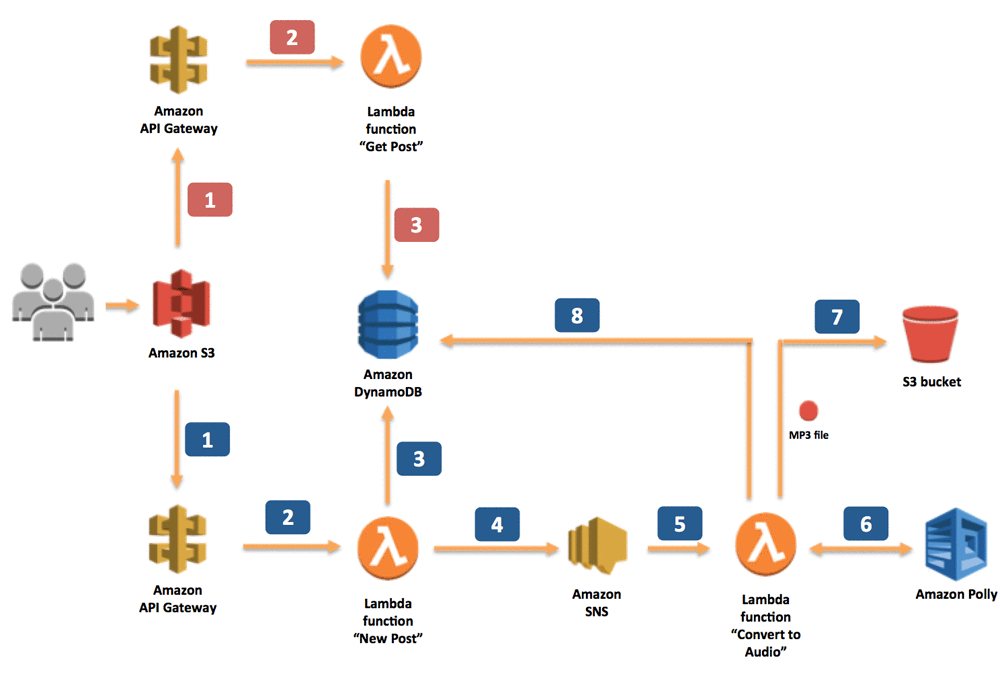
## **The application’s architecture**

The following diagram shows the application architecture. It uses a serverless approach, which means that we don’t need to work with servers – no provisioning, no patching, no scaling. The Cloud automatically takes care of this, allowing us to focus on our application.



The application provides two methods – one for sending information about a new post, which should be converted into an MP3 file, and one for retrieving information about the post (including a link to the MP3 file stored in an S3 bucket). Both methods are exposed as RESTful web services through Amazon API Gateway. Let’s look at how the interaction works in the application.

When the application sends information about new posts:

1. The information is received by the RESTful web service exposed by Amazon API Gateway. In our scenario, this web service is invoked by a static webpage hosted on Amazon Simple Storage Service (Amazon S3).
2. Amazon API Gateway sets off a dedicated Lambda function, “New Post,” which is responsible for initializing the process of generating MP3 files.
3. The Lambda function inserts information about the post into a DynamoDB table, where information about all posts is stored.
4. To run the whole process asynchronously, we use Amazon SNS to decouple the process of receiving information about new posts and starting their conversion.
5. Another Lambda function, “Convert to Speech,” is subscribed to our SNS topic whenever a new message appears (which means that a new post should be converted into an audio file). This is the trigger.
6. The “Convert to Speech” Lambda function uses Amazon Polly to convert the text into an audio file in the specified language (the same as the language of the text).
7. The new MP3 file is saved in a dedicated S3 bucket.
8. Information about the post is updated in the DynamoDB table. Then, the reference (URL) to the S3 bucket is saved with the previously stored data.

When the application retrieves information about posts:

1. The RESTful web service is deployed using Amazon API Gateway. Amazon API Gateway exposes the method for retrieving information about posts. These methods contain the text of the post and the link to the S3 bucket where the MP3 file is stored. In our scenario, this web service is invoked by a static webpage hosted on Amazon S3.
2. Amazon API Gateway invokes the “Get Post” Lambda function, which deploys the logic for retrieving the post data.
3. The “Get Post” Lambda function retrieves information about the post (including the reference to Amazon S3) from the DynamoDB table.

If you will want to replicate the following steps yourself, please be sure to choose a [region where Amazon Polly service is available](https://aws.amazon.com/about-aws/global-infrastructure/regional-product-services/).

## **Creating a DynamoDB table**

We store information about posts, including the text and URL for the MP3 file, on DynamoDB. From the DynamoDB console we create a single table, which we call “posts.” Our primary key (id) is a string, which the “New Post” Lambda function creates when new records (posts) are inserted into a database.

Provide the following information:

* id – The ID of the post
* status – UPDATED or PROCESSING, depending on whether an MP3 file has already been created
* text – The post’s text, for which an audio file is being created
* url – A link to an S3 bucket where an audio file is being stored
* voice – The Amazon Polly voice that was used to create audio file

## **Creating an S3 bucket**

We also need to create an S3 bucket to store all audio files created by the application. To do this, you need to go to the S3 console where you will find an option to create a new bucket. You can choose any name for the bucket as long as it’s globally unique.

Because we will want to make our audio files in S3 public, we will also need to configure our new S3 bucket and allow this kind of operation. In S3 console, check your new created bucket and click on “Edit public access settings”. In the new popup window just click ‘Save’ and confirm the action.

## **Creating an SNS topic**

As you probably noticed in our architecture diagram, we have split the logic of converting a post (text) into an audio file into two Lambda functions. We did this for a couple of reasons. First, it allows our application to use asynchronous calls so that the user who sends a new post to the application receives the ID of the new DynamoDB item; so it knows what to ask for later; and to eliminate waiting for the conversion to finish. With small posts, the process of converting to audio files can take milliseconds, but with bigger posts (100,000 words or more), converting the text can take a bit longer. In other use cases, when we want to do real-time streaming, size isn’t a problem, because Amazon Polly starts to stream speech back as soon as the first bytes are available.

The second reason is that we use a Lambda function, which allows a single execution to run as long as 5 minutes. This should be more than enough time to convert our posts. In the future we might want to convert something bigger. In that case, we might want to use AWS Batch instead of Lambda. Decoupling these two parts of the application makes this change much easier.

When we have two components (in our case two Lambda functions) we need to integrate them. In other words, the second one needs to know when to start. You could do this in many different ways. In our case, we will use Amazon SNS. It sends the message about the new post from the first function to the second one.

So let’s create a simple SNS topic. We can do it from the SNS console, where you will find a button for creating a new topic. Let’s call it new\_posts.

## **Creating an IAM role**

Before we dive into creating Lambda functions, we need to create an IAM role for the functions. The role specifies which AWS services (APIs) the functions can interact with. We will create one role for all three functions.

In the IAM console, find the Policies tab and then press the Create Policy button to open a wizard for creating a new policy. Click on the JSON tab and paste the following script.

{

"Version": "2012-10-17",

"Statement": [

{

"Sid": "Perm1",

"Effect": "Allow",

"Action": [

"polly:SynthesizeSpeech",

"s3:GetBucketLocation",

"logs:CreateLogGroup",

"logs:CreateLogStream",

"logs:PutLogEvents"

],

"Resource": "\*"

},

{

"Sid": "Perm2",

"Effect": "Allow",

"Action": [

"dynamodb:Query",

"dynamodb:Scan",

"dynamodb:PutItem",

"dynamodb:UpdateItem"

],

"Resource": "arn:aws:dynamodb:REGION:ACCOUNT\_ID:table/DYNAMODB\_TABLE\_NAME"

},

{

"Sid": "Perm3",

"Effect": "Allow",

"Action": [

"s3:PutObject",

"s3:PutObjectAcl",

"s3:GetBucketLocation"

],

"Resource": "arn:aws:s3:::BUCKET\_NAME/\*"

},

{

"Sid": "Perm4",

"Effect": "Allow",

"Action": [

"sns:Publish"

],

"Resource": "arn:aws:sns:REGION:ACCOUNT\_ID:SNS\_TOPIC\_NAME"

}

]

}

You will need to replace REGION:ACCOUNT\_ID, DYNAMODB\_TABLE\_NAME, BUCKET\_NAME, and REGION:ACCOUNT\_ID:SNS\_TOPIC\_NAME in the above policy to match the Region in which you are building your app, your account id number, and the resource names that you created earlier.

Click on the Review policy button and on the next tab provide the name of your policy, for example: MyServerlessAppPolicy. Click on Create policy button and that’s all! Your IAM Policy is done!

In the IAM console, find the Roles tab and then press Create New Role button to open a wizard for creating a new role.

In the first step of the wizard you will need to assign a new role to the AWS Lambda service.

After the role is created, on the Permissions tab, write the name of the IAM policy that you created in the previous step: MyServerlessAppPolicy. You should see one policy in the list, select it and then click on the Next: Tags button.

On the third step of the wizard you don’t need to provide any tags. Move to the last step of the wizard where you will need to provide the name of your IAM policy. Let’s name the role LambdaPostsReaderRole.

Click on Create role button and that’s all, your IAM role is created!

## **Creating the “New Post” Lambda function**

The first Lambda function that we create is the entry point for our application. It receives information about new posts that should be converted into audio files.

In the Lambda console, you will see a button for creating a new Lambda function. Let’s call it PostReader\_NewPost. For Runtime, we choose Python 2.7. For now, we don’t configure any triggers.

As shown in the following code, this Lambda function does the following:

1. Retrieves two input parameters:
   * Voice – one of dozens of voices that are supported by Amazon Polly
   * Text – the text of the post that we want to convert into an audio file
2. Creates a new record in the DynamoDB table with information about the new post
3. Publishes information about the new post to SNS (the ID of the DynamoDB item/post ID is published there as a message)
4. Returns the ID of the DynamoDB item to the user

import boto3

import os

import uuid

def lambda\_handler(event, context):

recordId = str(uuid.uuid4())

voice = event["voice"]

text = event["text"]

print('Generating new DynamoDB record, with ID: ' + recordId)

print('Input Text: ' + text)

print('Selected voice: ' + voice)

#Creating new record in DynamoDB table

dynamodb = boto3.resource('dynamodb')

table = dynamodb.Table(os.environ['DB\_TABLE\_NAME'])

table.put\_item(

Item={

'id' : recordId,

'text' : text,

'voice' : voice,

'status' : 'PROCESSING'

}

)

#Sending notification about new post to SNS

client = boto3.client('sns')

client.publish(

TopicArn = os.environ['SNS\_TOPIC'],

Message = recordId

)

return recordId

In addition, the New Post Lambda function needs to know the name of the DynamoDB table and the SNS topic. To provide these values, we use the following environment variables:

* SNS\_TOPIC – the Amazon Resource Name (ARN) of the SNS topic we created
* DB\_TABLE\_NAME – the name of the DynamoDB table (in our case, it’s posts)

Still in this same wizard, we assign the IAM role that we created for the Lambda functions.

The “New Post” Lambda function is ready. To test if, we invoke it with the following input data:

{

"voice": "Joanna",

"text": "This is working!"

}

The “New Post” Lambda function should return an ID, and a new record should appear in the DynamoDB table.

## **Creating the “Convert to Audio” Lambda function**

Now let’s create the Lambda function that converts text that is stored in a DynamoDB table into an audio file, “Convert to Audio.”

In the first step of the wizard, we specify the SNS topic that we created. This time, we configure and enable a trigger. Whenever our SNS topic receives a new message, it executes this function.

Let’s call our new function PostReader\_ConvertToAudio. As before, we use Python 2.7 for Runtime.

This Lambda function does the following:

1. Retrieves the ID of the DynamoDB item (post ID) which should be converted into an audio file from the input message (SNS event)
2. Retrieves the item from DynamoDB
3. Converts the text into an audio stream
4. Places the audio (MP3) file into an S3 bucket
5. Updates the DynamoDB table with a reference to the S3 bucket and the new status

The following code is mostly self explanatory, but let’s dive into the part that invokes Amazon Polly. The synthesize\_speech method receives the text that should be converted and the voice that should be used. In return, it provides the audio stream. The catch is that there is a size limit of 1,500 characters on the text that can be provided as input. Because our posts can be big, we need to divide them into blocks of about 1,000 characters, depending where the final word in the block ends. After converting the blocks into an audio stream, we join them together again.

import boto3

import os

from contextlib import closing

from boto3.dynamodb.conditions import Key, Attr

def lambda\_handler(event, context):

postId = event["Records"][0]["Sns"]["Message"]

print "Text to Speech function. Post ID in DynamoDB: " + postId

#Retrieving information about the post from DynamoDB table

dynamodb = boto3.resource('dynamodb')

table = dynamodb.Table(os.environ['DB\_TABLE\_NAME'])

postItem = table.query(

KeyConditionExpression=Key('id').eq(postId)

)

text = postItem["Items"][0]["text"]

voice = postItem["Items"][0]["voice"]

rest = text

#Because single invocation of the polly synthesize\_speech api can

# transform text with about 1,500 characters, we are dividing the

# post into blocks of approximately 1,000 characters.

textBlocks = []

while (len(rest) > 1100):

begin = 0

end = rest.find(".", 1000)

if (end == -1):

end = rest.find(" ", 1000)

textBlock = rest[begin:end]

rest = rest[end:]

textBlocks.append(textBlock)

textBlocks.append(rest)

#For each block, invoke Polly API, which will transform text into audio

polly = boto3.client('polly')

for textBlock in textBlocks:

response = polly.synthesize\_speech(

OutputFormat='mp3',

Text = textBlock,

VoiceId = voice

)

#Save the audio stream returned by Amazon Polly on Lambda's temp

# directory. If there are multiple text blocks, the audio stream

# will be combined into a single file.

if "AudioStream" in response:

with closing(response["AudioStream"]) as stream:

output = os.path.join("/tmp/", postId)

with open(output, "a") as file:

file.write(stream.read())

s3 = boto3.client('s3')

s3.upload\_file('/tmp/' + postId,

os.environ['BUCKET\_NAME'],

postId + ".mp3")

s3.put\_object\_acl(ACL='public-read',

Bucket=os.environ['BUCKET\_NAME'],

Key= postId + ".mp3")

location = s3.get\_bucket\_location(Bucket=os.environ['BUCKET\_NAME'])

region = location['LocationConstraint']

if region is None:

url\_begining = "https://s3.amazonaws.com/"

else:

url\_begining = "https://s3-" + str(region) + ".amazonaws.com/" \

url = url\_begining \

+ str(os.environ['BUCKET\_NAME']) \

+ "/" \

+ str(postId) \

+ ".mp3"

#Updating the item in DynamoDB

response = table.update\_item(

Key={'id':postId},

UpdateExpression=

"SET #statusAtt = :statusValue, #urlAtt = :urlValue",

ExpressionAttributeValues=

{':statusValue': 'UPDATED', ':urlValue': url},

ExpressionAttributeNames=

{'#statusAtt': 'status', '#urlAtt': 'url'},

)

Return

As with the New Post function, we need to tell this Lambda function which services it can interact with. To provide these values, we use the following environment variables and values:

* DB\_TABLE\_NAME – The name of the DynamoDB table (in our case, it’s posts )
* BUCKET\_NAME – The name of the S3 bucket that we created to store MP3 files

Still in this same wizard, we assign the IAM role that we created for the functions:

Because the posts that we want to convert can be quite big, we extend the maximum length of a single code execution to 5 minutes.

And that’s all. If you retest the “Get Post” function, you should notice that the “Convert to Audio” function also executes (thanks to the SNS integration). This time, a new MP3 file is placed in the S3 bucket.

## **Creating a “Get Post” Lambda function**

The third Lambda function provides a method for retrieving information about posts from our database. From the Lambda console we will create a new function. We’ll call this function PostReader\_GetPost. As before, we will use Python 2.7 as the runtime, but we won’t specify any triggers.

This time the code is very short. This function expects to get the post ID (the DynamoDB item ID) and, on the basis of this ID, it retrieves all information (including the S3 link to the audio file if it exists), and then returns it. To make it a little more user friendly if the input parameter is and asterisk (\*), our Lambda function returns all items from the database. (For a database with a lot of items, avoid this approach because it can degrade performance and might take a long time.)

import boto3

import os

from boto3.dynamodb.conditions import Key, Attr

def lambda\_handler(event, context):

postId = event["postId"]

dynamodb = boto3.resource('dynamodb')

table = dynamodb.Table(os.environ['DB\_TABLE\_NAME'])

if postId=="\*":

items = table.scan()

else:

items = table.query(

KeyConditionExpression=Key('id').eq(postId)

)

return items["Items"]

Again, we provide the name of the DynamoDB table as the environment variable for our function.

Still in this same wizard, we assign the LambdaPostsReaderRole IAM role to the function.

Test the function by executing it with the following input data:

{

"postId": "\*"

}

## **Exposing the Lambda function as a RESTful web service**

The last thing we need to do is expose our application logic as a RESTful web service so it can be invoked easily using a standard HTTP protocol. To do this, we use Amazon API Gateway. From API Gateway console, we choose Create API option. Let’s call the API PostReaderAPI.

After our API is created, we create two HTTP methods (from Actions button and then Create Method). The POST method invokes the PostReader\_NewPost Lambda function. For the GET method, our API invokes the PostReader\_GetPost Lambda function.

The last method is CORS (cross-origin resource sharing). This method enables invoking the API from a website with a different hostname.

Now, configure the GET method for a query parameter, postId, which provides information about the id of the post that should be returned. To do this, click on GET method, modify the configuration of the Method Request, and add information about the new URL query string parameter (the second screen shot below). Just provide the name of the parameter, in our case postId.

The Lambda function (PostReader\_GetPost) expects to receive the input data in JSON format, so we need to configure our API to map our parameter into this format. To do this, we add mapping to the Integration Request configuration.

We use the following mapping:

{

"postId" : "$input.params('postId')"

}

Our API is ready. After deploying it (in the popup window you can call this stage dev) we get a URL, which we can use to interact with our application.

## **Creating a serverless user interface**

Although our application is fully operational, it is exposed as a RESTful web service. Therefore, we need to make sure everything is working as it should. Let’s deploy a small web page on Amazon S3, which is a great choice for hosting static web pages. This web page uses JavaScript to connect to our API and provide all of our text-to-speech functionalities in a WWW page.

The package for the website contains three files:

* html
* css
* js

You can download the package from this link: <https://s3.amazonaws.com/aws-bigdata-blog/artifacts/ai-text-to-speech/text-to-speech-demo.zip>.

We need to modify the script’s.js file by providing the URL for the API that we created using Amazon API Gateway in the first line. After making this small modification, we can deploy our website on Amazon S3.

Let’s create a new S3 bucket (from S3 console) and put all three files in it.

In the properties for the bucket, choose Static Website Hosting, enable website hosting, and provide the name of the index file (index.html).

The last step is to change the bucket’s permissions so that our website is accessible by everybody. On the Permissions tab, edit the bucket policy by adding the following policy. Provide the name of the bucket in 12th line.

{

"Version": "2012-10-17",

"Statement": [

{

"Sid": "PublicReadGetObject",

"Effect": "Allow",

"Principal": "\*",

"Action": [

"s3:GetObject"

],

"Resource": [

"arn:aws:s3:::BUCKET\_NAME/\*"

]

}

]

}

And that’s it! You can now check if the website is working by finding the URL on the Static Website Hosting tab.