

Modernizing Applications with Apigee X

experiment Lab schedule 1 hour 30 minutes universal_currency_alt No cost

show_chart Intermediate

GSP842



Google Cloud Self-Paced Labs

Overview

Google Cloud's Apigee API Platform can be used to modernize existing applications by adding new functionality to your existing APIs.

In this lab, you deploy a backend service on Cloud Run. The backend service implements a REST API that stores and retrieves bank data (customers, accounts, ATMs, and transactions) in a Firestore database. You create an Apigee API proxy that proxies the backend service. You also create a shared flow that retrieves and caches content from an external service. You then call that shared flow from your API proxy and use JavaScript code to modify an API response.

Objectives

In this lab, you learn how to perform the following tasks:

- Deploy a backend service on Cloud Run
- Proxy a backend service using an Apigee X proxy
- Create a shared flow for functionality that can be used by multiple proxies
- Store configuration data in a property set
- Use a service callout policy to retrieve content from a service
- Use cache policies to cache reusable information
- Use JavaScript code to modify a response payload

Setup

Before you click the Start Lab button

Read these instructions. Labs are timed and you cannot pause them. The timer, which starts when you click **Start Lab**, shows how long Google Cloud resources will be made available to you.

This hands-on lab lets you do the lab activities yourself in a real cloud environment, not in a simulation or demo environment. It does so by giving you new, temporary credentials that you use to sign in and access Google Cloud for the duration of the lab.

To complete this lab, you need:

- Access to a standard internet browser (Chrome browser recommended).

Note: Use an Incognito or private browser window to run this lab. This prevents any conflicts between your personal account and the Student account, which may cause extra charges incurred to your personal account.

- Time to complete the lab---remember, once you start, you cannot pause a lab.

Note: If you already have your own personal Google Cloud account or project, do not use it for this lab to avoid extra charges to your account.

How to start your lab and sign in to the Google Cloud console

1. Click the **Start Lab** button. If you need to pay for the lab, a pop-up opens for you to select your payment method. On the left is the **Lab Details** panel with the following:

- The **Open Google Cloud console** button
- Time remaining
- The temporary credentials that you must use for this lab
- Other information, if needed, to step through this lab

2. Click **Open Google Cloud console** (or right-click and select **Open Link in Incognito Window** if you are running the Chrome browser).

The lab spins up resources, and then opens another tab that shows the **Sign in** page.

Tip: Arrange the tabs in separate windows, side-by-side.

Note: If you see the **Choose an account** dialog, click **Use Another Account**.

3. If necessary, copy the **Username** below and paste it into the **Sign in** dialog.

"Username"

content_co

You can also find the **Username** in the **Lab Details** panel.

4. Click **Next**.

5. Copy the **Password** below and paste it into the **Welcome** dialog.

"Password"

content_co

You can also find the **Password** in the **Lab Details** panel.

6. Click **Next**.

Important: You must use the credentials the lab provides you. Do not use your Google Cloud account credentials.

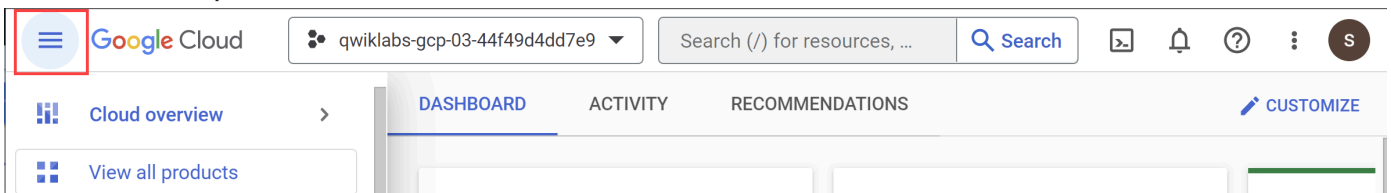
Note: Using your own Google Cloud account for this lab may incur extra charges.

7. Click through the subsequent pages:

- Accept the terms and conditions.
- Do not add recovery options or two-factor authentication (because this is a temporary account).
- Do not sign up for free trials.


After a few moments, the Google Cloud console opens in this tab.

Note: To view a menu with a list of Google Cloud products and services, click the **Navigation menu** at the top-left.



Activate Cloud Shell

Cloud Shell is a virtual machine that is loaded with development tools. It offers a persistent 5GB home directory and runs on the Google Cloud. Cloud Shell provides command-line access to your Google Cloud resources.

1. Click **Activate Cloud Shell**  at the top of the Google Cloud console.

When you are connected, you are already authenticated, and the project is set to your **Project_ID**, PROJECT_ID. The output contains a line that declares the **Project_ID** for this session:

```
Your Cloud Platform project in this session is set to "PROJECT_ID"
```

gcloud is the command-line tool for Google Cloud. It comes pre-installed on Cloud Shell and supports tab-completion.

2. (Optional) You can list the active account name with this command:

```
gcloud auth list
```

content_co

3. Click **Authorize**.

Output:

```
ACTIVE: *  
ACCOUNT: "ACCOUNT"  
  
To set the active account, run:  
$ gcloud config set account `ACCOUNT`
```

4. (Optional) You can list the project ID with this command:

```
gcloud config list project
```

content_co

Output:

```
[core]
```

```
project = "PROJECT_ID"
```

Note: For full documentation of `gcloud`, in Google Cloud, refer to the [gcloud CLI overview guide](#).

Open the Apigee UI

The Apigee UI is accessed on a page separate from the Google Cloud Console. This lab has automatically created an Apigee organization that has the same name as the Google Cloud project.

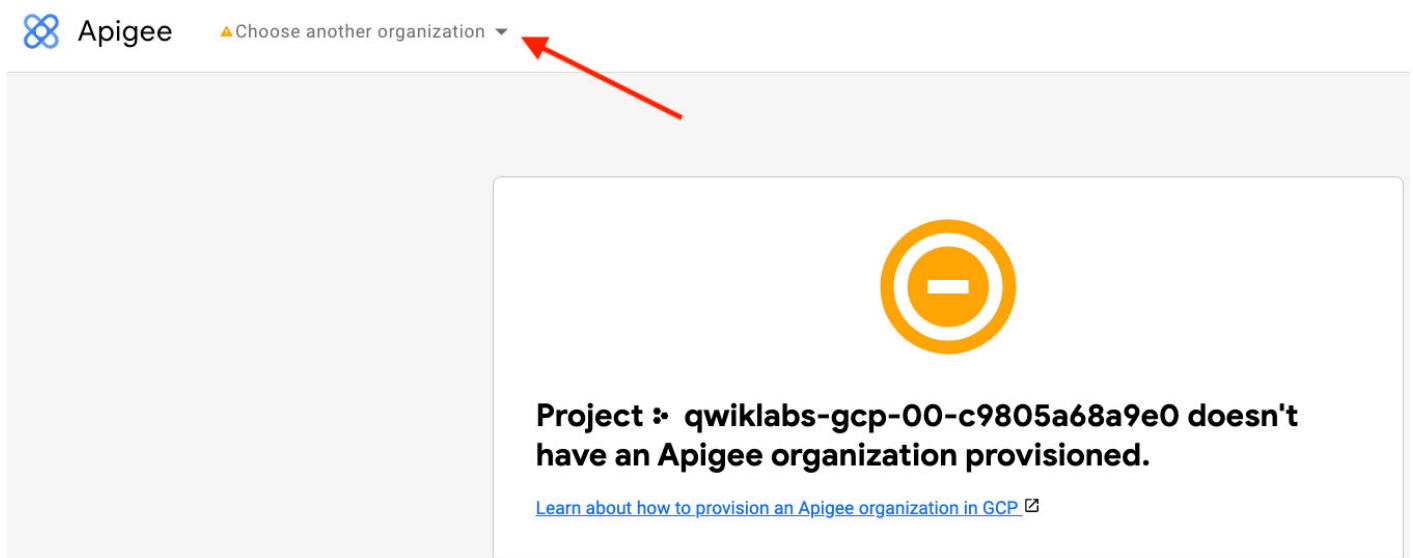
- Click to open the Apigee UI.

You may also open the Apigee UI from the Google Cloud Console by opening the **Navigation menu** (≡) and selecting **Apigee API Management > Apigee**.

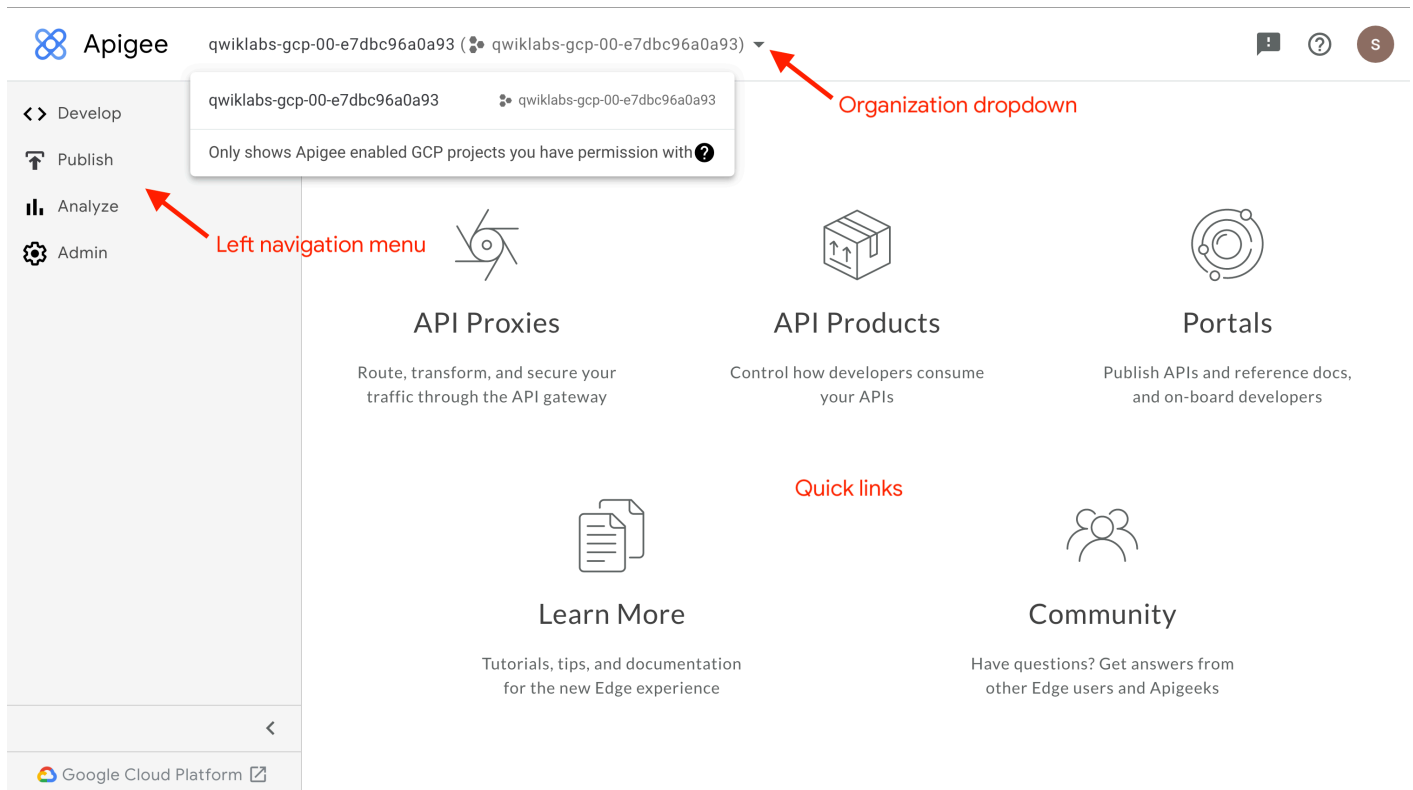
If you see an error indicating that the project does not have an organization provisioned, the tab might be trying to load the organization for a previous lab.

If you get this error:

- Click on the organization dropdown.



The organization dropdown should show an organization that has the same name as the Google Cloud project.



The organizations listed are those that are accessible by the logged-in user. For this lab, you should be logged in with the lab credentials provided in the **Lab Details** panel when you started the lab.

You can navigate the Apigee UI using its **left navigation menu**. The landing page also shows **quick links** for navigating to commonly used locations.

Task 1. Deploy a backend service on Cloud Run

In this task, you deploy a backend service on Cloud Run.

The service implements an API for SimpleBank. This API provides a simple representation of a bank, with customers, accounts, transactions, and ATMs. The SimpleBank service is built using Node.js with data stored in Firestore. The code is packaged in a Docker container, and this container is deployed to Cloud Run.

Clone the code repository

1. To clone the repository containing the code for the SimpleBank service, in Cloud Shell, run the following command:

```
git clone --depth 1  
https://github.com/GoogleCloudPlatform/training-data-analyst
```

content_copy

2. Create a soft link to the working directory:

```
ln -s ~/training-data-analyst/quests/develop-apis-apigee  
~/develop-apis-apigee
```

content_copy

3. To change to the directory containing the REST backend, run the following command:

```
cd ~/develop-apis-apigee/rest-backend
```

content_copy

Initialize the project

The project initialization script, **init-project.sh**, enables the Cloud Run API within the project. These APIs will be required to deploy Cloud Run services.

The database for the service will be Firestore in Native mode. A project can host a single Firestore database in either Native mode or Datastore mode. This script will create the Firestore in Native mode database.

1. To see the commands run by the **init-project.sh** script, enter this command:

```
cat init-project.sh
```

content_copy

The script enables the Cloud Run API and creates the Firestore in Native mode database.

2. To run the script, enter this command:

```
./init-project.sh
```

content_co

Click **check my progress** to verify the objective.



Enables the Cloud Run API and creates the Firestore database

Check my progress

Initialize the service

The service initialization script, **init-service.sh**, creates a service account named **simplebank-rest**. This service account is used as the identity for the Cloud Run service. The service account is given the role **roles/datastore.user**, which allows the service to read and update data in Firestore.

It is a best practice to create a service account for a service you create, and give permissions to the account using the principle of least privilege. This principle says that an account should have only those privileges essential to perform its unique function.

1. To see the commands run by the **init-service.sh** script, enter this command:

```
cat init-service.sh
```

content_co

The script creates the service account used by the service and adds the *roles/datastore.user* role to the service account.

2. To run the script, enter this command:

```
./init-service.sh
```

content_co

Deploy the backend service

The deploy script, **deploy.sh**, builds the simplebank service application using the code in the current directory. It then deploys the service to Cloud Run, using the **simplebank-rest** service account. The deploy script would be run each time you updated your application code.

The service is deployed to require authenticated access, so you cannot call the service without a valid OpenID Connect identity token.

1. In Cloud Shell, to see the commands run by the **deploy.sh** script, enter this command:

```
cat deploy.sh
```

content_co

The script builds the service using Cloud Build, and then deploys the **simplebank-grpc** service to Cloud Run.

Note: The deploy script uses the max-instances parameter to limit the number of instances in the Cloud Run cluster to 1. For a real-world production service, you would not want to specify a limit this low.

2. To deploy the script to Cloud Run, enter this command:

```
./deploy.sh
```

content_co

Click **check my progress** to verify the objective.



Deploy the backend service

[Check my progress](#)

Test the service

1. To verify that the service is running, make a curl request that calls the service:

```
export RESTHOST=$(gcloud run services describe simplebank-rest -  
-platform managed --region us-west1 --format  
'value(status.url)')  
echo "export RESTHOST=${RESTHOST}" >> ~/.bashrc  
curl -H "Authorization: Bearer $(gcloud auth print-identity-  
token)" -X GET "${RESTHOST}/_status"
```

content_copy

The command that sets the *RESTHOST* variable uses *gcloud* to retrieve the hostname for the *simplebank-rest* Cloud Run service. The variable is then added to the *.bashrc* file, which will cause the *RESTHOST* variable to be reloaded if Cloud Shell restarts.

The **GET /_status** command simply returns a JSON response indicating that the API is up and running. In this call you used **gcloud auth print-identity-token** to get an Open ID Connect identity token for the user logged in to Cloud Shell. You are logged in with the project owner role, and the project owner is given very broad permissions.

2. To verify that the service can write to Firestore, make a curl request that creates a customer:

```
curl -H "Authorization: Bearer $(gcloud auth print-identity-  
token)" -H "Content-Type: application/json" -X POST  
"${RESTHOST}/customers" -d '{"lastName": "Diallo", "firstName":  
"Temeka", "email": "temeka@example.com"}'
```

content_copy

The **POST /customers** command creates a customer. The *lastName*, *firstName*, and *email* parameters are all required. The email address must be unique and is used as the identifier for the customer. The

customer record is stored in Firestore.

Note: If you receive an "AlreadyExist" error, it may indicate that you have not successfully created the Firestore database. The database is created by running the `init-project.sh` script.

3. To verify that the service can read from Firestore, make a curl request to retrieve the customer that you just created:

```
curl -H "Authorization: Bearer $(gcloud auth print-identity-token)" -X GET "${RESTHOST}/customers/temeka@example.com" content_copy
```

The **GET /customers/** command retrieves a customer record from Firestore.

4. To load some additional sample data into Firestore, enter this command:

```
gcloud firestore import gs://cloud-training/api-dev-quest/firestore/example-data content_copy
```

This *gcloud* command will use the Firestore import/export feature to import customers, accounts, and ATMs into the database.

5. To retrieve the list of ATMs, run this curl command:

```
curl -H "Authorization: Bearer $(gcloud auth print-identity-token)" -X GET "${RESTHOST}/atms" content_copy
```

6. To retrieve a single ATM, run this curl command:

```
curl -H "Authorization: Bearer $(gcloud auth print-identity-token)" -X GET "${RESTHOST}/atms/spruce-goose" content_copy
```

The request retrieves an ATM by name, and the response contains the latitude and longitude for the ATM, but it does not contain an address:

```
{ "name": "spruce-goose", "longitude": -118.408207, "description": "", "latitude":
```

In a later task, you will use Apigee and the Geocoding API to add an address to the response returned when retrieving a specific ATM.

Task 2. Proxy the backend service with an Apigee API proxy

In this task, you create an Apigee API proxy that acts as a facade for the backend service. The API proxy will use a service account to allow it to present OpenID Connect identity tokens to the Cloud Run service.

Create a service account for the Apigee API proxy

1. To create a service account that can be used by the Apigee API proxy, enter this command:

```
gcloud iam service-accounts create apigee-internal-access \
  --display-name="Service account for internal access by Apigee
proxies" \
  --project=${GOOGLE_CLOUD_PROJECT}
```

The *gcloud* command creates a service account named **apigee-internal-access**, which will be used by your Apigee proxy when calling the backend service.

2. To grant the role that allows access to the service, enter this command:

```
gcloud run services add-iam-policy-binding simplebank-rest \
--member="serviceAccount:apigee-internal-
access@${GOOGLE_CLOUD_PROJECT}.iam.gserviceaccount.com" \
--role=roles/run.invoker --region=us-west1 \
--project=${GOOGLE_CLOUD_PROJECT}
```

content_co

This *gcloud* command grants the service account the **roles/run.invoker** role for the *simplebank-rest* Cloud Run service, allowing the service account to invoke the service.

3. To retrieve the URL for the backend service, use the following command:

```
gcloud run services describe simplebank-rest --platform managed
--region us-west1 --format 'value(status.url)'
```

content_co

Save this URL. It will be used when creating the API proxy.

Click **check my progress** to verify the objective. There may be a short delay before the granted role is detected.



Create a service account for the Apigee API proxy

Check my progress

Create the Apigee proxy

1. Select the Apigee UI tab in your browser window.
2. On the left navigation menu, select **Develop > API Proxies**.
3. To create a new proxy using the proxy wizard, click **Create New**.

You will create a reverse proxy for your backend service.

4. Click the **Reverse proxy** box.

Note: Do not click the "Use OpenAPI Spec" link within the reverse proxy box.

5. Specify the following for the **Proxy details**:

Property	Value
Name	bank-v1
Base path	/bank/v1
Target (Existing API)	<i>backend URL</i>

Note: Confirm that you are using "/bank/v1" for the base path, not "/bank-v1".

The target should be the backend URL you retrieved earlier in the task, which should look something like this:

```
https://simplebank-rest-mtdtzt7yzq-ue.a.run.app
```

6. Click **Next**.

7. Leave the Common Policies settings at their defaults, and click **Next**.

8. On the summary page, leave the settings at their defaults, and click **Create**.

9. Click **Edit proxy**.

10. If a **Switch to Classic** link is in the upper right corner, click that link.

Confirm that the runtime instance is available

1. In Cloud Shell, paste and run the following set of commands:

```
export INSTANCE_NAME=eval-instance; export ENV_NAME=eval; export
PREV_INSTANCE_STATE=; echo "waiting for runtime instance
${INSTANCE_NAME} to be active"; while : ; do export
INSTANCE_STATE=$(curl -s -H "Authorization: Bearer $(gcloud auth
print-access-token)" -X GET
"https://apigee.googleapis.com/v1/organizations/${GOOGLE_CLOUD_PROJ
| jq "select(.state != null) | .state" --raw-output); [[
"${INSTANCE_STATE}" == "${PREV_INSTANCE_STATE}" ]] || (echo;
echo "INSTANCE_STATE=${INSTANCE_STATE}"); export
PREV_INSTANCE_STATE=${INSTANCE_STATE}; [[ "${INSTANCE_STATE}" !=
"ACTIVE" ]] || break; echo -n "."; sleep 5; done; echo; echo
"instance created, waiting for environment ${ENV_NAME} to be
attached to instance"; while : ; do export
ATTACHMENT_DONE=$(curl -s -H "Authorization: Bearer $(gcloud
auth print-access-token)" -X GET
"https://apigee.googleapis.com/v1/organizations/${GOOGLE_CLOUD_PROJ
| jq "select(.attachments != null) | .attachments[] |
select(.environment == \"${ENV_NAME}\") | .environment" --join-
output); [[ "${ATTACHMENT_DONE}" != "${ENV_NAME}" ]] || break;
echo -n "."; sleep 5; done; echo "***ORG IS READY TO USE***";
```

This series of commands uses the Apigee API to determine when the Apigee runtime instance has been created and the eval environment has been attached.

2. Wait until the instance is ready.

When the text `***ORG IS READY TO USE***` is displayed, the instance is ready. The Apigee organization (org) may have been created before you started the lab, so you might not have to wait for the instance to be created.

If you are waiting for the org to be ready, you can learn more about Apigee, explore the Apigee X architecture, or learn about APIs and API proxies.

Deploy the API proxy

1. Select the Apigee UI tab in your browser window.
2. On the left navigation menu, select **Develop > API Proxies**, and then click **bank-v1**.
3. Click the **Develop** tab.
4. Click **Deploy to eval**.

A dialog asks you to confirm the deployment.

5. For *Service Account*, specify the service account's email address:

```
apigee-internal-access@PROJECT.iam.gserviceaccount.com
```

content_co

6. Click **Deploy**.
7. Click the **Overview** tab, and wait for the **eval** deployment status to show that the proxy has been deployed.

Test the API proxy

The eval environment in the Apigee organization can be called using the hostname *eval.example.com*. The DNS entry for this hostname has been created within your project, and it resolves to the IP address of the Apigee runtime instance. This DNS entry has been created in a private zone, which means it is only visible on the internal network.

Cloud Shell does not reside on the internal network, so Cloud Shell commands cannot resolve this DNS entry. A virtual machine (VM) within your project can access the private zone DNS. A virtual machine named **apigee-test-vm** was automatically created. You can use this machine to call the API proxy.

1. In Cloud Shell, open an SSH connection to your test VM:

```
TEST_VM_ZONE=$(gcloud compute instances list --filter="name=  
( 'apigee-test-vm' )" --format "value(zone)")
```

content_co

```
gcloud compute ssh apigeex-test-vm --zone=${TEST_VM_ZONE} --  
force-key-file-overwrite
```

The first *gcloud* command retrieves the zone of the test VM, and the second opens the SSH connection to the VM.

2. If asked to authorize, click **Authorize**.

For each question asked in the Cloud Shell, click **Enter** or **Return** to specify the default input.

Your logged in identity is the owner of the project, so SSH to this machine is allowed.

Your Cloud Shell session is now running inside the VM.

3. Call the deployed **bank-v1** API proxy in the **eval** environment:

```
curl -i -k "https://eval.example.com/bank/v1/_status"
```

content_c

The `-k` option tells `curl` to skip verification of the TLS certificate. The Apigee runtime is using a self-signed certificate instead of a certificate that has been created by a trusted certificate authority (CA).

Note: You should not use the `-k` option to bypass certificate verification for production use cases.

A 403 Forbidden status code is returned, with an error message indicating that your client does not have permission to get the URL. The request to the backend service was rejected because the client did not provide the required token with the request. The API proxy is running with the correct identity, but you still need to force the OpenId Connect identity token to be sent with the request.

4. Return to the **bank-v1** proxy in the Apigee UI, and click the **Develop** tab.
5. In the Navigator menu for the proxy, in the **Target Endpoints** section, click **PreFlow**.
6. Find the following code (your URL will be different):

```
<HTTPTargetConnection>
  <URL>https://simplebank-rest-zce6j3rjwq-uw.a.run.app</URL>
</HTTPTargetConnection>
```

Note: If you do not see the HTTPTargetConnection section, make sure you have clicked on the PreFlow in the Target Endpoints section, not in the Proxy Endpoints section.

7. Below the URL, add an *Authentication* section that looks like this:

```
<Authentication>
  <GoogleIDToken>
    <Audience>AUDIENCE</Audience>
  </GoogleIDToken>
</Authentication>
```

content_co

8. Replace *AUDIENCE* with the URL value already in the *HTTPTargetConnection* section. Your code should now look similar to this, except with your specific URL in the URL and Audience elements:

```
<TargetEndpoint name="default">
  <PreFlow name="PreFlow">
    <Request/>
    <Response/>
  </PreFlow>
  <Flows/>
  <PostFlow name="PostFlow">
    <Request/>
    <Response/>
  </PostFlow>
  <HTTPTargetConnection>
    <URL>https://simplebank-rest-zce6j3rjwq-uw.a.run.app</URL>
    <Authentication>
      <GoogleIDToken>
        <Audience>https://simplebank-rest-zce6j3rjwq-
uw.a.run.app</Audience>
      </GoogleIDToken>
    </Authentication>
```

```
</HTTPTargetConnection>  
</TargetEndpoint>
```

9. Click **Save**, and then click **OK**.
10. Click **Deploy to eval**, and then click **Deploy**.

Note: Do not change the service account. It should remain unchanged from the previous deployment.

11. Click the **Overview** tab, and wait for the **eval** deployment status to show that the new revision has been deployed.

Click **check my progress** to verify the objective.



Create the Apigee proxy

Check my progress

12. If your SSH login has timed out, run this command in Cloud Shell to reestablish a connection:

```
TEST_VM_ZONE=$(gcloud compute instances list --filter="name=  
( 'apigee-test-vm' )" --format "value(zone)")  
gcloud compute ssh apigee-test-vm --zone=${TEST_VM_ZONE} --  
force-key-file-overwrite
```

content_co

13. Rerun the status command inside the VM:

```
curl -i -k "https://eval.example.com/bank/v1/_status"
```

content_co

You should now see a successful (200) response similar to this:

```
HTTP/2 200
x-powered-by: Express
content-type: application/json; charset=utf-8
etag: W/"41-x4uozCo6q/yN+kzizriXxryNZvc"
x-cloud-trace-context: 5c810a7faa3353bcc085473fd58805b7
date: Thu, 11 Nov 2021 22:54:35 GMT
server: Google Frontend
content-length: 65
x-request-id: cf109193-6d6f-49a1-b323-7f66f63c5e28
via: 1.1 google

{"serviceName":"simplebank-rest","status":"API up","ver":"1.0.0"}
```

This response indicates that the API proxy is successfully calling the backend service.

14. Enter the command `exit` to leave the SSH session and return to Cloud Shell.

Task 3. Enable use of the Google Cloud Geocoding API

In this task, you enable the Geocoding API, which will be used in your API proxy to add address information to the response when retrieving an ATM from the SimpleBank service.

1. In Cloud Shell, to enable the Geocoding API, run the following command:

```
gcloud services enable geocoding-backend.googleapis.com
```

content_c

Next, you will create an API key that can access the Geocoding API.

2. To create the API key, run the following command:

```
API_KEY=$(gcloud alpha services api-keys create --
project=${GOOGLE_CLOUD_PROJECT} --display-name="Geocoding API
key for Apigee" --api-target=service=geocoding_backend --format
"value(response.keyString)")
echo "export API_KEY=${API_KEY}" >> ~/.bashrc
echo "API_KEY=${API_KEY}"
```

content_co

Note: If you receive an error indicating that the project property is set to the empty string, make sure you have exited the VM SSH session and returned to Cloud Shell.

The *gcloud* command creates the API key with the ability to send requests to the Geocoding API. By supplying the *--format* parameter, you select the *keyString* field in the response and store it in the *API_KEY* shell variable. The *API_KEY* variable is then stored in the *.bashrc* file in Cloud Shell.

Click **check my progress** to verify the objective.



Enable use of the Google Cloud Geocoding API

Check my progress

3. To retrieve the geocoding information for a given latitude and longitude, run the following curl command:

```
curl "https://maps.googleapis.com/maps/api/geocode/json?
key=${API_KEY}&latlng=37.404934,-122.021411"
```

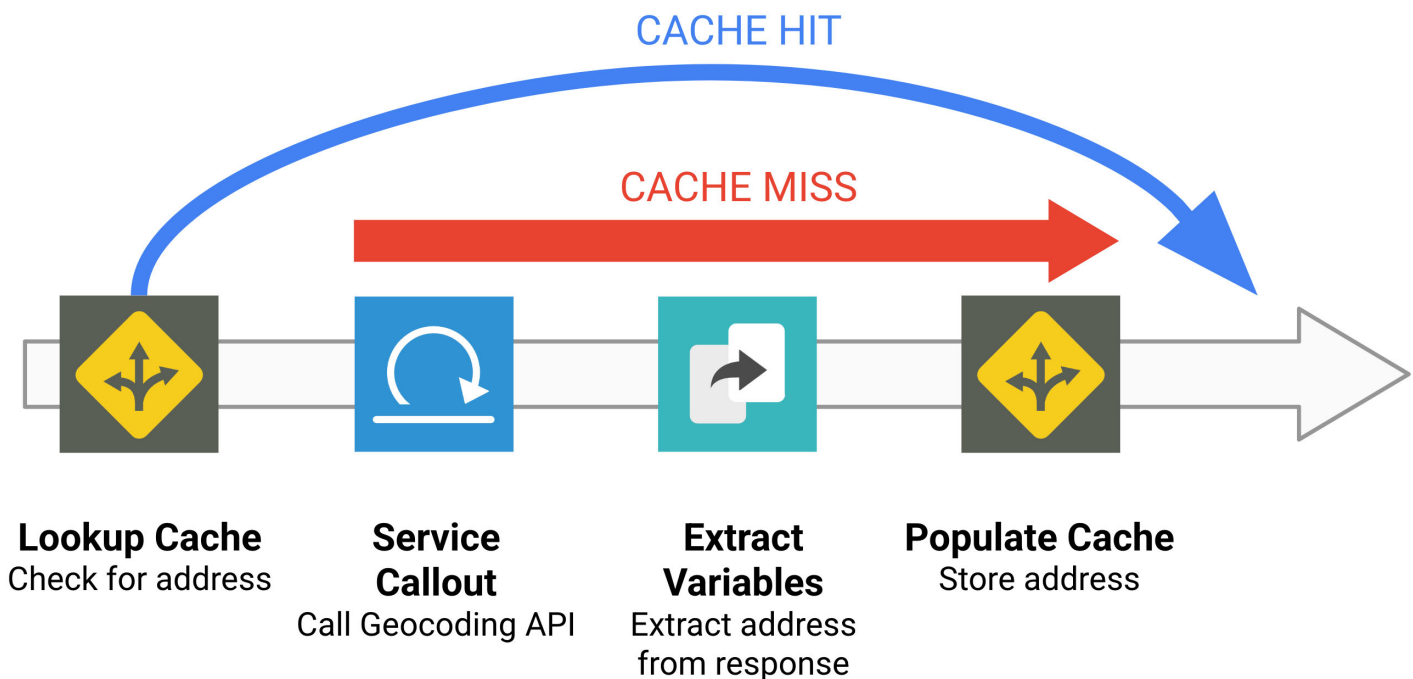
content_co

This command calls the Geocoding API, providing the API key and the desired latitude and longitude. The response contains an array of results, each of which has a formatted address. In your API proxy, you will use the formatted address of the first result to add an address to the API response when retrieving the details for a single ATM.

Task 4. Create a shared flow to call the Geocoding API

In this task, you will create a shared flow for calling the Google Geocoding API. Shared flows allow you to combine policies and resources in a single flow so that it can be consumed from multiple API proxies or other shared flows.

The shared flow will use this pattern:



The number of ATMs in our database is limited, and ATMs latitudes and longitudes do not change. To avoid excessive calls to the Geocoding API, the retrieved address will be cached, using the latitude and longitude for the cache key. If the address is not in the cache for the given latitude and longitude, the Geocoding API will be called, and the returned address will be stored in the cache.

Create the shared flow

1. Return to the Apigee UI.
2. On the left navigation menu, select **Develop > Shared Flows**.
3. Click **Create New**.

4. Set the Name to `get-address-for-location` , and then click **Create**.
5. If a **Switch to Classic** link is in the upper right corner, click that link.
6. Click the **Develop** tab.
7. In the Navigator menu for the shared flow, in the **Shared Flows** section, click **default**.

Add a LookupCache policy

The LookupCache policy will retrieve an address if it has previously been cached.

1. Above the flow arrow, click + **Step**.
2. In the *Traffic Management* section, select the **Lookup Cache** policy type. Specify the following:

Property	Value
Display Name	LC-LookupAddress
Name	LC-LookupAddress

3. Click **Add**.

The policy is added to the flow, and the policy's configuration XML is shown in the pane below the flow.

4. Replace the LookupCache XML configuration with:

```
<LookupCache continueOnError="false" enabled="true" name="LC-  
LookupAddress">  
  <CacheResource>AddressesCache</CacheResource>  
  <Scope>Exclusive</Scope>  
  <CacheKey>  
    <KeyFragment ref="geocoding.latitude"/>  
    <KeyFragment ref="geocoding.longitude"/>  
  </CacheKey>  
  <AssignTo>geocoding.address</AssignTo>  
</LookupCache>
```

content_co

The policy looks in the *AddressesCache* for an entry matching the specified latitude and longitude and, if found, assigns the value to the variable address.

Add a Service Callout policy

The Service Callout policy will call the Google Geocoding API.

- 1. In the Navigator menu for the shared flow, in the **Shared Flows** section, click **default**.
- 2. To add another policy, click + **Step**.
- 3. In the *Extension* section, select the **Service Callout** policy type. Specify the following:

Property	Value
Display Name	SC-GoogleGeocode
Name	SC-GoogleGeocode

- 4. Leave the HTTP Target field empty, and click **Add**.
- 5. Replace the ServiceCallout XML configuration with:

```
<ServiceCallout continueOnError="false" enabled="true" name="SC-GoogleGeocode">
  <Request>
    <Set>
      <QueryParams>
        <QueryParam name="latlng">{geocoding.latitude},
{geocoding.longitude}</QueryParam>
        <QueryParam name="key">{geocoding.apikey}</QueryParam>
      </QueryParams>
    </Set>
    <Verb>GET</Verb>
  </Request>
  <Response>calloutResponse</Response>
  <HTTPTargetConnection>
```

```
<URL>https://maps.googleapis.com/maps/api/geocode/json</URL>
  </HTTPTargetConnection>
</ServiceCallout>
```

This policy calls the *Geocoding API*, using the *geocoding.latitude*, *geocoding.longitude*, and *geocoding.apikey* variables. The API call response is stored in the *calloutResponse* variable.

Add an ExtractVariables policy

The ExtractVariables policy will extract the formatted address from the Google Geocoding API response.

1. In the Navigator menu for the shared flow, in the **Shared Flows** section, click **default**.
2. To add another policy, click + **Step**.
3. In the *Mediation* section, select the **Extract Variables** policy type. Specify the following:

Property	Value
Display Name	EV-ExtractAddress
Name	EV-ExtractAddress

4. Click **Add**.
5. Replace the ExtractVariables XML configuration with:

```
<ExtractVariables continueOnError="false" enabled="true"
name="EV-ExtractAddress">
  <IgnoreUnresolvedVariables>true</IgnoreUnresolvedVariables>
  <JSONPayload>
    <Variable name="address">
      <JSONPath>$.results[0].formatted_address</JSONPath>
    </Variable>
  </JSONPayload>
  <Source
clearPayload="false">calloutResponse.content</Source>
```

content_co

```
<VariablePrefix>geocoding</VariablePrefix>
</ExtractVariables>
```

This policy uses JSONPath to extract the *formatted_address* from the first result in the *calloutResponse* message JSON payload. The address is stored in the *geocoding.address* variable.

Add a PopulateCache policy

The PopulateCache policy will store the address into the cache.

1. In the Navigator menu for the shared flow, in the **Shared Flows** section, click **default**.
2. To add another policy, click + **Step**.
3. In the *Traffic Management* section, select the **Populate Cache** policy type. Specify the following:

Property	Value
Display Name	PC-StoreAddress
Name	PC-StoreAddress

4. Click **Add**.

The policy is added to the flow, and the policy's configuration XML is shown in the pane below the flow.

5. Replace the PopulateCache XML configuration with:

```
<PopulateCache continueOnError="false" enabled="true" name="PC-StoreAddress">
  <CacheResource>AddressesCache</CacheResource>
  <Scope>Exclusive</Scope>
  <Source>geocoding.address</Source>
  <CacheKey>
    <KeyFragment ref="geocoding.latitude"/>
    <KeyFragment ref="geocoding.longitude"/>
  </CacheKey>
  <ExpirySettings>
    <ContentCache>content_cache</ContentCache>
  </ExpirySettings>
</PopulateCache>
```

```
<TimeoutInSec>3600</TimeoutInSec>
</ExpirySettings>
</PopulateCache>
```

The policy stores the value in the *address* variable into the *AddressesCache* using the same latitude and longitude key fragments used by the *LookupCache* policy, in the same order.

The *ExpirySettings/TimeoutInSec* setting specifies that stored data will be cached for 3600 seconds, or 1 hour.

Conditionally skip policies

When the address is found in the cache for a specific latitude and longitude (a cache hit), the ServiceCallout, ExtractVariables, and PopulateCache policies are not necessary and should be skipped.

1. In the Navigator menu for the shared flow, in the **Shared Flows** section, click **default**.

The Code pane contains the *default flow*, which lists the four policies that have been attached:

```
<SharedFlow name="default">
  <Step>
    <Name>LC-LookupAddress</Name>
  </Step>
  <Step>
    <Name>SC-GoogleGeocode</Name>
  </Step>
  <Step>
    <Name>EV-ExtractAddress</Name>
  </Step>
  <Step>
    <Name>PC-StoreAddress</Name>
  </Step>
</SharedFlow>
```

Each *Step* specifies a policy that has been attached. The *Name* specifies the name of the attached policy. A *Condition* element can also be added, specifying a boolean condition that determines whether the policy should execute.

Look back at the shared flow pattern at the beginning of the task. When the address lookup succeeds, there is no need to call the service or store the data back in the cache. In this case, the second through fourth policy steps should be skipped.

The *LookupCache* policy sets a variable that indicates whether the item was found in the cache. If the variable `lookupcache.{policyName}.cachehit` is false, then the item was not found. The policies on the second to fourth steps should execute only when there was not a cache hit.

2. For each of the second through fourth steps, add the following condition inside the Step element:

```
<Condition>lookupcache.LC-LookupAddress.cachehit == content_co  
false</Condition>
```

After you have added them all, the shared flow should resemble this:

```
<SharedFlow name="default">  
  <Step>  
    <Name>LC-LookupAddress</Name>  
  </Step>  
  <Step>  
    <Condition>lookupcache.LC-LookupAddress.cachehit ==  
false</Condition>  
    <Name>SC-GoogleGeocode</Name>  
  </Step>  
  <Step>  
    <Condition>lookupcache.LC-LookupAddress.cachehit ==  
false</Condition>  
    <Name>EV-ExtractAddress</Name>  
  </Step>  
  <Step>  
    <Condition>lookupcache.LC-LookupAddress.cachehit ==  
false</Condition>  
    <Name>PC-StoreAddress</Name>  
  </Step>  
</SharedFlow>
```

3. Click **Save**, and then click **Deploy to eval**.

4. Leave the *Service Account* blank, and then click **Deploy**.

The shared flow uses an API key to call the Geocoding API, so a service account is not required.

A shared flow can only be tested by calling it from an API proxy.

Click **check my progress** to verify the objective.



Create a shared flow to call the Geocoding API

Check my progress

Task 5. Add the ATM's address when retrieving a single ATM

In this task, you add a FlowCallout policy to the API proxy to call the shared flow that was just created. When you retrieve a specific ATM, your API proxy must extract the latitude and longitude from the Cloud Run service response and call the shared flow to retrieve the corresponding address. Then, a JavaScript policy will add the address to the API response.

Add a property set for the API key

A property set can be used to store non-expiring data that is easily accessed from within the API proxy. A property set value will hold the API key.

1. On the left navigation menu, select **Develop > API Proxies**.
2. Click **bank-v1**, and then select the **Develop** tab.
3. In the Navigator menu for the proxy, in the **Resources** section, click +.
4. In the *File Type* dropdown, select *Property Set*.
5. Specify `geocoding.properties` for the *File Name*, and then click **Add**.
6. In the *geocoding.properties* pane, add the following property:

```
apikey=<APIKEY>
```

content_co

7. Replace `<APIKEY>` with the *API_KEY* you created in Task 3.
8. You can retrieve the *API_KEY* using this command in Cloud Shell:

```
echo ${API_KEY}
```

content_co

Your *geocoding.properties* file should look similar to this:

```
apikey=AIzaSyC8-B6nt7M240wwZtsxR205sb0xznuhQWc
```

Create a conditional flow

1. In the Navigator menu for the API proxy, inside the **Proxy Endpoints** section, on the line with **default**, click +.
2. In the *New Conditional Flow* dialog, specify the following values:

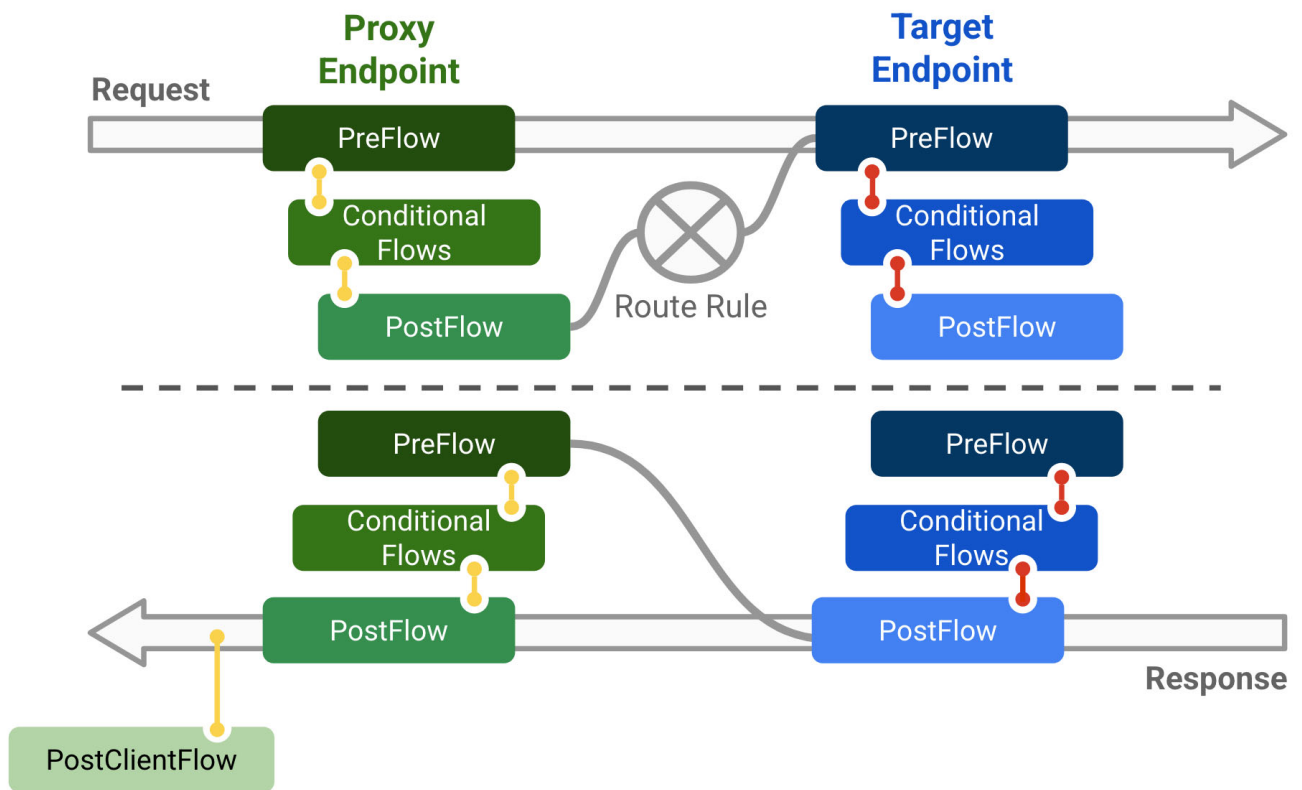
Property	Value
----------	-------

Flow Name	GetATM
Description	retrieve a single ATM
Condition Type	select <i>Path and Verb</i>
Path	/atms/{name}
Verb	select <i>GET</i>

Leave *Optional Target URL* blank.

3. Click **Add**.

An API proxy is made up of many flows. Each flow provides a location for attaching policies as steps. Here is a diagram of an API proxy:



A conditional flow configuration will only be run when the condition is true. For this conditional flow, the *proxy.pathsuffix* variable must match the format `/atms/{name}`, and the *request.verb* variable must be `GET`.

You will attach some policies to the *GetATM* conditional flow so that they only execute for a `GET /atms/{name}` request. The policies must execute after the backend service is called, so they must be attached in a *Proxy Endpoint Response* conditional flow.

Extract the latitude and longitude

1. Click the *GetATM* flow if it is not highlighted, and then click the + **Step** button in the lower left below the *Response* flow.
2. In the *Mediation* section, select the **Extract Variables** policy type. Specify the following:

Property	Value
Display Name	EV-ExtractLatLng
Name	EV-ExtractLatLng

3. Click **Add**.
4. Replace the ExtractVariables XML configuration with:

```
<ExtractVariables name="EV-ExtractLatLng">
  <Source>response</Source>
  <JSONPayload>
    <Variable name="latitude">
      <JSONPath>$.latitude</JSONPath>
    </Variable>
    <Variable name="longitude">
      <JSONPath>$.longitude</JSONPath>
    </Variable>
  </JSONPayload>
  <IgnoreUnresolvedVariables>true</IgnoreUnresolvedVariables>
</ExtractVariables>
```

content_c

This policy extracts the *latitude* and *longitude* from the GET `/atms/{name}` JSON response from the backend service. The *IgnoreUnresolvedVariables* element is set to true, which causes processing to continue even if the latitude and longitude are not found in the response.

Call the shared flow

1. Click the *GetATM* flow if it is not highlighted, and then click the + **Step** button in the lower left below the *Response* flow.
2. In the *Extension* section, select the **Flow Callout** policy type. Specify the following:

Property	Value
Display Name	FC-GetAddress
Name	FC-GetAddress
Shared Flow	<i>select get-address-for-location</i>

3. Click **Add**.
4. In the Navigator menu for the API proxy, click on the **FC-GetAddress** policy.

You should now see the *FC-GetAddress* configuration in the *Code* pane.

5. Replace the *FC-GetAddress* XML configuration with:

```
<FlowCallout continueOnError="false" enabled="true" name="FC-
GetAddress">
  <Parameters>
    <Parameter name="geocoding.latitude">{latitude}
  </Parameter>
    <Parameter name="geocoding.longitude">{longitude}
  </Parameter>
    <Parameter name="geocoding.apikey">
{propertyset.geocoding.apikey}</Parameter>
  </Parameters>
  <SharedFlowBundle>get-address-for-
location</SharedFlowBundle>
</FlowCallout>
```

content_c

This policy sets the latitude, longitude, and apikey variables as shared flow parameters and calls the shared flow. The shared flow sets the *geocoding.address* variable.

Add the address

1. Click the *GetATM* flow, and then click the + **Step** button in the lower left below the *Response* flow.
2. In the *Extension* section, select the **JavaScript** policy type. Specify the following:

Property	Value
Display Name	JS-AddAddress
Name	JS-AddAddress
Script File	select <i>Create new script</i>
Script Name	addAddress.js

3. Click **Add**.
4. In the Navigator menu for the API proxy, inside the **Resources** section, click **addAddress.js**.

The code pane for the addAddress.js code is empty.

5. Add this JavaScript code to add the address to the response:

```
// get the flow variable 'geocoding.address'
var address = context.getVariable('geocoding.address');

// parse the response payload into the responsePayload object
var responsePayload =
JSON.parse(context.getVariable('response.content'));
try {
    // add address to the response
    responsePayload.address = address;

    // convert the response object back into JSON
    context.setVariable('response.content',
JSON.stringify(responsePayload));
} catch(e) {
    // catch any exception
    print('Error occurred when trying to add the address to the
response.');
```

content_co

This code parses the JSON response payload into an object, adds an address field to the object, converts the object back into a JSON string, and then stores it in the response.

A try/catch block is used so that an exception isn't thrown out of the JavaScript policy. A fault is raised if an exception isn't caught, which would cause the API proxy processing to abort.

Conditionally skip policies

1. In the Navigator menu for the proxy, in the **Proxy Endpoints** section, click **GetATM**.

The Code pane contains the *Get ATM* flow, which lists the three policies that have been attached:

```
<Flows>
  <Flow name="GetATM">
    <Description>retrieve a single ATM</Description>
    <Request/>
    <Response>
      <Step>
        <Name>EV-ExtractLatLng</Name>
      </Step>
      <Step>
        <Name>FC-GetAddress</Name>
      </Step>
      <Step>
        <Name>JS-AddAddress</Name>
      </Step>
    </Response>
    <Condition>(proxy.pathsuffix MatchesPath "/atms/{name}") and
(request.verb = "GET")</Condition>
  </Flow>
</Flows>
```

If the `GET /atms/{name}` response does not contain both a latitude and a longitude, as happens when an invalid ATM name is supplied, the *FC-GetAddress* and *JS-AddAddress* policies should be skipped.

2. For the second and third steps, add the following condition inside the Step element:

```
        <Condition>latitude != null AND longitude !=
null</Condition>
```

content_co

After you have added the conditions, the conditional flow should look like this:

```
<Flow name="GetATM">
  <Description>retrieve a single ATM</Description>
  <Request/>
  <Response>
    <Step>
      <Name>EV-ExtractLatLng</Name>
    </Step>
    <Step>
      <Condition>latitude != null AND longitude !=
null</Condition>
      <Name>FC-GetAddress</Name>
    </Step>
    <Step>
      <Condition>latitude != null AND longitude !=
null</Condition>
      <Name>JS-AddAddress</Name>
    </Step>
  </Response>
  <Condition>(proxy.pathsuffix MatchesPath "/atms/{name}") and
(request.verb = "GET")</Condition>
</Flow>
```

3. Click **Save**. If you are notified that the proxy was saved as a new revision, click **OK**.
4. Click **Deploy to eval**, and to confirm that you want the new revision deployed to the eval environment, click **Deploy**.
5. Click the **Overview** tab, and wait for the **eval** deployment status to show that the new revision has been deployed.

Click **check my progress** to verify the objective.



Add the ATM's address when retrieving a single ATM

[Check my progress](#)

Test the updated API proxy

1. In Cloud Shell, open an SSH connection to your test VM:

```
TEST_VM_ZONE=$(gcloud compute instances list --filter="name=
('apigeex-test-vm')" --format "value(zone)")
gcloud compute ssh apigeex-test-vm --zone=${TEST_VM_ZONE} --
force-key-file-overwrite
```

content_co

2. Use the following command to call the *bank-v1* proxy and retrieve all ATMs:

```
curl -i -k "https://eval.example.com/bank/v1/atms"
```

content_co

The response does not contain addresses, because the request does not use the `GET /atms/{name}` flow.

3. Retrieve a single ATM:

```
curl -i -k "https://eval.example.com/bank/v1/atms/spruce-goose"
```

content_co

This response now contains the address that was added in the API proxy:

```
{"longitude":-118.408207,"latitude":33.977601,"description":"","name":"spru
```



Congratulations!

In this lab, you deployed a backend service on Cloud Run