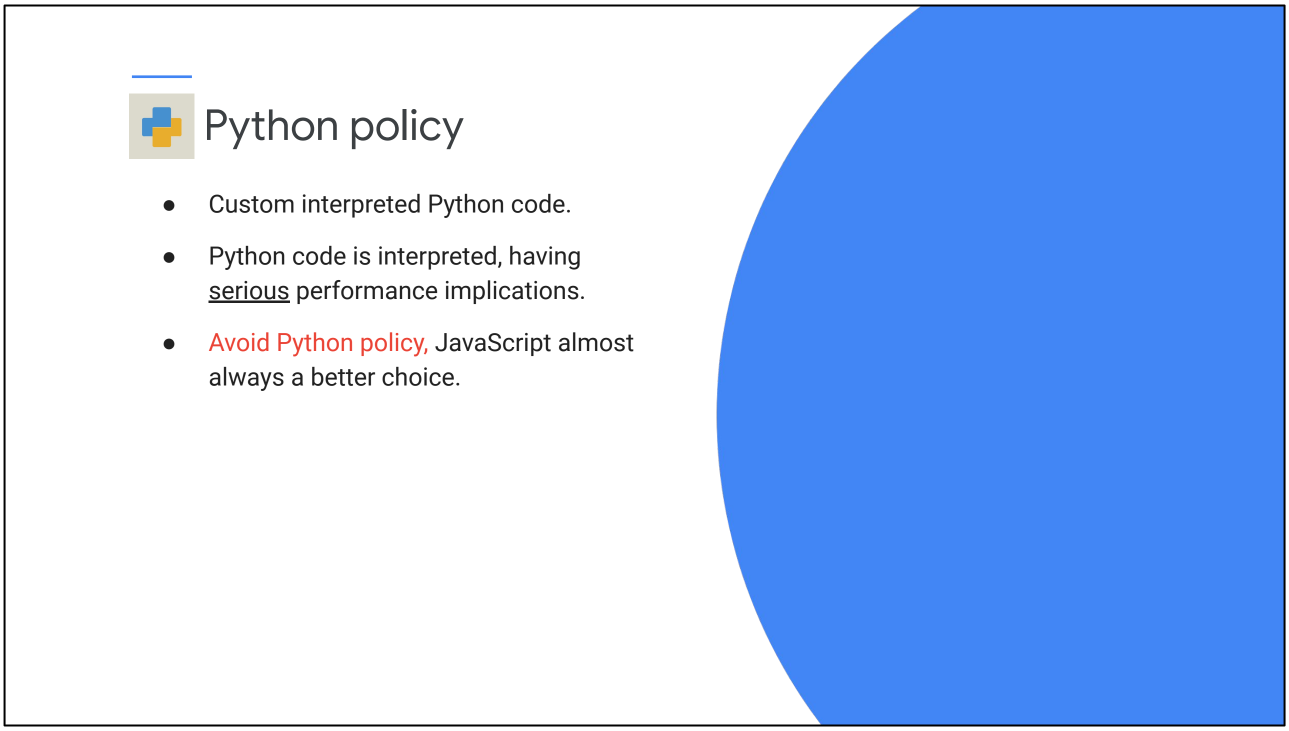
JavaCallout policy

- Custom Java code.
- Harder to maintain and debug because code is stored in jars.
- Most system calls not allowed, including network socket I/O, process info, cpu/memory info, and file system reads and writes.
- Can access flow variables.

```
<JavaCallout continueOnError="false" enabled="true" timeLimit="200"
name="J-ExampleCallout">
  <ClassName>com.apigeek.ApigeekExample</ClassName>
  <ResourceURL>java://apigeek-example.jar</ResourceURL>
  <Properties>
    <Property name="varName">request.formparam.username</Property>
  </Properties>
</Javascript>
```

```
public class ApigeekExample implements Execution {
  public ApigeekExample(Map<string, string> props) {
    // extract properties
  }
  public ExecutionResult execute(MessageContext msgContext,
    ExecutionContext execContext) {
    try {
      // code here
      msgContext.getMesssage().setHeader("Accept","application/json");
      return ExecutionResult.SUCCESS;
    } catch (Exception e) {
      return ExecutionResult.ABORT; // raises a fault
    }
  }
}
```

You can create, update, and remove flow variables and message elements.



Python policy

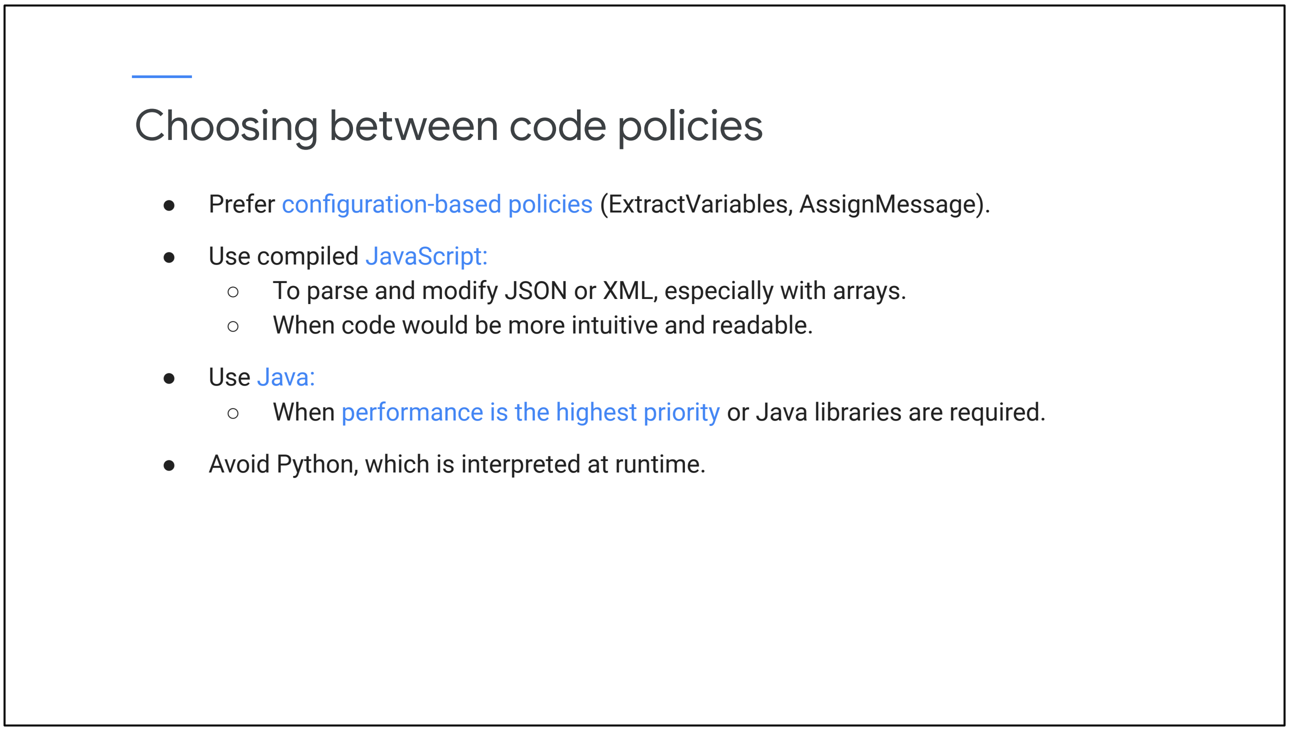
- Custom interpreted Python code.
- Python code is interpreted, having serious performance implications.
- Avoid Python policy, JavaScript almost always a better choice.

Apigee also has a policy that can execute Python code.

The Python policy runs interpreted Python code inside the Java virtual machine and has functionality similar to the JavaScript policy, including flow variable access.

However, since Python is interpreted at runtime, you can run into serious performance problems under load.

We recommend that you avoid the Python policy, especially for any high-traffic proxies. JavaScript can be used to solve the same problems and is almost always a better choice.



Choosing between code policies

- Prefer configuration-based policies (ExtractVariables, AssignMessage).
- Use compiled JavaScript:
  - To parse and modify JSON or XML, especially with arrays.
  - When code would be more intuitive and readable.
- Use Java:
  - When performance is the highest priority or Java libraries are required.
- Avoid Python, which is interpreted at runtime.

So, how do you choose which code policy to use, if any?

First, if you can use the configuration-based policies like ExtractVariables and AssignMessage, you probably should. These policies tend to perform better than custom code.

The second choice is the JavaScript policy.

JavaScript is pre-compiled, so it performs well.

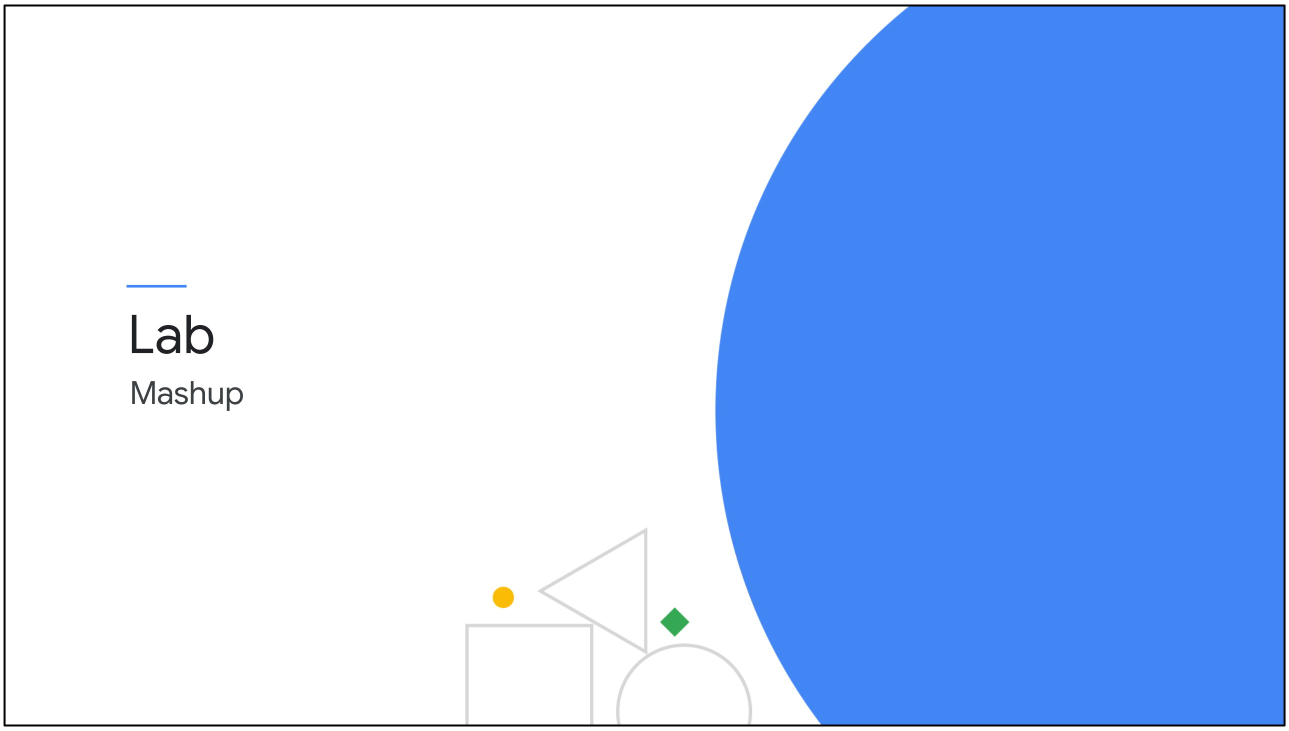
It is easy to write small bits of code to do things like parsing and modifying JSON and XML.

Use JavaScript for problems that can't be solved easily with ExtractVariables and AssignMessage. You should use JavaScript for most custom code use cases.

Java should be used when performance is the highest priority. If you are doing heavy, compute-intensive cryptography, for example, Java might be the right choice.

You also might use the JavaCallout when you have Java libraries that perform the functionality you are adding to your proxy.

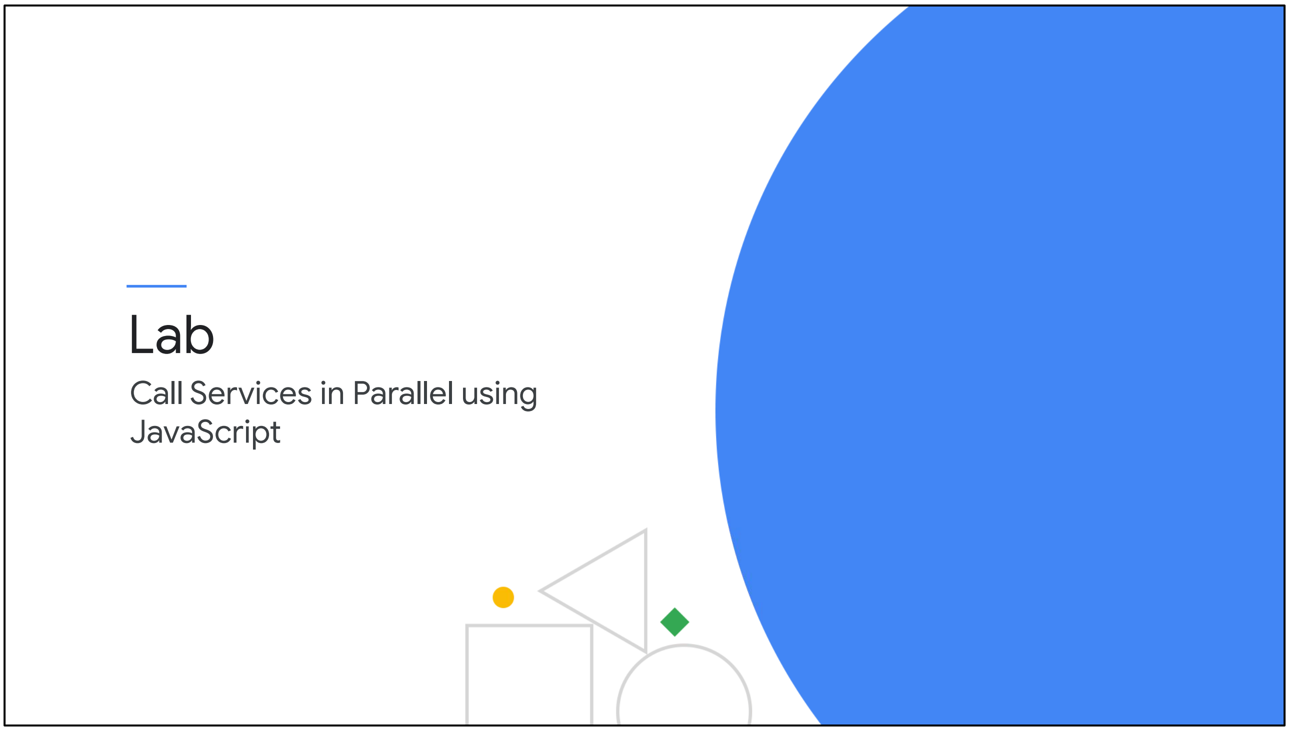
Finally, you should avoid the Python policy if at all possible. Python is interpreted and can be slow. Performance can especially suffer during heavy load.



Lab

Mashup

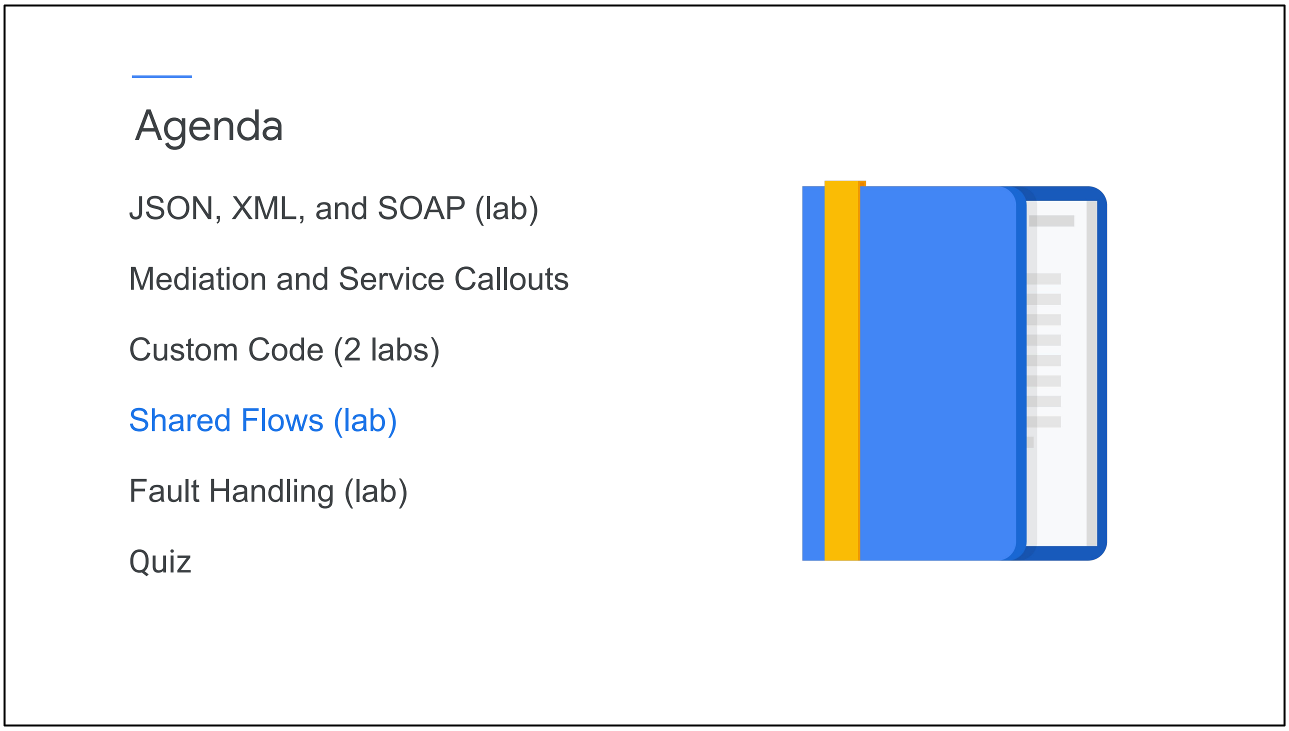
In this lab you add additional data to your API response by calling a geocoding API and extracting an address from its response. You will add this address to your response payload, thus adding a feature to your API without modifying the backend service.



Lab

Call Services in Parallel using JavaScript

In this lab you create a new proxy that uses a JavaScript policy to call services in parallel.



Agenda

JSON, XML, and SOAP (lab)

Mediation and Service Callouts

Custom Code (2 labs)

Shared Flows (lab)

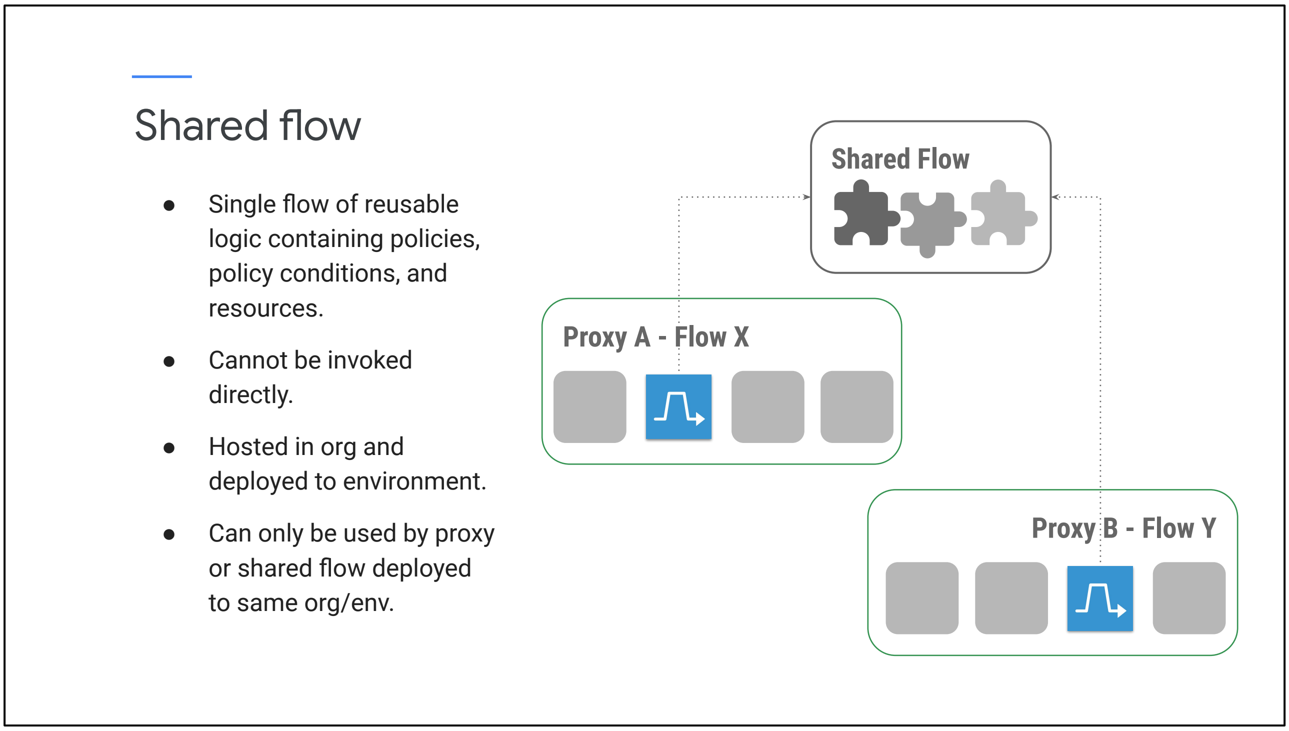
Fault Handling (lab)

Quiz

In a previous section we talked about proxy chaining, a technique for sharing code by calling into another proxy within the same organization and environment.

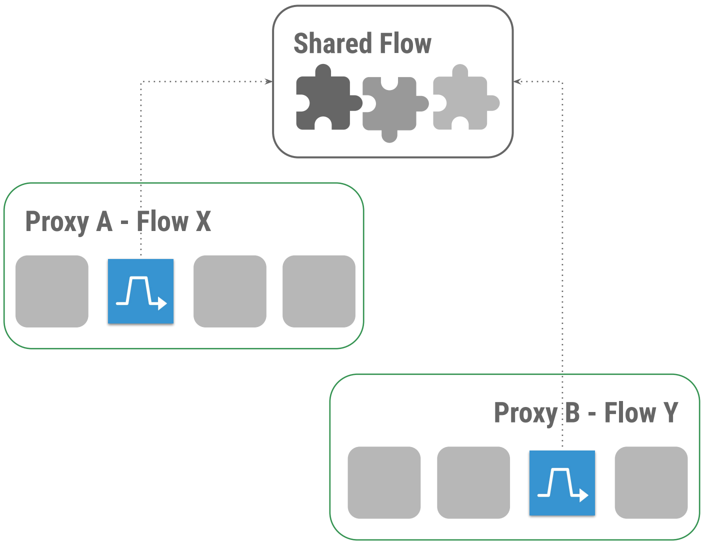
In this section we'll discuss shared flows, another method for sharing code inside proxies.

We will end this section with a lab.



Shared flow

- Single flow of reusable logic containing policies, policy conditions, and resources.
- Cannot be invoked directly.
- Hosted in org and deployed to environment.
- Can only be used by proxy or shared flow deployed to same org/env.



Proxy chaining is calling another proxy from within a proxy to share code. To call a proxy, you have to build an HTTP request message, and you get back an HTTP response message that you have to parse.

A shared flow is a single flow of policies, policy conditions, and resources.

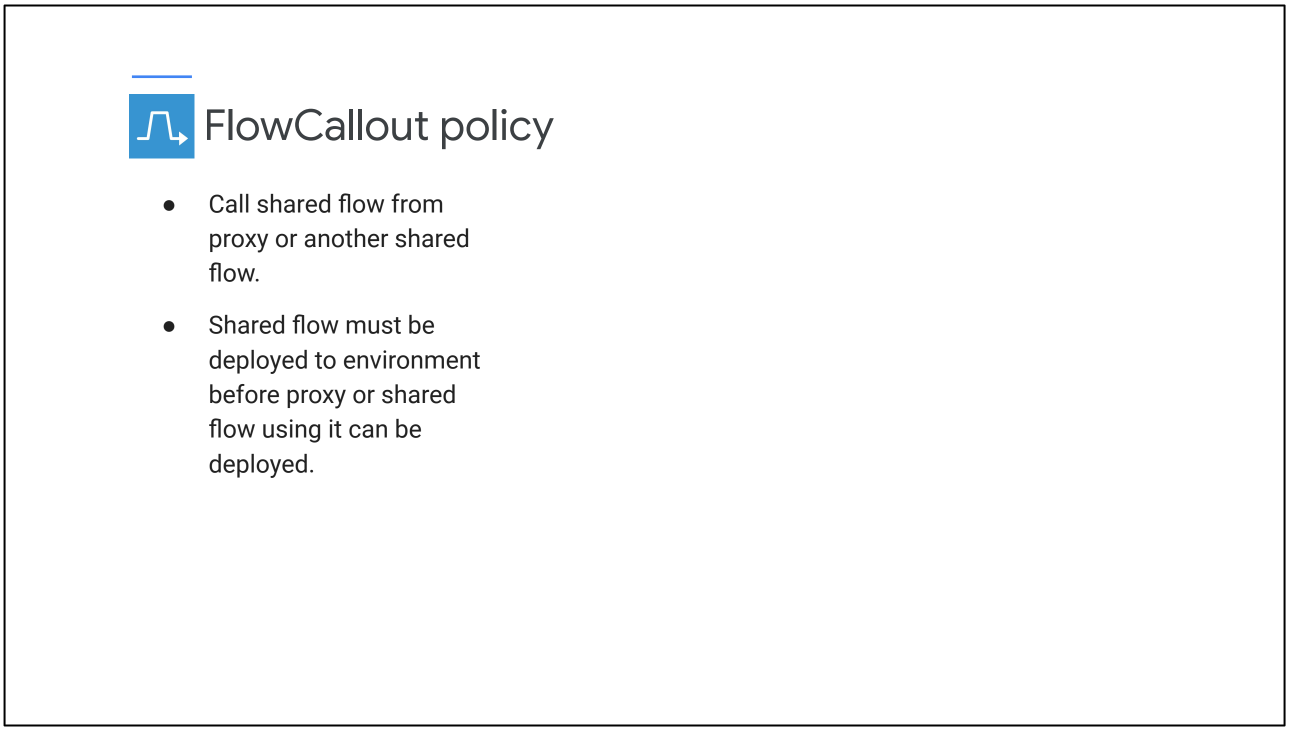
The shared flow executes within the context of the hosting proxy's flow. The shared flow has read and write access to the proxy's flow variables, so there is no need to build and parse messages to use a shared flow. When calling a shared flow, the proxy executes as if the policies in the shared flow are in the proxy flow.

Shared flows are typically better than proxy chaining when sharing reusable logic between proxies.

Shared flows live in the context of a hosting proxy's flow, so there is no way to invoke a shared flow directly. To test a shared flow, you need to call it from a proxy.

A shared flow is hosted in an organization and can be deployed to an environment. This is the same way proxies work.

A proxy deployed to an organization and environment can only call shared flows deployed to the same organization and environment.



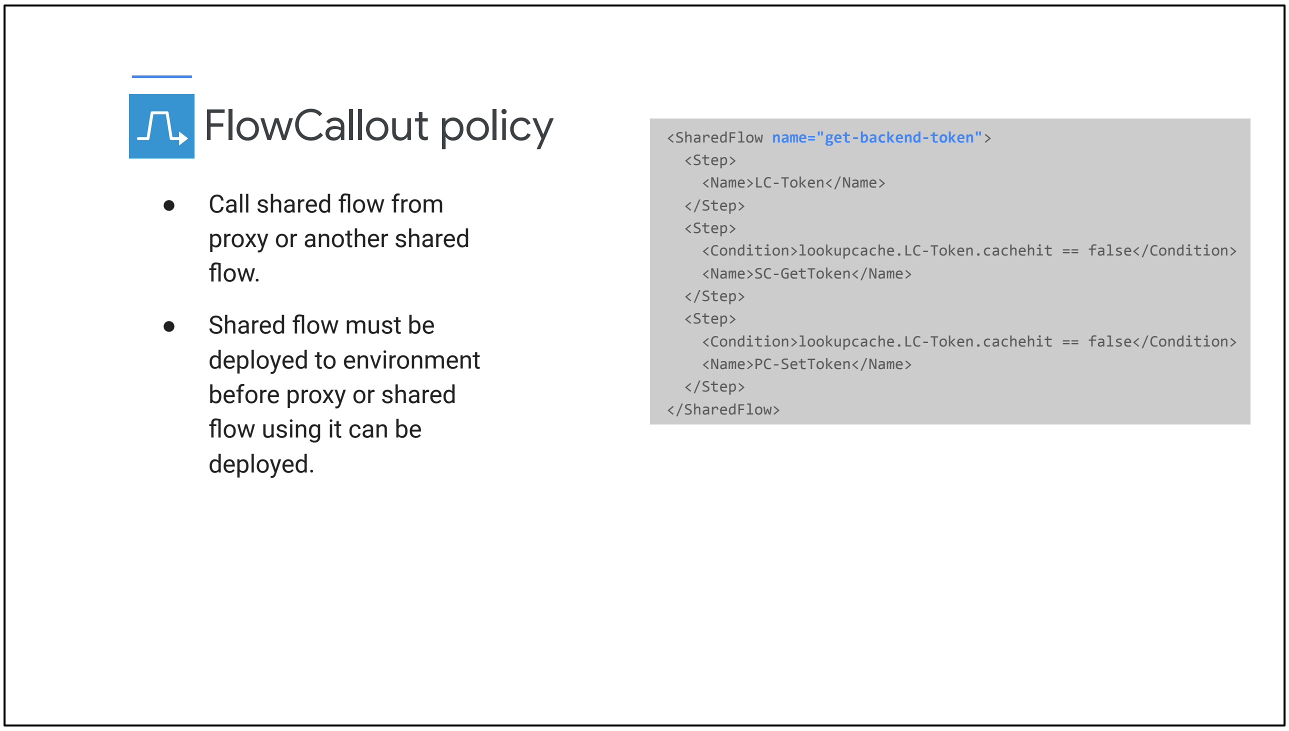
FlowCallout policy

- Call shared flow from proxy or another shared flow.
- Shared flow must be deployed to environment before proxy or shared flow using it can be deployed.

Shared flows are invoked by using the FlowCallout policy.

The FlowCallout policy can be used in proxies or in other shared flows.

A shared flow must be deployed to an environment before a proxy or shared flow using the FlowCallout policy can be deployed.



FlowCallout policy

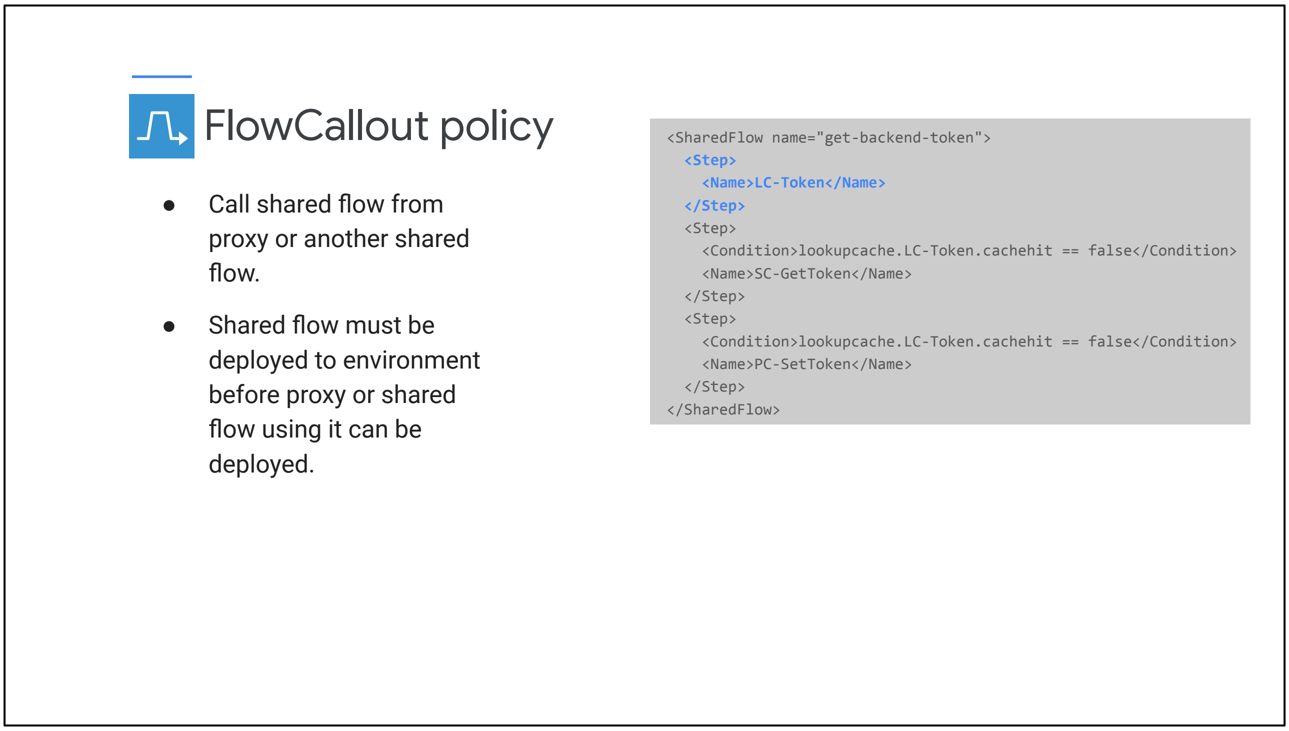
- Call shared flow from proxy or another shared flow.
- Shared flow must be deployed to environment before proxy or shared flow using it can be deployed.

```
<SharedFlow name="get-backend-token">
  <Step>
    <Name>LC-Token</Name>
  </Step>
  <Step>
    <Condition>lookupcache.LC-Token.cachehit == false</Condition>
    <Name>SC-GetToken</Name>
  </Step>
  <Step>
    <Condition>lookupcache.LC-Token.cachehit == false</Condition>
    <Name>PC-SetToken</Name>
  </Step>
</SharedFlow>
```

Let's look at an example.

Here is the XML configuration for a shared flow named "get backend token."

The steps are at the top level of the XML hierarchy.



FlowCallout policy

- Call shared flow from proxy or another shared flow.
- Shared flow must be deployed to environment before proxy or shared flow using it can be deployed.

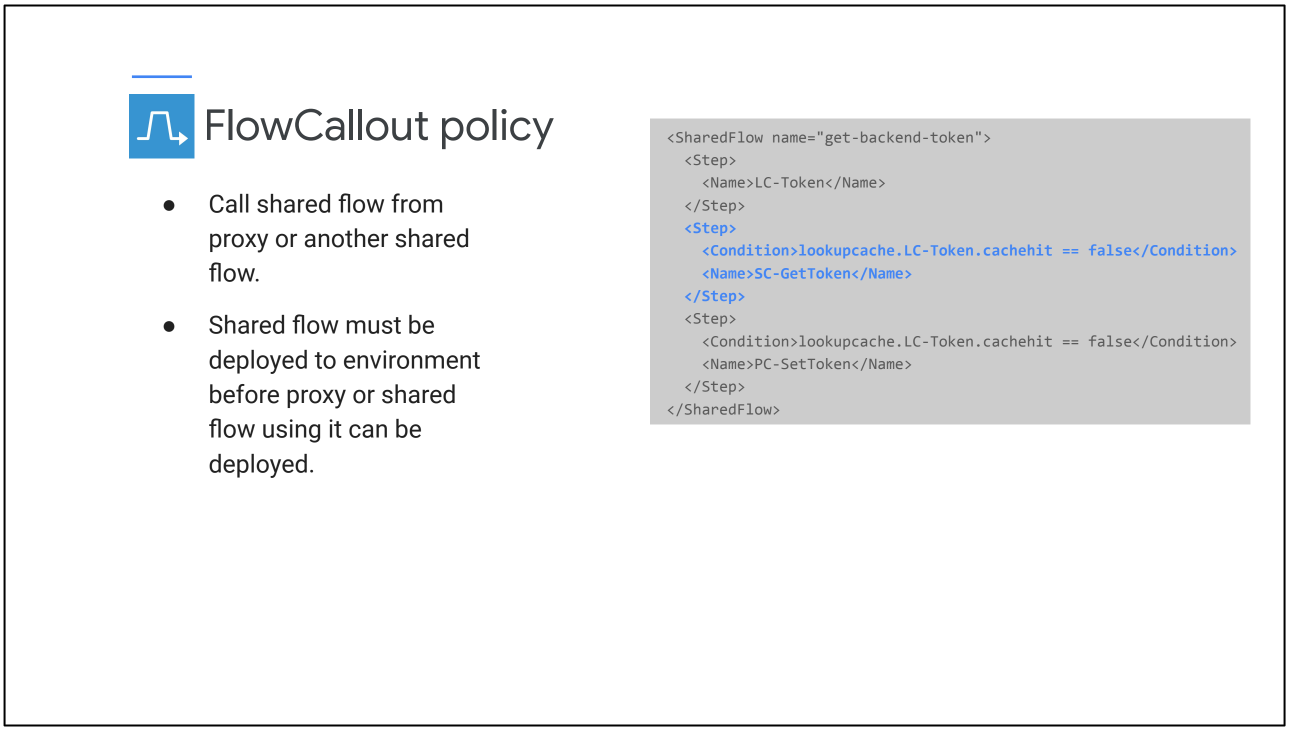
```
<SharedFlow name="get-backend-token">
  <Step>
    <Name>LC-Token</Name>
  </Step>
  <Step>
    <Condition>lookupcache.LC-Token.cachehit == false</Condition>
    <Name>SC-GetToken</Name>
  </Step>
  <Step>
    <Condition>lookupcache.LC-Token.cachehit == false</Condition>
    <Name>PC-SetToken</Name>
  </Step>
</SharedFlow>
```

The first step is a policy named LC-Token. LC stands for lookup cache.

We'll discuss more about the cache policies later, but this policy is just looking for a token in the cache.

When the token was put into the cache, it was configured with a time-to-live that expired shortly before the token itself expired.

If the token is in the cache, it should be active and should be used.



FlowCallout policy

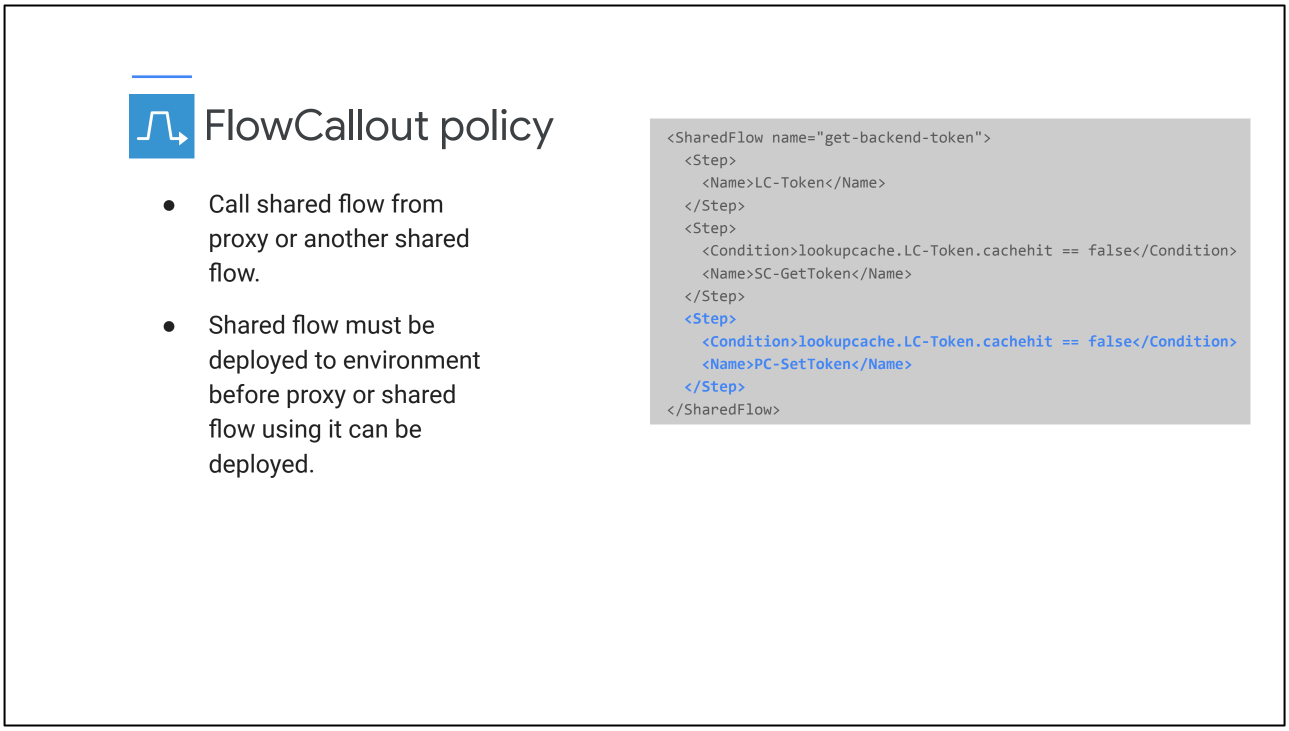
- Call shared flow from proxy or another shared flow.
- Shared flow must be deployed to environment before proxy or shared flow using it can be deployed.

```
<SharedFlow name="get-backend-token">
  <Step>
    <Name>LC-Token</Name>
  </Step>
  <Step>
    <Condition>lookupcache.LC-Token.cachehit == false</Condition>
    <Name>SC-GetToken</Name>
  </Step>
  <Step>
    <Condition>lookupcache.LC-Token.cachehit == false</Condition>
    <Name>PC-SetToken</Name>
  </Step>
</SharedFlow>
```

The second step is a policy named SC-GetToken. SC stands for service callout, so this is a call to get a new backend token from an external service.

Note that this step has a condition. The step will only be executed if the condition evaluates to true.

The condition is checking a variable called lookupcache.LC-Token.cachehit. This variable was set by the lookup cache policy. If this variable is false, the token was not found in the cache. If the value is true, this step will be skipped.



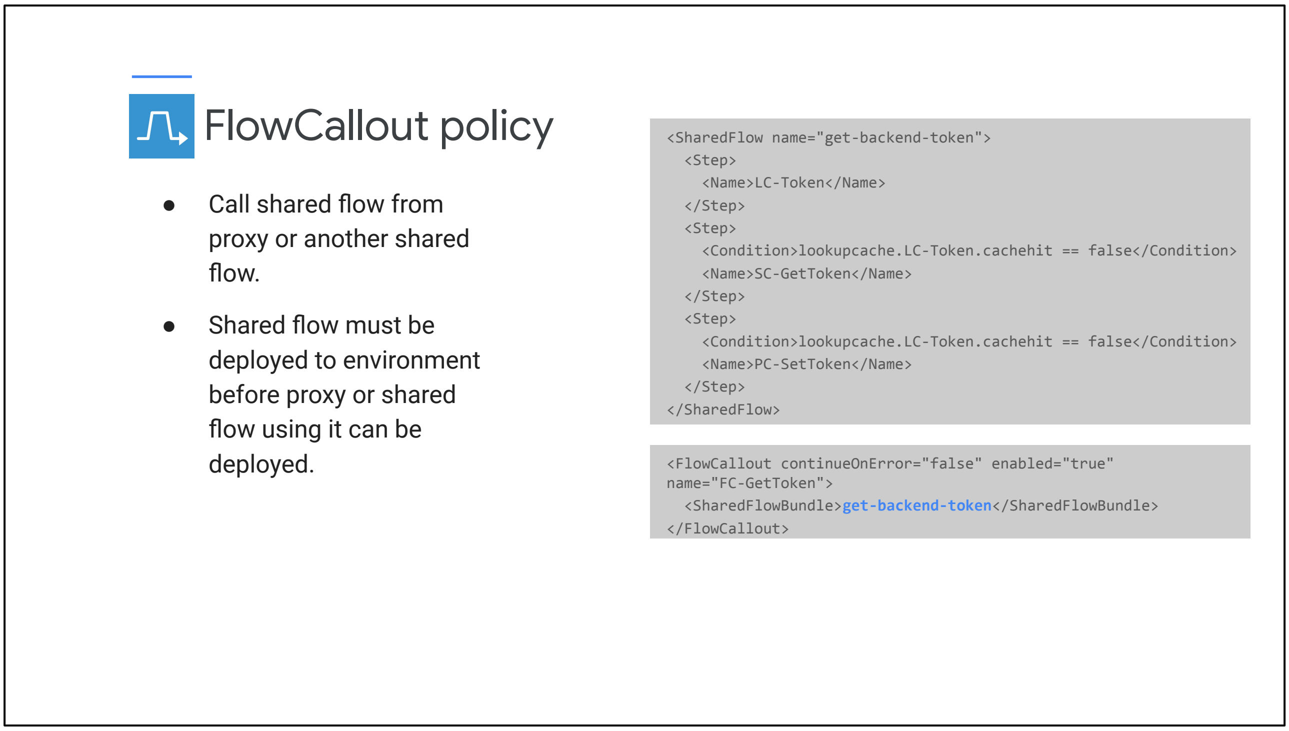
FlowCallout policy

- Call shared flow from proxy or another shared flow.
- Shared flow must be deployed to environment before proxy or shared flow using it can be deployed.

```
<SharedFlow name="get-backend-token">
  <Step>
    <Name>LC-Token</Name>
  </Step>
  <Step>
    <Condition>lookupcache.LC-Token.cachehit == false</Condition>
    <Name>SC-GetToken</Name>
  </Step>
  <Step>
    <Condition>lookupcache.LC-Token.cachehit == false</Condition>
    <Name>PC-SetToken</Name>
  </Step>
</SharedFlow>
```

The third step is a policy called PC-SetToken. PC stands for populate cache, so this policy stores the token we got from the ServiceCallout into the cache and makes it available for the next API call.

The condition is the same as for the previous step, so this policy is only run when the token wasn't found in the cache originally.



FlowCallout policy

- Call shared flow from proxy or another shared flow.
- Shared flow must be deployed to environment before proxy or shared flow using it can be deployed.

```
<SharedFlow name="get-backend-token">
  <Step>
    <Name>LC-Token</Name>
  </Step>
  <Step>
    <Condition>lookupcache.LC-Token.cachehit == false</Condition>
    <Name>SC-GetToken</Name>
  </Step>
  <Step>
    <Condition>lookupcache.LC-Token.cachehit == false</Condition>
    <Name>PC-SetToken</Name>
  </Step>
</SharedFlow>
```

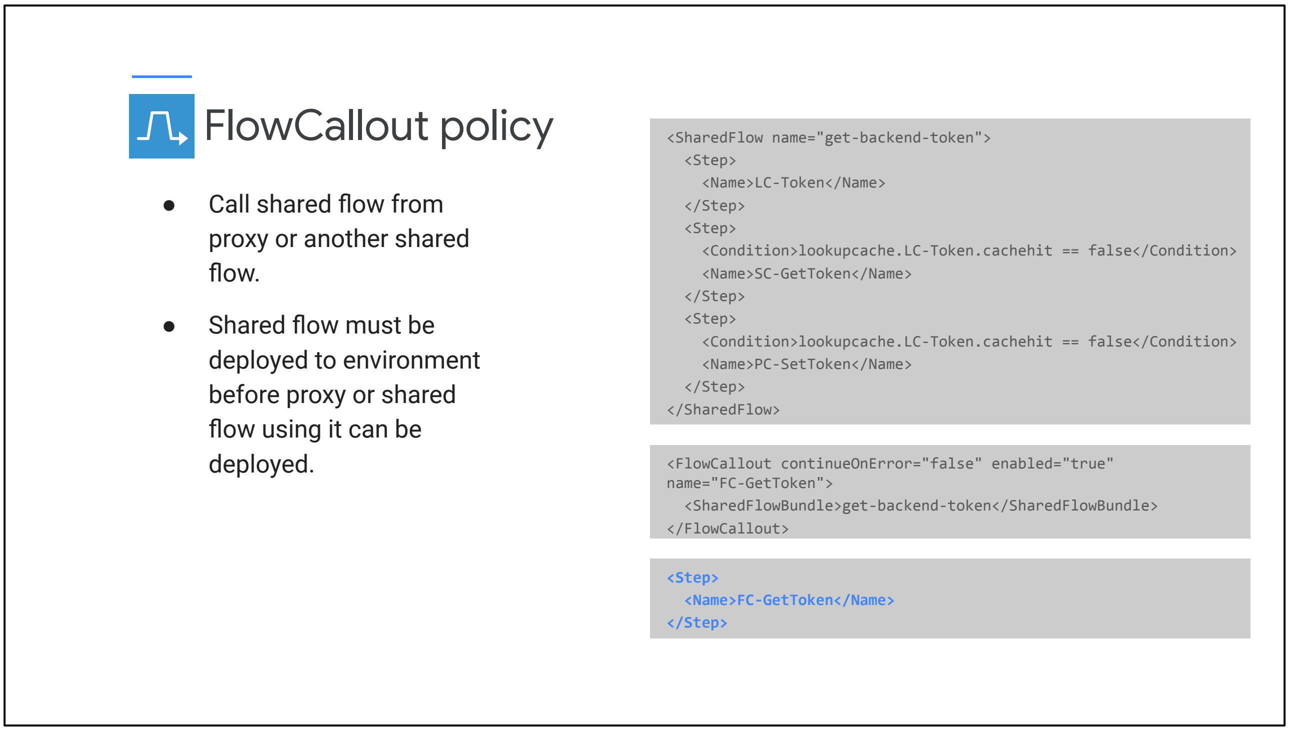
```
<FlowCallout continueOnError="false" enabled="true"
name="FC-GetToken">
  <SharedFlowBundle>get-backend-token</SharedFlowBundle>
</FlowCallout>
```

The FlowCallout policy is simple.

The policy is named FC-GetToken, with FC standing for Flow Callout.

The SharedFlowBundle element contains the name of the shared flow to call.

You could also add parameter elements to create variables that are only visible during execution of the shared flow.



FlowCallout policy

- Call shared flow from proxy or another shared flow.
- Shared flow must be deployed to environment before proxy or shared flow using it can be deployed.

```
<SharedFlow name="get-backend-token">
  <Step>
    <Name>LC-Token</Name>
  </Step>
  <Step>
    <Condition>lookupcache.LC-Token.cachehit == false</Condition>
    <Name>SC-GetToken</Name>
  </Step>
  <Step>
    <Condition>lookupcache.LC-Token.cachehit == false</Condition>
    <Name>PC-SetToken</Name>
  </Step>
</SharedFlow>
```

```
<FlowCallout continueOnError="false" enabled="true"
name="FC-GetToken">
  <SharedFlowBundle>get-backend-token</SharedFlowBundle>
</FlowCallout>
```

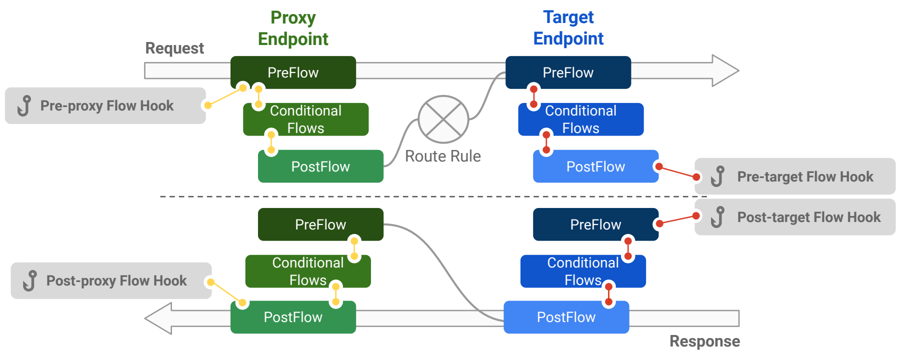
```
<Step>
  <Name>FC-GetToken</Name>
</Step>
```

Within the hosting proxy's flow, the flow callout policy can be called as you would call any other policy. The functionality would be the same as if you included the three shared flow steps directly in the proxy.

When you trace the proxy, you have full access to the trace information for the shared flow steps.

Flow hooks

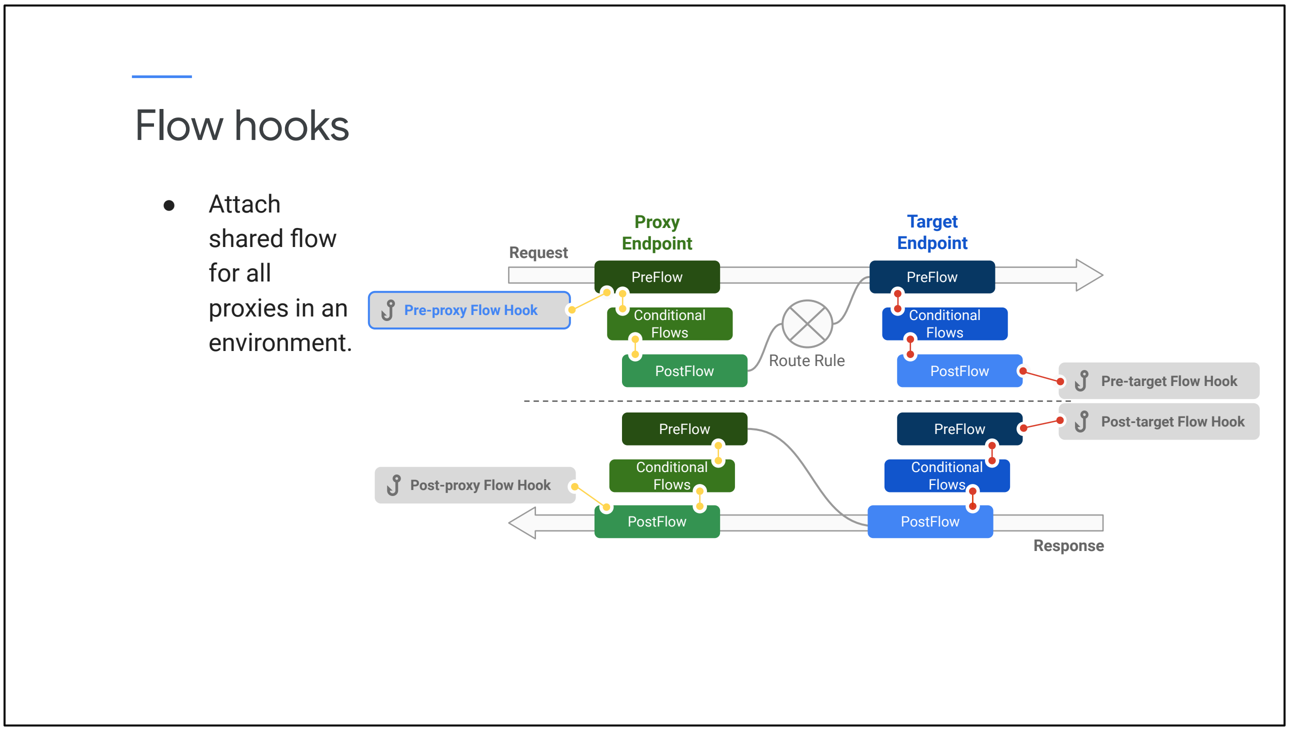
- Attach shared flow for all proxies in an environment.



Another way to attach shared flows to a proxy is to use flow hooks.

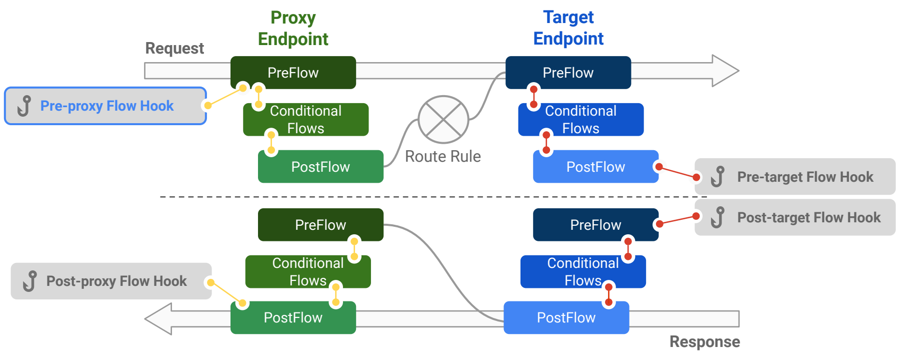
Shared flows attached to flow hooks will execute for all proxies deployed in the environment.

There are four flow hooks.



Flow hooks

- Attach shared flow for all proxies in an environment.

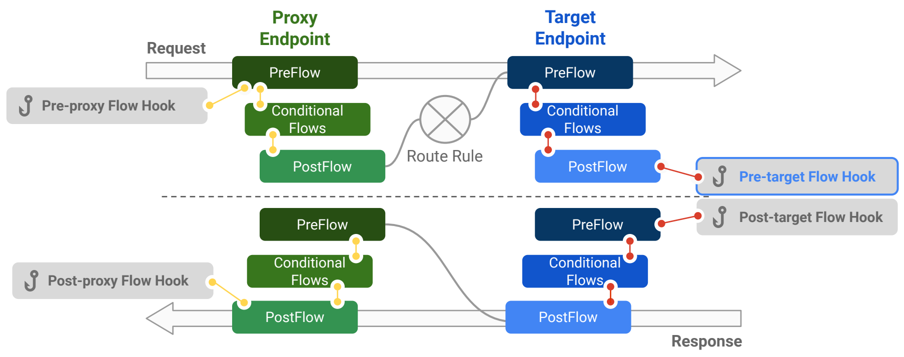


The Pre-proxy flow hook is used for logic that needs to be enforced before the proxy executes.

This might be used to execute security policies before letting a call get into the proxy.

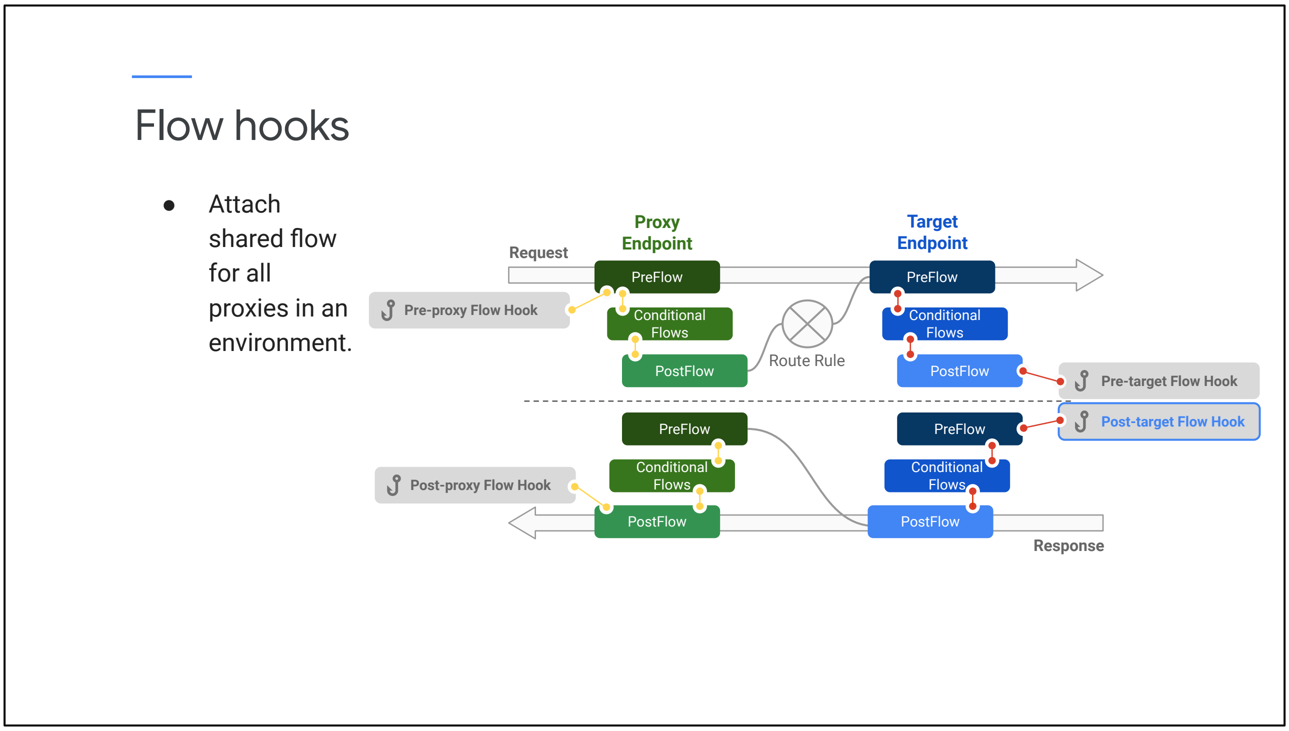
Flow hooks

- Attach shared flow for all proxies in an environment.



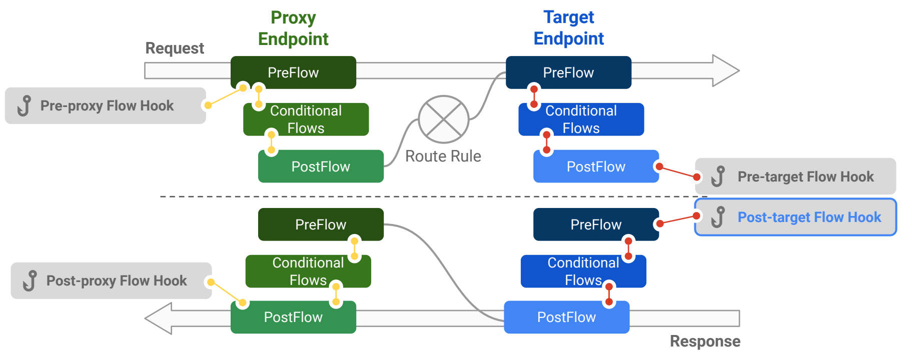
The Pre-target flow hook is used to attach logic after the target endpoint request completes but before the target is called.

You might use the Pre-target flow hook to strip authorization information from the request message before sending it to the backend.



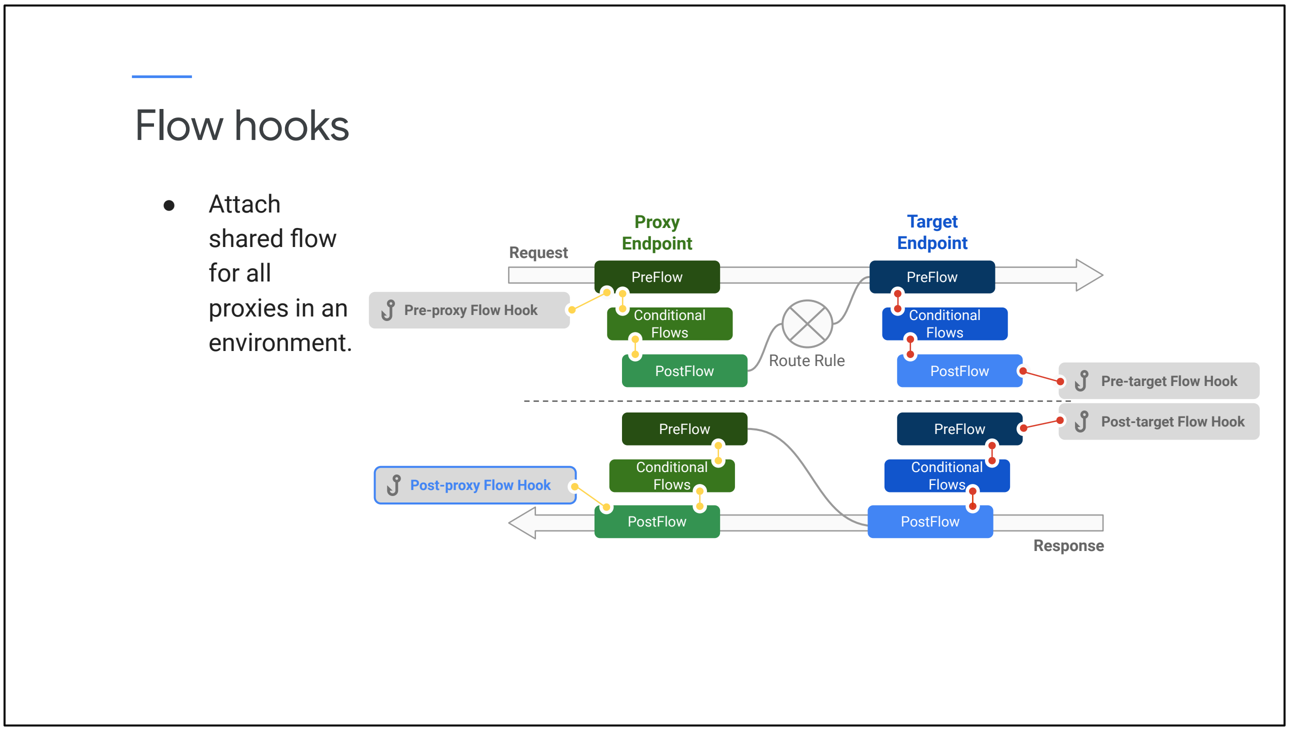
Flow hooks

- Attach shared flow for all proxies in an environment.



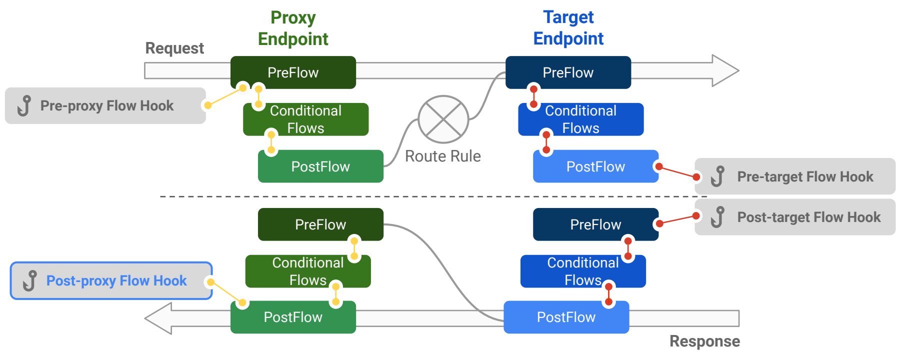
The Post-target flow hook executes after the target response is received but before policies in the target endpoint response are executed.

This might be used to strip sensitive fields from the backend response.



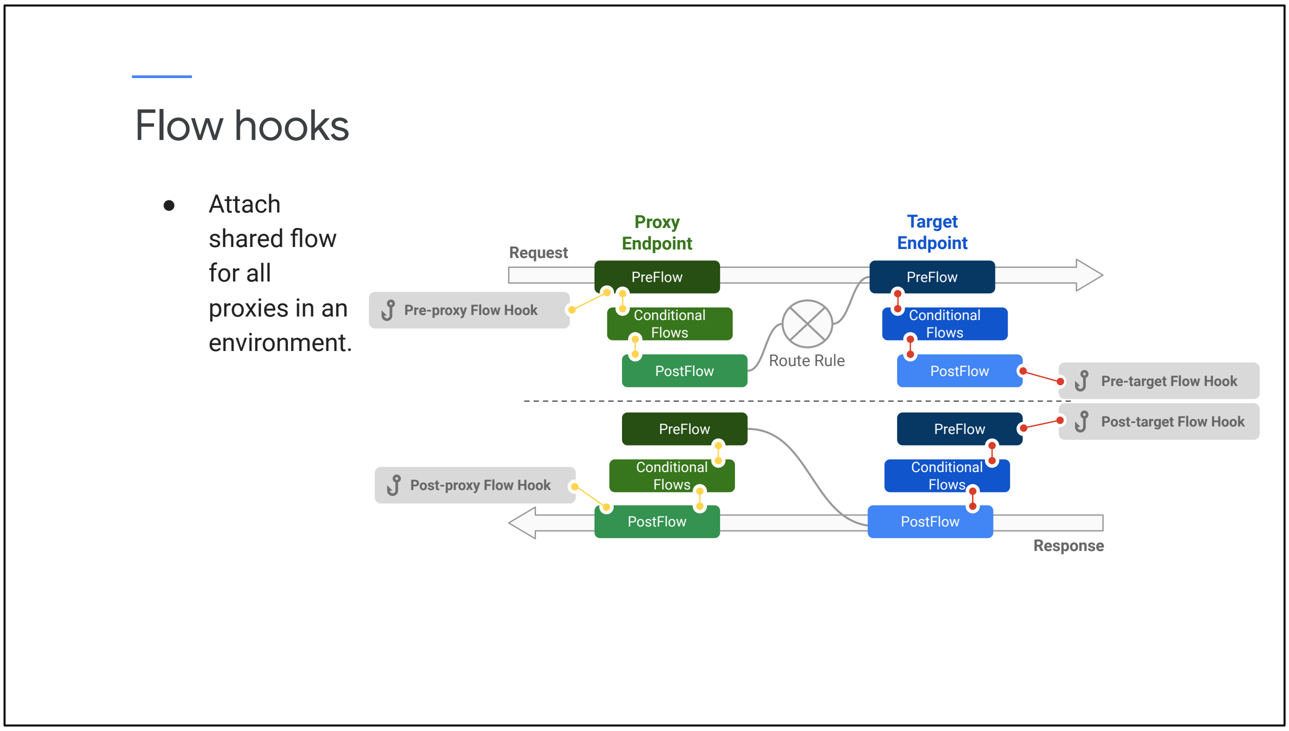
Flow hooks

- Attach shared flow for all proxies in an environment.



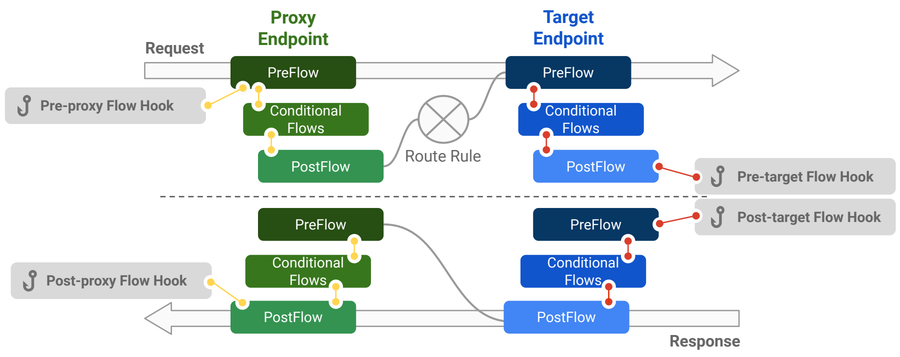
A shared flow attached to the Post-proxy flow hook executes after the proxy endpoint response completes, and just before the response is sent to the client.

This can be used to add response headers or modify the response format in some other way.



Flow hooks

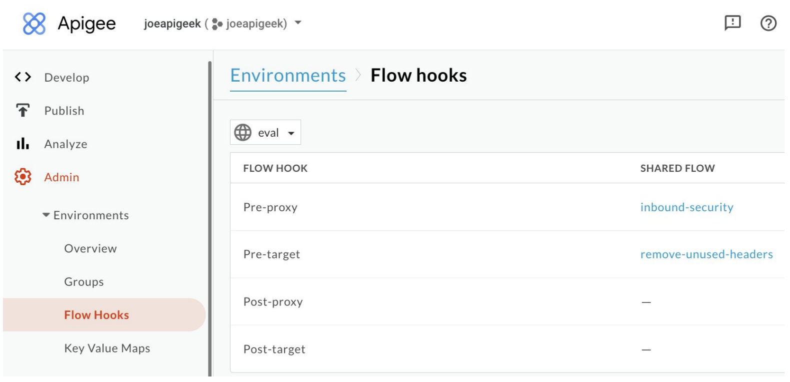
- Attach shared flow for all proxies in an environment.



Flow hooks execute for all proxies deployed to an environment.

There is no option to have a proxy execute without running a configured flow hook, but you can use variables and conditions within the shared flow to conditionally skip the shared flow steps.

Flow hook configuration



FLOW HOOK	SHARED FLOW
Pre-proxy	inbound-security
Pre-target	remove-unused-headers
Post-proxy	—
Post-target	—

- Elevated privileges required to configure flow hooks.
- Only one shared flow per hook, but shared flows can be nested.

Flow hooks are configured in the environment administration section of the Apigee console.

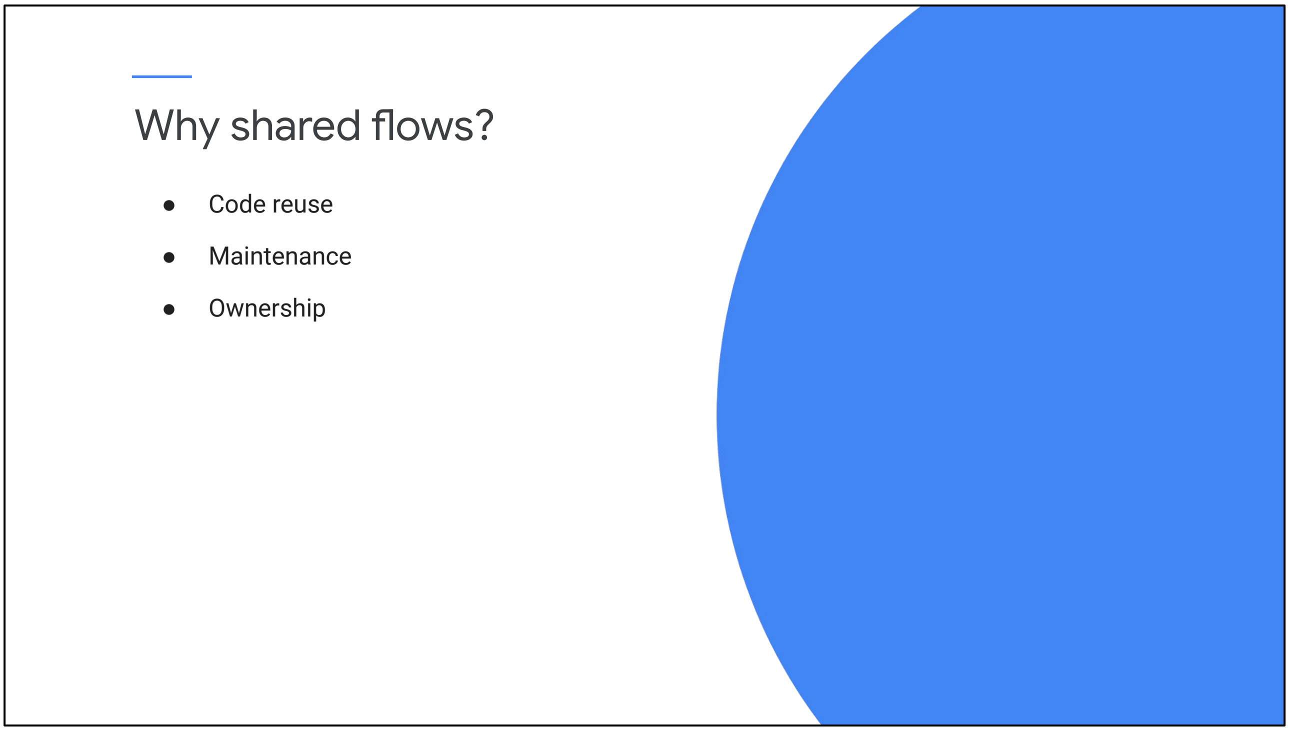
Flow hooks are completely optional; none need to be configured for an environment.

An Organization Admin or Environment Admin can configure flow hooks, but most Apigee roles cannot.

When you configure a flow hook to call a shared flow, that flow will execute for every API proxy deployed in the same environment, so be very careful when configuring flow hooks.

A flow hook can only have a single shared flow configured for it.

Remember that you can nest shared flows if necessary.



Why shared flows?

- Code reuse
- Maintenance
- Ownership

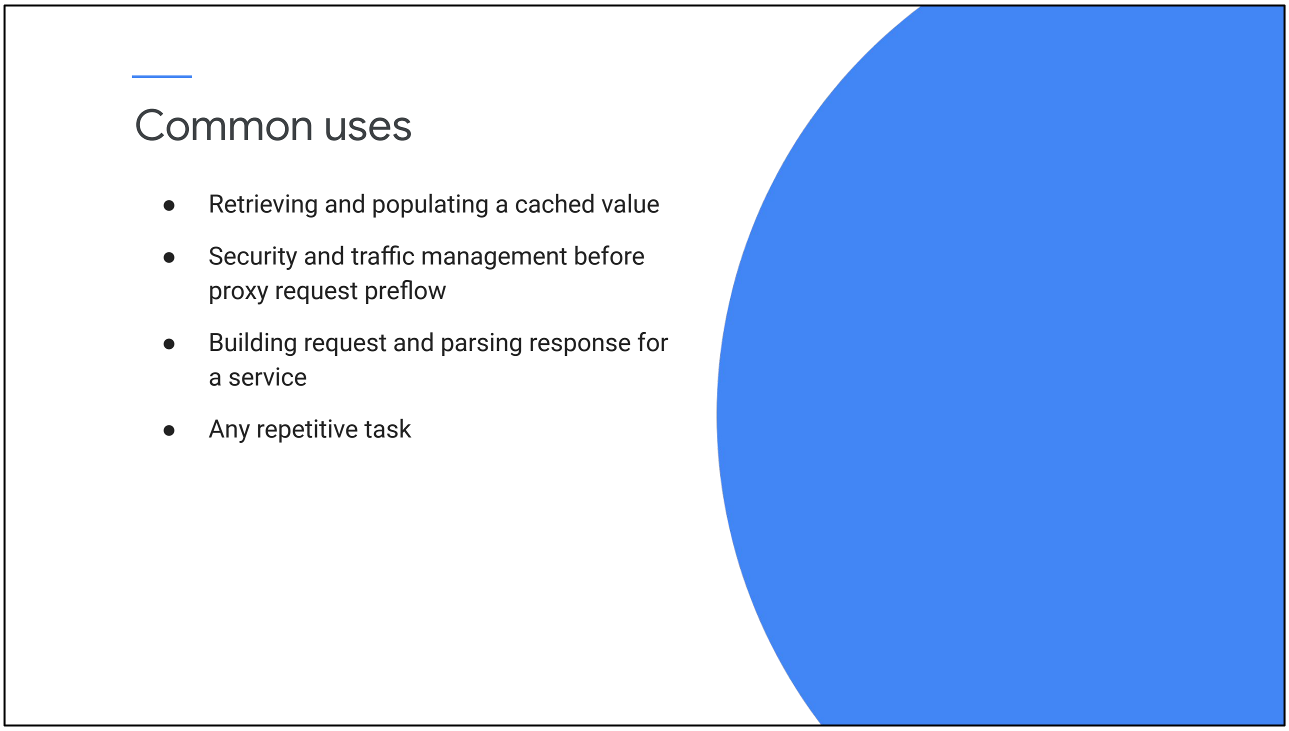
So why should you use shared flows?

We use shared flows for code reuse. It is easier and faster to write and debug a series of steps once rather than doing it for every proxy. Each proxy using the shared flows should perform the task in the same way.

It is also easier to maintain the shared code in one place rather than in several, or even hundreds, of proxies. If the flow needs to be changed, it can be changed in a single place, rather than changing all of the proxies.

For significant changes to a shared flow, we recommend that you test all of the proxies that use the shared flow before making a change to the shared flow in production.

Shared flows can be created to allow one team to own the shared flow and allow other teams to use it. This way you might have your API security team design, build, and maintain a shared flow that protects against malicious requests. As changes need to be made to security functionality, the team best positioned to make those changes can update the shared flow without updating all of the proxies.



Common uses

- Retrieving and populating a cached value
- Security and traffic management before proxy request preflow
- Building request and parsing response for a service
- Any repetitive task

Let's look at some common uses for shared flows.

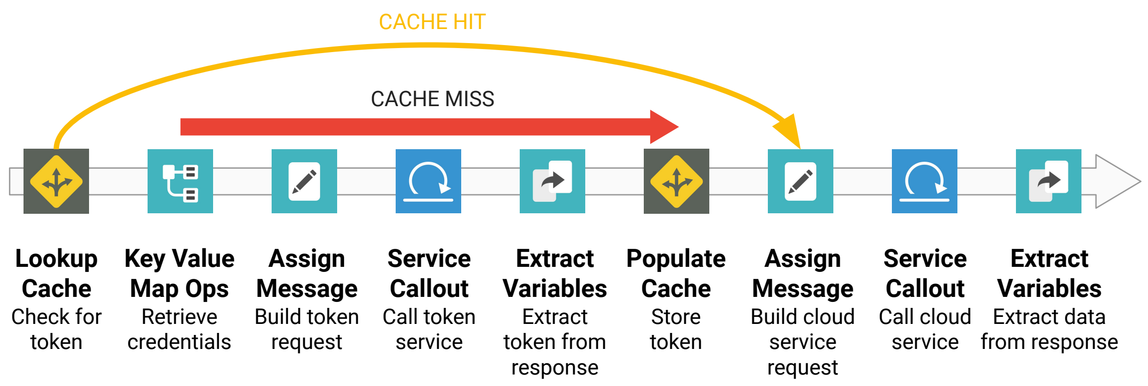
We saw an example of retrieving and populating a cached value earlier. We used a shared flow to get a cached token for a backend service. If the token was not found in the cache, a service callout would call the backend service to get a token, and then the token would be stored in the cache.

Security and traffic management policies that should be run before allowing proxy access can be stored in a shared flow. This can be done in the pre-proxy flow hook if you want to enforce this shared flow for all proxies.

Shared flows can also be used for calling external services, especially if there is a significant amount of work to build the request and parse the response. If your process requires several policies, you greatly reduce the work for your API developers, and you make sure that all proxies are accessing the service in the same way.

Any repetitive, non-trivial task can generally be implemented using a shared flow. Wherever you see a common pattern in your proxies, you should be thinking about shared flows.

Calling a service

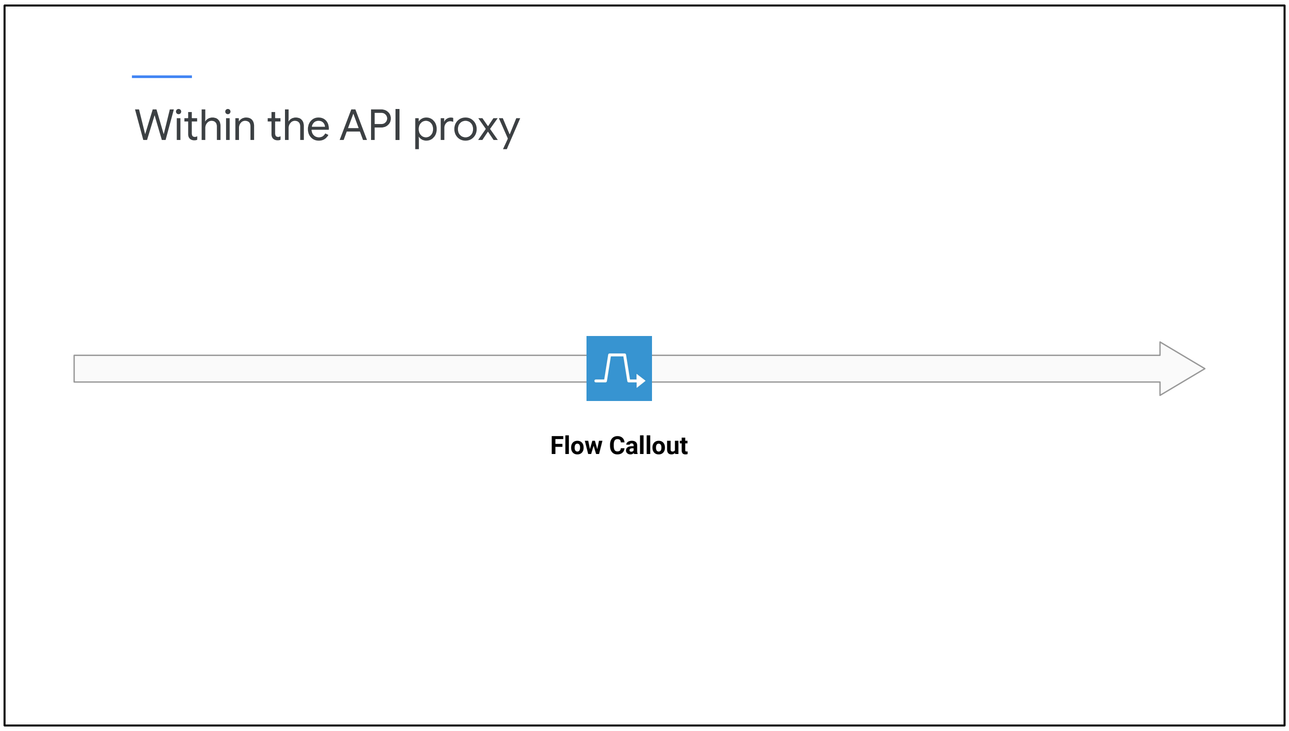


Here's an example of steps that may be required to call an external service.

This sequence first checks a cache for a token. If the token is expired or missing, a request is built to retrieve and cache a new token.

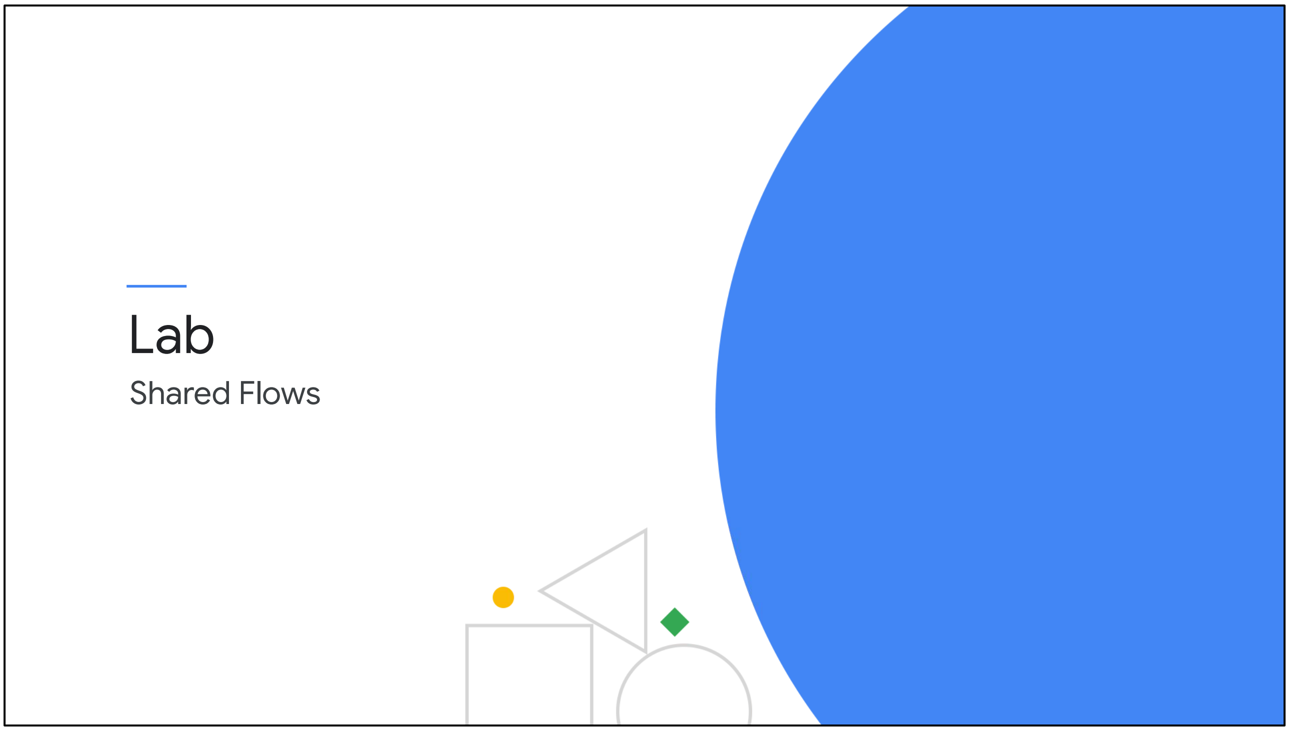
The service is then called with the token, and variables are extracted.

This process may be deployed as a shared flow.



Within the API proxy

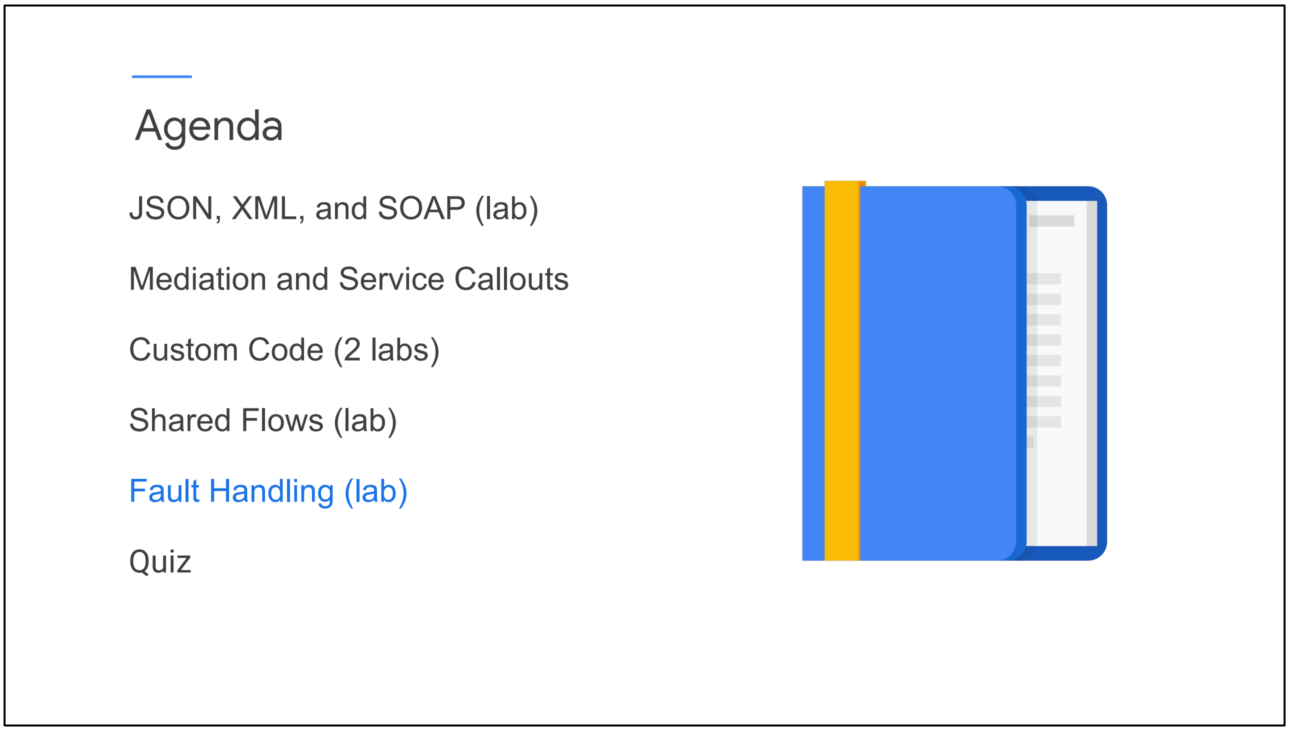
Instead of adding those nine policies to each API proxy, a single flow callout to the shared flow can be used.



Lab

Shared Flows

In this lab you use a shared flow for the logic required when calling the backend. You create a new shared flow which contains the policies to access the encrypted key/value map and build the basic authentication header. In your retail API, you will remove those policies and instead use a flow callout to call this new shared flow.



Agenda

JSON, XML, and SOAP (lab)

Mediation and Service Callouts

Custom Code (2 labs)

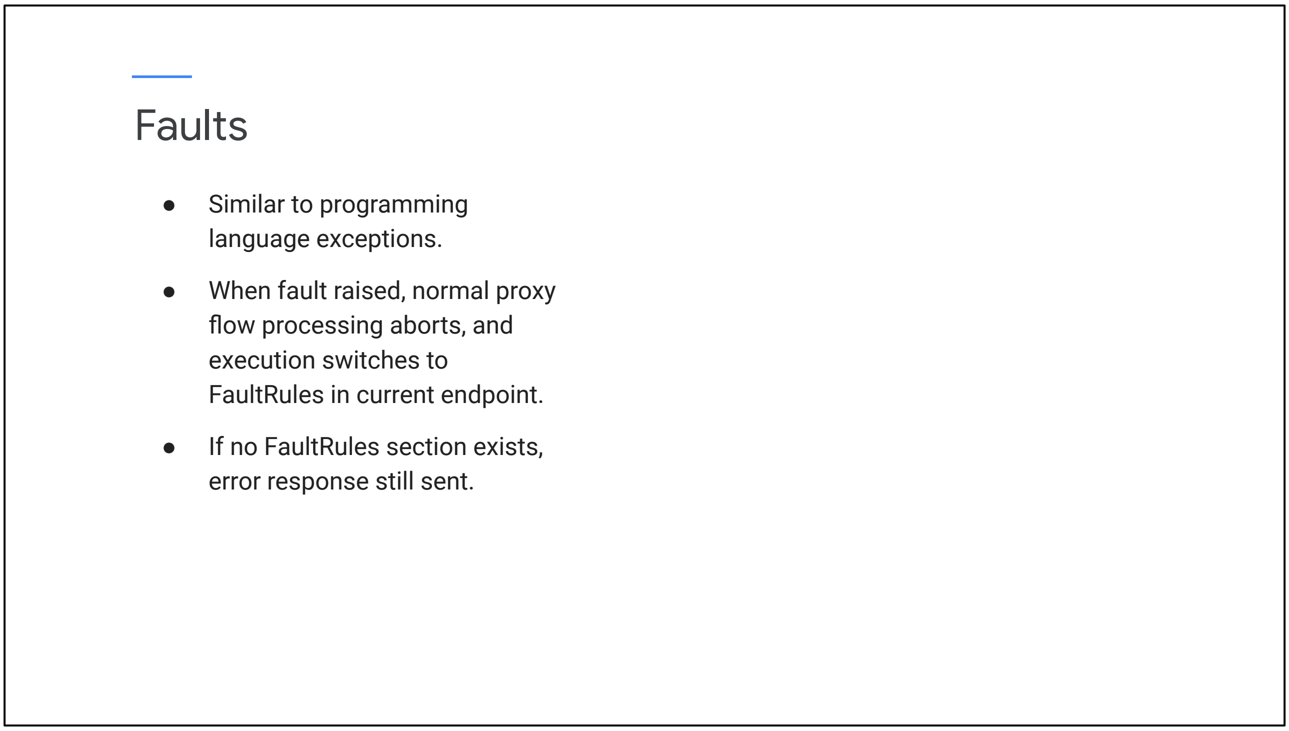
Shared Flows (lab)

Fault Handling (lab)

Quiz

We've been talking about policies raising faults, so now let's understand what that means.

This section will talk about faults, how they are raised, and some best practices for handling faults.



Faults

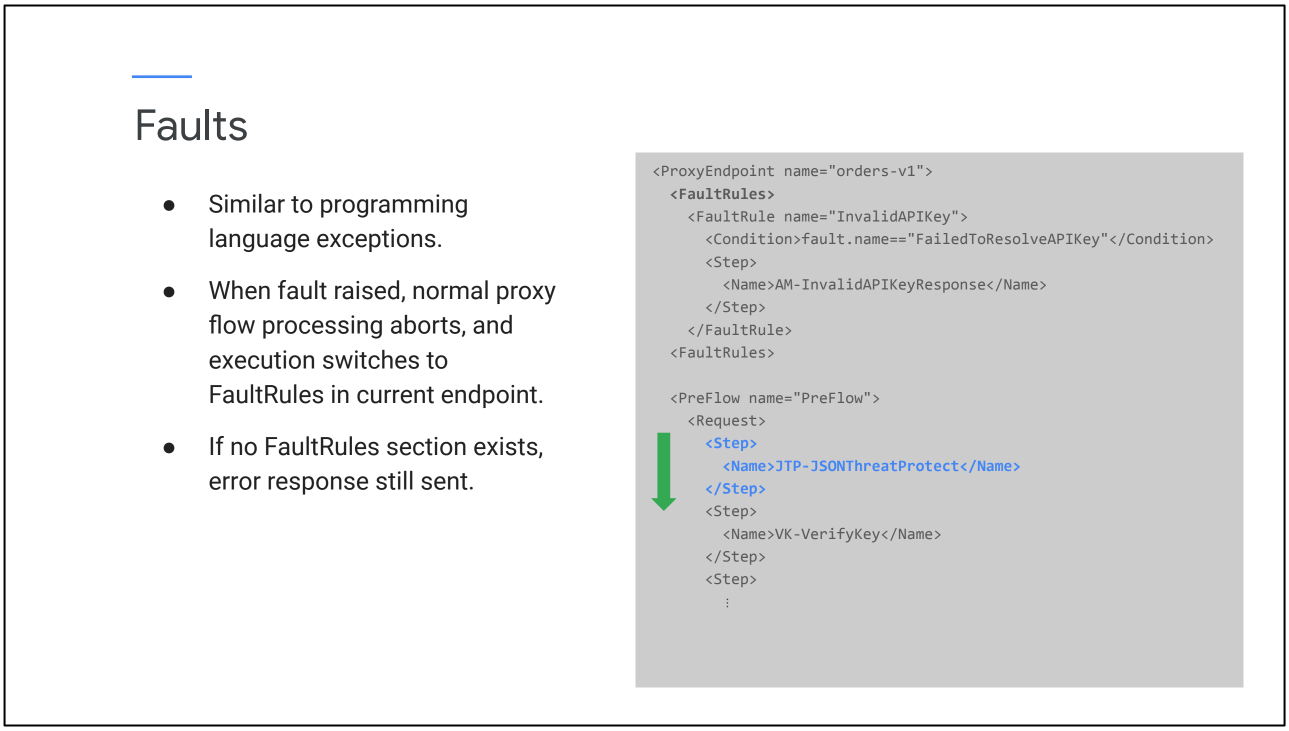
- Similar to programming language exceptions.
- When fault raised, normal proxy flow processing aborts, and execution switches to FaultRules in current endpoint.
- If no FaultRules section exists, error response still sent.

So, what are faults?

Faults are similar to exceptions in other programming languages like Java or JavaScript.

When a fault is raised, the normal flow processing immediately aborts. Proxy execution switches to the FaultRules element in the current endpoint. Processing will never return to the original flow.

Note that the FaultRules element is optional. If no FaultRules element exists, an error response will still be sent to the client. We'll talk about the format for the error response shortly.



Faults

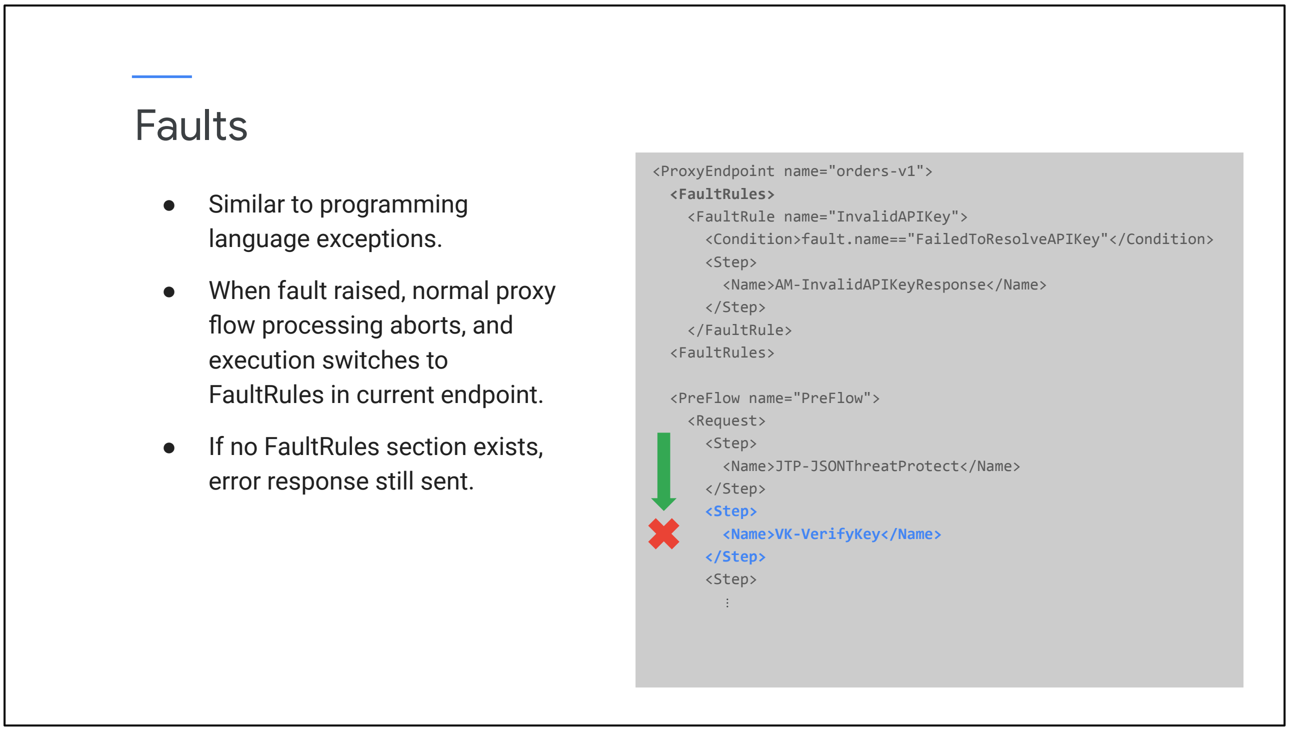
- Similar to programming language exceptions.
- When fault raised, normal proxy flow processing aborts, and execution switches to FaultRules in current endpoint.
- If no FaultRules section exists, error response still sent.

```
<ProxyEndpoint name="orders-v1">
  <FaultRules>
    <FaultRule name="InvalidAPIKey">
      <Condition>fault.name=="FailedToResolveAPIKey"</Condition>
      <Step>
        <Name>AM-InvalidAPIKeyResponse</Name>
      </Step>
    </FaultRule>
  <FaultRules>

  <PreFlow name="PreFlow">
    <Request>
      <Step>
        <Name>JTP-JSONThreatProtect</Name>
      </Step>
      <Step>
        <Name>VK-VerifyKey</Name>
      </Step>
      <Step>
        ⋮
```

Let's look at an example.

We are processing the proxy endpoint request preflow.

The JSONThreatProtection step executes successfully...



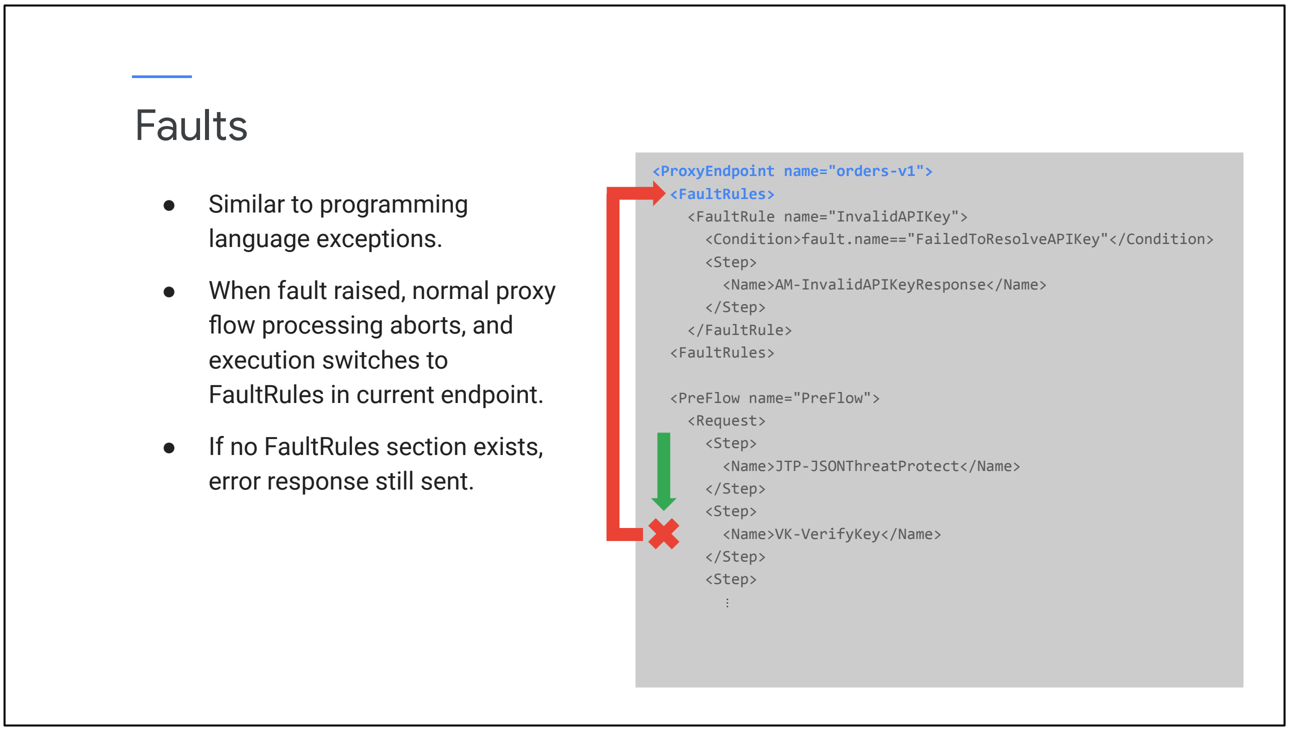
Faults

- Similar to programming language exceptions.
- When fault raised, normal proxy flow processing aborts, and execution switches to FaultRules in current endpoint.
- If no FaultRules section exists, error response still sent.

```
<ProxyEndpoint name="orders-v1">
  <FaultRules>
    <FaultRule name="InvalidAPIKey">
      <Condition>fault.name=="FailedToResolveAPIKey"</Condition>
      <Step>
        <Name>AM-InvalidAPIKeyResponse</Name>
      </Step>
    </FaultRule>
  <FaultRules>

  <PreFlow name="PreFlow">
    <Request>
      <Step>
        <Name>JTP-JSONThreatProtect</Name>
      </Step>
      <Step>
        <Name>VK-VerifyKey</Name>
      </Step>
      <Step>
        ⋮
```

...but when we get to the verify key step, a fault is raised because the API key was not valid.

No further steps will be processed in this flow.

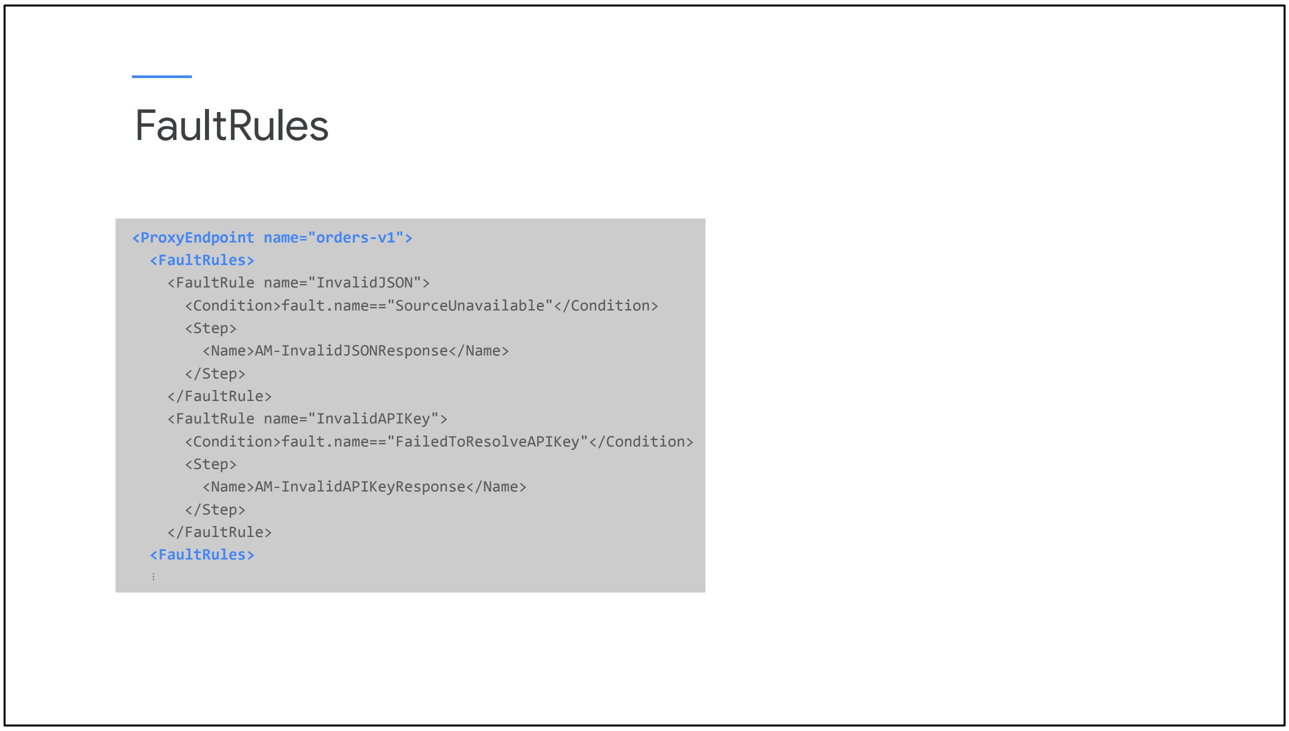


Faults

- Similar to programming language exceptions.
- When fault raised, normal proxy flow processing aborts, and execution switches to FaultRules in current endpoint.
- If no FaultRules section exists, error response still sent.

```
<ProxyEndpoint name="orders-v1">
  <FaultRules>
    <FaultRule name="InvalidAPIKey">
      <Condition>fault.name=="FailedToResolveAPIKey"</Condition>
      <Step>
        <Name>AM-InvalidAPIKeyResponse</Name>
      </Step>
    </FaultRule>
  <FaultRules>

  <PreFlow name="PreFlow">
    <Request>
      <Step>
        <Name>JTP-JSONThreatProtect</Name>
      </Step>
      <Step>
        <Name>VK-VerifyKey</Name>
      </Step>
      <Step>
        ⋮
```

Execution will continue with the FaultRules element in the same endpoint, the ProxyEndpoint named orders-v1.



FaultRules

```
<ProxyEndpoint name="orders-v1">
  <FaultRules>
    <FaultRule name="InvalidJSON">
      <Condition>fault.name=="SourceUnavailable"</Condition>
      <Step>
        <Name>AM-InvalidJSONResponse</Name>
      </Step>
    </FaultRule>
    <FaultRule name="InvalidAPIKey">
      <Condition>fault.name=="FailedToResolveAPIKey"</Condition>
      <Step>
        <Name>AM-InvalidAPIKeyResponse</Name>
      </Step>
    </FaultRule>
  <FaultRules>
    ⋮
```

The FaultRules element is specified in a ProxyEndpoint or TargetEndpoint.

By default, when you create a new proxy, there is no FaultRules element. When you add a new endpoint to a proxy, the FaultRules element also isn't added.

Flows are shown visually in the proxy editor, and a flow's policies can be edited using drag and drop. Fault rules are not shown visually in the proxy editor. You'll need to add fault rules manually to the XML configuration of the proxy endpoint or target endpoint.

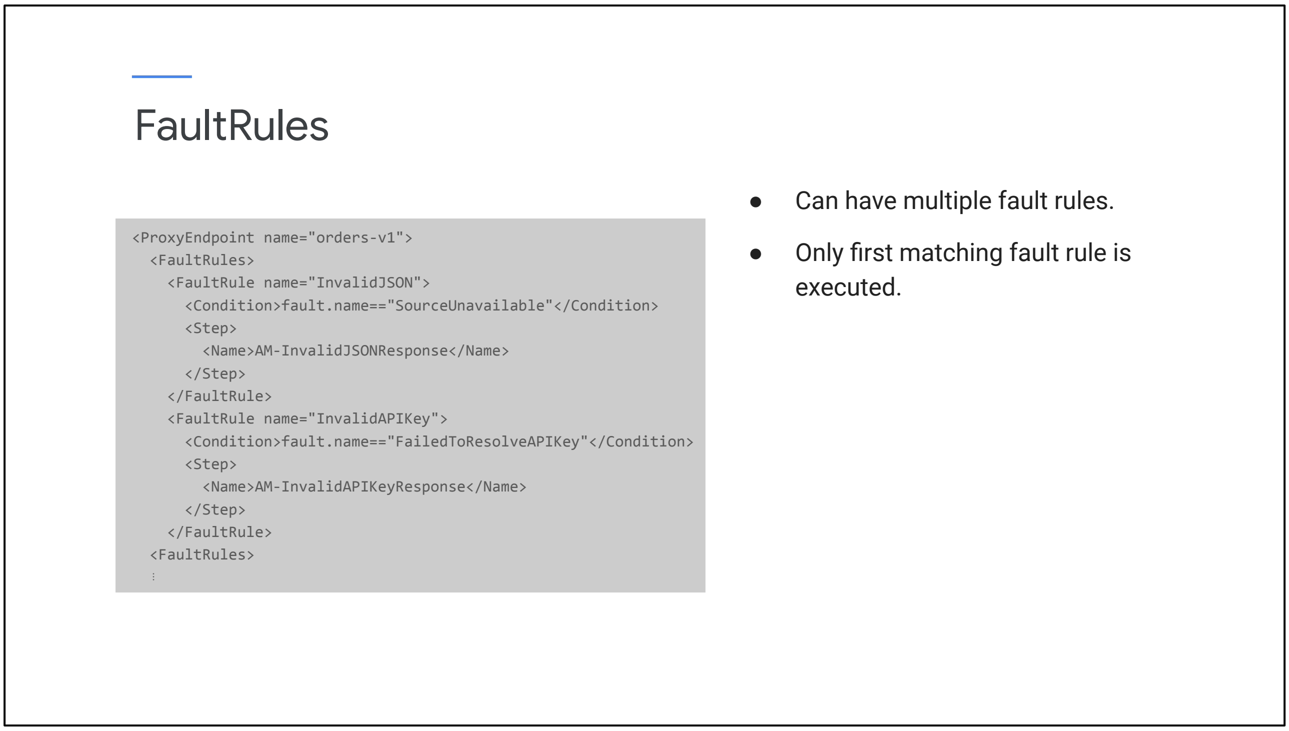
FaultRules

```
<ProxyEndpoint name="orders-v1">
  <FaultRules>
    <FaultRule name="InvalidJSON">
      <Condition>fault.name=="SourceUnavailable"</Condition>
      <Step>
        <Name>AM-InvalidJSONResponse</Name>
      </Step>
    </FaultRule>
    <FaultRule name="InvalidAPIKey">
      <Condition>fault.name=="FailedToResolveAPIKey"</Condition>
      <Step>
        <Name>AM-InvalidAPIKeyResponse</Name>
      </Step>
    </FaultRule>
  <FaultRules>
    ⋮
```

- Can have multiple fault rules.

The FaultRules element contains zero or more FaultRule entries.

Each FaultRule can have multiple steps, and the steps can have conditions.



FaultRules

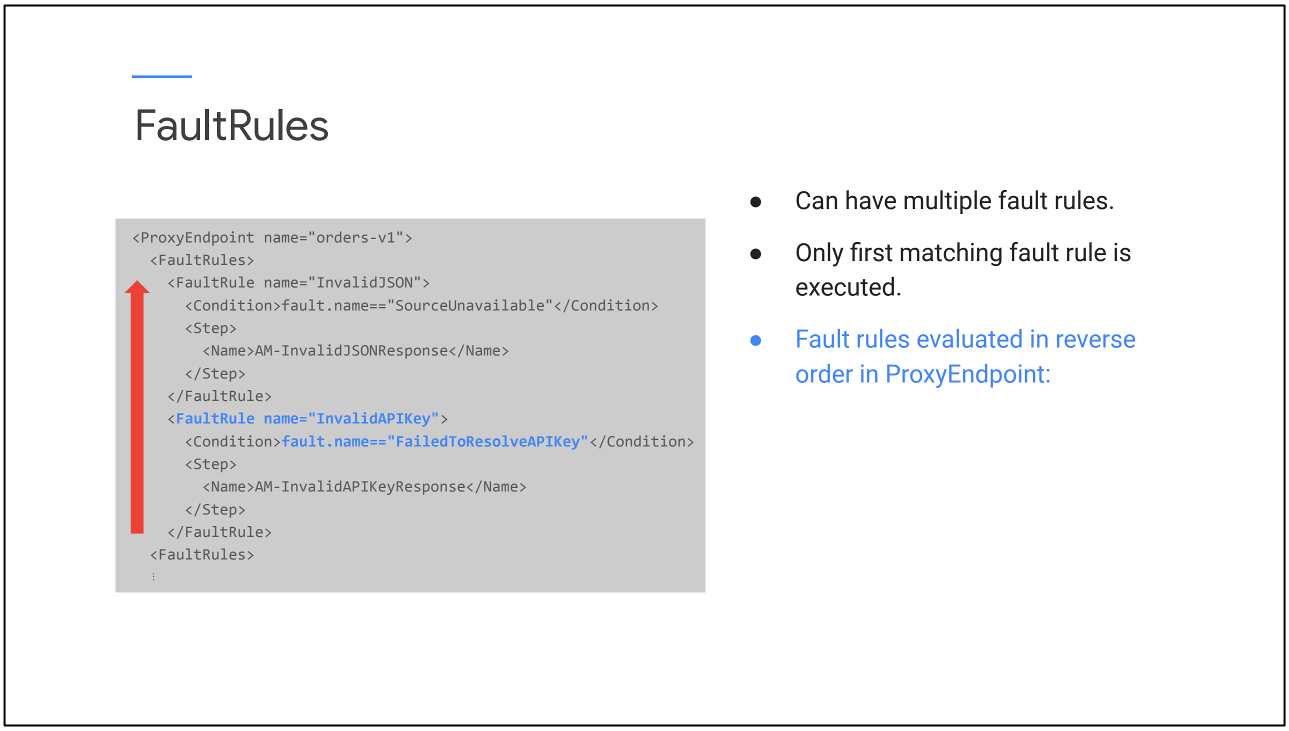
```
<ProxyEndpoint name="orders-v1">
  <FaultRules>
    <FaultRule name="InvalidJSON">
      <Condition>fault.name=="SourceUnavailable"</Condition>
      <Step>
        <Name>AM-InvalidJSONResponse</Name>
      </Step>
    </FaultRule>
    <FaultRule name="InvalidAPIKey">
      <Condition>fault.name=="FailedToResolveAPIKey"</Condition>
      <Step>
        <Name>AM-InvalidAPIKeyResponse</Name>
      </Step>
    </FaultRule>
  <FaultRules>
  ⋮
```

- Can have multiple fault rules.
- Only first matching fault rule is executed.

Only the first FaultRule with a condition that evaluates to true is executed.

The remaining fault rules will be skipped.

Fault rules without a condition always evaluate to true.



FaultRules

```
<ProxyEndpoint name="orders-v1">
  <FaultRules>
    <FaultRule name="InvalidJSON">
      <Condition>fault.name=="SourceUnavailable"</Condition>
      <Step>
        <Name>AM-InvalidJSONResponse</Name>
      </Step>
    </FaultRule>
    <FaultRule name="InvalidAPIKey">
      <Condition>fault.name=="FailedToResolveAPIKey"</Condition>
      <Step>
        <Name>AM-InvalidAPIKeyResponse</Name>
      </Step>
    </FaultRule>
  <FaultRules>
    ⋮
```

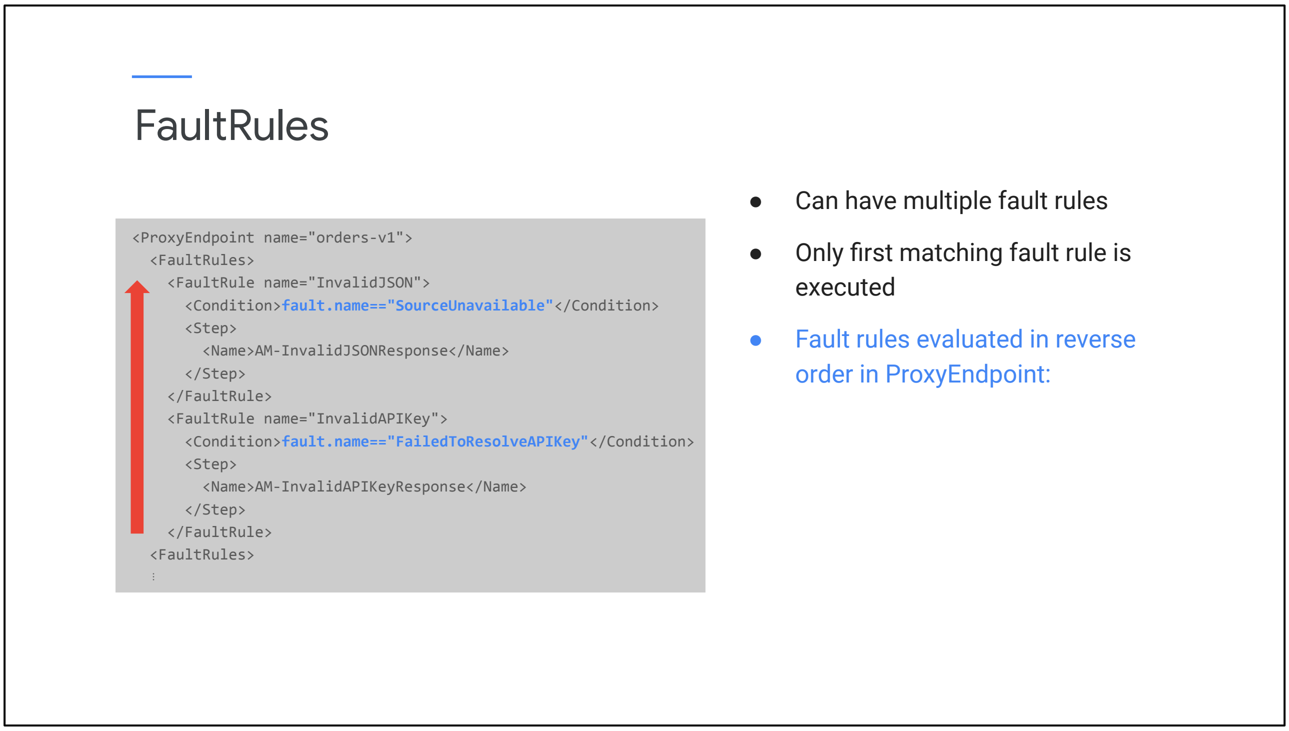
- Can have multiple fault rules.
- Only first matching fault rule is executed.
- Fault rules evaluated in reverse order in ProxyEndpoint:

Always remember that the fault rules are evaluated in reverse order, from bottom to top, in a proxy endpoint.

In this example, the InvalidAPIKey fault rule condition would be evaluated first. The condition is true if the fault.name variable is "FailedToResolveAPIKey."

Fault.name contains the name of the fault raised by a policy, in this case the VerifyAPIKey policy.

Policies will set this variable when raising a fault, so it is a useful variable for fault rules, where we often build custom error responses.



FaultRules

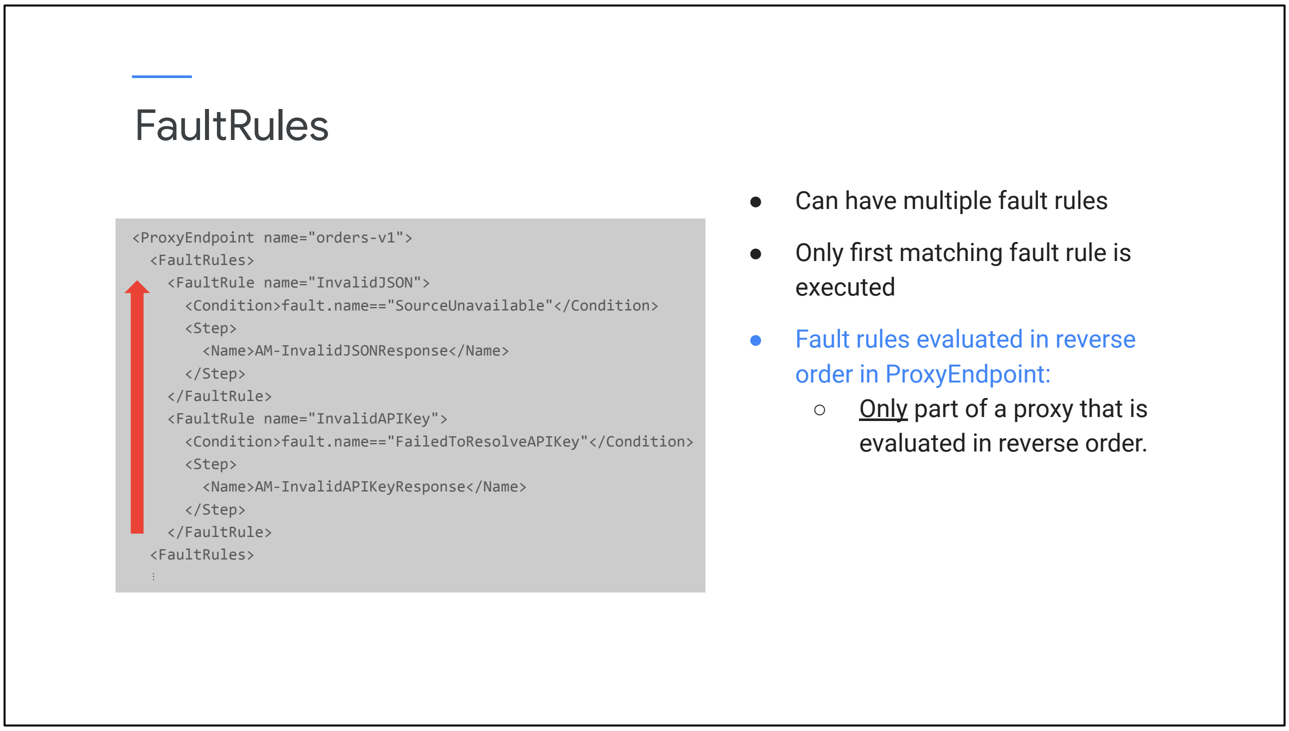
```
<ProxyEndpoint name="orders-v1">
  <FaultRules>
    <FaultRule name="InvalidJSON">
      <Condition>fault.name=="SourceUnavailable"</Condition>
      <Step>
        <Name>AM-InvalidJSONResponse</Name>
      </Step>
    </FaultRule>
    <FaultRule name="InvalidAPIKey">
      <Condition>fault.name=="FailedToResolveAPIKey"</Condition>
      <Step>
        <Name>AM-InvalidAPIKeyResponse</Name>
      </Step>
    </FaultRule>
  <FaultRules>
    ⋮
```

- Can have multiple fault rules
- Only first matching fault rule is executed
- Fault rules evaluated in reverse order in ProxyEndpoint:

The conditions for the two fault rules are mutually exclusive. If the first condition matches, the second wouldn't, and vice versa. This means that you could evaluate the fault rules from top to bottom or bottom to top and you'd get the same result.

If you can, it is a best practice to set up your fault rule conditions this way, so that you don't have to worry about the fault rules evaluation order.

If your fault rules do depend on this reverse ordering, add an XML comment to the fault rules section to remind the reader that the evaluation of fault rules is from bottom to top.

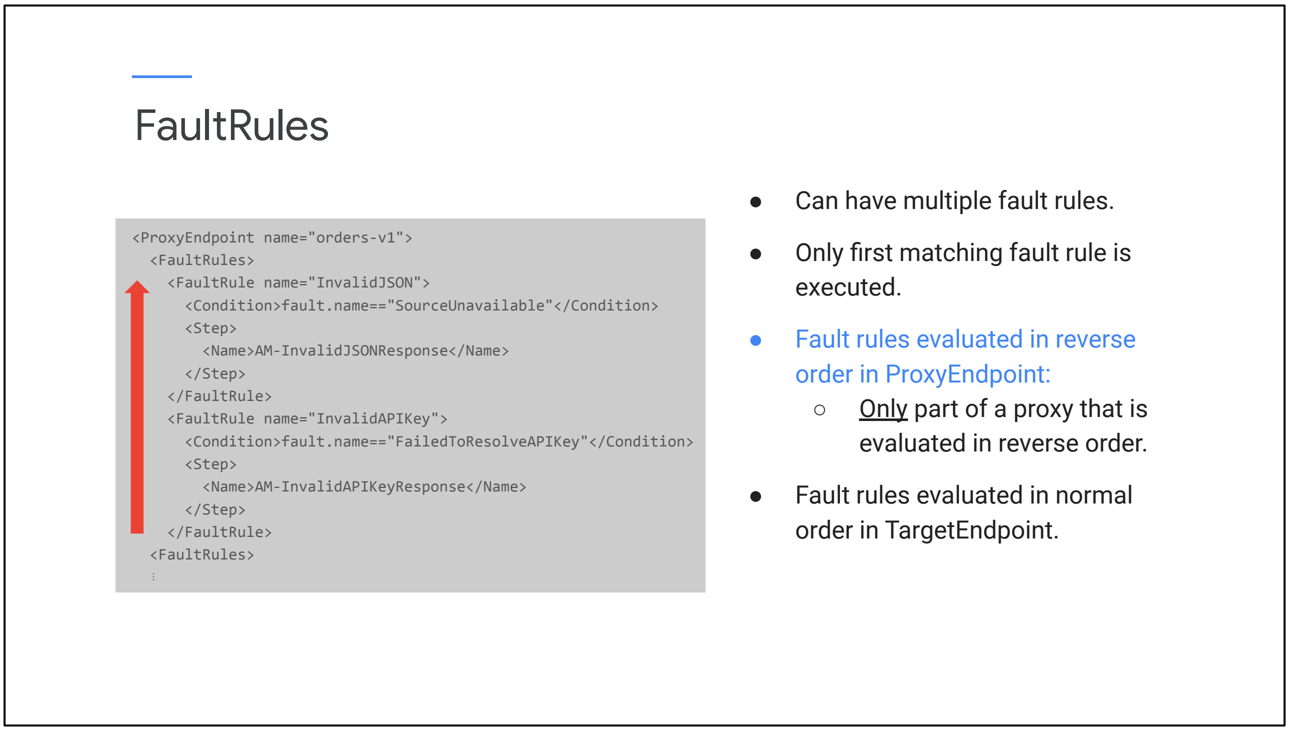


FaultRules

```
<ProxyEndpoint name="orders-v1">
  <FaultRules>
    <FaultRule name="InvalidJSON">
      <Condition>fault.name=="SourceUnavailable"</Condition>
      <Step>
        <Name>AM-InvalidJSONResponse</Name>
      </Step>
    </FaultRule>
    <FaultRule name="InvalidAPIKey">
      <Condition>fault.name=="FailedToResolveAPIKey"</Condition>
      <Step>
        <Name>AM-InvalidAPIKeyResponse</Name>
      </Step>
    </FaultRule>
  <FaultRules>
  ⋮
```

- Can have multiple fault rules
- Only first matching fault rule is executed
- Fault rules evaluated in reverse order in ProxyEndpoint:
  - Only part of a proxy that is evaluated in reverse order.

The ProxyEndpoint fault rules are the only part of a proxy that are evaluated in reverse order.

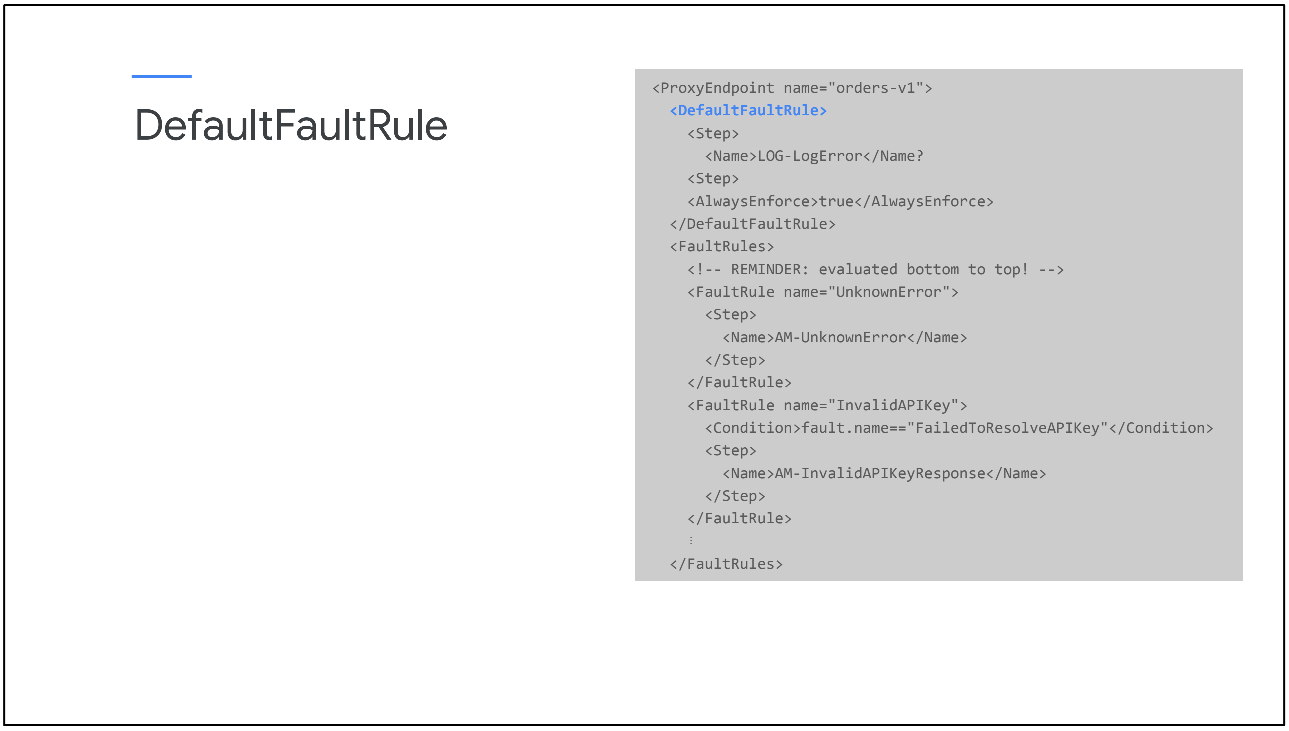


FaultRules

```
<ProxyEndpoint name="orders-v1">
  <FaultRules>
    <FaultRule name="InvalidJSON">
      <Condition>fault.name=="SourceUnavailable"</Condition>
      <Step>
        <Name>AM-InvalidJSONResponse</Name>
      </Step>
    </FaultRule>
    <FaultRule name="InvalidAPIKey">
      <Condition>fault.name=="FailedToResolveAPIKey"</Condition>
      <Step>
        <Name>AM-InvalidAPIKeyResponse</Name>
      </Step>
    </FaultRule>
  <FaultRules>
    ⋮
```

- Can have multiple fault rules.
- Only first matching fault rule is executed.
- Fault rules evaluated in reverse order in ProxyEndpoint:
  - Only part of a proxy that is evaluated in reverse order.
- Fault rules evaluated in normal order in TargetEndpoint.

TargetEndpoint fault rules are evaluated in the normal top-to-bottom order.



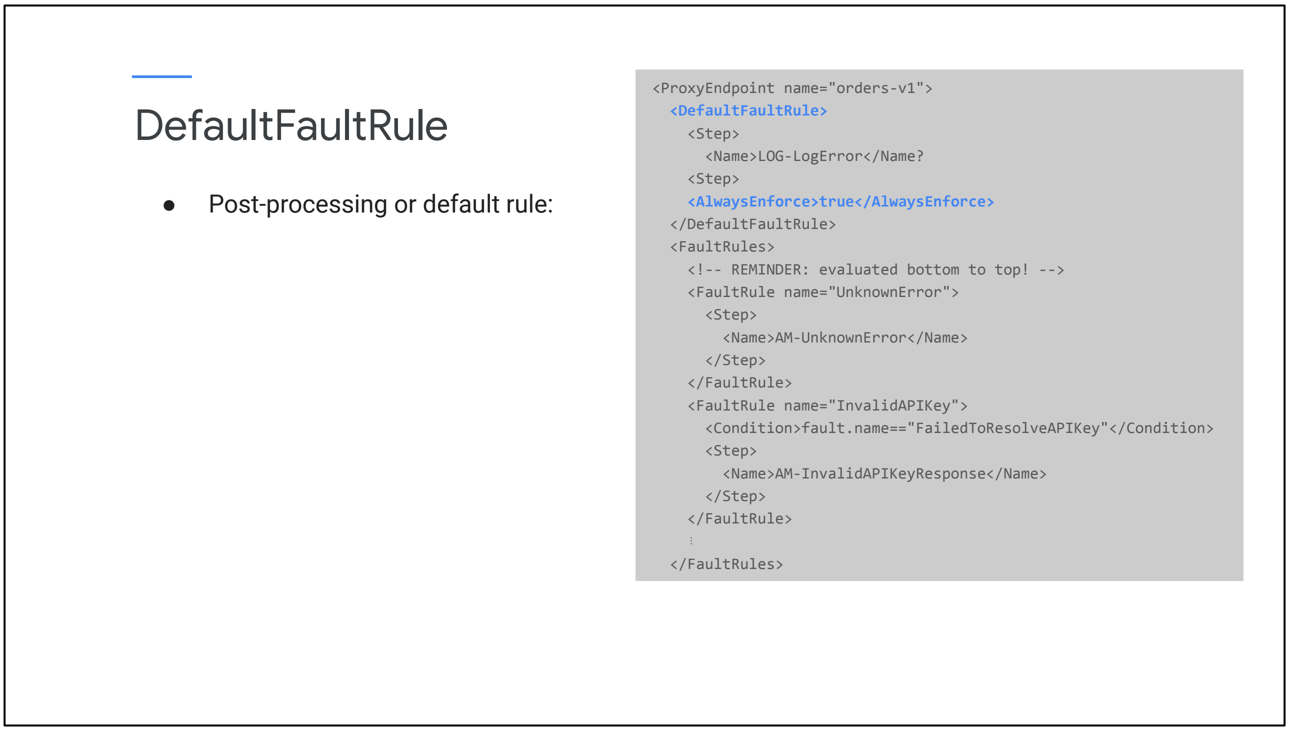
DefaultFaultRule

```
<ProxyEndpoint name="orders-v1">
  <DefaultFaultRule>
    <Step>
      <Name>LOG-LogError</Name?
    <Step>
    <AlwaysEnforce>true</AlwaysEnforce>
  </DefaultFaultRule>
  <FaultRules>
    <!-- REMINDER: evaluated bottom to top! -->
    <FaultRule name="UnknownError">
      <Step>
        <Name>AM-UnknownError</Name>
      </Step>
    </FaultRule>
    <FaultRule name="InvalidAPIKey">
      <Condition>fault.name=="FailedToResolveAPIKey"</Condition>
      <Step>
        <Name>AM-InvalidAPIKeyResponse</Name>
      </Step>
    </FaultRule>
     ⋮
  </FaultRules>
```

There is an optional fault rule called the DefaultFaultRule.

It is specified outside of the FaultRules section.

It does not have a condition.

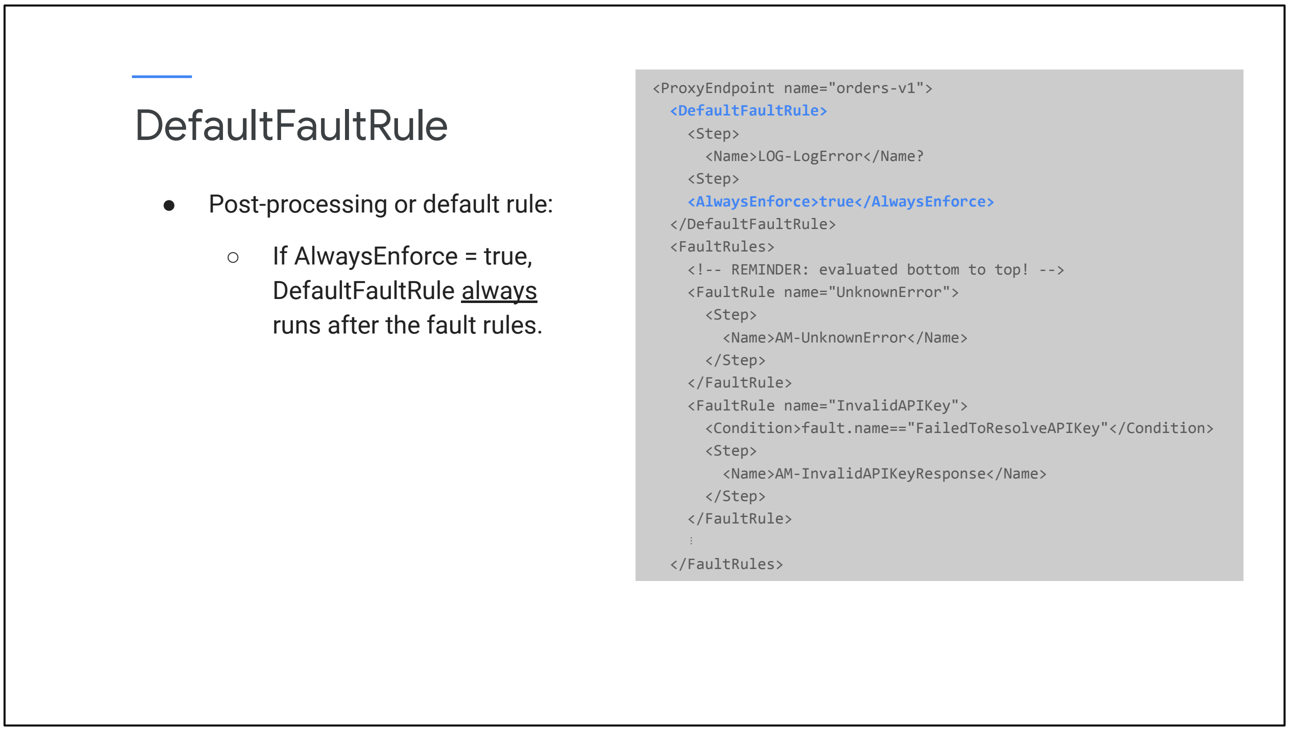


DefaultFaultRule

- Post-processing or default rule:

```
<ProxyEndpoint name="orders-v1">
  <DefaultFaultRule>
    <Step>
      <Name>LOG-LogError</Name?
    <Step>
    <AlwaysEnforce>true</AlwaysEnforce>
  </DefaultFaultRule>
  <FaultRules>
    <!-- REMINDER: evaluated bottom to top! -->
    <FaultRule name="UnknownError">
      <Step>
        <Name>AM-UnknownError</Name>
      </Step>
    </FaultRule>
    <FaultRule name="InvalidAPIKey">
      <Condition>fault.name=="FailedToResolveAPIKey"</Condition>
      <Step>
        <Name>AM-InvalidAPIKeyResponse</Name>
      </Step>
    </FaultRule>
    ⋮
  </FaultRules>
```

Instead, there is an element called AlwaysEnforce, which specifies whether the DefaultFaultRule is treated as a post-processing or default rule.



DefaultFaultRule

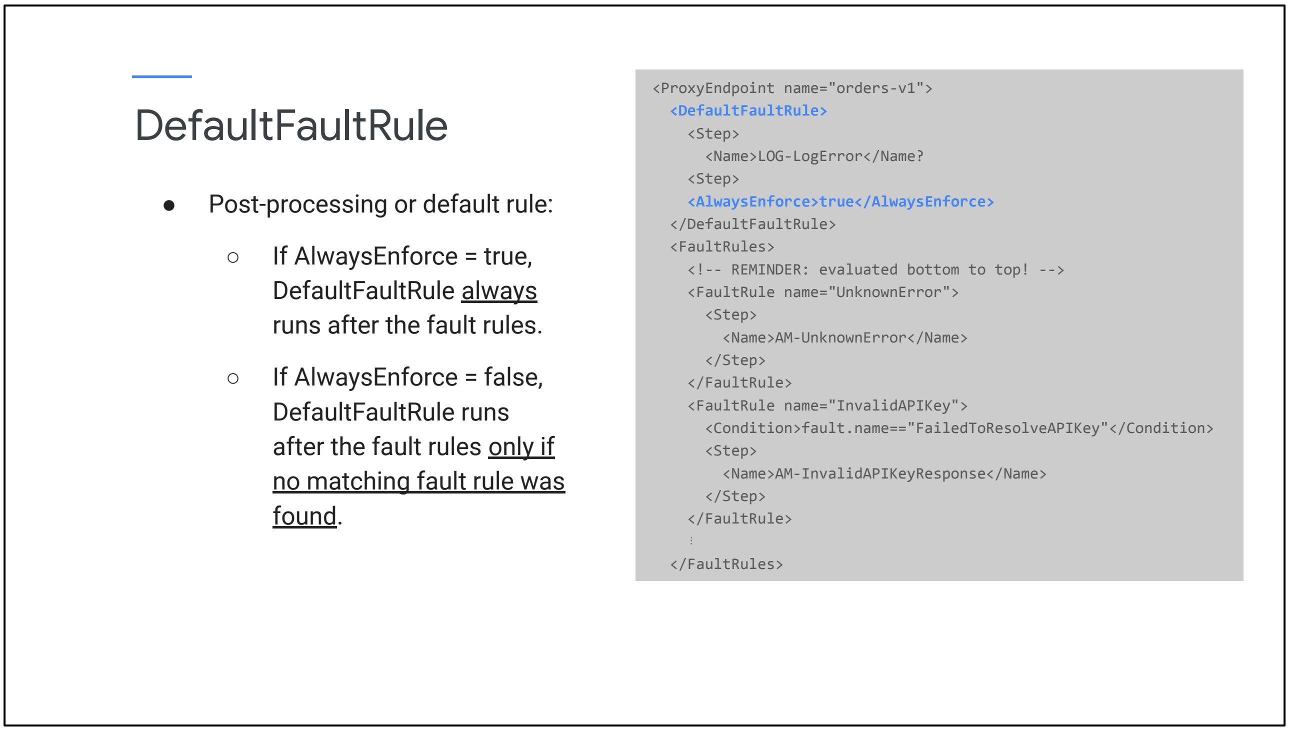
- Post-processing or default rule:
    - If AlwaysEnforce = true, DefaultFaultRule always runs after the fault rules.

```
<ProxyEndpoint name="orders-v1">
  <DefaultFaultRule>
    <Step>
      <Name>LOG-LogError</Name?
    <Step>
    <AlwaysEnforce>true</AlwaysEnforce>
  </DefaultFaultRule>
  <FaultRules>
    <!-- REMINDER: evaluated bottom to top! -->
    <FaultRule name="UnknownError">
      <Step>
        <Name>AM-UnknownError</Name>
      </Step>
    </FaultRule>
    <FaultRule name="InvalidAPIKey">
      <Condition>fault.name=="FailedToResolveAPIKey"</Condition>
      <Step>
        <Name>AM-InvalidAPIKeyResponse</Name>
      </Step>
    </FaultRule>
      ⋮
  </FaultRules>
```

If AlwaysEnforce is set to true, the default fault rule will always run after the fault rules.

A FaultRule might execute, or none might have true conditions.

Either way, the DefaultFaultRule would run at the end before responding to the client.

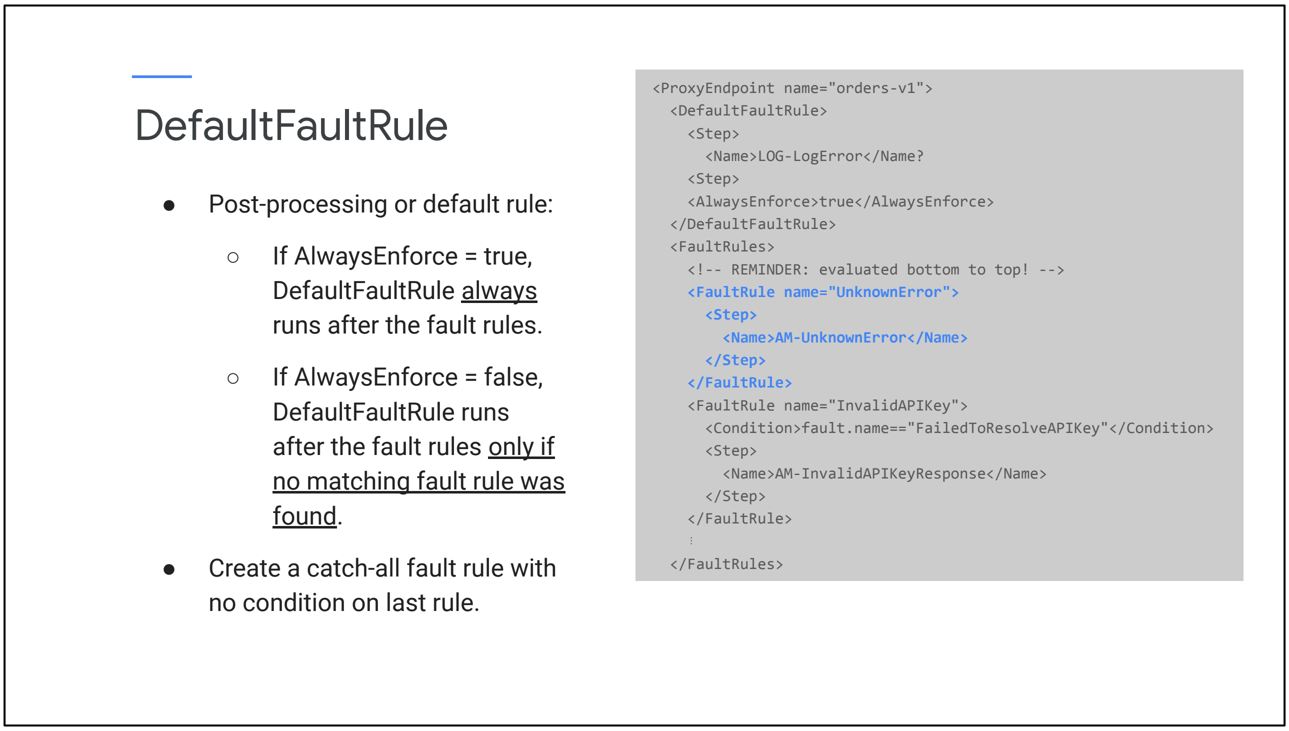


DefaultFaultRule

- Post-processing or default rule:
  - If AlwaysEnforce = true, DefaultFaultRule always runs after the fault rules.
  - If AlwaysEnforce = false, DefaultFaultRule runs after the fault rules only if no matching fault rule was found.

```
<ProxyEndpoint name="orders-v1">
  <DefaultFaultRule>
    <Step>
      <Name>LOG-LogError</Name?
    <Step>
    <AlwaysEnforce>true</AlwaysEnforce>
  </DefaultFaultRule>
  <FaultRules>
    <!-- REMINDER: evaluated bottom to top! -->
    <FaultRule name="UnknownError">
      <Step>
        <Name>AM-UnknownError</Name>
      </Step>
    </FaultRule>
    <FaultRule name="InvalidAPIKey">
      <Condition>fault.name=="FailedToResolveAPIKey"</Condition>
      <Step>
        <Name>AM-InvalidAPIKeyResponse</Name>
      </Step>
    </FaultRule>
      ⋮
  </FaultRules>
```

If AlwaysEnforce is set to false, the DefaultFaultRule would be executed only if no FaultRules had conditions evaluating to true.



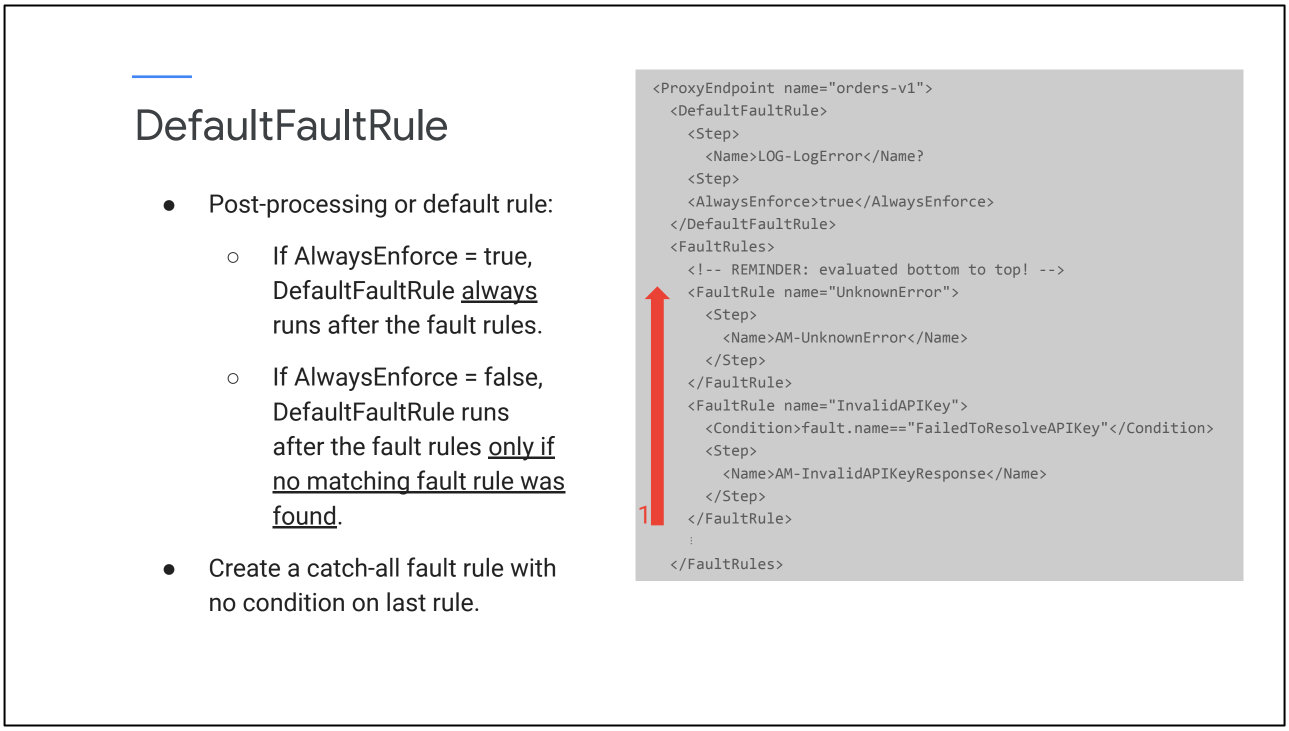
DefaultFaultRule

- Post-processing or default rule:
  - If AlwaysEnforce = true, DefaultFaultRule always runs after the fault rules.
  - If AlwaysEnforce = false, DefaultFaultRule runs after the fault rules only if no matching fault rule was found.
- Create a catch-all fault rule with no condition on last rule.

```
<ProxyEndpoint name="orders-v1">
  <DefaultFaultRule>
    <Step>
      <Name>LOG-LogError</Name?
    <Step>
    <AlwaysEnforce>true</AlwaysEnforce>
  </DefaultFaultRule>
  <FaultRules>
    <!-- REMINDER: evaluated bottom to top! -->
    <FaultRule name="UnknownError">
      <Step>
        <Name>AM-UnknownError</Name>
      </Step>
    </FaultRule>
    <FaultRule name="InvalidAPIKey">
      <Condition>fault.name=="FailedToResolveAPIKey"</Condition>
      <Step>
        <Name>AM-InvalidAPIKeyResponse</Name>
      </Step>
    </FaultRule>
      ⋮
  </FaultRules>
```

You can also create the same functionality as AlwaysEnforce equals false by not giving the last evaluated FaultRule a condition.

A FaultRule without a condition is the same as having a FaultRule with a condition that is always true.



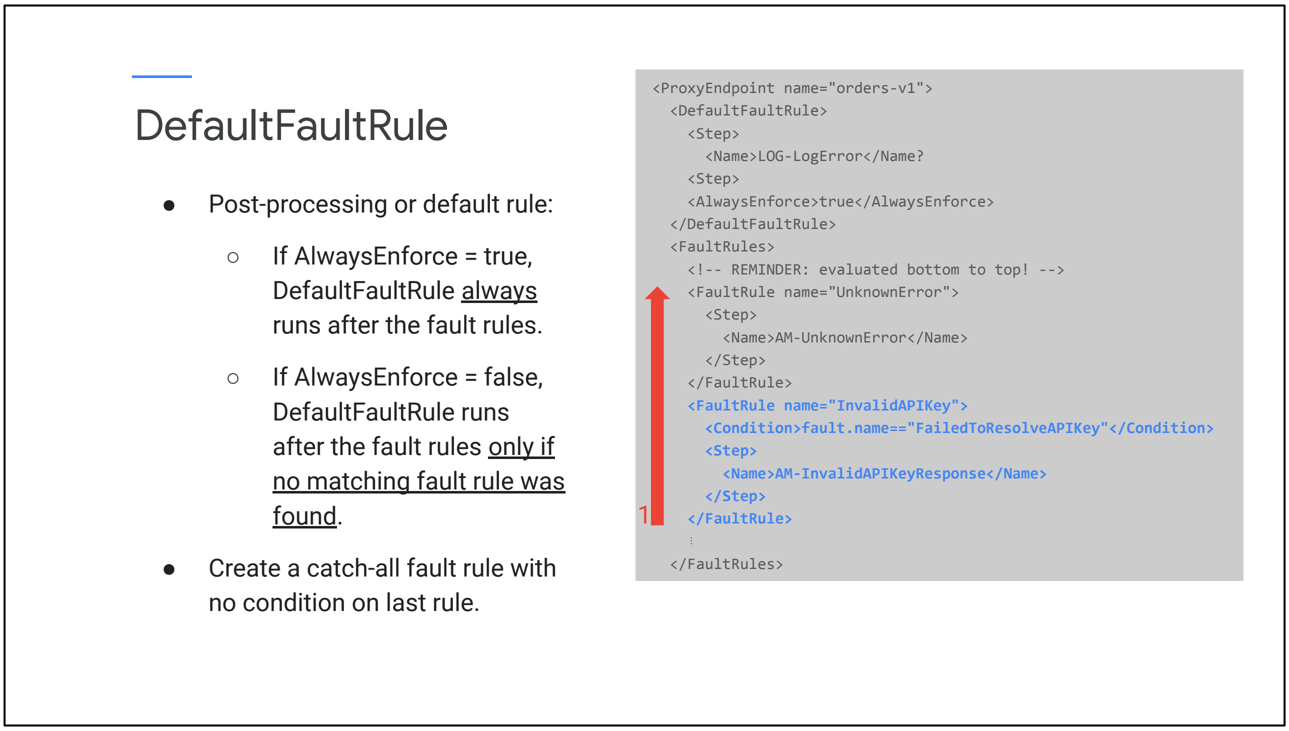
DefaultFaultRule

- Post-processing or default rule:
  - If AlwaysEnforce = true, DefaultFaultRule <u>always</u> runs after the fault rules.
  - If AlwaysEnforce = false, DefaultFaultRule runs after the fault rules <u>only if no matching fault rule was found</u>.
- Create a catch-all fault rule with no condition on last rule.

```
<ProxyEndpoint name="orders-v1">
  <DefaultFaultRule>
    <Step>
      <Name>LOG-LogError</Name?
    <Step>
    <AlwaysEnforce>true</AlwaysEnforce>
  </DefaultFaultRule>
  <FaultRules>
    <!-- REMINDER: evaluated bottom to top! -->
    <FaultRule name="UnknownError">
      <Step>
        <Name>AM-UnknownError</Name>
      </Step>
    </FaultRule>
    <FaultRule name="InvalidAPIKey">
      <Condition>fault.name=="FailedToResolveAPIKey"</Condition>
      <Step>
        <Name>AM-InvalidAPIKeyResponse</Name>
      </Step>
1   </FaultRule>
    ⋮
  </FaultRules>
```

Looking at this example, we first evaluate the FaultRules list.

Since this is in the ProxyEndpoint, we evaluate from bottom to top.



DefaultFaultRule

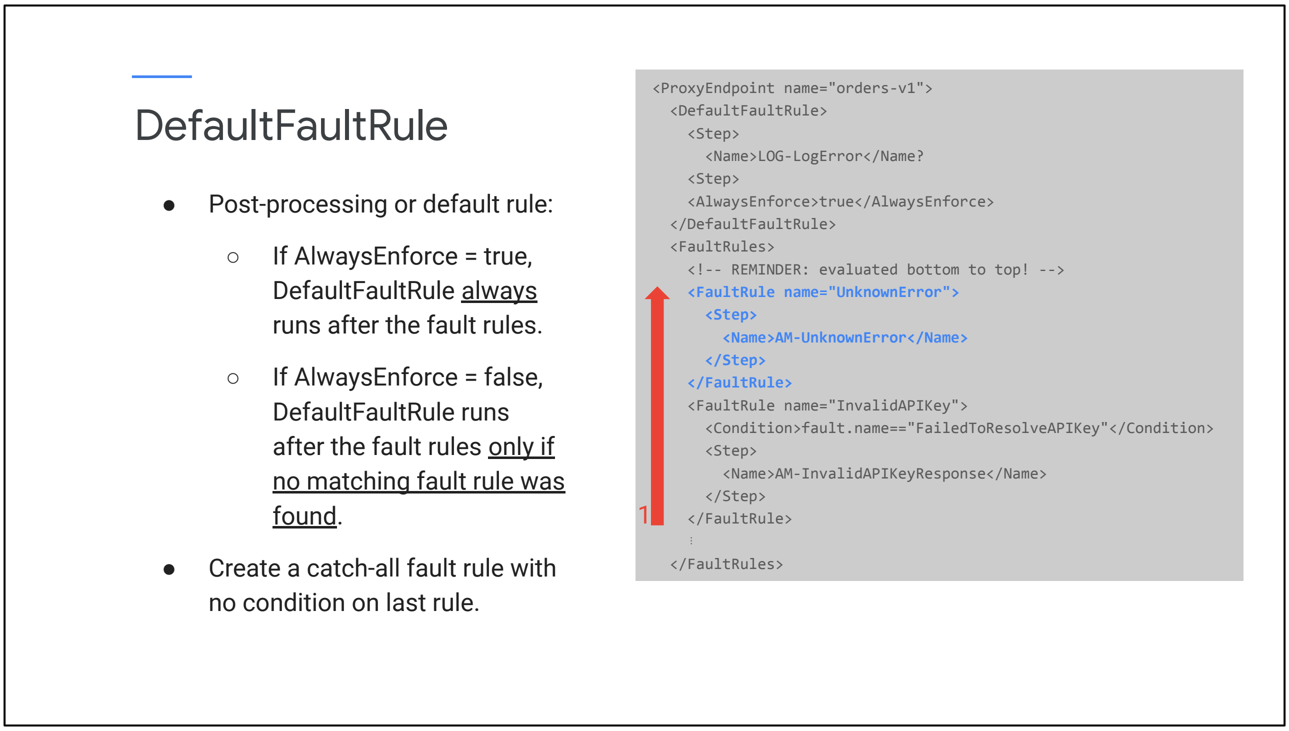
- Post-processing or default rule:
  - If AlwaysEnforce = true, DefaultFaultRule <u>always</u> runs after the fault rules.
  - If AlwaysEnforce = false, DefaultFaultRule runs after the fault rules <u>only if no matching fault rule was found</u>.
- Create a catch-all fault rule with no condition on last rule.

```
<ProxyEndpoint name="orders-v1">
  <DefaultFaultRule>
    <Step>
      <Name>LOG-LogError</Name?
    <Step>
    <AlwaysEnforce>true</AlwaysEnforce>
  </DefaultFaultRule>
  <FaultRules>
    <!-- REMINDER: evaluated bottom to top! -->
    <FaultRule name="UnknownError">
      <Step>
        <Name>AM-UnknownError</Name>
      </Step>
    </FaultRule>
    <FaultRule name="InvalidAPIKey">
      <Condition>fault.name=="FailedToResolveAPIKey"</Condition>
      <Step>
        <Name>AM-InvalidAPIKeyResponse</Name>
      </Step>
1   </FaultRule>
      ⋮
  </FaultRules>
```

The InvalidAPIKey fault rule is evaluated first.

If fault.name is "FailedToResolveAPIKey," the AssignMessage policy AM-InvalidAPIKeyResponse would be executed, and no other fault rules would be evaluated.

If fault dot name is not "FailedToResolveAPIKey," we check the next fault rule.



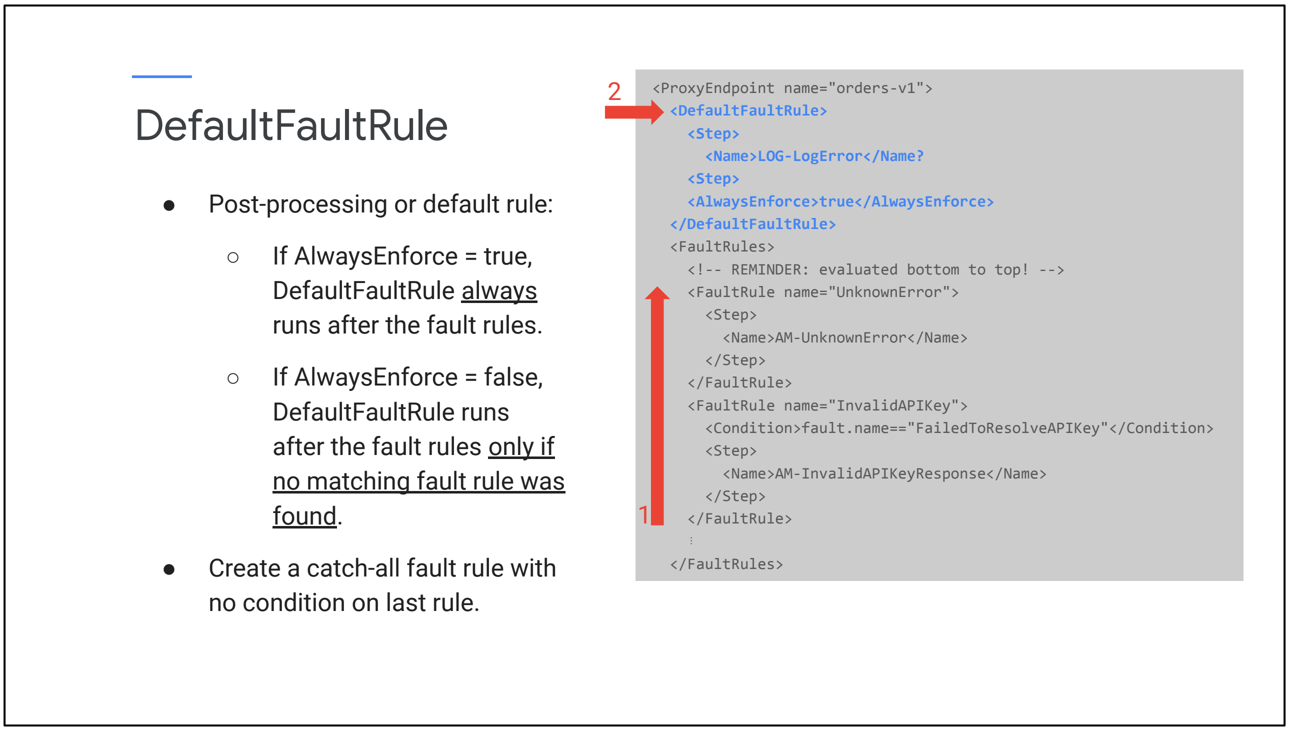
DefaultFaultRule

- Post-processing or default rule:
    - If AlwaysEnforce = true, DefaultFaultRule <u>always</u> runs after the fault rules.
    - If AlwaysEnforce = false, DefaultFaultRule runs after the fault rules <u>only if no matching fault rule was found</u>.
- Create a catch-all fault rule with no condition on last rule.

```
<ProxyEndpoint name="orders-v1">
  <DefaultFaultRule>
    <Step>
      <Name>LOG-LogError</Name?
    <Step>
    <AlwaysEnforce>true</AlwaysEnforce>
  </DefaultFaultRule>
  <FaultRules>
    <!-- REMINDER: evaluated bottom to top! -->
    <FaultRule name="UnknownError">
      <Step>
        <Name>AM-UnknownError</Name>
      </Step>
    </FaultRule>
    <FaultRule name="InvalidAPIKey">
      <Condition>fault.name=="FailedToResolveAPIKey"</Condition>
      <Step>
        <Name>AM-InvalidAPIKeyResponse</Name>
      </Step>
    </FaultRule>
      ⋮
  </FaultRules>
```

The UnknownError fault rule has no condition, so it will be evaluated.

The AssignMessage policy, AM-UnknownError, is executed.



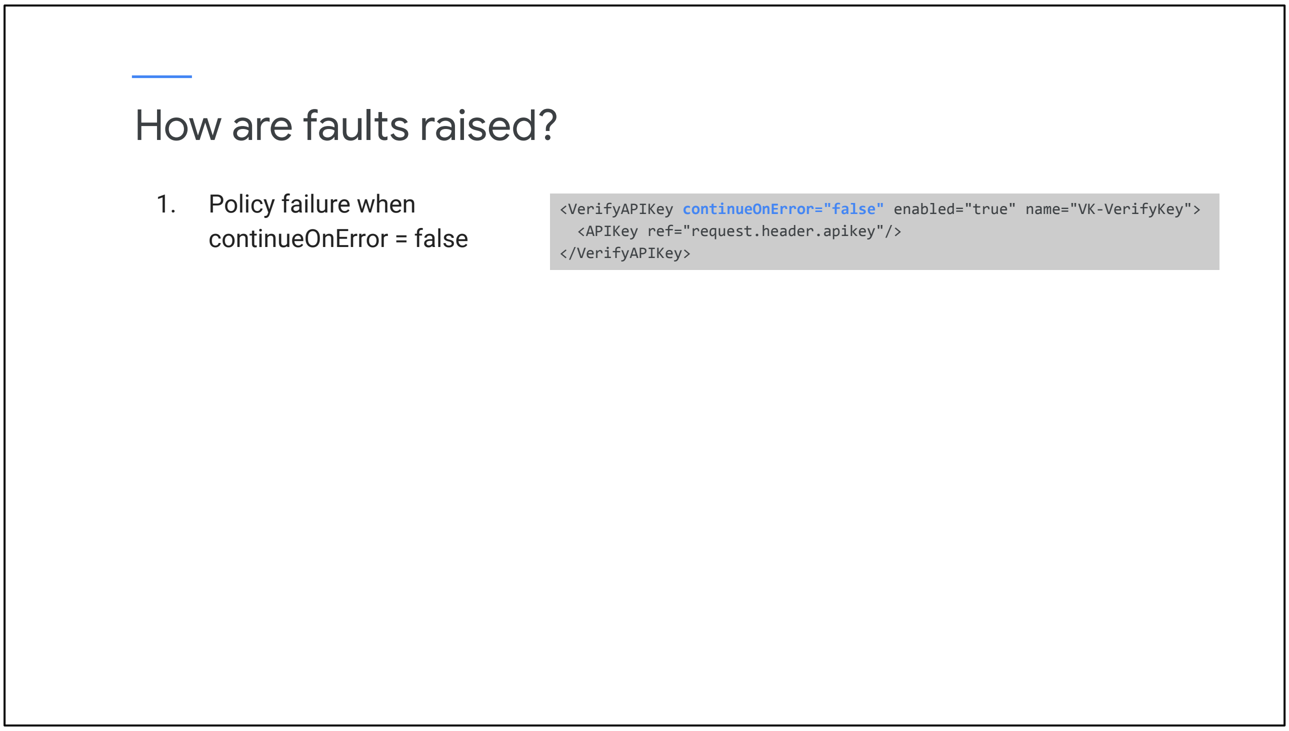
DefaultFaultRule

- Post-processing or default rule:
  - If AlwaysEnforce = true, DefaultFaultRule always runs after the fault rules.
  - If AlwaysEnforce = false, DefaultFaultRule runs after the fault rules only if no matching fault rule was found.
- Create a catch-all fault rule with no condition on last rule.

```
<ProxyEndpoint name="orders-v1">
  <DefaultFaultRule>
    <Step>
      <Name>LOG-LogError</Name?
    <Step>
    <AlwaysEnforce>true</AlwaysEnforce>
  </DefaultFaultRule>
  <FaultRules>
    <!-- REMINDER: evaluated bottom to top! -->
    <FaultRule name="UnknownError">
      <Step>
        <Name>AM-UnknownError</Name>
      </Step>
    </FaultRule>
    <FaultRule name="InvalidAPIKey">
      <Condition>fault.name=="FailedToResolveAPIKey"</Condition>
      <Step>
        <Name>AM-InvalidAPIKeyResponse</Name>
      </Step>
    </FaultRule>
    ⋮
  </FaultRules>
```

After the FaultRules, we evaluate the DefaultFaultRule element if it exists.

In this case AlwaysEnforce is true, so the DefaultFaultRule steps are executed. LOG-LogError would execute, regardless of whether any fault rule was executed.



How are faults raised?

1. Policy failure when continueOnError = false

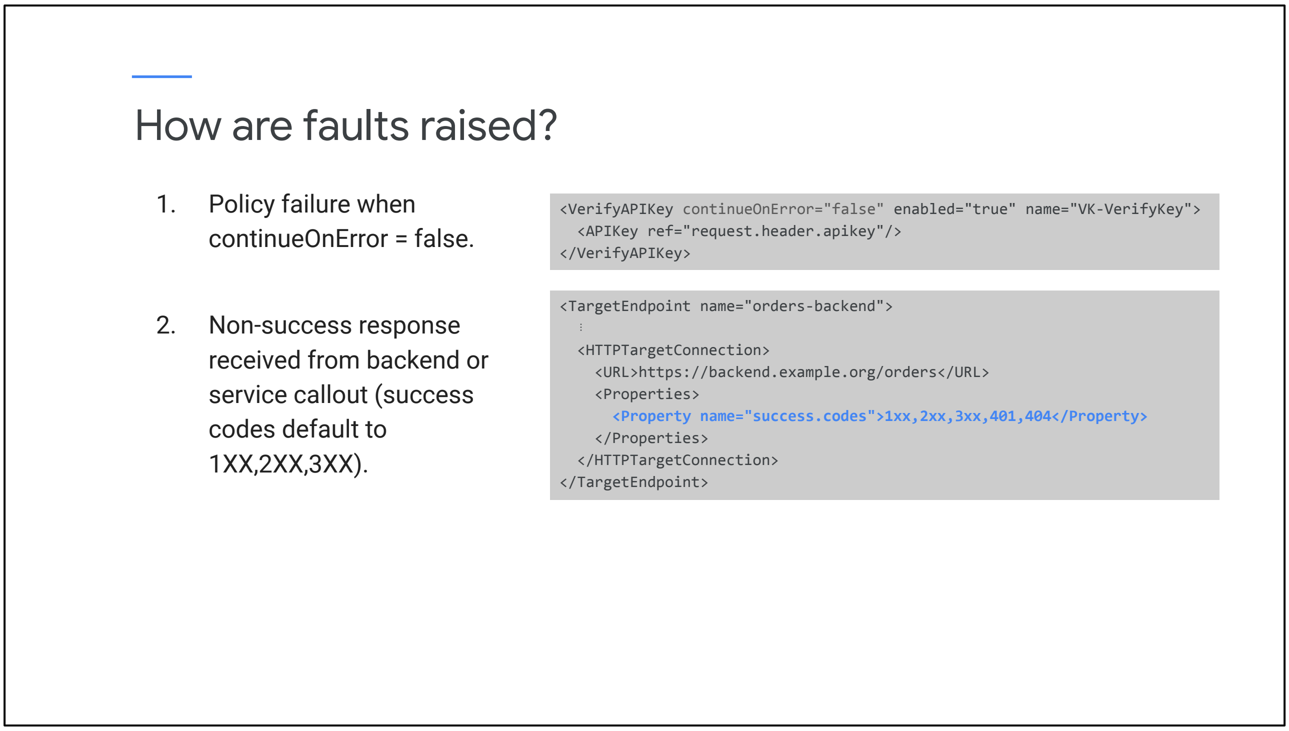
```
<VerifyAPIKey continueOnError="false" enabled="true" name="VK-VerifyKey">
  <APIKey ref="request.header.apikey"/>
</VerifyAPIKey>
```

When faults are raised, processing is transferred to the fault rules. Faults may be raised in a few ways.

When a policy error occurs, and the policy attribute continueOnError is set to false, a fault is raised.

If continueOnError is set to true, the fault variables will be set but processing continues in the current flow.

Uncaught exceptions from JavaScript and Java code also result in policy errors, as well as raised faults if ContinueOnError is false.



How are faults raised?

1. Policy failure when continueOnError = false.

```
<VerifyAPIKey continueOnError="false" enabled="true" name="VK-VerifyKey">
  <APIKey ref="request.header.apikey"/>
</VerifyAPIKey>
```

2. Non-success response received from backend or service callout (success codes default to 1XX,2XX,3XX).

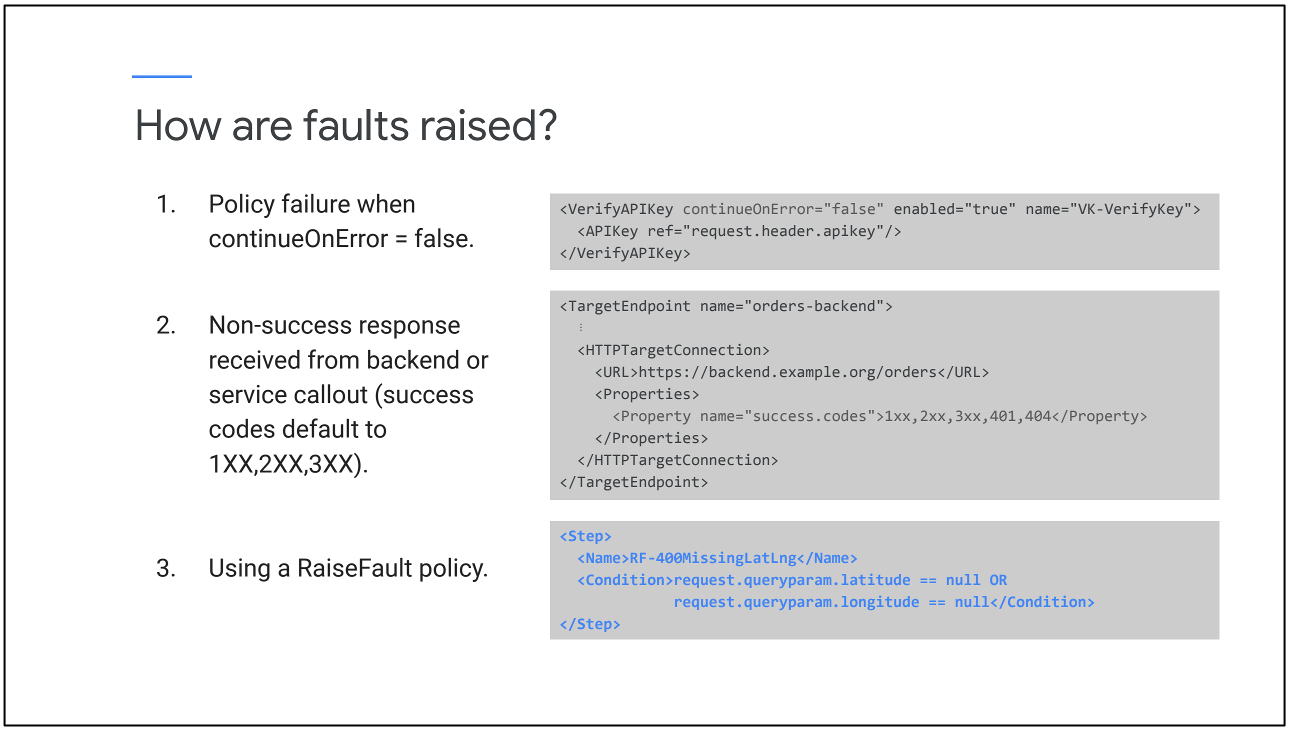
```
<TargetEndpoint name="orders-backend">
  ⋮
  <HTTPTargetConnection>
    <URL>https://backend.example.org/orders</URL>
    <Properties>
      <Property name="success.codes">1xx,2xx,3xx,401,404</Property>
    </Properties>
  </HTTPTargetConnection>
</TargetEndpoint>
```

Faults are also raised when a non-success response is received from a service callout or call to a target. By default, any status code in the 400s or 500s is considered an error.

If an error status code is returned, processing will be transferred to the fault rules in the current endpoint. For targets, processing would continue in the target endpoint.

You can change the status codes that should be treated as successful by using the success.codes property. In this example, 401 and 404 status codes are treated as successful, along with the default success status codes in the 100s, 200s, and 300s.

Error status codes should be treated as success codes when you plan to handle the processing of them in the normal proxy flow. 401, for example, might be handled by using a service callout to request another backend token and then resubmitting the original request using the new token.



How are faults raised?

1. Policy failure when continueOnError = false.

```
<VerifyAPIKey continueOnError="false" enabled="true" name="VK-VerifyKey">
  <APIKey ref="request.header.apikey"/>
</VerifyAPIKey>
```

2. Non-success response received from backend or service callout (success codes default to 1XX,2XX,3XX).

```
<TargetEndpoint name="orders-backend">
  ⋮
  <HTTPTargetConnection>
    <URL>https://backend.example.org/orders</URL>
    <Properties>
      <Property name="success.codes">1xx,2xx,3xx,401,404</Property>
    </Properties>
  </HTTPTargetConnection>
</TargetEndpoint>
```

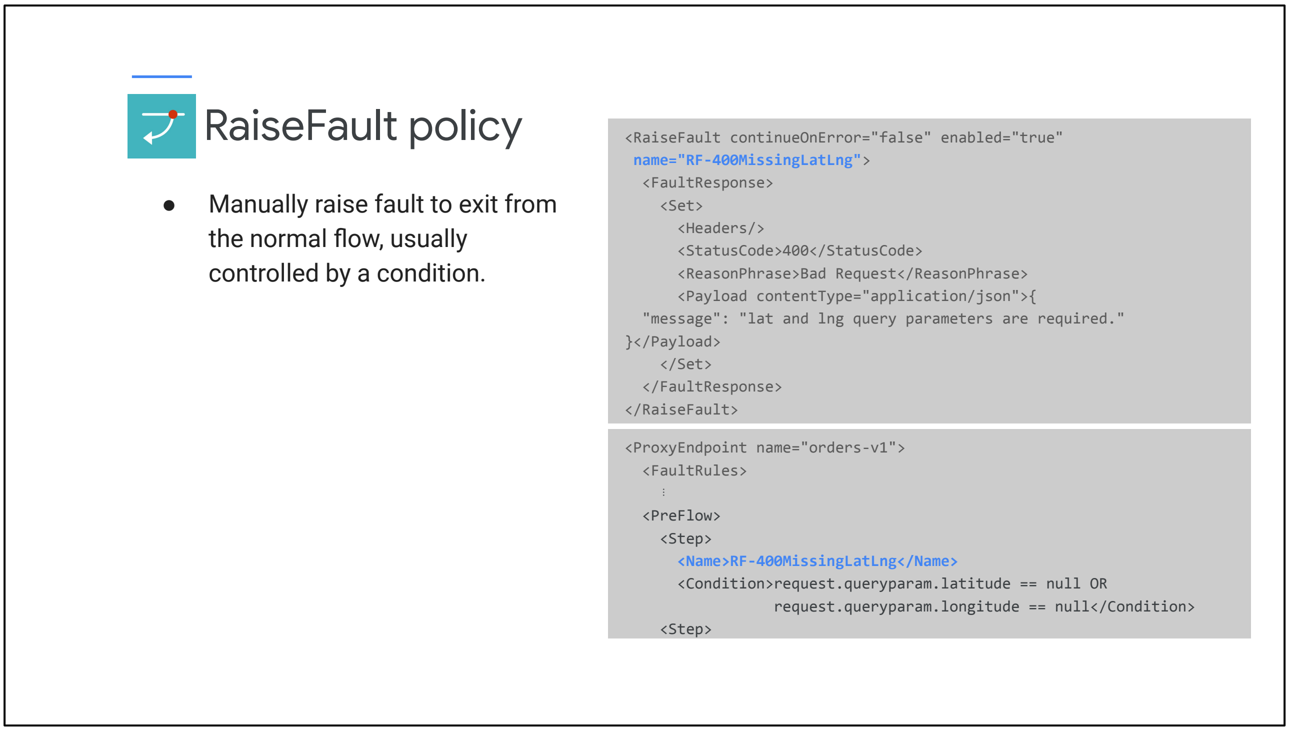
3. Using a RaiseFault policy.

```
<Step>
  <Name>RF-400MissingLatLng</Name>
  <Condition>request.queryparam.latitude == null OR
           request.queryparam.longitude == null</Condition>
</Step>
```

Faults can also be raised manually by using the RaiseFault policy.

In this example, the request is checked for two required parameters, latitude and longitude.

If either is null, we return an error to the client using a RaiseFault policy.



RaiseFault policy

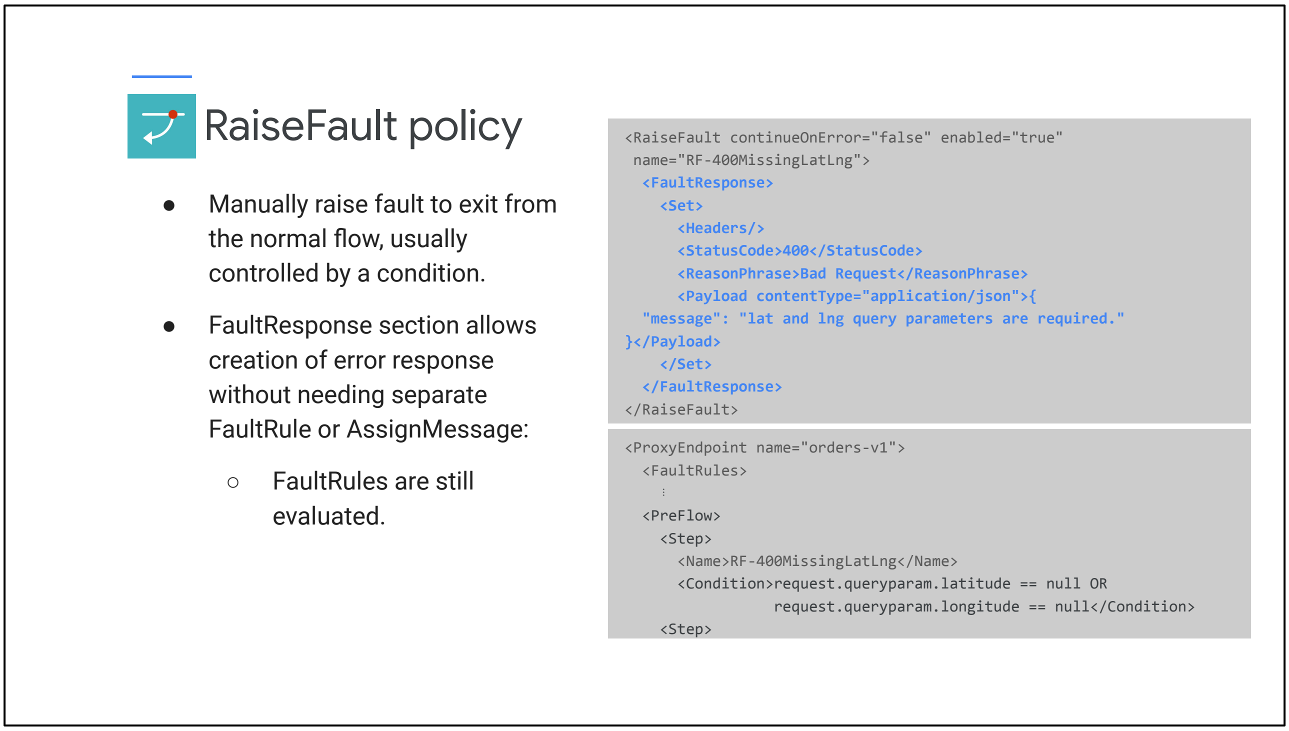
- Manually raise fault to exit from the normal flow, usually controlled by a condition.

```
<RaiseFault continueOnError="false" enabled="true"
 name="RF-400MissingLatLng">
  <FaultResponse>
    <Set>
      <Headers/>
      <StatusCode>400</StatusCode>
      <ReasonPhrase>Bad Request</ReasonPhrase>
      <Payload contentType="application/json">{
  "message": "lat and lng query parameters are required."
}</Payload>
    </Set>
  </FaultResponse>
</RaiseFault>
```

```
<ProxyEndpoint name="orders-v1">
  <FaultRules>
    ⋮
  <PreFlow>
    <Step>
      <Name>RF-400MissingLatLng</Name>
      <Condition>request.queryparam.latitude == null OR
                 request.queryparam.longitude == null</Condition>
    <Step>
```

The RaiseFault policy allows you to raise a fault manually, exiting from the normal flow into the FaultRules.

Typically, RaiseFaults are conditionally executed.



RaiseFault policy

- Manually raise fault to exit from the normal flow, usually controlled by a condition.
- FaultResponse section allows creation of error response without needing separate FaultRule or AssignMessage:
  - FaultRules are still evaluated.

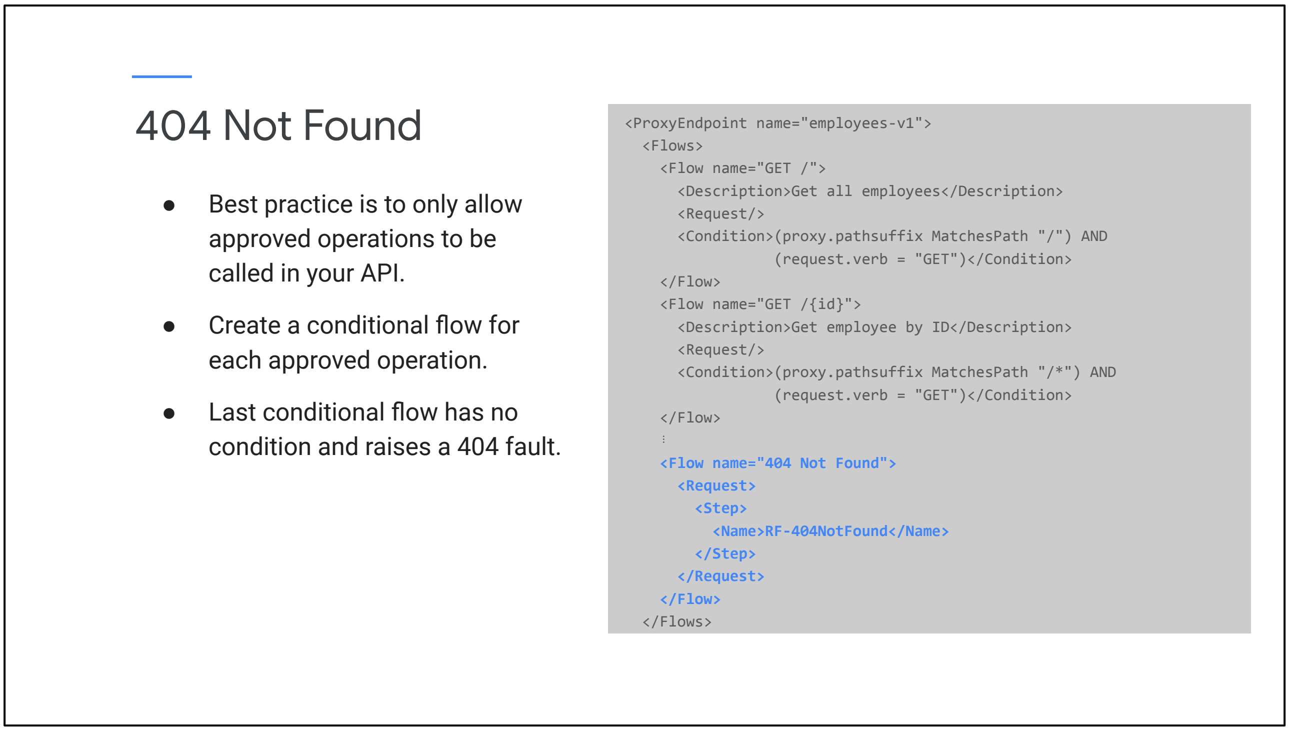
```
<RaiseFault continueOnError="false" enabled="true"
 name="RF-400MissingLatLng">
  <FaultResponse>
    <Set>
      <Headers/>
      <StatusCode>400</StatusCode>
      <ReasonPhrase>Bad Request</ReasonPhrase>
      <Payload contentType="application/json">{
  "message": "lat and lng query parameters are required."
}</Payload>
    </Set>
  </FaultResponse>
</RaiseFault>
```

```
<ProxyEndpoint name="orders-v1">
  <FaultRules>
    ⋮
  <PreFlow>
    <Step>
      <Name>RF-400MissingLatLng</Name>
      <Condition>request.queryparam.latitude == null OR
                 request.queryparam.longitude == null</Condition>
    <Step>
```

Within the RaiseFault policy, you can use the FaultResponse element to build the response message as you would in an AssignMessage policy.

This is generally easier than using an AssignMessage policy in the FaultRules to rewrite the response for your error conditions.

The FaultRules you've defined, if any, will still be evaluated.



404 Not Found

- Best practice is to only allow approved operations to be called in your API.
- Create a conditional flow for each approved operation.
- Last conditional flow has no condition and raises a 404 fault.

```
<ProxyEndpoint name="employees-v1">
  <Flows>
    <Flow name="GET /">
      <Description>Get all employees</Description>
      <Request/>
      <Condition>(proxy.pathsuffix MatchesPath "/") AND
                 (request.verb = "GET")</Condition>
    </Flow>
    <Flow name="GET /{id}">
      <Description>Get employee by ID</Description>
      <Request/>
      <Condition>(proxy.pathsuffix MatchesPath "/*") AND
                 (request.verb = "GET")</Condition>
    </Flow>
      ⋮
    <Flow name="404 Not Found">
      <Request>
        <Step>
          <Name>RF-404NotFound</Name>
        </Step>
      </Request>
    </Flow>
  </Flows>
```

When you create an API that is proxying a backend service, it is a best practice to only allow operations you have specifically approved through to the backend, and reject others with a 404 Not Found error.

Even if some of your backend calls are not documented, it is possible for bad actors to attempt to discover calls in your backend that they should not be calling.

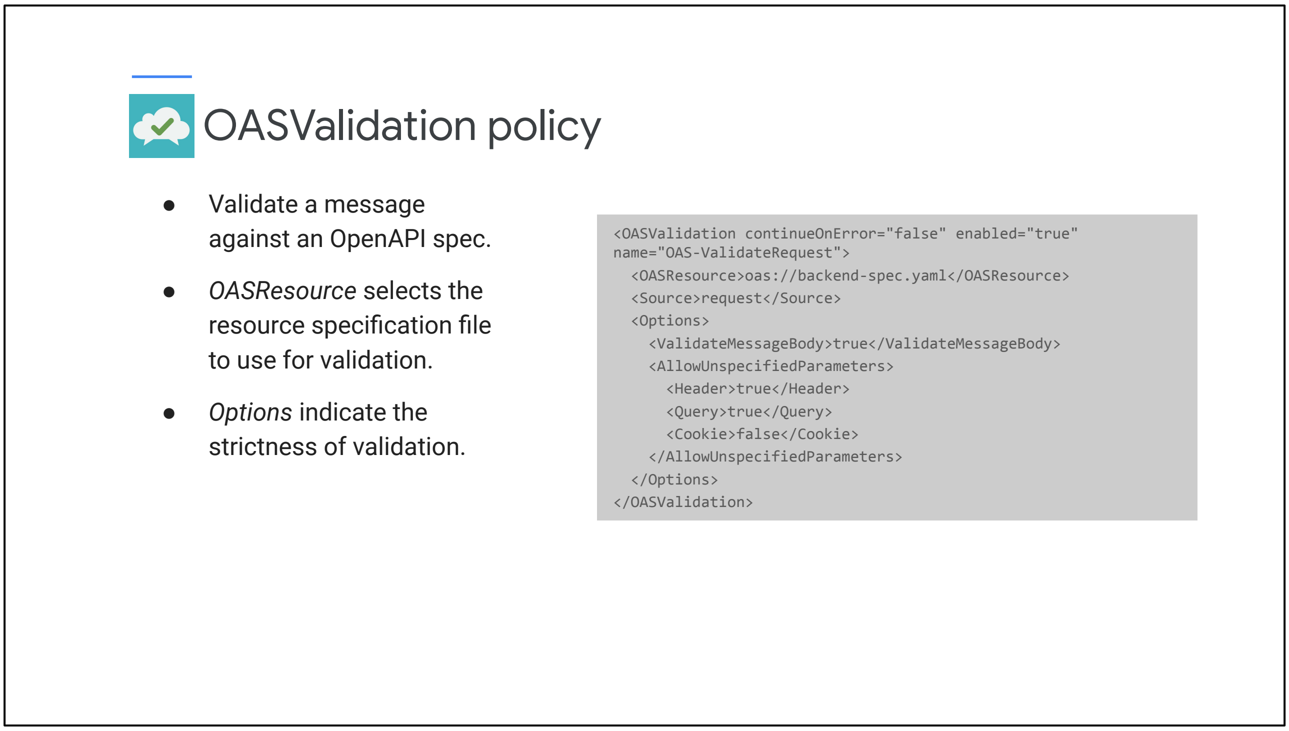
Also, if you are using your API proxy to protect against malicious data, you want to make sure that you have protected all calls to the backend. New backend calls that haven't been added to the proxy would not have code checking for data validity.

One way to specify the approved operations is to create a conditional flow in your ProxyEndpoint request flow for each operation that is allowed.

Any data validation required for the operation could be put in the conditional flow request pipeline.

The last conditional flow has no condition, and will therefore always execute if none of the other flow conditions are executed. In this flow you raise a fault with 404 Not Found.

With this pattern, calls that don't match one of the other conditional flows will never get through.



OASValidation policy

- Validate a message against an OpenAPI spec.
- *OASResource* selects the resource specification file to use for validation.
- *Options* indicate the strictness of validation.

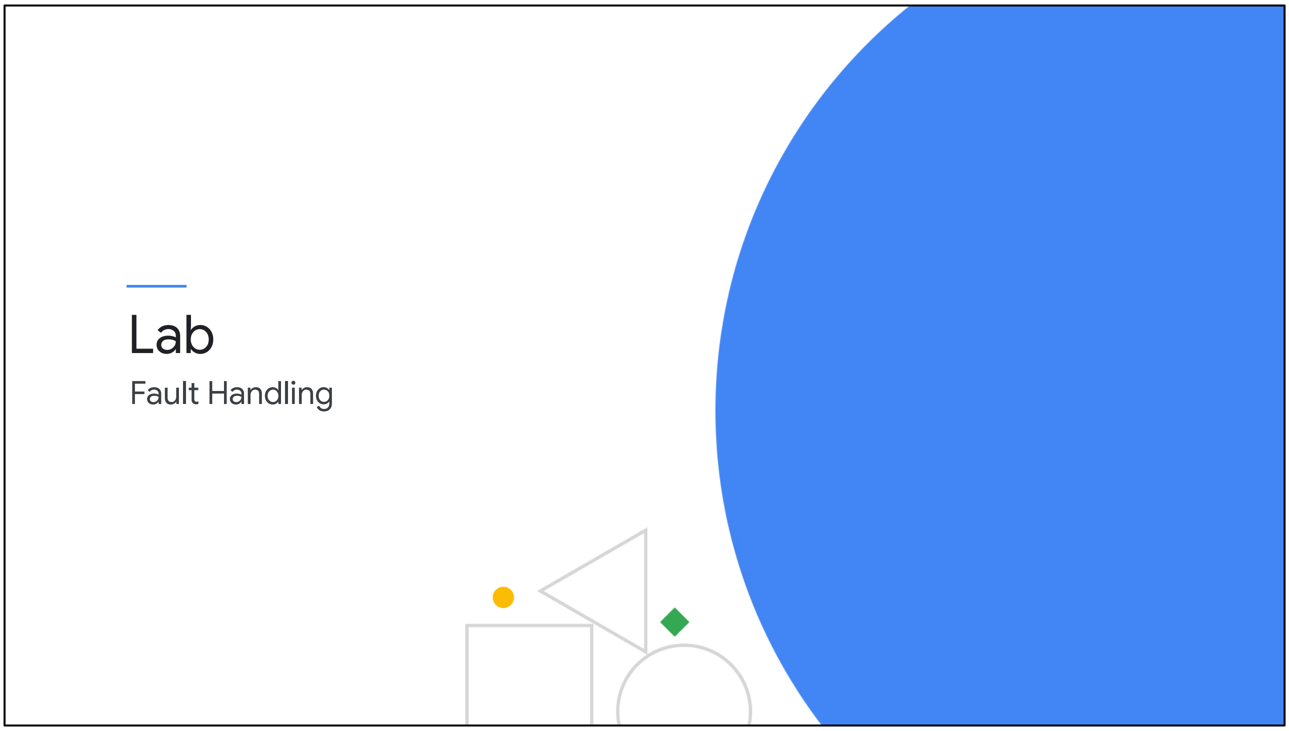
```
<OASValidation continueOnError="false" enabled="true"
name="OAS-ValidateRequest">
  <OASResource>oas://backend-spec.yaml</OASResource>
  <Source>request</Source>
  <Options>
    <ValidateMessageBody>true</ValidateMessageBody>
    <AllowUnspecifiedParameters>
      <Header>true</Header>
      <Query>true</Query>
      <Cookie>false</Cookie>
    </AllowUnspecifiedParameters>
  </Options>
</OASValidation>
```

Another way to protect your backend service is to allow only those calls which match an OpenAPI specification. The OASValidation policy enables you to validate an incoming JSON request or response message against an OpenAPI specification.

The OASResource element is used to specify the OpenAPI specification. This specification must be an OpenAPI version 3.0 specification that is stored as a resource in the proxy resources section.

The Options indicate how strict the validation should be. ValidateMessageBody must be set to true to validate the payload against the operation's schema in the OpenAPI specification. If set to false, which is the default, then only the existence of the body is checked.

AllowUnspecifiedParameters determines whether headers, query parameters, or cookies that are not in the specification should be allowed. By default, extraneous headers, query parameters, and cookies are allowed.



Lab

Fault Handling

In this lab you add fault handling to your retail API. You use fault rules to rewrite error messages, check the request for a content type header, and raise a fault if it is not included. You will also use the 404 Not Found pattern to reject operations that are not handled in the proxy.

Agenda

JSON, XML, and SOAP (lab)

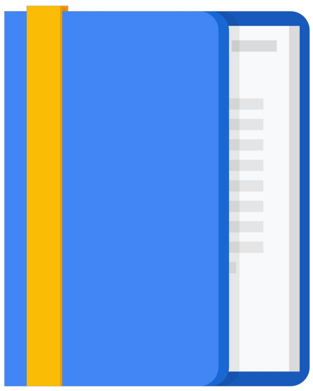
Mediation and Service Callouts

Custom Code (2 labs)

Shared Flows (lab)

Fault Handling (lab)

Quiz



Quiz

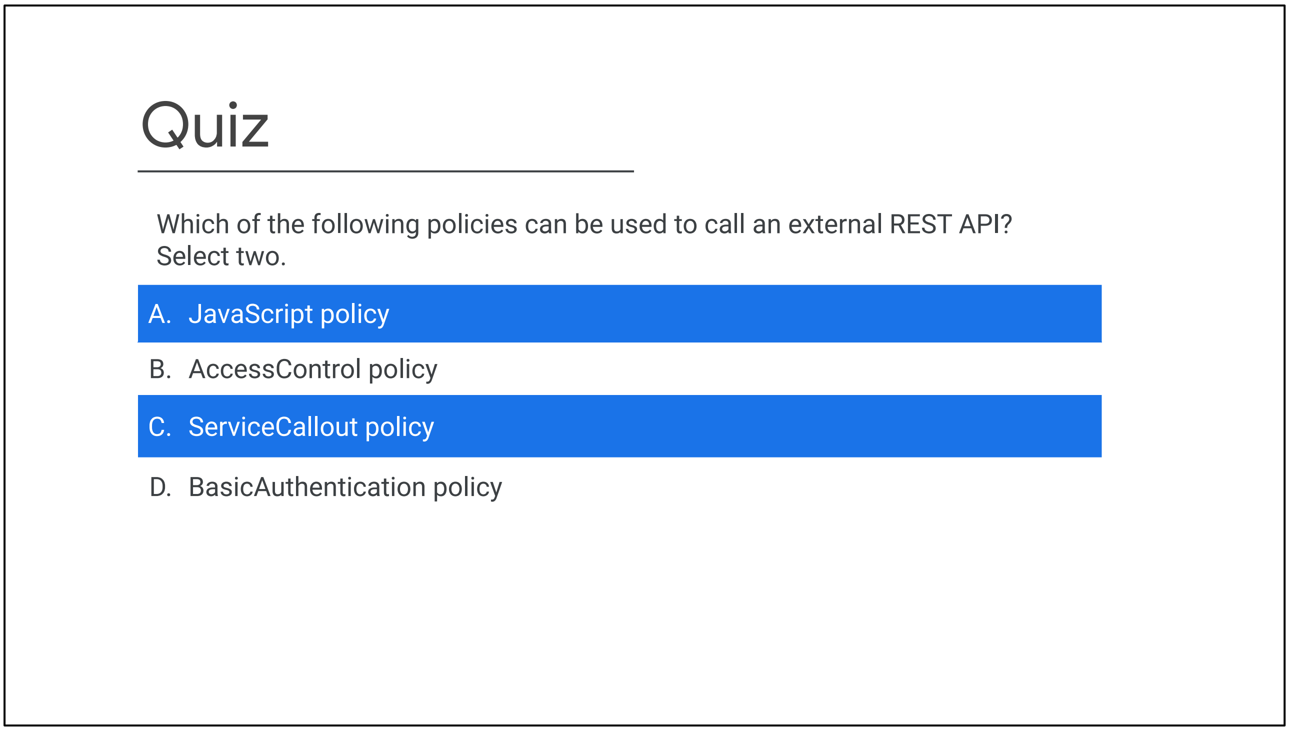
Which of the following policies can be used to call an external REST API?
Select two.

A. JavaScript policy

B. AccessControl policy

C. ServiceCallout policy

D. BasicAuthentication policy



Quiz

Which of the following policies can be used to call an external REST API? Select two.

A. JavaScript policy [✓]

B. AccessControl policy

C. ServiceCallout policy [✓]

D. BasicAuthentication policy

Explanation:

*A: JavaScript policy - Correct! The httpclient can be used to call a service from JavaScript code.

B: AccessControl policy - No, that's not correct. The AccessControl policy controls access by IP address.

*C: ServiceCallout policy - Correct! A ServiceCallout policy is used to call a service over HTTP, including REST APIs.

D: BasicAuthentication policy - No, that's not correct. The BasicAuthentication policy builds and parses Basic Authentication headers.

Quiz

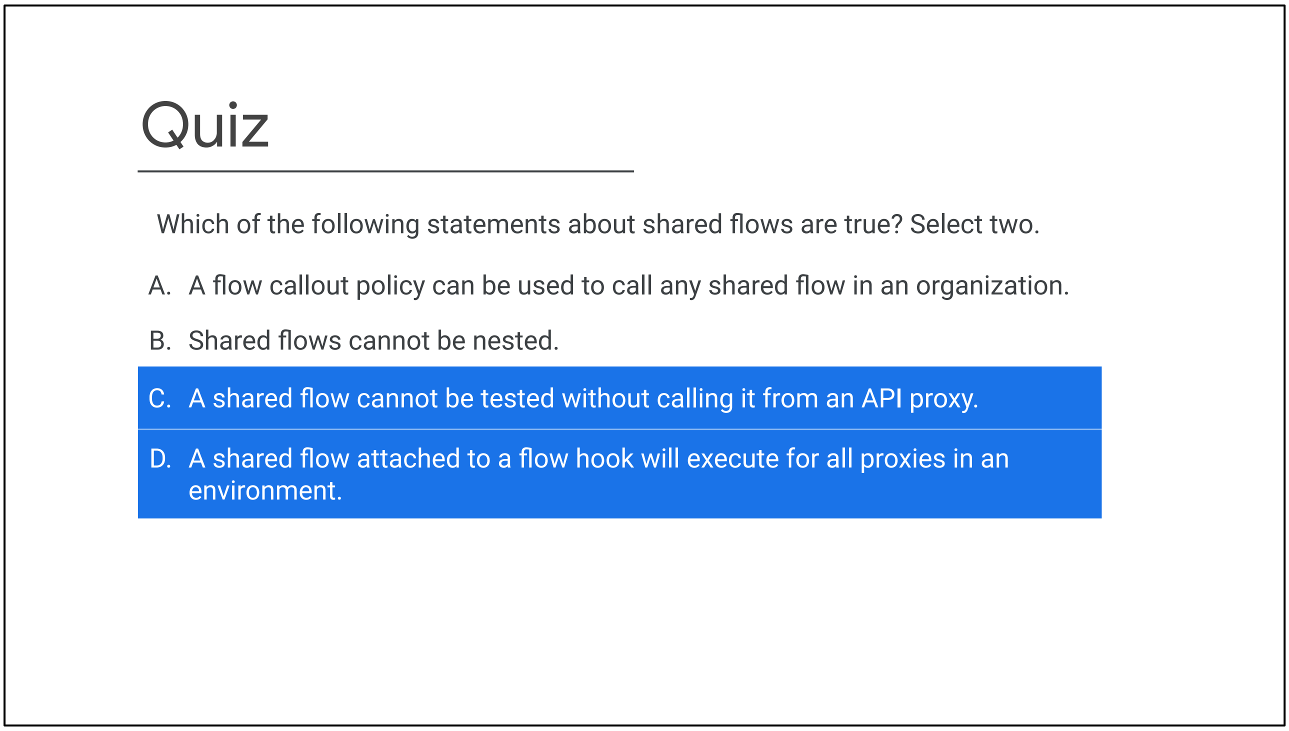
Which of the following statements about shared flows are true? Select two.

A. A flow callout policy can be used to call any shared flow in an organization.

B. Shared flows cannot be nested.

C. A shared flow cannot be tested without calling it from an API proxy.

D. A shared flow attached to a flow hook will execute for all proxies in an environment.



Quiz

Which of the following statements about shared flows are true? Select two.

A. A flow callout policy can be used to call any shared flow in an organization.

B. Shared flows cannot be nested.

[✓ C. A shared flow cannot be tested without calling it from an API proxy.]

[✓ D. A shared flow attached to a flow hook will execute for all proxies in an environment.]

Explanation:

A: A flow callout policy can be used to call any shared flow in an organization. - No, only shared flows deployed to the same environment as the API proxy can be called.

B: Shared flows cannot be nested. - No, that is not correct. Shared flows can be nested.

*C: A shared flow cannot be tested without calling it from an API proxy. - Correct!

*D: A shared flow attached to a flow hook will execute for all proxies in an environment. - Correct!

Quiz

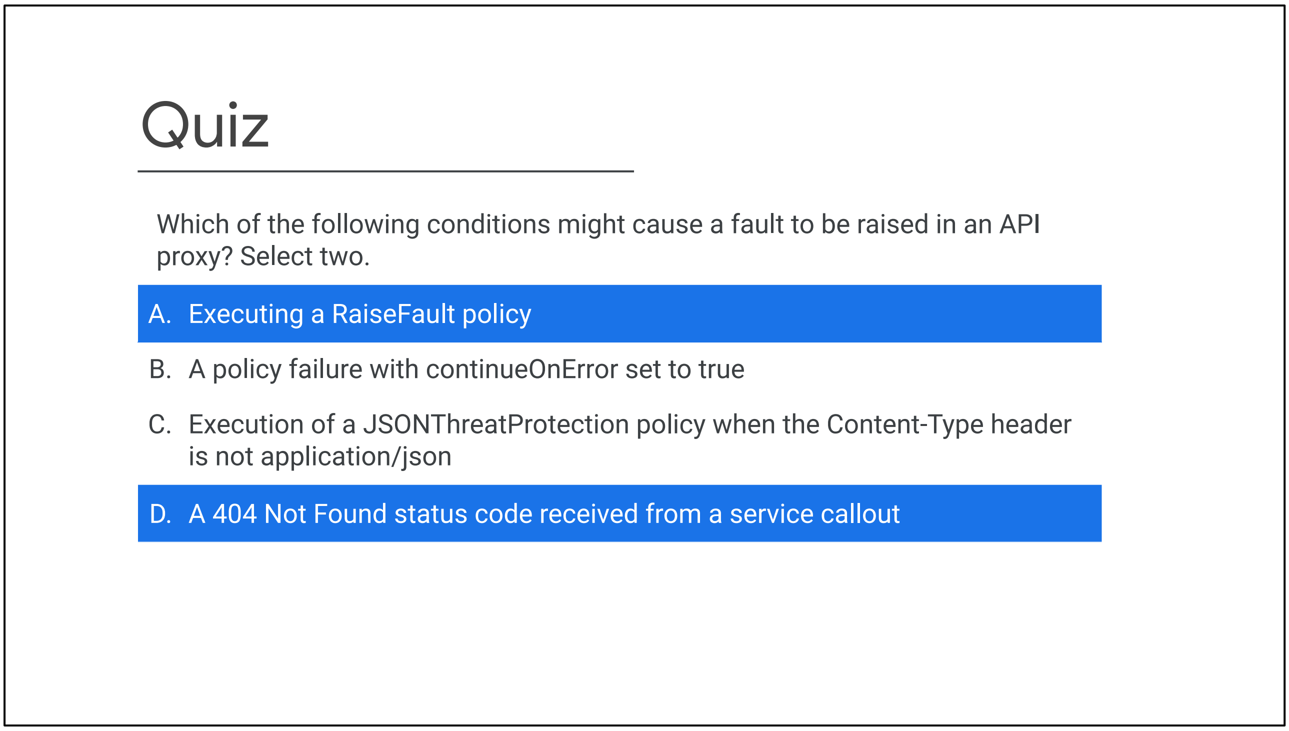
Which of the following conditions might cause a fault to be raised in an API proxy? Select two.

A. Executing a RaiseFault policy

B. A policy failure with continueOnError set to true

C. Execution of a JSONThreatProtection policy when the Content-Type header is not application/json

D. A 404 Not Found status code received from a service callout



Quiz

Which of the following conditions might cause a fault to be raised in an API proxy? Select two.

A. Executing a RaiseFault policy

B. A policy failure with continueOnError set to true

C. Execution of a JSONThreatProtection policy when the Content-Type header is not application/json

D. A 404 Not Found status code received from a service callout

Explanation:

*A: Executing a RaiseFault policy - Correct!

B: A policy failure with continueOnError set to true - No, continueOnError set to true blocks the fault from being raised.

C: Execution of a JSONThreatProtection policy when the Content-Type header is not application/json - No, the JSONThreatProtection policy functionality will be silently skipped if the content type is not JSON.

*D: A 404 Not Found status code received from a service callout - Correct! By default, all 4XX or 5XX status codes returned from a service callout will cause a fault to be raised.

Quiz

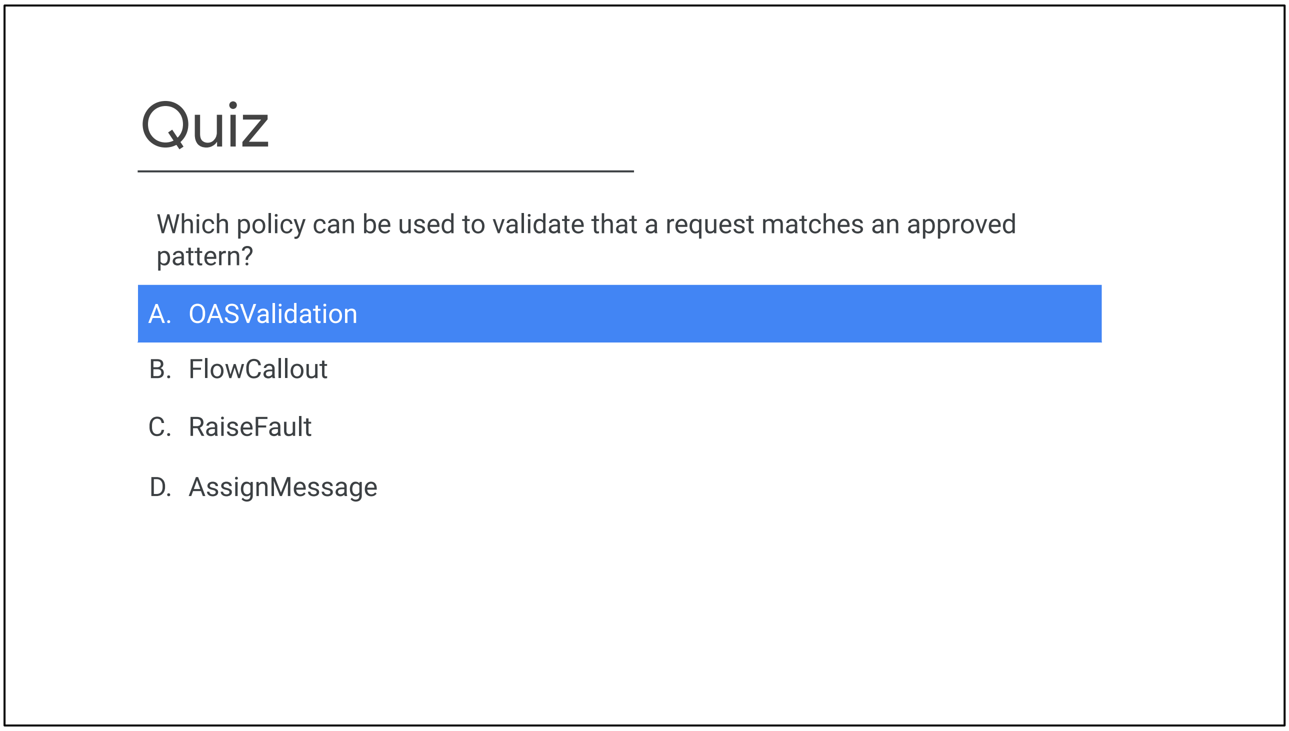
Which policy can be used to validate that a request matches an approved pattern?

A. OASValidation

B. FlowCallout

C. RaiseFault

D. AssignMessage



Quiz

Which policy can be used to validate that a request matches an approved pattern?

[✓ A. OASValidation]

B. FlowCallout

C. RaiseFault

D. AssignMessage

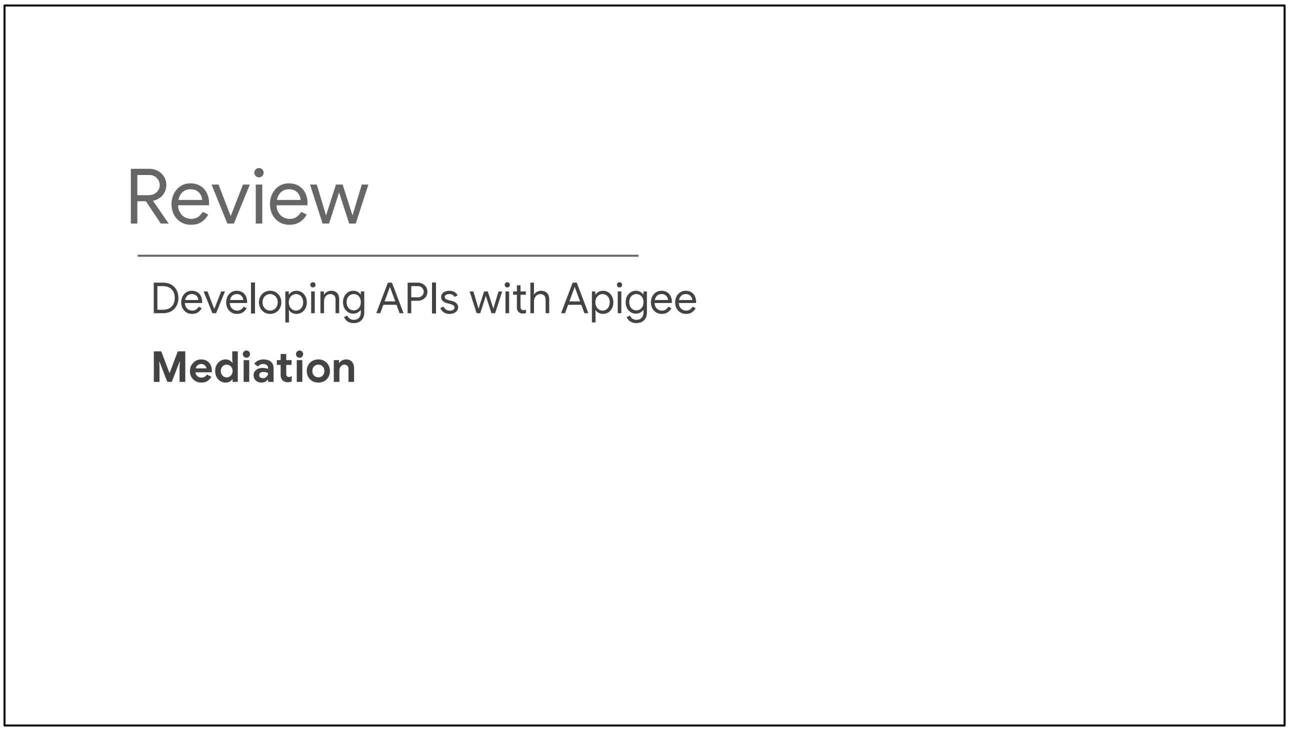
Explanation:

*A: OASValidation - Correct! The OASValidation policy can validate that a request matches an operation in an OpenAPI specification.

B: FlowCallout - No, FlowCallout cannot validate a request.

C: RaiseFault - No, RaiseFault cannot validate a request.

D: AssignMessage - No, AssignMessage cannot validate a request.



Review

Developing APIs with Apigee

Mediation

In this module you were introduced to API mediation.

You learned about XML, JSON, and SOAP, and how you can manage these different formats in your API proxies.

We discussed mediation patterns, using the ServiceCallout policy to call external services, and writing custom code in our proxies.

We learned about sharing proxy logic between API proxies using shared flows, and fault handling in proxies.

You completed several labs, learning about mediation, shared flows, and fault handling.