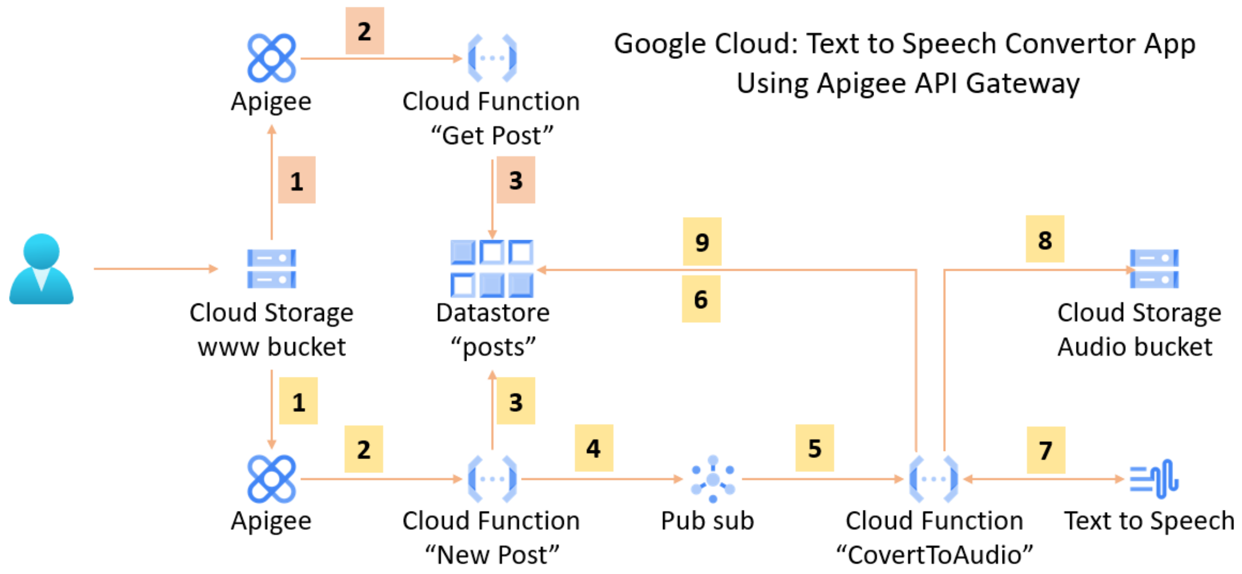
Text to Speech demo using Google Cloud Platform and Apigee

I created this demo application as part of a training that I recently delivered. This application is conceptually similar to another demo available from AWS. I have rewritten code to leverage Google cloud platform and leveraged APIGEE as an API Gateway.

This demo application allows a user to submit a test string for text-to-speech conversion. The user can return later to retrieve the converted audio files. Google platform is still evolving. The platform may change by the time you come across this post and you may need to modify some features during your implementation.

All source code for this demo application can be downloaded from github repository at https://github.com/dgoyal1504/text_to_speech_gcp_apigee



Logic flow

1. User submits a new text to speech conversion request through webform. This request contains the text and the voice to be used during conversion.
    a. [1] Request is directed to Apigee gateway
    b. [2] Gateway directs call to the "New Post" cloud function
    c. "New Post" cloud function
        i. [3] creates a new cord with three fields: text, voice and a status of "PROCESSING" in datastore entity "posts"
        ii. [4] posts the ID of new datastore record as an event in the pub/sub topic new_post.
    d. [5] "ConvertToAudio" cloud function is triggered by the event in the topic "new_post".
        i. [6] Cloud function retrieves the corresponding record from the datastore.

   ii. [7] Cloud function invokes text to speech API supplying the text and the voice as the parameters.
   iii. [8] Returned audio is stored in the “audio” bucket
   iv. [9] Datastore record is updated with the URL of the audio file. Status code is updated to “UPDATED”

2. User submits a query through web form. This request contain one field that can either be a post ID or “*”.
   a. [1] Web action results in invocation of the GET api exposed by APIGEE.
   b. [2] APIGEE directs the call to the cloud function GET POST.
   c. [3] Cloud function retrieves records from datastore and return to the user interface.

Following are the steps involved in this implementation. In case you are new to Google Cloud, please refer to the Google Cloud product documentation for step specific details. As I didn’t find good examples of Apigee cloud function interface, I will attempt to provide more details on that part of the implementation. Please note that this implementation was done as part of a training and is not production ready. Security, single signoff, throttling and lot more needs to be considered to make a production ready system.

Let us create the backend infrastructure and services.

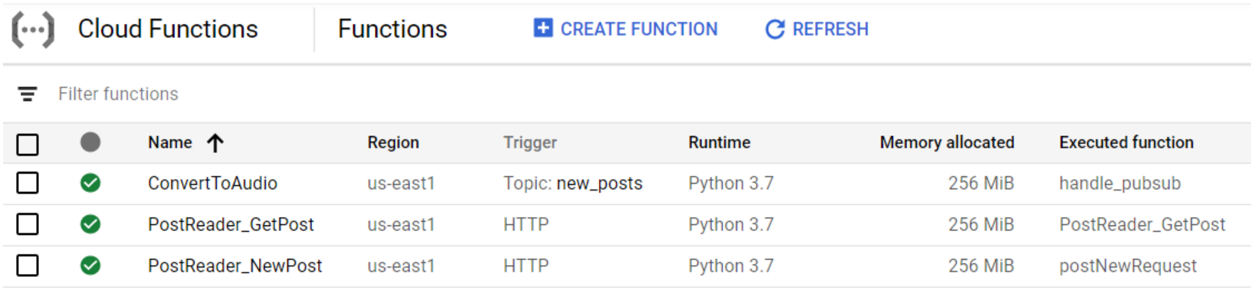
BACK-END

The backend has been implemented using cloud functions. The source for the cloud functions can be downloaded from CloudsFunctions folder of source code repository. You must configure environment variables for the cloud functions. You should also configure datastore, pubsub and cloud storage. Following are the detailed steps.

1. If you don’t already have a Google cloud account and a project, create one.
2. Create a new service account named **voice-demo-lab** and assign following permissions
   a. Cloud Datastore User
   b. Cloud Datastore Viewer
   c. Pub/Sub Publisher
   d. Pub/Sub Subscriber
   e. Pub/Sub Viewer
   f. Storage Admin
   g. Storage Object Admin
   h. Storage Object Creator
   i. Storage Object Viewer
3. Create another service account **apigee-voice-demo** and assign following permissions
   a. Cloud Functions Invoker
4. Create a cloud Datastore Entity (kind = **posts**, key = default (numeric ID auto generated), namespace = default).
5. Create a cloud storage bucket to store all audio files. Give a unique name to this bucket (**audio-xxxx**). Configure public view access to this cloud bucket. (Add “Storage Object Viewer” permission for allUsers.)

6. Create a pub/sub topic id = **new_posts**. The cloud **PostReader_NewPost** function processing new posts will create a new record in the Datastore and also post to the topic. Newly created post will trigger the "**convert to audio**" cloud function.
7. Create & deploy a cloud function **PostReader_NewPost** to process the new post. Name= "PostReader_NewPost", Runtime: Python 3.7, trigger = HTTP, service_account=Voice-demo-lab. Set runtime environment variables for TOPIC=new_posts, ENTITY_TYPE=posts, PROJECT_ID=your-project-id. Test this function with event {"voice" : "Amit", "text" : "This is a test"}. Check newly created records in DataStore entity with status PROCESSING and new posts in pub/sub topic. Note the HTTP trigger URL.
8. Create second cloud function "**Convert to Audio**" to process newly posted conversion requests. (Name="ConvertToAudio", runtime=Python 3.7, trigger=Topic:new_posts, service_account=Voice-demo-lab. Set environment variables AUDIO_BUCKET=audioposts-1506, PROJECT_ID=steam-form-246511, TOPIC=new_posts, ENTITY_TYPE=posts. Deploy). Newly deployed function will process the events already accumulated in "posts" topic and update the status of Datastore entity records to "UPDATED". You can also test this function by posting the ID of any entity from DataStore entity "posts" in a text message to "new_posts" topic.
9. Create third cloud function **PostReader_Get_Post**. This function will be used by the user interface for inquiring the request status. Name = "PostReader_GetPost. Python 3.7, trigger=HTTP, service_account=Voice-demo-lab. Set environment variables AUDIO_BUCKET=audioposts-1506, PROJECT_ID=steam-form-246511, TOPIC=new_posts, ENTITY_TYPE=posts. Deploy. Test using JSON post '{"postID" : "*" }'. Note the HTTP trigger URL for this cloud function.

Summary so far: At this point we have a working backend. We have two HTTP callable cloud functions. First function, **PostReader_NewPost**, accepts two query parameters (text and voice) and returns a JSON object containing ID (**postID**) of the request. Second function, **PostReader_Get_Post,** accepts a postID in the body of the post and returns a JSON object containing records that match the query condition. Third function, **CovertToAudio** calls Speech API to convert the text and update the datastore with the URL of converted audio file. All configuration has been externalized to environment variables. The Cloud Function console of the google project now looks like following.

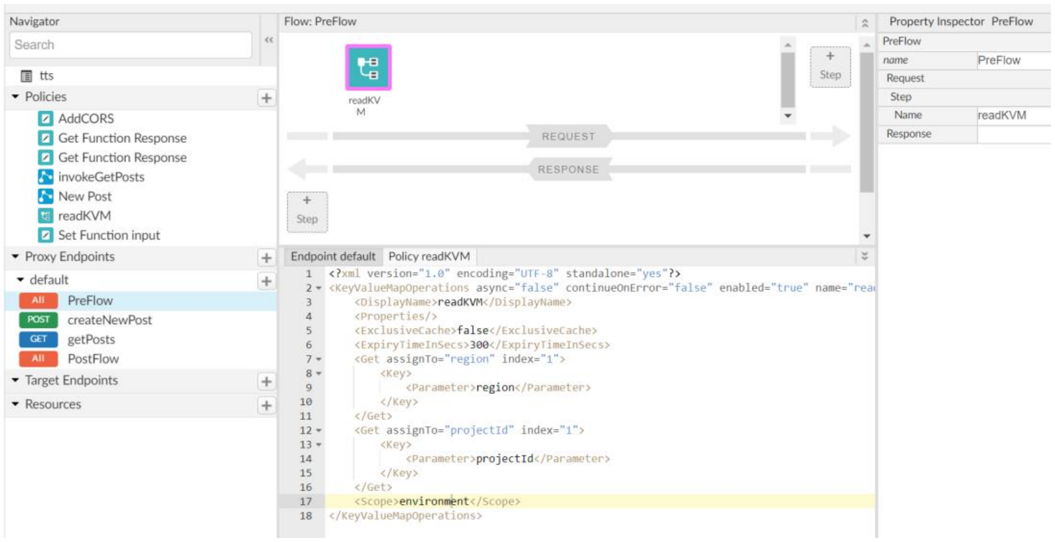


☐	●	Name ↑	Region	Trigger	Runtime	Memory allocated	Executed function
☐	✓	ConvertToAudio	us-east1	Topic: new_posts	Python 3.7	256 MiB	handle_pubsub
☐	✓	PostReader_GetPost	us-east1	HTTP	Python 3.7	256 MiB	PostReader_GetPost
☐	✓	PostReader_NewPost	us-east1	HTTP	Python 3.7	256 MiB	postNewRequest

API GATEWAY

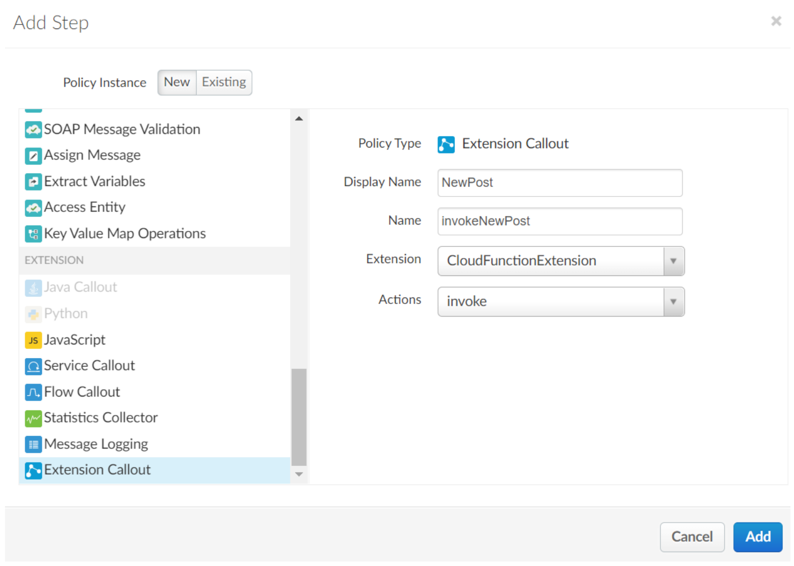
It is never a good idea to expose your back-end infrastructure to the web. Hence, we will implement an API Gateway that will hide the details of the backend from the web server layer. This sample implementation is quite rudimentary. You can enhance it later to include request throttling, authentication and other good features needed for a production implementation. Google cloud provides two options for API Gateway. We will use Apigee as part of this implementation.

10. If you don't already have done so, create an Apigee account at http://login.apigee.com
11. Create a RESTFUL service that will be leveraged by web pages to access the backend functionality. In this demo we are using Apigee to create this RESTFUL web service.
    a. Login to Apigee edge (http://Apigee.com/edge)
    b. Create a key-value map "tts" for test environment (admin->environments->key value maps). Add two keys "projectId" and "region". Assign correct value from your GCP environment.
    c. Create a new API proxy. Name=tts, base path=/tts, target=none, security=Pass through, virtual hosts=secure, deployment environments= prod, test
    d. Edit proxy and select Development tab.
    e. Add a "KeyValueMapOperations" policy in the PreFlow request pipeline.



```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<KeyValueMapOperations async="false" continueOnError="false" enabled="true" name="rea
    <DisplayName>readKVM</DisplayName>
    <Properties/>
    <ExclusiveCache>false</ExclusiveCache>
    <ExpiryTimeInSecs>300</ExpiryTimeInSecs>
    <Get assignTo="region" index="1">
        <Key>
            <Parameter>region</Parameter>
        </Key>
    </Get>
    <Get assignTo="projectId" index="1">
        <Key>
            <Parameter>projectId</Parameter>
        </Key>
    </Get>
    <Scope>environment</Scope>
</KeyValueMapOperations>
```

    f. Let us first implement the **POST** operation which will be used to create a new text-to-speech request. Under Proxy Endpoints->Default, add a new flow after PreFlow. Name=createNewPost, Description="create new post", Condition= (proxy.pathsuffix MatchesPath "/") and (request.verb = "POST")
    g. Add an **Extension Callout** step. This step will invoke the **PostReader_NewPost** cloud function.


Add Step

Policy Instance: New | Existing

SOAP Message Validation
Assign Message
Extract Variables
Access Entity
Key Value Map Operations
EXTENSION
Java Callout
Python
JavaScript
Service Callout
Flow Callout
Statistics Collector
Message Logging
Extension Callout

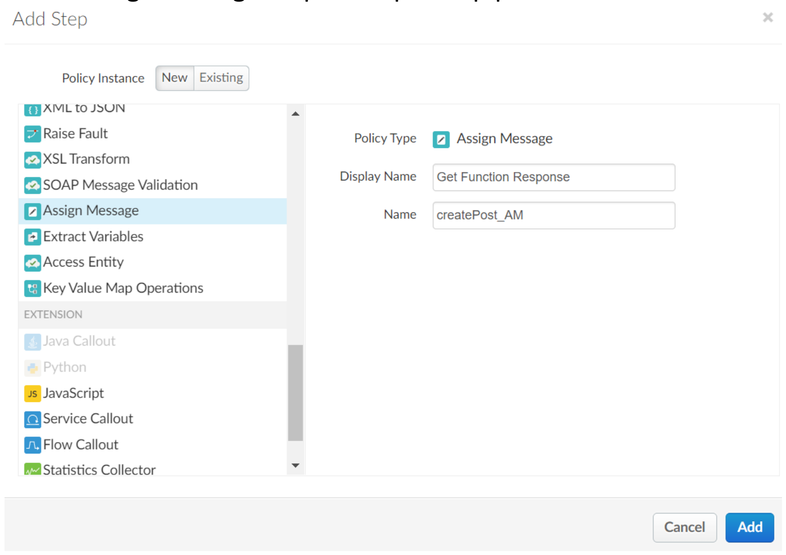
Policy Type: Extension Callout
Display Name: NewPost
Name: invokeNewPost
Extension: CloudFunctionExtension
Actions: invoke

Cancel | Add

h. Edit XML and replace INPUT and OutPUT tags with following

```
<Input><![CDATA[
    {
      "region" : "{region}",
      "projectId" : "{projectId}",
      "functionName" : "PostReader_NewPost",
      "method" : "POST",
      "payload" : {response.content}
    }
 ]]></Input>
<Output>function.response</Output>
```

i. Add a AssignMessage step to response pipeline.


Add Step

Policy Instance: New | Existing

XML to JSON
Raise Fault
XSL Transform
SOAP Message Validation
Assign Message
Extract Variables
Access Entity
Key Value Map Operations
EXTENSION
Java Callout
Python
JavaScript
Service Callout
Flow Callout
Statistics Collector

Policy Type: Assign Message
Display Name: Get Function Response
Name: createPost_AM

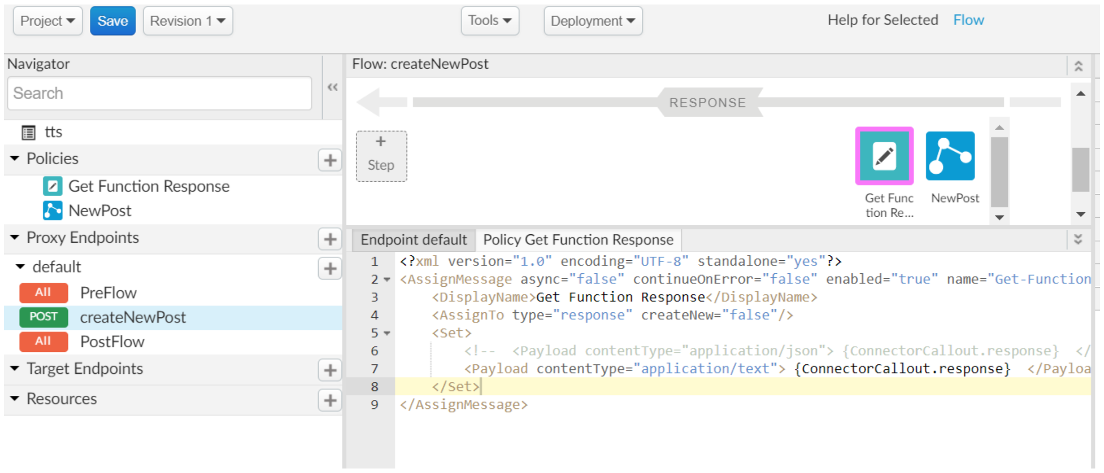
Cancel | Add

j. Replace everything between AssignMessage tags with following

```
<DisplayName>Get Function Response</DisplayName>
<AssignTo type="response" createNew="false"/>
<Set>
```

```
            <!--  <Payload contentType="application/json"> {ConnectorCallout.response}
</Payload> -->

            <Payload contentType="application/text"> {ConnectorCallout.response}
</Payload>
    </Set>
```



```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<AssignMessage async="false" continueOnError="false" enabled="true" name="Get-Function
    <DisplayName>Get Function Response</DisplayName>
    <AssignTo type="response" createNew="false"/>
    <Set>
        <!--  <Payload contentType="application/json"> {ConnectorCallout.response}  </
        <Payload contentType="application/text"> {ConnectorCallout.response}  </Payloa
    </Set>
</AssignMessage>
```

k. Now we will handle the **GET** operation. In the Proxy Endpoints panel, create a new flow "getPosts" under default endpoint. Configure the condition as (proxy.pathsuffix MatchesPath "/") and (request.verb = "GET")

l. In request pipeline, add a AssignMessage policy step. Name it as "set-request-variable". In policy XML view, replace everything between AssignMessageTags with following

```
<DisplayName>Set Function input</DisplayName>
<AssignTo type="request" createNew="false"/>
<Set>
  <Payload contentType="application/json">
     { "postId" : "{request.queryparam.postId}" }
  </Payload>
</Set>
```

m. In response pipeline, add a cloud function extension callout. Name= invokeGetPosts. Replace Input and output tags with following

```
<Input><![CDATA[
   {
     "region" : "{region}",
     "projectId" : "{projectId}",
     "functionName" : "PostReader_GetPost",
     "method" : "POST",
     "payload" : {response.content}
   }
 ]]></Input>
<Output parsed="false">
  function.response
</Output>
```

n. In the response pipeline, add an AssignMessage step. Name=getPostAssignResponse. Replace everything between AssignMessage tags with

```
<DisplayName>Get Function Response</DisplayName>
<AssignTo type="response" createNew="false"/>
<Set>
  <Payload contentType="application/json"> {function.response}  </Payload>
</Set>
```

o. Finally, in the PostFlow response pipeline, add an AssignMessage step. Name=addCORS. Replace everything between AssignMessage tags with

```
<DisplayName>Add CORS</DisplayName>
<FaultRules/>
<Properties/>
<Set>
  <Headers>
    <Header name="Access-Control-Allow-Origin">{request.header.origin}</Header>
    <Header name="Access-Control-Allow-Headers">
        origin, x-requested-with, accept, content-type, authorization
    </Header>
    <Header name="Access-Control-Max-Age">3628800</Header>
    <Header name="Access-Control-Allow-Methods">
      GET, PUT, POST, DELETE
    </Header>
  </Headers>
</Set>
<IgnoreUnresolvedVariables>true</IgnoreUnresolvedVariables>
<AssignTo createNew="false" transport="http" type="response"/>
```

p. Configure Apigee CloudFunction. During this step you configure GCP cloud function access authorization for APIGee CloudFunction connector.

q. Configure Stackdriver logging connector for Apigee.

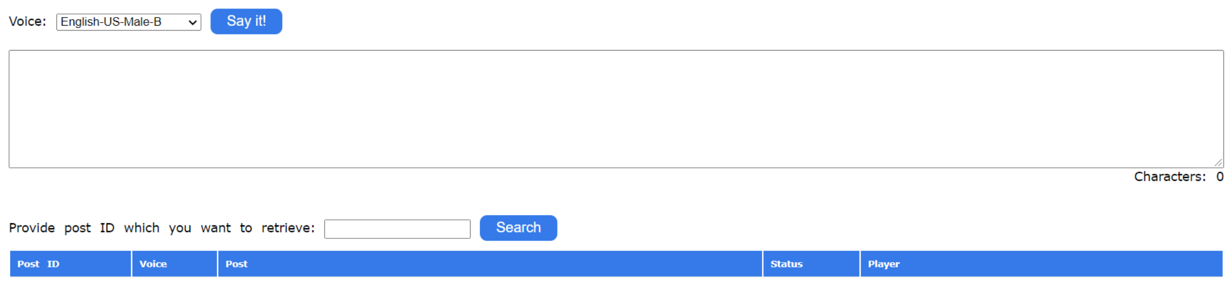
r. Test get and post methods of your Apigee API using postman.

Summary so far: We implemented an API abstraction layer using Apigee. You can test this layer using postman or another client. As earlier, configuration has been externalized to environment variables and must be updated to suit your environment.

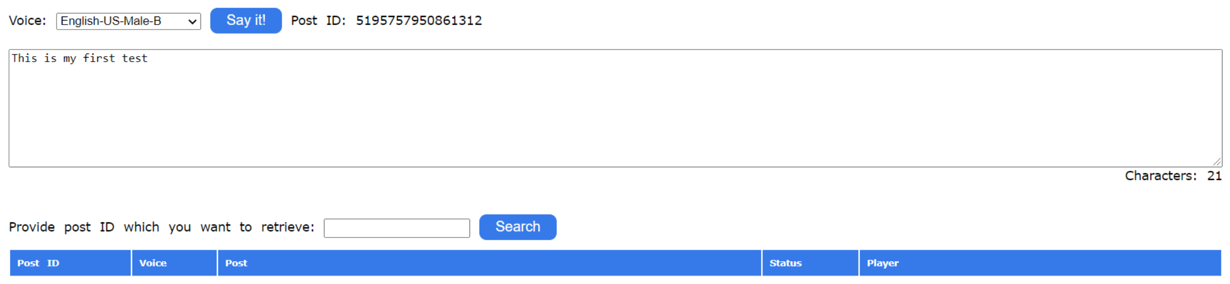
WEB LAYER

12. Let us create a serverless static web user interface.
    a. Download webserver files index.html, styles.css, scripts_gcp.js
    b. Edit scripts.js. Modify API_ENDPOINT to point to the URL of APIGEE API.
    c. Create a cloud storage bucket **www-BUCKET**. Configure public view access to this cloud bucket. (Add “Storage Object Viewer” permission for allUsers.)
    d. Upload web files.
    e. Deselect Block all public access.
    f. Note public URL of index.html.

Now we have all layers in place. Access public URL of index.html from a web browser window. You should see following user interface.



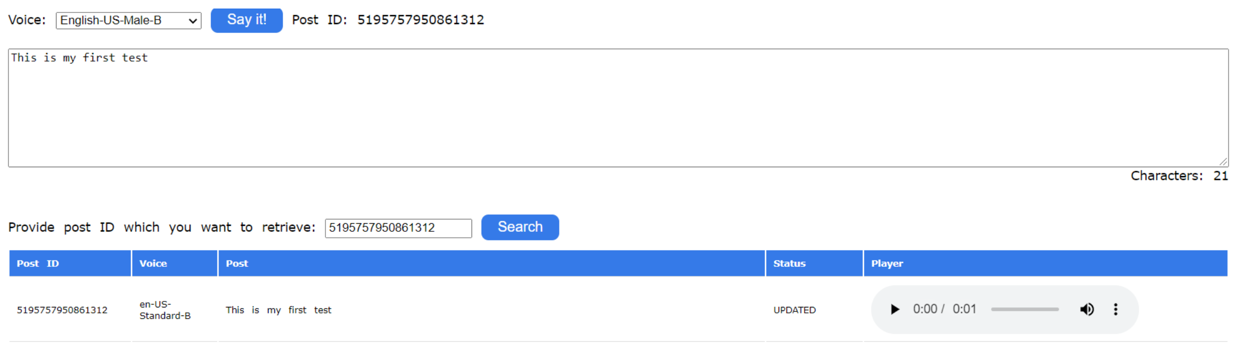
In the first text box, enter a string and click on "Say it" button. If everything works well, you will get a post ID.



Post ID: 5195757950861312

```
This is my first test
```

Now enter the post ID or an asterisk "*" in second text box. Click on search. You should see a list of requests, corresponding status and, if processed, the URL of the audio file.



Post ID: 5195757950861312

```
This is my first test
```

5195757950861312

Post ID	Voice	Post	Status	Player
5195757950861312	en-US-Standard-B	This is my first test	UPDATED	

FINALLY

Please note that this code is not production ready. It was created to deliver a training program. Please ensure that your production code always implements best practices for security, logging, error handling and other requirements. If you are using a trial google account, please remember to remove public access from all resources or delete them.