

V1.0 2/16/2021

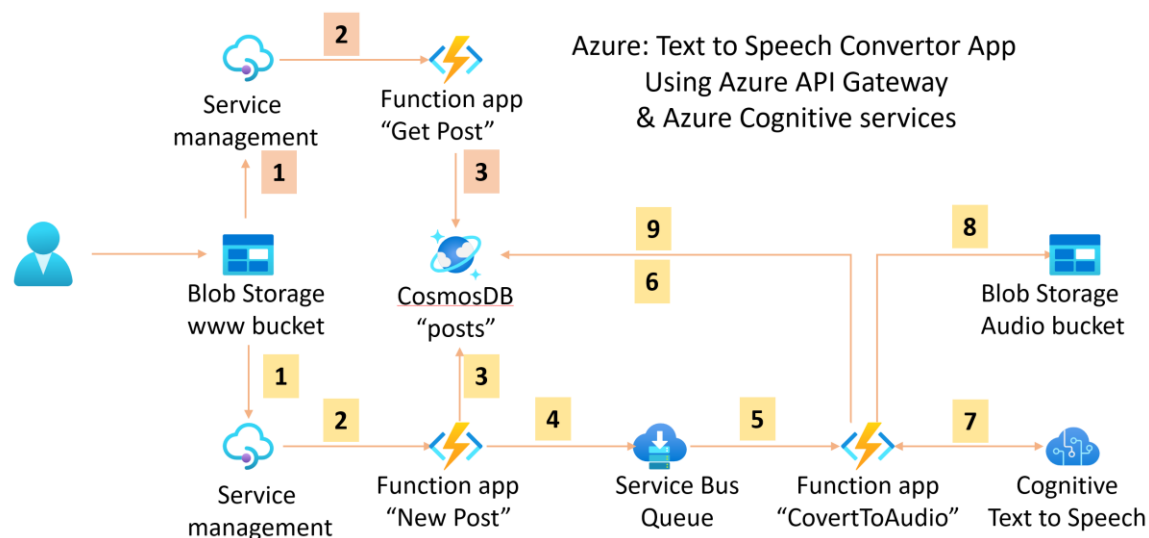
Text to Speech conversion using Azure cloud services

This demo application was created as part of a training that I recently delivered. This application is conceptually similar to another demo available from AWS. I have rewritten the code to leverage Azure cloud platform. I also leveraged Azure API management as an API Gateway.

This demo application allows a user to submit a text string for text-to-speech conversion. Application asynchronously converts the text to audio using Azure Cognitive API. The user can later return to retrieve the converted audio files. Azure 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_azure.git

Following is the logical design of this implementation



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] POST request is directed to the tts API managed by Azure API management.
 - b. [2] API management directs call to the "post_tts_request" function app.
 - c. "post_tts_request" function app
 - i. [3] creates a new record in CosmosDB collection "posts" with three fields: text, voice and status (initial value "PROCESSING")
 - ii. [4] adds a message in the azure service bus queue "posts". Message body contains the ID of new CosmosDB record
 - d. [5] "Process_tts_request" function app is triggered by the message in the queue "posts".

V1.0 2/16/2021

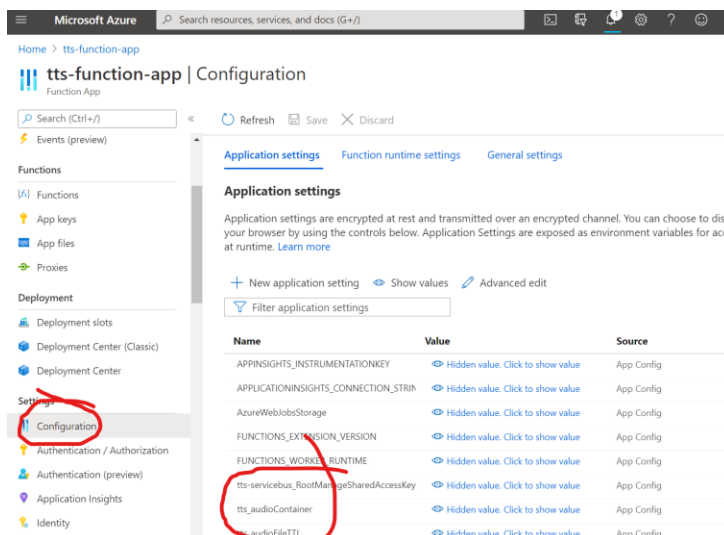
- i. [6] Function app retrieves the corresponding record from the CosmosDB.
 - ii. [7] Function app invokes text to speech API supplying the text and the voice as the parameters.
 - iii. [8] Returned audio is stored in the “audio” bucket
 - iv. [9] CosmosDB record is updated with the URL of the audio file. Status code is updated to “UPDATED”
2. User submits a query through web form. This GET request contains only one field that can either be a post ID or “*”.
 - a. [1] Web action results in the invocation of the GET api exposed by API management.
 - b. [2] API management directs the call to the function app get_tts_request.
 - c. [3] Function app retrieves records from CosmosDB and returns to the user interface.

Following are the steps involved in this implementation. In case you are new to Azure, please refer to the Azure product documentation for step specific details. As I saw challenges with the documentation around few features of Azure API management, 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 function apps. The source for the function apps can be downloaded from function_apps folder of source code repository. Wherever possible, the application configuration has been externalized to the environment variables. You must configure the values specific to your own environment.



You will also need to configure the CosmosDB, Service Bus and Storage Accounts. Following are the detailed steps.

1. If you don't already have an Azure account create one.

V1.0 2/16/2021

2. Create a new resource group “<name your own resource-group>”. Make sure that you create all subsequent resources in this resource group and the location.

[Home](#) > [New](#) > [Resource group](#)

Create a resource group

[Basics](#) [Tags](#) [Review + create](#)

Resource group - A container that holds related resources for an Azure solution. The resource group can include all the resources for the solution, or only those resources that you want to manage as a group. You decide how you want to allocate resources to resource groups based on what makes the most sense for your organization. [Learn more](#)

Project details

Subscription *

Resource group *

Resource details

Region *

3. Create NOSQL Cosmos DB account “<Cosmos-DB-Account>”

[Home](#) > [Azure Cosmos DB](#)

Create Azure Cosmos DB Account

For a limited time, create a new Azure Cosmos DB account with multi-region writes in any region, and receive up to 33% off for the life of your account. Restrictions apply.

Project Details

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription *

Resource Group *

[Create new](#)

Instance Details

Account Name *

API *

Notepad (Preview)

Location *

Capacity mode

Account Type

[Home](#) > [Azure Cosmos DB](#)

Create Azure Cosmos DB Account

For a limited time, create a new Azure Cosmos DB account with multi-region writes in any region, and receive up to 33% off for the life of your account. Restrictions apply.

[Basics](#) [Networking](#) [Backup Policy](#) [Encryption](#) [Tags](#) [Review + create](#)

Network connectivity

You can connect to your Cosmos DB account either publicly, via public IP addresses or service endpoints, or privately, using a private endpoint.

Connectivity method *

☐ All networks

☐ Public endpoint (selected networks)

☒ Private endpoint

Configure Firewall

Allow access from Azure Portal

Allow access from my IP (71.140.145.91)

Private endpoint

Create a private endpoint to allow private connection to this resource. [Learn More](#)

4. After creation of the Cosmos DB account, go to settings and copy the “Primary connection String”. The value of the CosmosDB primary connection string will be used by function app code.
5. Create a collection “posts” in CosmosDB database “ttscollection” with shard key “_id” and storage capacity “unlimited”. Allow access from all networks. Optionally, under collection settings, configure a timeout of 1 hour. This will get rid of our old test data and reduce the Azure bill.
6. Create a service bus namespace <service-bus-namespace> at basic pricing tier.

V1.0 2/16/2021

Home > Service Bus >

Create namespace

Service Bus

Basics Tags Review + create

PROJECT DETAILS

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription *

Resource group *

INSTANCE DETAILS

Enter required settings for this namespace.

Namespace name *

Location *

Pricing tier (View full pricing details) *

Review + create < Previous Next: Tags >

7. Create a queue "posts".

Home > Service Bus >

Service Bus Namespace

Search (Ctrl+F)

Overview Queue Topic Refresh Delete

Essentials

Resource group (change)

Status Active

Location East US

Subscription (change) Pay-As-You-Go

Subscription ID

Tags (change) Click here to add tags

Show data for the last: 1 hour 6 hours 12 hours 1 day 7 days 30 days

Requests

Create queue

Service Bus

Name *

Max queue size

Max delivery count *

Message time to live Days Hours Minutes Seconds

Lock duration Days Hours Minutes Seconds

☐ Enable dead lettering on message expiration

☐ Enable partitioning

Create

8. Create a storage account "<storage account>"

- Using Storage explorer, create two BLOB containers. Configure "anonymous read access to blob".
- Container "www-tts" will contain static web server content.
- Container "www-tts-audio" will be used to store audio file generated by the application.
- Under "Settings->Access Keys" for this storage account, copy the content of "key" and "Connection String" for key1.

9. Configure Cognitive services. We will use these services for text to speech conversion. Create an account under "Home->Cognitive Services -> Marketplace". Once created, click on "Resource management" and copy the value of Key1, Endpoint and Location. These values will be used later.

Home > Cognitive Services > Marketplace > Cognitive Services >

Create Cognitive Services

Prerequisites

Select the subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription *

Resource group *

Instance details

Region *

Location specifies the region only for included regional services. This does not specify a region for included non-regional services. Click here for more details.

Name *

Pricing tier *

View full pricing details

By checking this box, I certify that use of this service is not by or for a police department in the United States. ☐

I confirm I have read and understood the notice below. ☐

Review + create < Previous Next: Tags >

V1.0 2/16/2021

Back-end services

10. Create a “node.js 14 LTS” function app <tts-function-app> on Windows runtime

Home > Function App > Create Function App

Create a function app, which lets you group functions as a logical unit for easier management, deployment and sharing of resources. Functions lets you execute your code in a serverless environment without having to first create a VM or publish a web application.

Project Details

Select a subscription to manage deployed resources and costs. Use resource groups like folders to organize and manage all your resources.

Subscription *

Resource Group * [Create new](#)

Instance Details

Function App name * [.azurewebsites.net](#)

Publish * ☒ Code ☐ Docker Container

Runtime stack *

Version *

Region *

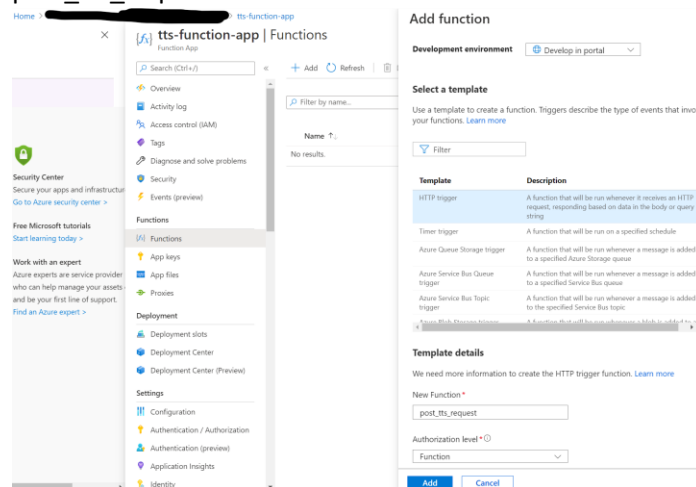
11. Configure following function app environment variables under “Settings->Configuration->Application settings”. We will refer to these environment variables in our function app code.

tts-servicebus-RootManagedSharedAccessKey	<your service bus access key>
Tts_audio_container	<Your audio blob container>
Tts_audioFileTTL	3600 (seconds)
Tts_audioFormat	audio-16khz-64kbps-mono-mp3
Tts_cognitive_subscriptionkey	
Tts_cognitiveUri	https://eastus.tts.speech.microsoft.com/ Choose the endpoint and URL specific to your region https://docs.microsoft.com/en-us/azure/cognitive-services/speech-service/rest-text-to-speech
Tts_collection	posts
Tts_issueTokenUri	https://eastus.api.cognitive.microsoft.com/sts/v1.0/issueToken (please see comment on tts_cognitiveURI)
Tts_language	en-US
Tts_mongo_dbname	<name of your CosmosDB mongodb instance>
Tts_mongo_url	<url of your mongo instance>
Tts_queue	posts
Tts_serviceBusConnectionString	<your connection string>
Tts_storage_account	<your storage account>
Tts_storageConnectionString	<your storage connection string>
Tts_voiceName	(default voice to be used) en-US-Guy24kRUS

12. Download the function app source code from repository.
13. In a browser window navigate to <https://<your function app name>.scm.azurewebsites.net/DebugConsole/?shell=powershell>
 - a. Switch to c:\home\site\wwwroot directory

V1.0 2/16/2021

- b. Edit package.json and copy content of package.json from the source code. This file lists node.js dependencies needed for our project.
 - c. Execute “npm install” under c:\home\site\wwwroot. This will result in installation of all node.js dependencies.
 - d. Create a folder “c:\home\site\wwwroot\shared” and upload the file “mongo.js”. This file contains mongo specific code shared by our function apps.
14. Create a new “Function” to handle new text to speech conversion requests:
- a. Switch to Azure portal console for function app. Add a new HTTP triggered function “post_tts_request”. This function will accept a text to speech conversion request in a JSON message. Click on “Code and Test” link and copy the content of index.js from post_tts_request source folder.



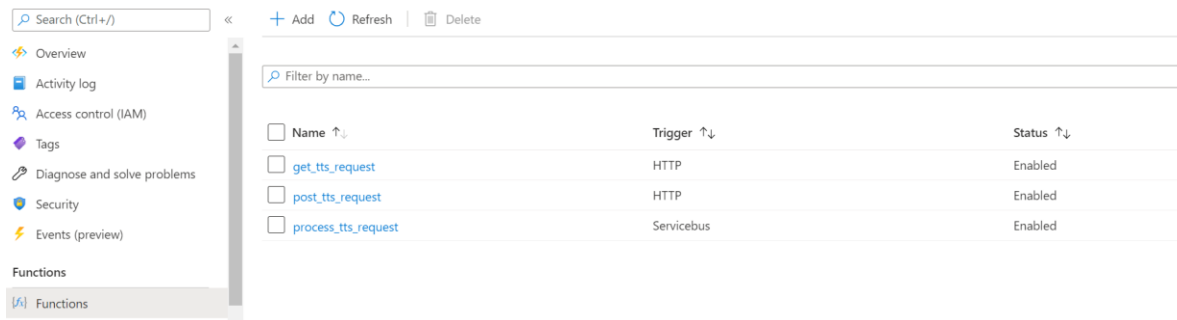
- b. This function retrieves “text” and “voice” inputs from the request, creates a new entry in CosmosDB with status “PROCESSING” and publishes a message in the service bus queue.
 - c. Test this function by providing following JSON input payload ‘{ “text” : “First request”, “voice” : “en-US-Guy24kRUS” }’. A successful execution will return a text string and also result in a new record in the CosmosDB.
15. Create a new “Function” to return status of the submitted requests:
- a. From Azure function app console, add a new HTTP triggered function “get_tts_request”. This function will optionally accept ID of a previous request. Click on “Code and Test” link and copy the content of index.js from get_tts_request source folder.
 - b. This function queries the CosmosDB and returns the results.
 - c. Test this function by providing following inputs ‘{“postId” : “*”}’. A successful execution will return a JSON object containing list of earlier submitted requests.
16. Create a new “Function” to process the events in the service bus queue and invoke text to speech API.
- a. From Azure function app console, add a new service bus message triggered function “process_tts_request”. Click on “Code and test” link and copy the content of index.js from process_tts_request source folder.
 - b. This function retrieves the request Id from the message body, queries corresponding CosmosDB record for the “text” and “voice” variables, invokes text to speech API, stores the converted audio to www-tts-audio folder, and finally updates to cosmodDB record

V1.0 2/16/2021

status to “UPDATED”. The code also implements a TTL on the audio files. This will allow us to keep our azure bill under control.

- c. Test this function by publishing a message to queue. Message body should contain the ID of an existing CosmosDB record. A successful execution will result in update of cosmosDB record to “UPDATED”.

Summary so far: At this point we have a working backend. We have two HTTP callable function apps. First function, HTTP triggered, **post_tts_request**, accepts two query parameters (text and voice) and returns a JSON object containing ID (**postId**) of the request. Second function, HTTP triggered, **get_tts_request**, accepts a postId in the body of the post and returns a JSON object containing records that match the query condition. Third function, queue triggered, **process_tts_request** calls Cognitive Speech API to convert the text and update the cosmosDB with the URL of converted audio file. All configuration has been externalized to environment variables. The Function app console of the Azure project now looks like following.



Name	Trigger	Status
get_tts_request	HTTP	Enabled
post_tts_request	HTTP	Enabled
process_tts_request	Servicebus	Enabled

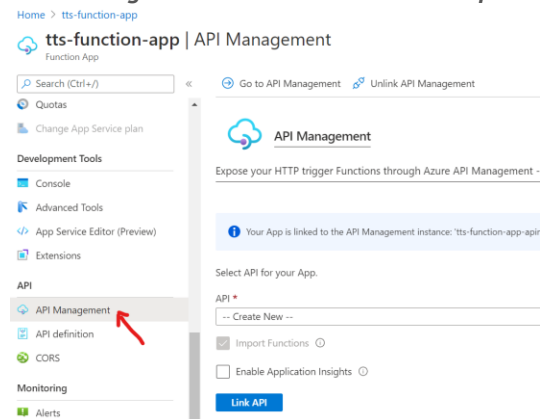
API GATEWAY

It is never a good idea to expose your back-end infrastructure directly 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. We will use Azure API management. You can configure an API Management service through Azure portal UI for API management. In this case we will use a simpler approach to configure it directly from function app.

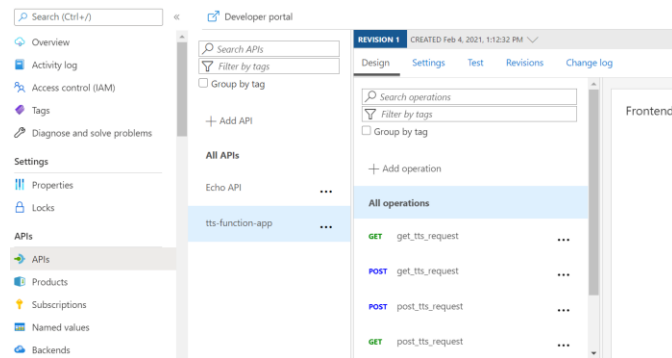
1. On the “function app” UI, scroll down in left navigation panel and click on “API Management” link. Select “create new”, click on “Link API”, select both get_tts_request and post_tts_request functions, copy the base URL of the API and click on create. Base URL will be like <https://<your>>

V1.0 2/16/2021

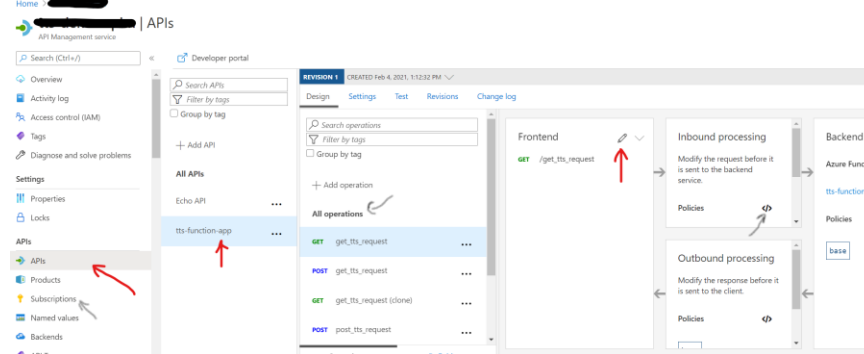
API management instance>.azure-api.net



2. This process creates an API with four operations – GET get_tts_request, POST get_tts_request, GET post_tts_request and POST post_tts_request.



3. Even though we could have used the APIs in this format itself, our design decision was to create a REST like interface where POST results in the creation of a resource and GET results in a query. Our goal is to expose an API with URL of format “<https://<your API management instance>.azure-api.net/tts>” for POST as well as GET.
4. Select GET get_tts_request operation and click on pen icon in front-end window.



5. Edit URL name to “tts” and URL to GET /tts. You can ignore other values. Click on “SAVE”.
6. Repeat this operation for “POST post_tts_request” operation. Edit URL name to “tts” and URL to POST /tts. Click on “SAVE”.
7. Now click on three “...” to the right of “POST get_tts_request” operation and select “DELETE” to delete this operation. Repeat this process for “GET post_tts_request” operation.
8. We are left with two operations – GET tts and POST tts.

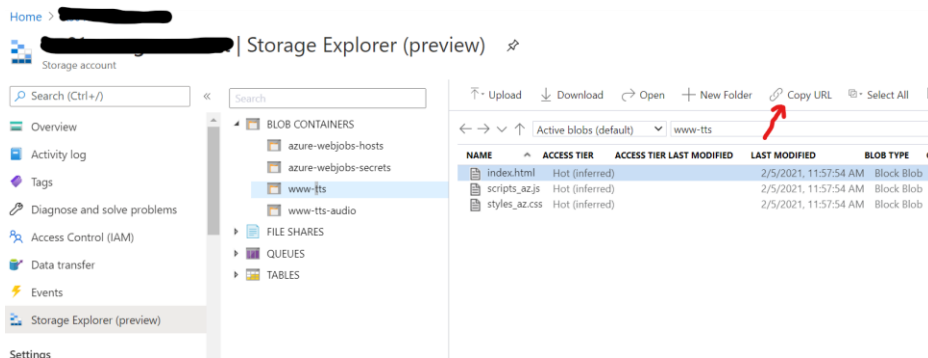
V1.0 2/16/2021

9. Highlight “All operations”, click on “</>” icon in the inbound processing box. It opens a policy editor. Copy the content of policy.xml downloaded from the “API Management” folder of the source code repository. Ensure that the value of “backend-id” matches the name of your function app, and the values of GET and POST rewrite-uri templates match your function app functions. This policy implements CORS as well as a VERB based routing. Save the policy.
10. Azure API management is secured by subscription keys. Click on “subscriptions” in left navigation panel, add a new subscription with a scope of all APIs and save. Copy the primary key. While making any request, you must pass this key in the HTTP header variable “Ocp-Apim-Subscription-Key”.
11. Earlier you copied the base URL of the API. Complete URL of the “tts” operation will be <BASE URL>/tts. Use Postman or some other HTTP test client to test both GET and POST operation. The URL used during this test will be used during next step as the API_ENDPOINT value.

Summary so far: We implemented an API abstraction layer using Azure API management. 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_az.css, scripts_az.js
 - b. Edit scripts.js. Edit top two lines. Enter values of API_ENDPOINT and Ocp_Apim_Subscription_Key specific to your environment.
 - c. Access the storage account that you created earlier, click on “Storage explorer” in the left navigation panel, select “www-tts” BLOB container and upload all web files.
 - d. Once uploaded, highlight index.html and copy its URL.



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

V1.0 2/16/2021

Voice:

Characters: 0

Provide post ID which you want to retrieve:

Post ID	Voice	Post	Status	Player
---------	-------	------	--------	--------

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

Voice: Post ID: 60229951e78bb401fc7c3391

My first test of Azure demo application

Characters: 39

Provide post ID which you want to retrieve:

Post ID	Voice	Post	Status	Player
---------	-------	------	--------	--------

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.

Voice: Post ID: 60229951e78bb401fc7c3391

My first test of Azure demo application

Characters: 39

Provide post ID which you want to retrieve:

Post ID	Voice	Post	Status	Player
60229951e78bb401fc7c3391	en-US-Guy24kRUS	My first test of Azure demo application	UPDATED	<div> <div></div> <div>0:03 / 0:03</div> <div></div> <div></div> </div>

V1.0 2/16/2021

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 Azure account, please remember to remove public access from all resources or delete them.