Build a Serverless Text-to-Speech Application with Amazon Polly

Overview

In general, speech synthesis is not easy. You cannot assume that when an application reads each letter of a sentence, the output will make sense. A few common challenges for text-to-speech applications include:

- Words that are written the same way, but that are pronounced differently: *I **live** in Las Vegas* compared to *This presentation broadcasts **live** from Las Vegas.*
- Text normalization: Disambiguating abbreviations, acronyms, and units: **St.**,

which can be expanded as **Street** or **Saint**.
- Converting text to phonemes in languages with complex mapping, such as, in English, **tough**, **through**, and **though**. In this example, similar parts of different words can be pronounced differently depending on the word and context.
- Foreign words (**déjà vu**), proper names (**François Hollande**) and slang (**ASAP**, **LOL**).

Amazon Polly provides speech synthesis functionality that overcomes these challenges, allowing you to focus on building applications that use text-to-speech instead of addressing interpretation challenges.

Amazon Polly turns text into life-like speech. It lets you create applications that talk naturally, enabling you to build entirely new categories of speech-enabled products. Amazon Polly is an Amazon AI service that uses advanced deep learning technologies to synthesize speech that sounds like a human voice. It currently includes dozens of lifelike voices in over 20 languages, so you can select the ideal voice and build speech-enabled applications that work in many different countries.

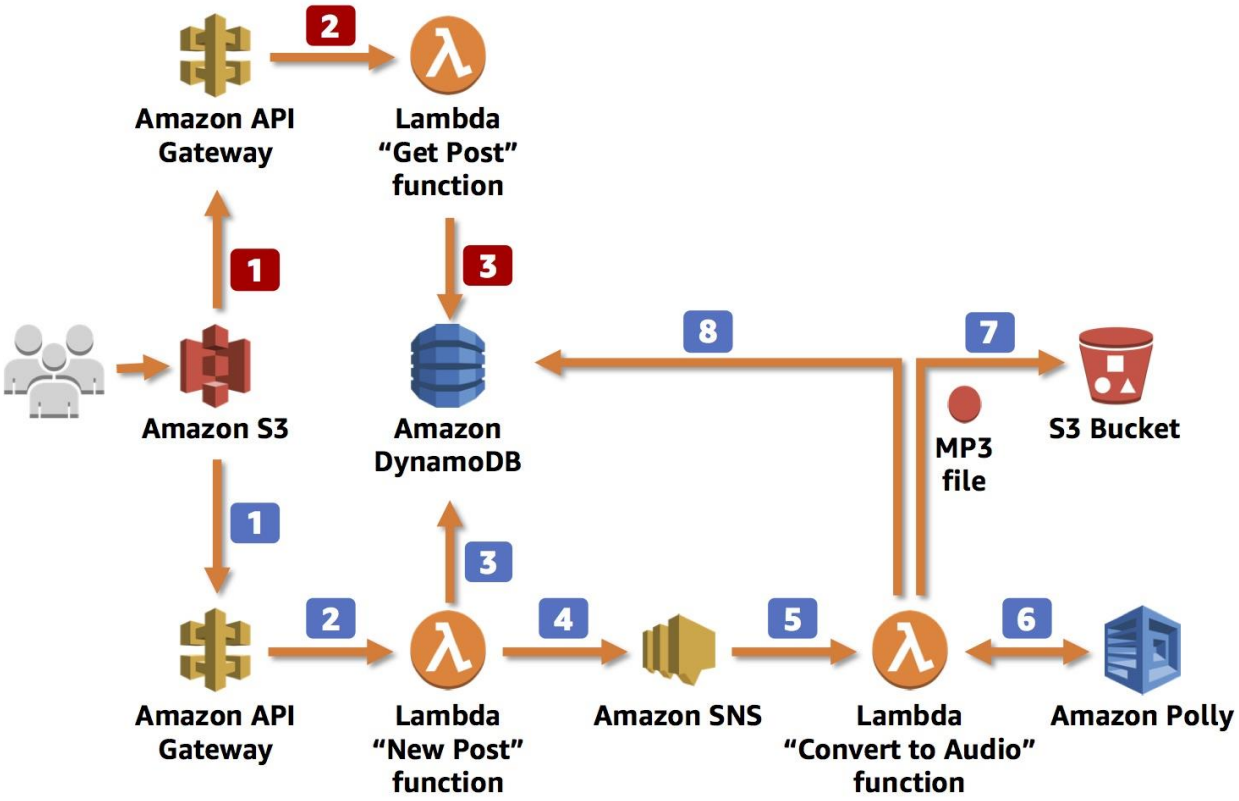
In addition, Amazon Polly delivers the consistently fast response times required to support real-time, interactive dialog. You can cache and save Polly's audio files for offline replay or redistribution. (In other words, what you convert and save is yours. There are no additional text-to-speech charges for using the speech.) Polly is also easy to use. You simply send the text you want to convert into speech to the Amazon Polly API. Amazon Polly immediately returns the audio stream to your application so that your application can play it directly or store it in a standard audio file format such as an MP3.

In this lab you create a basic, serverless application that uses Amazon Polly to convert text to speech. The application has a simple user interface that accepts text in many different languages and then converts it into audio files that you can play from a web browser. This lab uses blog posts, but you can use any type of text. For example, you can use the application to read recipes while you are preparing a meal, or news articles or books while you are driving or riding a bike.

Application Architecture

You build a **serverless application**, which means that you will not need to work with servers — no provisioning, no patching, no scaling. The AWS Cloud automatically takes care of this, allowing you to focus on your 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 Amazon S3 bucket). Both methods are exposed as RESTful web services through **Amazon API Gateway**.



When the application **sends** information about new posts:

1 The information is received by the RESTful web service exposed by **Amazon API Gateway**. This web service is invoked by a static webpage hosted on Amazon Simple Storage Service (**Amazon S3**).

2 Amazon API Gateway triggers an **AWS 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 an **Amazon DynamoDB table**, where information about all posts is stored.

4 To run the whole process asynchronously, you use Amazon Simple Notification Service (**Amazon SNS**) to decouple the process of receiving information about new posts and starting their audio conversion.

5 Another Lambda function, *Convert to Audio*, is subscribed to your SNS topic and is triggered whenever a new message appears (which means that a new post should be converted into an audio file).

6 The *Convert to Audio* 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. The URL to the audio file stored in 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. The 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 and returns the information.

Topics covered

By the end of this lab, you will be able to:

- Create an Amazon DynamoDB to store data
- Create an Amazon API Gateway RESTful API
- Create AWS Lambda functions triggered by API Gateway
- Connect AWS Lambda functions with Amazon Simple Notification Service (SNS)
- Use Amazon Polly to synthesize speech in a variety of languages and voices

Icon key

Various icons are used throughout this lab to call attention to different types of instructions and notes. The following list explains the purpose for each icon:

- A command that you must run
- A sample output that you can use to verify the output of a command or edited file
- A hint, tip, or important guidance
- Information of special interest or importance (not so important to cause problems with the equipment or data if you miss it, but it could result in the need to repeat certain steps)
- **WARNING:** An action that is irreversible and could potentially impact the failure of a command or process (including warnings about configurations that cannot be changed after they are made).

Task 1: Create a DynamoDB Table

The application stores information about blog posts, including the text and URL of the MP3 file, in Amazon DynamoDB. You start by creating a *posts* table. The primary key (id) is a string, which the *New Post* Lambda function creates when new records (posts) are inserted into the database.

3. At the top of the AWS Management Console, to the right of **Services** menu, in the search bar, search for DynamoD and then choose **DynamoDB** from the list.

4. Choose **Create table** .

5. Create a new DynamoDB table with:

- **Table name:** posts
- **Partition key:** id (String)
- Use default settings

6. Choose **Create table** to create your DynamoDB table posts.

There is no need to define the whole structure of the table now. Instead, the application populates it with records like this:

id	status	text	voice	url
5efdb728-a193-	UPDATED	Hallo, mein N...	Marlene	https://s3-eu-...
763a841c-9507	UPDATED	Hello! My nam...	Joanna	https://s3-eu-...
b98fdc51-563b-	UPDATED	Cześć! Jeste...	Maja	https://s3-eu-...
7366bec3-76ff-	UPDATED	Hola! Mi nom...	Enrique	https://s3-eu-...

The application stores:

- **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
- **voice:** The Amazon Polly voice that was used to create audio file
- **url:** A link to an S3 bucket where an audio file is being stored

Task 2: Create an Amazon S3 Bucket

You also need to create an Amazon S3 bucket to store all audio files created by the application. You create a bucket with a unique name, such as *audioposts-123*.

7. At the top of the AWS Management Console, to the right of **Services** menu, in the search bar, search for S3 and then choose **S3** from the list.

8. Choose **Create bucket** and configure the following details:

- **Bucket name:** udioposts-NUMBER
  - Replace **NUMBER** with a random number
  - Copy the name of your bucket to your text editor as you use the bucket name later.
- **AWS Region:** Do not change the region
- Under **Object Ownership**, select **ACLs enabled**
- Under **Block Public Access settings for this bucket** deselect the **Block *all* public access** option, and then leave all other options **deselected**.

Notice all of the individual options remain deselected. When deselecting all public access, you must then select the individual options that apply to your situation and security objectives. In a production environment, it is recommended to use the least permissive settings possible. A warning box appears saying that:
Turning off block all public access might result in this bucket and the objects within becoming public
AWS recommends that you turn on block all public access, unless public access is required for specific and verified use cases such as static website hosting.
- Select the check box next to: I acknowledge that the current settings might result in this bucket and the objects within becoming public.
- Choose **Create bucket** .

Every Amazon S3 bucket must have a unique name.

If you receive an error stating The requested bucket name is not available, select the top Edit link, change the bucket name, and try again until it works.

Task 3: Create an SNS Topic

As you probably noticed in the architecture diagram, the logic of converting a post (text) into an audio file is split into two AWS Lambda functions. This was done for a couple of reasons.

First, it allows the application to use *asynchronous* calls so that the user who sends a new post to the application immediately receives the ID of the new DynamoDB item, so it knows what to ask for later without having to wait 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 longer. In other use cases, such as real-time streaming, size isn't a problem because Amazon Polly starts to stream speech back as soon as the first bytes are available.

Second, the system uses a Lambda function to convert the posts.

Given that the process has been divided into two processes, there needs to be a way to integrate them together. You use **Amazon SNS** to send the message about the new post from the first function to the second function.

9. At the top of the AWS Management Console, to the right of **Services** menu, in the search bar, search for Simple Notification Service and then choose **Simple Notification Service** from the list.

10. Choose **Topics** within the navigation pane on the left.

You may need to expand the navigation pane by choosing the menu icon

11. Choose **Create topic** and configure the following details:
- **Type:** Choose Standard
- **Name:** new_posts
- **Display name:** New posts
12. At the bottom of the page, choose **Create topic** .

13. Copy the **Topic ARN** and paste it into a text editor for later use.

It should look similar to:

```
arn:aws:sns:us-west-2:123456789012:new_posts
content_copy
```

You configure the Lambda functions to use this **Topic ARN** later in the lab.

Task 4: Create a New Post Lambda Function

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

14. At the top of the AWS Management Console, to the right of **Services** menu, in the search bar, search for Lambd and then choose **Lambda** from the list.

You may need to expand the navigation pane by choosing the menu icon

15. Choose **Create function** .

16. Choose **Author from scratch** and use the following settings:

- **Function name:** PostReader_NewPost
- **Runtime:** *Python 3.9*
- **Expand Change default execution role**
- **Execution role:** Create a new role with basic Lambda permissions

17. Scroll down and choose **Create function** .

18. In the **Code source** section, select **lambda_function.py**, open the context(right-click) menu and choose **Open**.

19. Delete the existing code and paste the following code:

```
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
```
content_copy

Examine the code. The Lambda function does the following:

- 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
- Creates a new record in the DynamoDB table with information about the new post
- Publishes information about the new post to SNS (the ID of the DynamoDB item/post ID is published there as a message)
- Returns the ID of the DynamoDB item to the user

20. Choose **Deploy** .

You should see a message that says Changes deployed .

The Lambda function needs to know the name of the DynamoDB table and the SNS topic. To provide these values, you use environment variables. This is an excellent way to pass information to a function without hard-coding values into the function itself.

21. Choose the **Configuration** tab to configure the environment variables.

22. In the left navigation pane, choose **Environment variables**.

23. In the **Environment variables** section, choose **Edit** .

- Choose **Add environment variable** .

- **Key:** Enter `SNS_TOPIC`
- **Value:** Paste the SNS topic you copied earlier in the lab (It looks similar to: *arn:aws:sns:us-west-2:123456789012:new_posts*)

- Choose **Add environment variable** .

- **Key:** Enter `B_TABLE_NAME`
- **Value:** Enter `posts`

24. Choose **Save** .

25. In the left navigation pane of the **Configuration** tab, choose **General configuration**.

26. In the **General configuration** section, choose **Edit** .

- Update the **Timeout** to 10 Seconds

27. Choose **Save** .

The *New Post* Lambda function is ready! You can now test that the function works.

28. Choose the **Test** tab and configure the following details:

- **Event name:** `Joanna`
- Delete the existing code in the code block, and paste this code:

```
{
  "voice": "Joanna",
  "text": "This is working!"
}
```
content_copy

29. Choose **Save** .

30. Choose **Test** to run your test event.

31. You will get an error message (AccessDeniedException)

    1. How can fix that error?

32. Once you fix the permissions error, test once again.

You should see the message: *Execution result: succeeded*

You can expand the **Details** section to view the execution log.

The *New Post* Lambda function returns an ID and you can see the input values in the **Log output**.

Task 5: Create a Convert to Audio Lambda Function

You now create a Lambda function that converts text that is stored in the DynamoDB table into an audio file.

31. Choose **Functions** in the top-left navigation pane.

You may need to expand the navigation pane by choosing the menu icon

32. Choose **Create function**.

33. Choose **Author from scratch** and use the following settings:

- **Function name:** ConvertToAudio
- **Runtime:** *Python 3.9*
- **Expand Change default execution role**
- **Execution role:** Create a new role with basic Lambda permissions

34. Scroll down and choose **Create function** .

35. In the **Code source** section, select **lambda_function.py**, open the context(right-click) menu and choose **Open**.

36. Delete the existing code and paste the following code: :

```
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 3000 characters, we are dividing the
    # post into blocks of approximately 2500 characters.
    textBlocks = []
    while (len(rest) > 2600):
        begin = 0
        end = rest.find(".", 2500)

        if (end == -1):
            end = rest.find(" ", 2500)

        textBlock = rest[begin:end]
        rest = rest[end:]
        textBlocks.append(textBlock)
    textBlocks.append(rest)
```

```
    # For each block, invoke Polly API, which transforms 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
        # is combined into a single file.
        if "AudioStream" in response:
            with closing(response["AudioStream"]) as stream:
                output = os.path.join("/tmp/", postId)
                with open(output, "wb") 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_beginning = "https://s3.amazonaws.com/"
    else:
        url_beginning = "https://s3-" + str(region) + ".amazonaws.com/"

    url = url_beginning \
            + 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
```

Examine the code. The Lambda function does the following:

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

The **synthesize_speech** method receives the text to be converted and the voice to used. In return, it provides the **audio stream**. The catch is that there is a size limit of 3000 characters on the text that can be provided as input. Because a post can be big, posts need to be divided into blocks of about 2500 characters, depending where the final word in the block ends. After converting the blocks into an audio stream, they are joined together again.

37. Choose **Deploy** .

You should see a message that says Changes deployed .

As with the *New Post* function, you need to tell this Lambda function which services it can interact with via Environment variables.

38. Choose the **Configuration** tab to configure the environment variables.

39. In the left navigation pane, choose **Environment variables**.

40. In the **Environment variables** section, choose **Edit** .

- Choose **Add environment variable**
  - **Key:** Enter 3_TABLE_NAME
  - **Value:** Enter posts
- Choose **Add environment variable**

- **Key:** Enter BUCKET_NA
- **Value:** Enter the name of the bucket you created earlier. It should look similar to: *audioposts-123*

41. Choose **Save** .

The posts to be converted can be quite big, so you need to extend the maximum time of a single code execution to 5 minutes.

42. In the **General configuration** section, choose **Edit** .
- Update the **Timeout** to 5 Minutes

43. Choose **Save** .

- In the Permissions section. Click on the Lambda Execuion role
- Attach below IAM policy to the role
    1. **AdministratorAccess**

You now configure the function to trigger automatically when a message is sent to the SNS topic that you created earlier.

44. In the **Triggers** section, choose **Add trigger** and then configure:
- **Select a trigger : *SNS***
- **SNS topic:** Select new_posts from available topics.

45. Choose **Add** .

You are now ready to test that the two Lambda functions communicate successfully via SNS and create a Polly audio file.

Task 6: Test the functions

You now test the following workflow:

- Manually trigger the *New Post* **Lambda function**

- It stores data in **DynamoDB** and send a message to the **SNS topic**
- SNS triggers the *Convert To Audio* function, which uses **Polly** to create an audio file and store it in the **S3 bucket**

46. Choose **Functions** in the top-left corner.

47. Choose **PostReader_NewPost** function .

48. Choose **Test |** .

You should see the message: *Execution result: succeeded*

This indicates that this function was executed. You now confirm that the other steps have also completed successfully.

49. At the top of the AWS Management Console, to the right of **Services** menu, in the search bar, search for DynamoD and then choose **DynamoDB** from the list.

50. In the left navigation pane, choose **Explore items**.

51. Choose **posts**.

You should see two entries because you have run the test twice. The second execution should have also triggered the *Convert to Audio* Lambda function, so there is also an entry for the **url**.

52. At the top of the AWS Management Console, to the right of **Services** menu, in the search bar, search for Lambd and then choose **Lambda** from the list.

53. Choose the **ConvertToAudio** function.

54. Choose the **Monitor** tab.

The monitoring charts should indicate that the function has been invoked.

If the **Error count and success rate** chart indicates that an error occurred, then you need to investigate the error:

- Choose **View logs in CloudWatch**
- Choose the Log Stream shown in the list
- Expand the log entries to discover the error message

For example, if you received the error *The specified bucket does not exist*, then you need to confirm that the bucket name you entered in the Environment variables matches the name of the S3 bucket you created earlier in the lab.

If *Convert to Audio* function executed successfully, there should be an MP3 file in your S3 bucket.

55. At the top of the AWS Management Console, to the right of **Services** menu, in the search bar, search for S3 and then choose **S3** from the list.

56. Choose your *audioposts-* bucket.

You should see an MP3 file. Download it and listen to the contents — you should hear Polly's *Joanna* voice saying "This is working!"

Task 7: Create a Get Post Lambda Function

The final *Get Post* Lambda function provides a method for retrieving information about posts from the database.

57. At the top of the AWS Management Console, to the right of **Services** menu,

in the search bar, search for Lambd and then choose **Lambda** from the list.

58. Choose **Functions** in the left navigation pane by expanding icon.

59. Choose **Create function**.

60. Choose **Author from scratch** and use the following settings:

- **Function name:** PostReader_GetPost
- **Runtime:** *Python 3.9*
- Expand **Change default execution role**
- **Execution role:** Create a new role with basic Lambda permissions

61. Scroll down and choose **Create function**.

62. In the **Code source** section, select **lambda_function.py**, open the context(right-click) menu and choose **Open**.

63. Delete the existing code and paste the following code: :

```
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"]
```

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 an asterisk (*), the 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.)

64. Choose **Deploy** .

You should see a message that says Changes deployed .

Again, you need to provide the name of the DynamoDB table as an Environment variable for the function.

65. Choose the **Configuration** tab to configure the environment variables.

66. In the left navigation pane, choose **Environment variables**.

67. In the **Environment variables** section, choose **Edit** .

- Choose **Add environment variable** .
  - **Key:** Enter B_TABLE_NAME
  - **Value:** Enter posts

68. Choose **Save** .

You can now test the function!

69. In the **Test** tab, create your test event using the following parameters:

- **Event name** AllPosts
- Replace the existing code with:

```
{
  "postId": "*"
}
```

70. Choose **Save** .

71. Choose **Test** to run the test event.

You should see the message: *Execution result: succeeded*

If you expand the **Details** section you should see a list of all records from the DynamoDB table.

Task 8: Expose the Lambda Function as a RESTful Web Service

The last thing you need to do is expose the application logic as a RESTful web service so it can be invoked easily using a standard HTTP protocol. To do this, you use **Amazon API Gateway**.

72. At the top of the AWS Management Console, to the right of **Services** menu, in the search bar, search for `API Gateway` and then choose **API Gateway** from the list.

73. Click on **Create API** → In the **Rest API** panel, choose **Build** .

Note : If a pop-up appears with the title **Create your first API**, choose **OK** .

74. In the **Create new API** section, choose **New API**

75. In the **Settings** section, using the following parameters:

- **API name:** `PostReaderAPI`
- **Description:** `API for PostReader Application`
- **Endpoint Type:** *Regional*

76. Choose **Create API** .

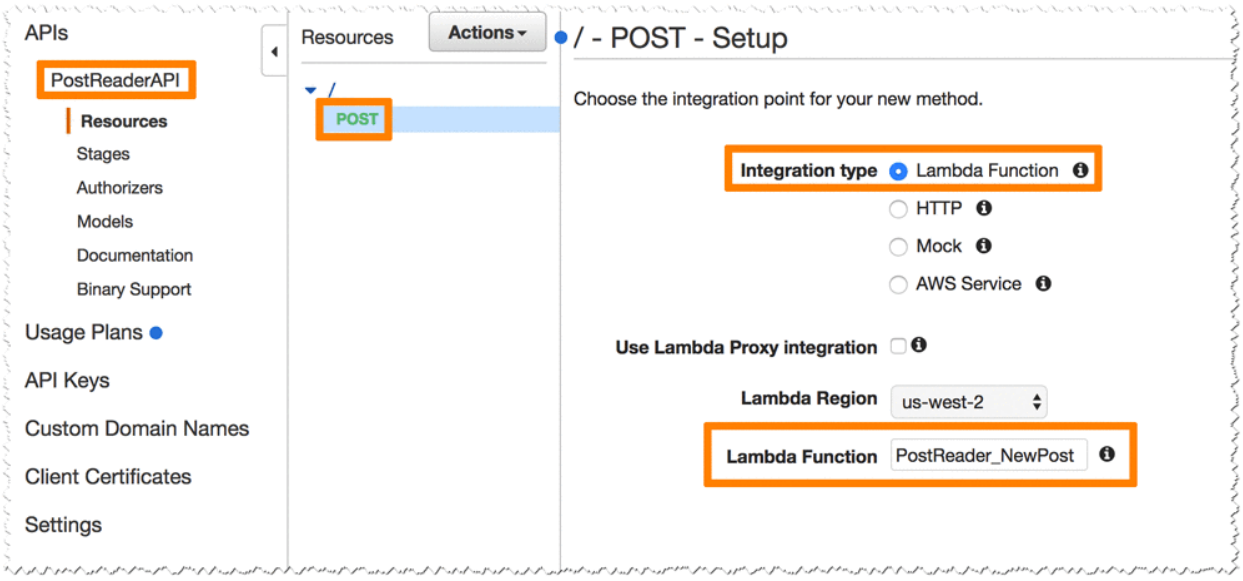
After the API is created, you need to create two **HTTP methods**.

You start by configuring the **POST** method to invoke the *PostReader_NewPost* Lambda function.

77. In the **Resources** pane, choose **Actions** .

78. Select **Create Method** from the dropdown list, then select **POST** and choose .

79. For **Lambda Function**, enter: PostReader_NewPost



80. Choose **Save** .

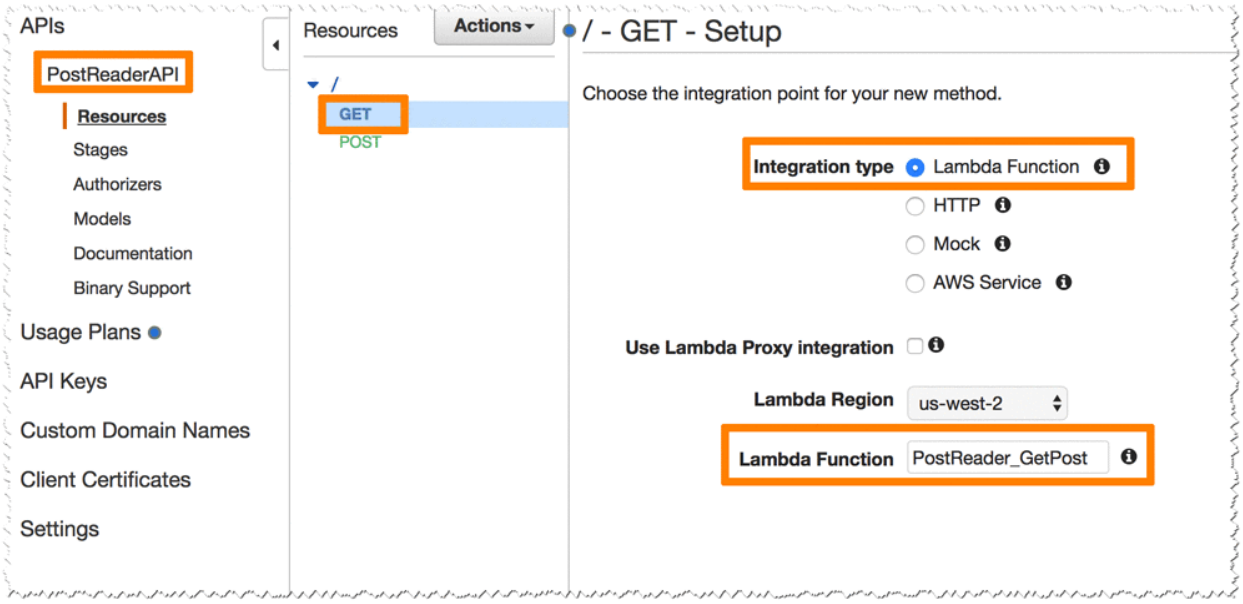
81. A pop-up appears asking to give API Gateway permissions to call the PostReader_NewPost Lambda function. Choose **OK** .

For the **GET** method, the API invokes the *PostReader_GetPost* Lambda function.

82. In the **Resources** pane, choose **Actions** .

83. Select **Create Method** from the dropdown list, then select **GET** and choose .

84. For **Lambda Function**, enter: PostReader_GetPost



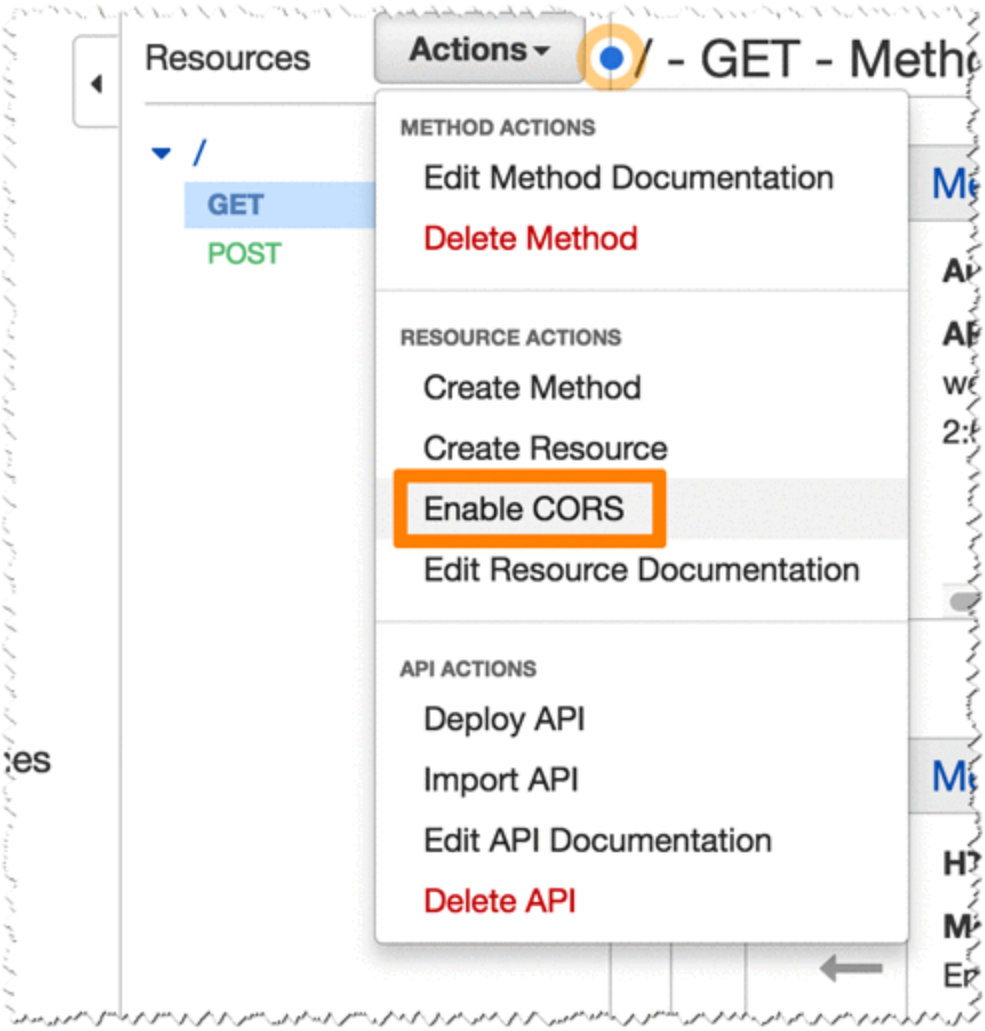
85. Choose **Save** .

86. A pop-up appears asking to give API Gateway permissions to call the PostReader_GetPost Lambda function. Choose **OK**.

The last method to configure is for CORS (Cross-Origin Resource Sharing). This method enables invoking the API from a website with a different hostname.

87. In the **Resources** pane, choose **Actions** .

88. Select **Enable CORS** from the dropdown list.

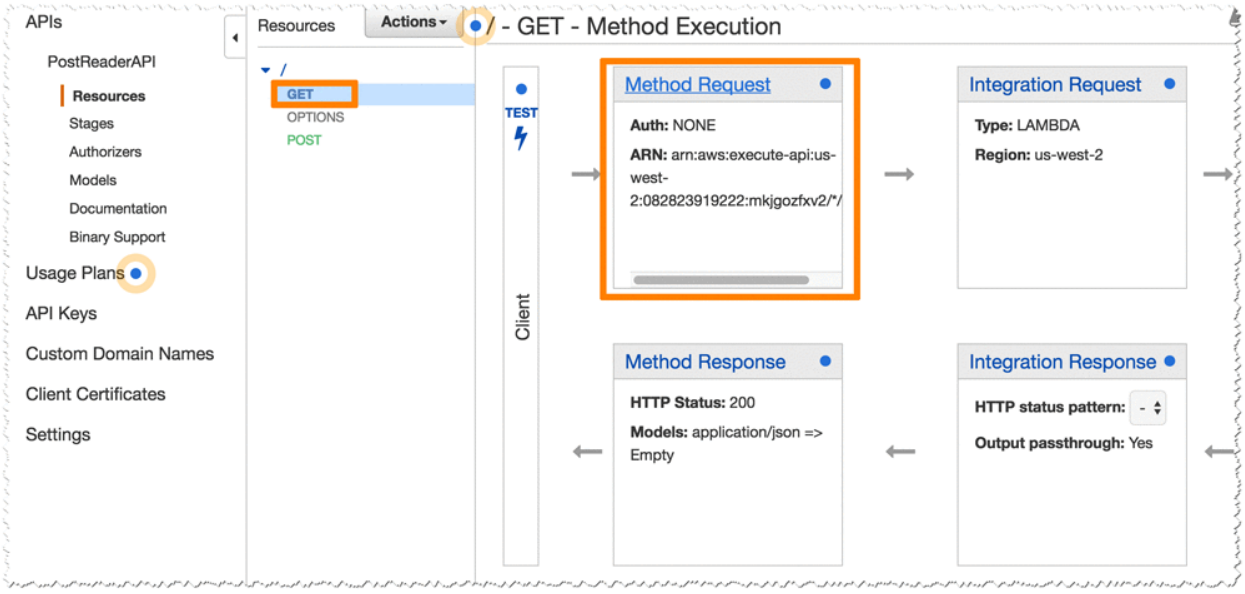


89. Choose **Enable CORS and replace existing CORS headers**

90. Choose **Yes, replace existing values**.

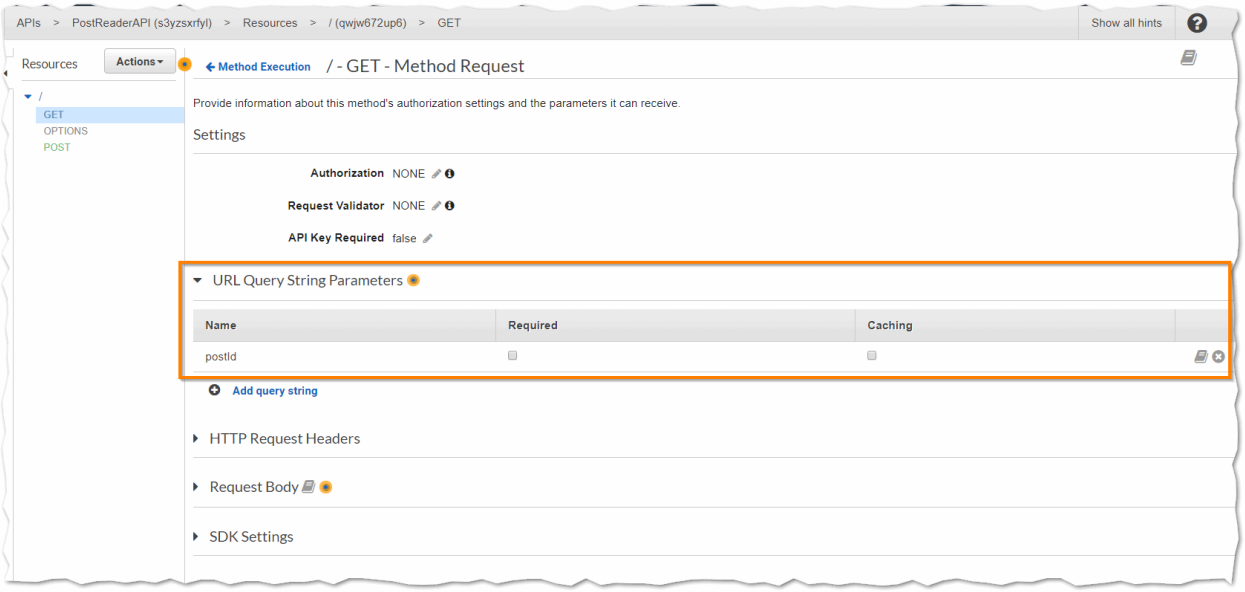
You now configure the GET method for a **query parameter**, *postId*, which provides information about the id of the post that should be returned.

91. Choose the **GET** method.



92. Choose **Method Request**.

93. Expand **URL Query String Parameters**.



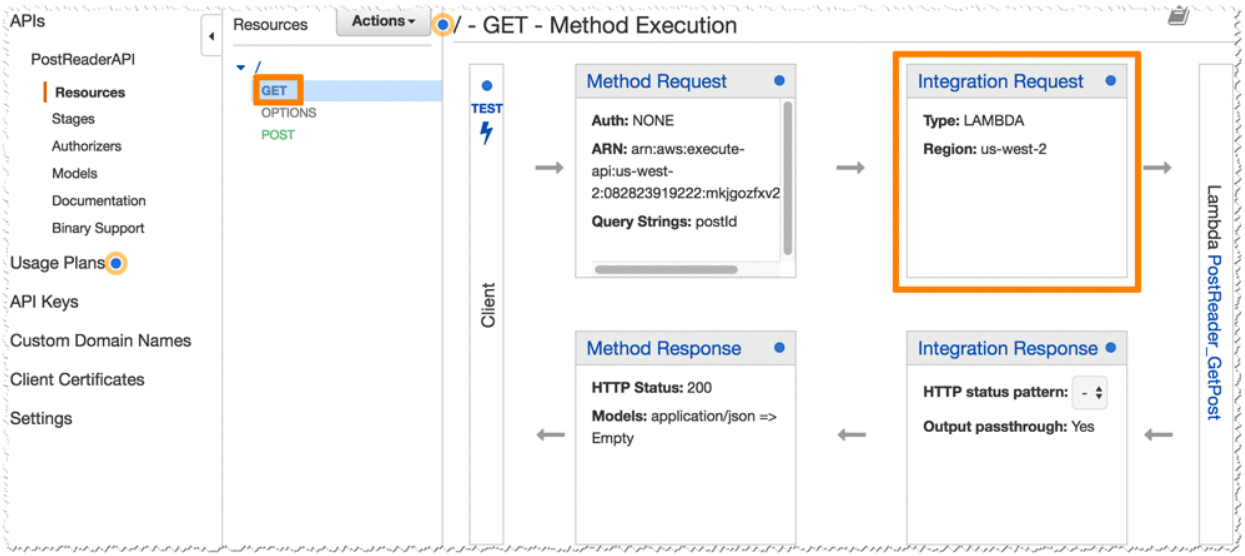
94. Choose **Add query string**.

95. For **Name**, enter postId and choose .

96. Choose **Method Execution** to go back to the GET configuration.

The *PostReader_GetPost* Lambda function expects to receive input data in JSON format, so the API needs to be configured to map the parameter into this format. To do this, you can add mapping to the Integration Request configuration.

97. Choose **Integration Request**.



98. Expand **Mapping Templates**.

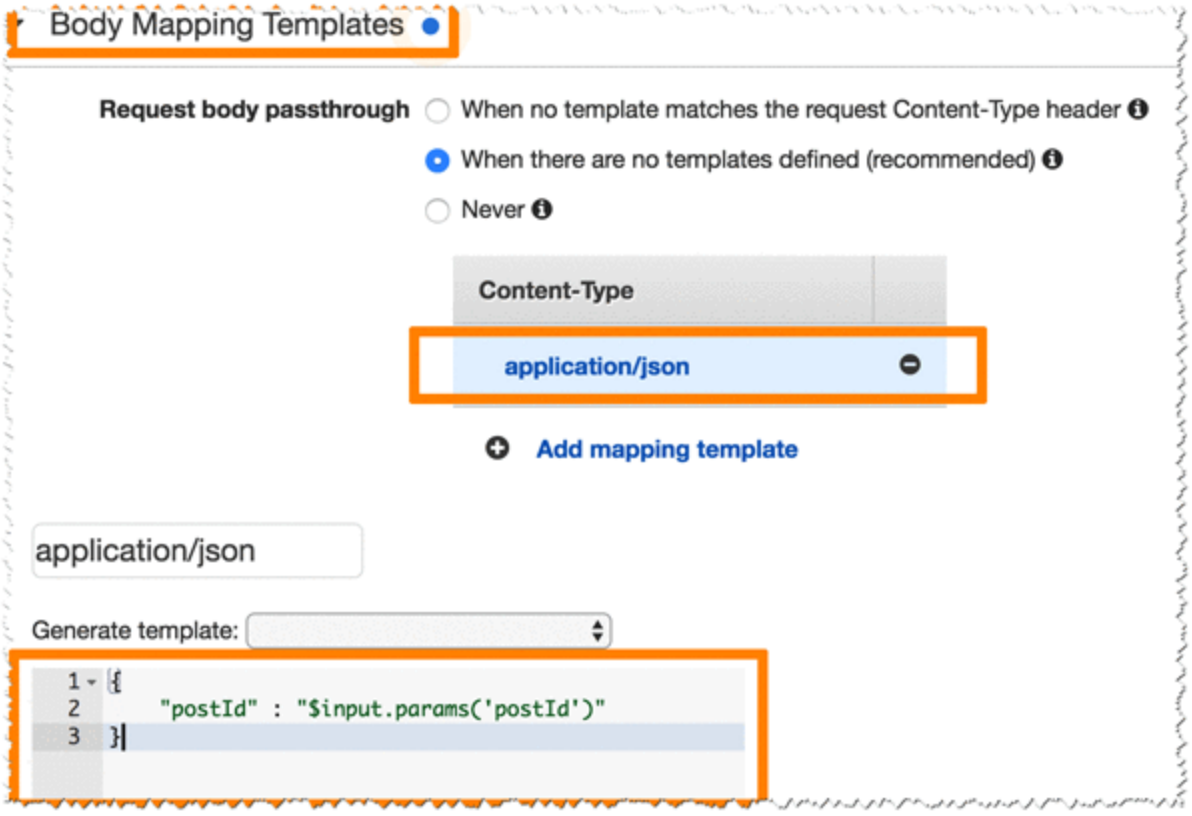
99. Select **When there are no templates defined**.

100. Choose **Add mapping template**.

101. Enter application/json and click .

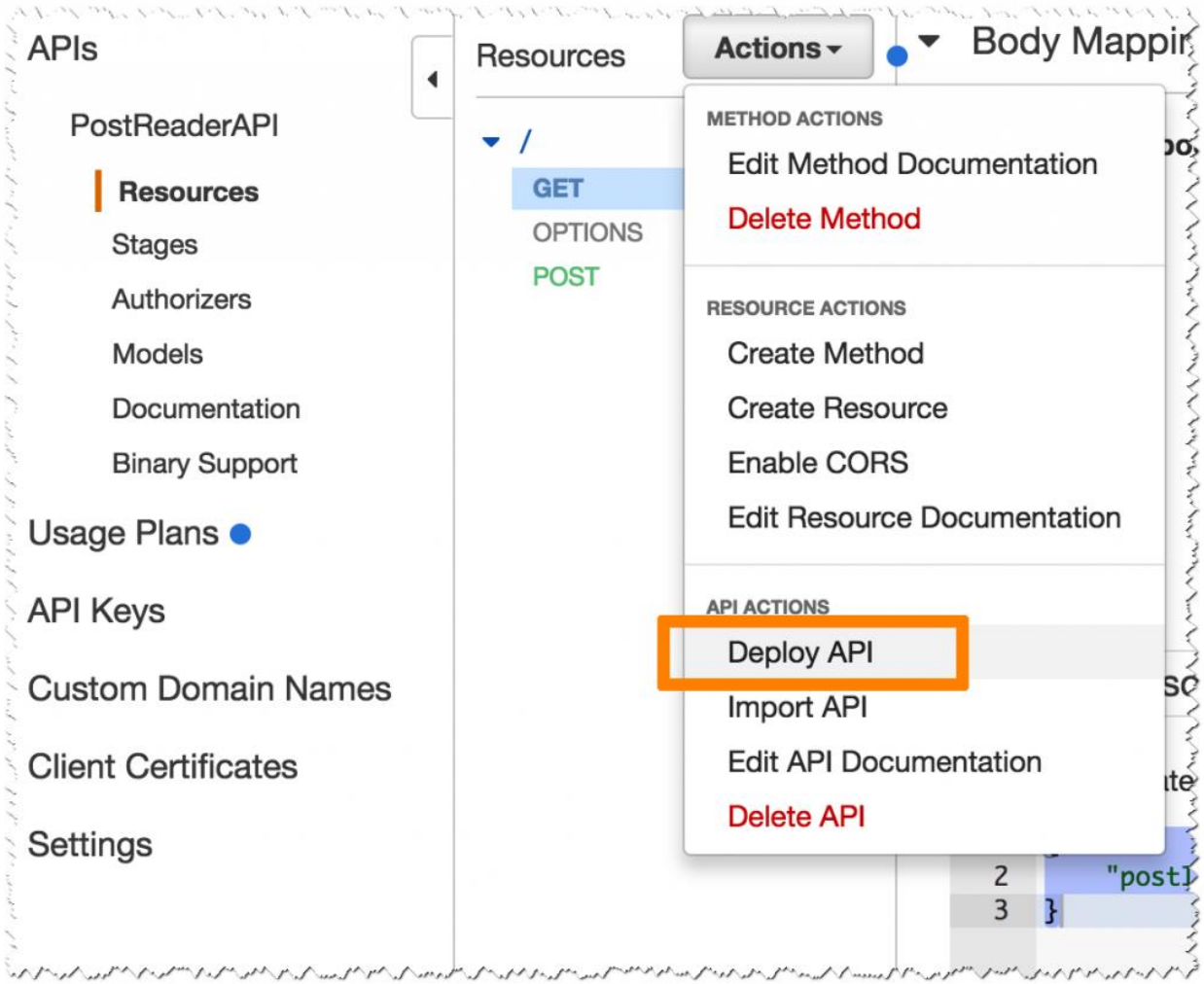
102. Under **Generate template**, enter:

```
{
    "postId" : "$input.params('postId')"
}
```



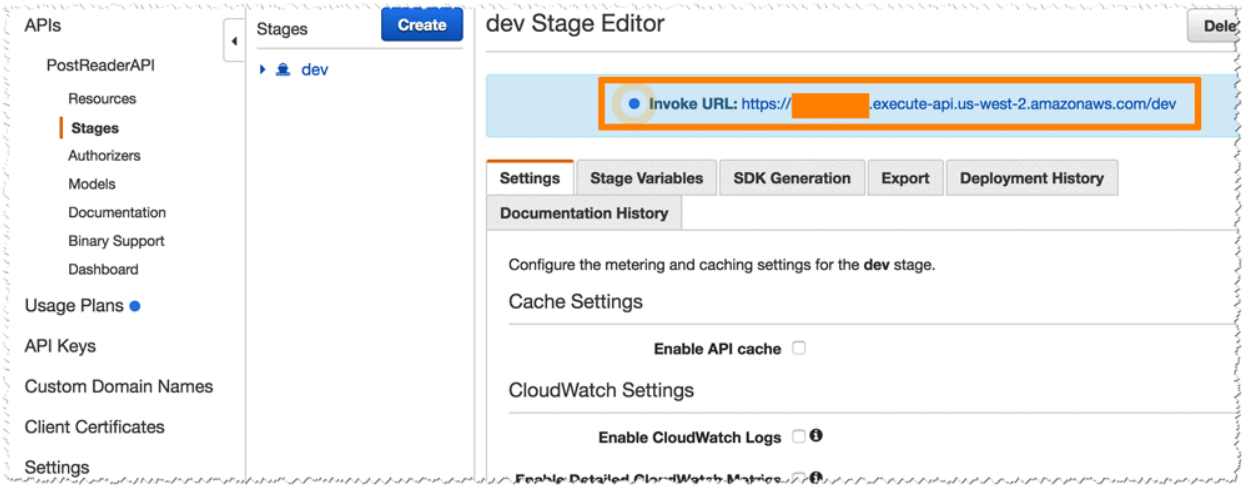
103. Choose **Save** .

The API is ready to deploy!



104. In the **Actions** menu, choose **Deploy API**, then configure:
- **Deployment stage:** *[New Stage]*
- **Stage name:** Dev
- Choose **Deploy**

105. Copy the **Invoke URL** and save it in a text editor for later use.



The URL is used later to interact with the application.

Task 9: Create a Serverless User Interface

Although the application is fully operational, it is only exposed as a RESTful web service. You now 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 the API and provide text-to-speech functionalities in a web page.

106. Right-click each of these links and download the files to your computer:

Ensure that each file keeps the same filename, including the extension!

- index.html
- scripts.js
- styles.css

107. Edit your **scripts.js** file with a Text Editor, replacing *YOUR_API_GATEWAY_ENDPOINT* (on the first line) with the **Invoke URL** you copied earlier.

The result should look similar to:

```
var API_ENDPOINT = "https://pf7fx5.execute-api.us-west-2.amazonaws.com/Dev"
```
content_copy

You now upload these files to an Amazon S3 bucket.

108. At the top of the AWS Management Console, to the right of **Services** menu, in the search bar, search for S3 and then choose **S3** from the list.

109. Choose **Create bucket** and configure the following details:

- **Bucket name:** ww-BUCKET
  - Replace **BUCKET** with the name of your audioposts bucket
  - Copy the name of your bucket to your text editor. You use the bucket name later.
- **Region:** Do not change the region
- You change the bucket's permissions so that the website is accessible to everybody.
- Under **Block Public Access settings for this bucket** deselect the **Block *all* public access** option, and then leave all other options **deselected**.

Notice all of the individual options remain deselected. When deselecting all public access, you must then select the individual options that apply to your situation and security objectives. In a production environment, it is recommended to use the least permissive settings possible.
A warning box appears saying that:
Turning off block all public access might result in this bucket and the objects within becoming public
AWS recommends that you turn on block all public access, unless public access is required for specific and verified use cases such as static website hosting.
- Select the check box next to: I acknowledge that the current settings might result in this bucket and the objects within becoming public.
- Choose **Create bucket**

110. After the bucket has been created, select it from the bucket list and upload the three files to your new *www* bucket.

The files must be named: **index.html**, **scripts.js** and **styles.css**

111. On the bucket page, select the **Permissions** tab at the top.

112. Scroll down to the **Bucket Policy** section and choose the **Edit** button.

113. Paste this policy into the editor:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "PublicReadGetObject",
            "Effect": "Allow",
            "Principal": "*",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": [
                "arn:aws:s3:::www-BUCKET/*"
            ]
        }
    ]
}
```
content_copy

114. Replace **www-BUCKET** with the name of your www-audioposts bucket.

115. Choose **Save changes** .

If you receive an error that *Policy has invalid resource*, confirm that you have edited the *Resource* line to match the name of your bucket.

You can ignore the warning that *This bucket has public access*. This is intentional.

Finally, you activate **static website hosting**, which makes the bucket operate like a static website.

116. Choose the **Properties** tab.

117. Ignore the **AWS CloudTrail** Permission error.

118. Scroll down to the **Static website hosting** section and choose **Edit** .

119. Choose **Enable** for **Static website hosting**.

- **Index document:** index.html
- **Error document:** index.html

For now we are using the index.html file as error document.

- Choose **Save changes**

120. Copy the **Endpoint** URL to your clipboard.

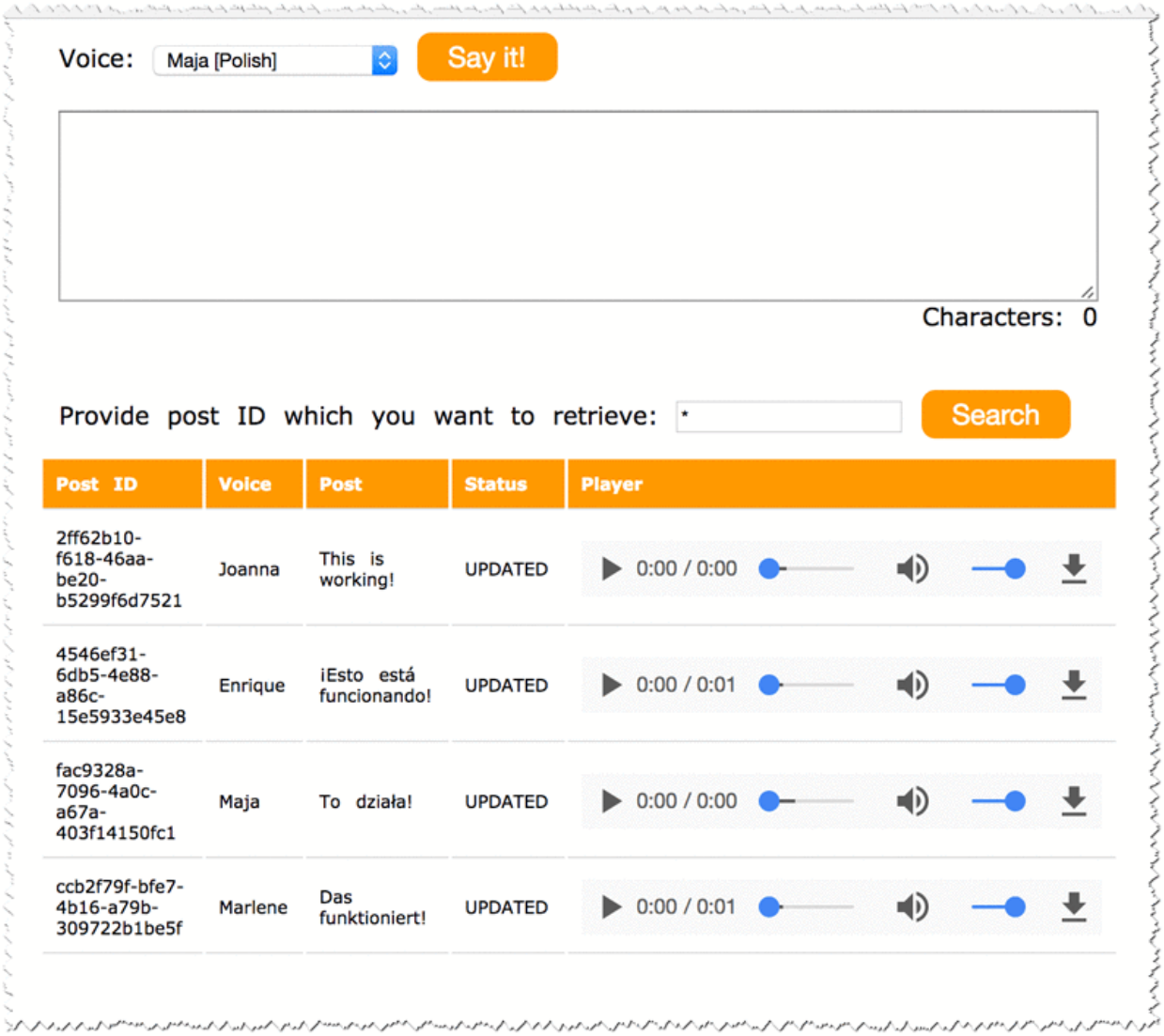
And that's it! You can now check if the website is working.

121. Open a new web browser tab and paste the **Endpoint** URL that you just copied.

You should see a page that looks like this:

If you write something in the text area and choose **Say it!**, the event is sent to your application. The application asynchronously converts the text into an audio file. Depending on the size of the text you provide, it can take a couple of seconds or a couple of minutes to convert it to an audio file.

To view the posts and their audio files, type the post ID or * in the Search box:



122. Choose a **Play** button to hear the audio.

Conclusion

Congratulations, you have completed the lab!

In this lab, you created an application that converts text into speech in dozens of languages and voices. Although the application converts blog posts into speech, it can be used for many other purposes, such as converting text on websites or adding speech functionality to web applications.

The application is *completely serverless*. There are no servers to maintain or patch. By default, the application is *highly available* because AWS Lambda, Amazon API Gateway, Amazon S3, and Amazon DynamoDB use multiple Available Zones.

So now what? Use this approach to imagine and build new applications that provide a much better user experience than previously possible.