**Requirements**

This guide only assumes that you have AWS as your cloud provider, and that you have access to the API Gateway and Lambda management consoles.

The solution is using an AWS Lambda function written in Python, but a similar solution is possible for other languages supported by AWS Lambda.

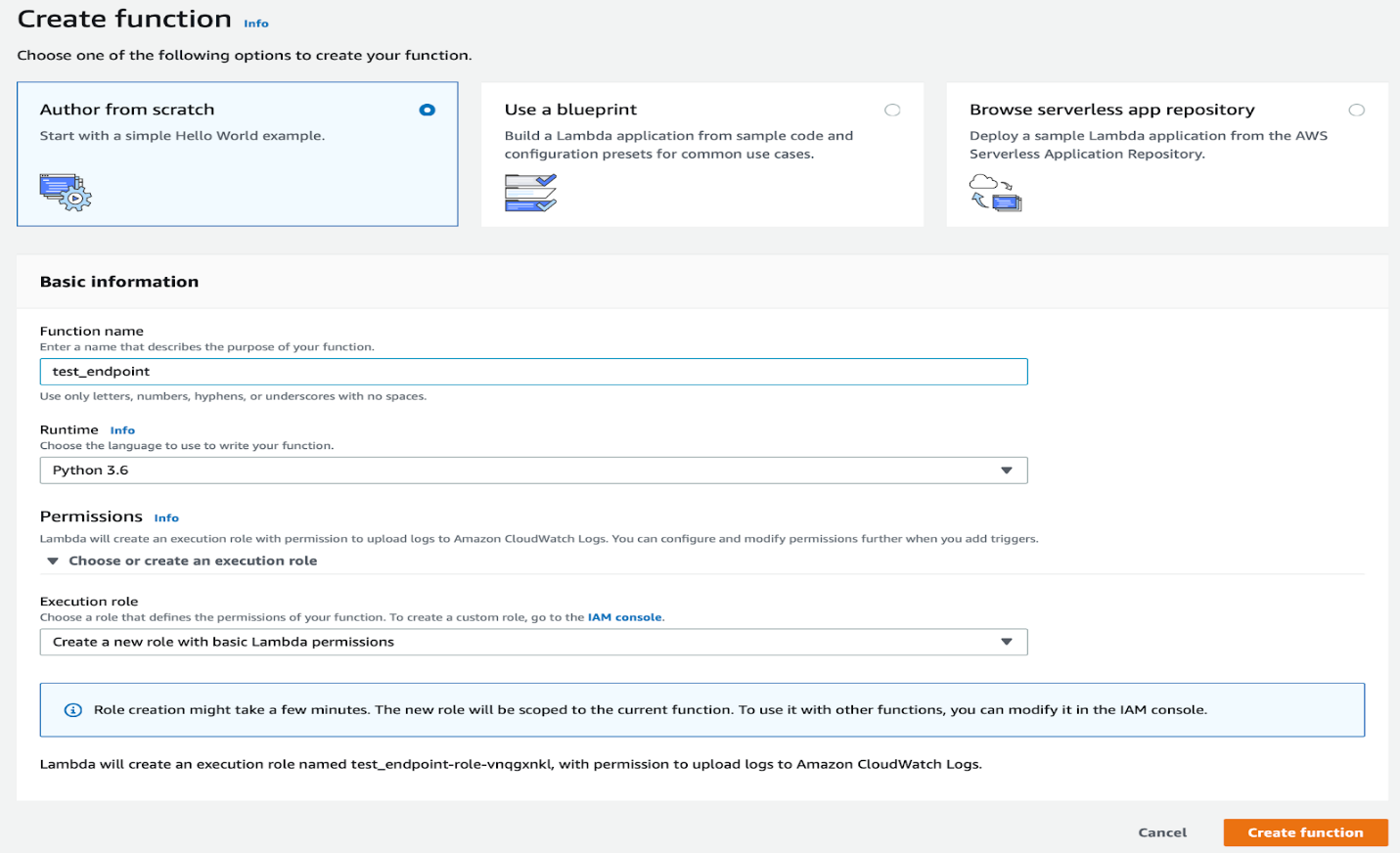
**Writing the AWS Lambda function**

We will start by creating the AWS Lambda function that handles our endpoint incoming requests.

**Creating the function**

Open the AWS Management Console, and from the Services menu select Lambda.

In the Lambda page click on *Create function*. Choose *Author from scratch*, type a name, and select *Python 3.6*or *Python 3.7*runtime. Expand the *Permissions*section, and choose *Create a new role with basic Lambda permissions*. This will setup CloudWatch Logs for your Lambda function, so you’ll be able to track executions. However, we also need S3 permissions to write our files. We will handle that later. Click on *Create function* to finish.



**Writing the function code**

Once your function is created, you will see the function management screen, where you handle configuration, test the function, and of course write the code.

The Python Lambda code editor comes with a default *lambda\_function.py* file with a *lambda\_handler*method. We will use this default file and method for our example.

In the code editor, delete the content of the *lambda\_function.py* file, and type the following code instead (Don’t forget to replace the placeholders with your S3 bucket name and file path):

import json  
import base64  
import boto3

BUCKET\_NAME = 'YOUR\_S3\_BUCKET\_NAME\_HERE'

def lambda\_handler(event, context):  
 file\_content = base64.b64decode(event['content'])  
 file\_path = 'YOUR\_FILE\_PATH\_HERE'  
 s3 = boto3.client('s3') try:  
 s3\_response = s3.put\_object(Bucket=BUCKET\_NAME, Key=file\_path, Body=file\_content) except Exception as e:  
 raise IOError(e)

return {  
 'statusCode': 200,  
 'body': {  
 'file\_path': file\_path  
 }  
 }

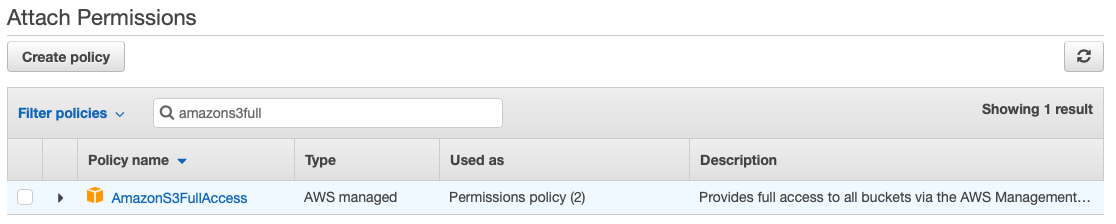
This function receives base64 encoded binary content, and uploads it to a certain bucket in S3. Then it returns the file path on a successful response.

**Granting S3 access to the function**

For our Lambda function to work, we need to grant it access to S3.

In the function management page, go to the *Execution role* section, and click on *View the … role on the IAM console*.

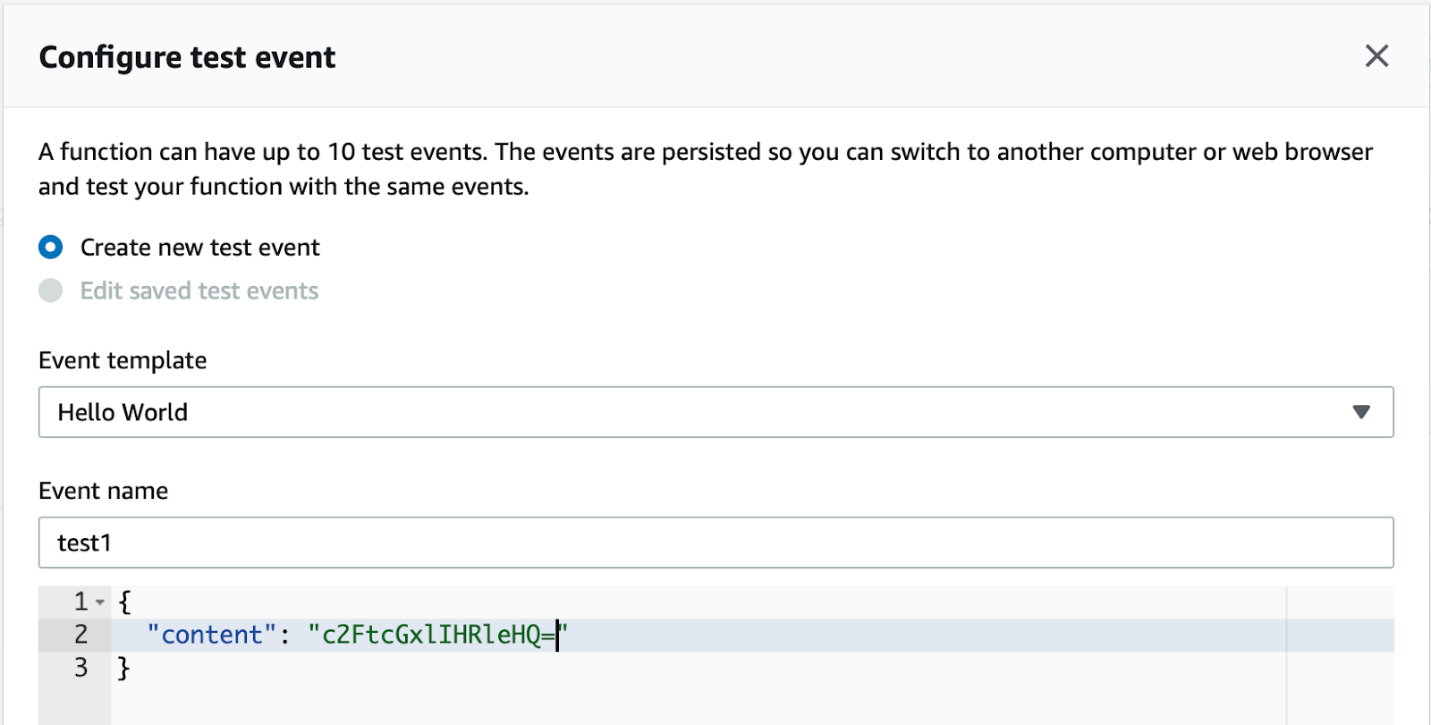
In the role IAM page click on *Attach policies*. Select the *AmazonS3FullAccess* policy and click on *Attach policy.*



You can also create a new policy that grants less access, such as write access only, or access to a specific bucket. However, since the access is hard-coded in your Lambda function (and only writes new files), it is safe to use the full access policy here.

**Testing the function**

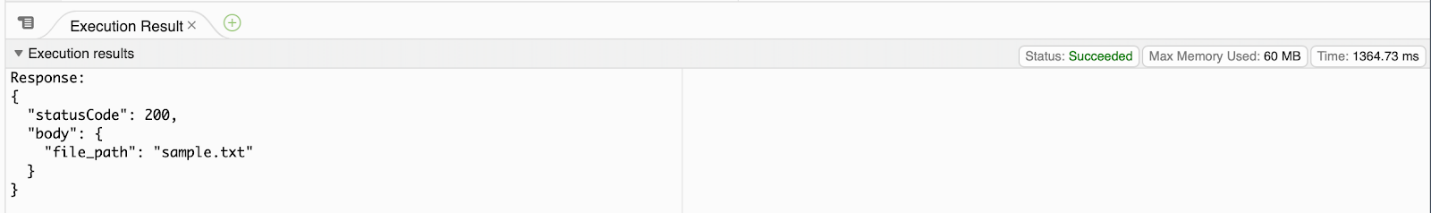
Before even setting up the API Gateway, you can test your Lambda function using a sample message.



In the Lambda function management page click on *Test*, then select *Create new test event*, type a name, and replace the sample data with a simple JSON object that has a key named *content* as follows (The content should be base64 encoded):

{  
 "content": "c2FtcGxlIHRleHQ="  
}

Click on *Create* to create the test event, and then in the function management page click on *Test* again to test your function. A successful test should give you a response similar to this:



You should also be able to see your sample file in S3.

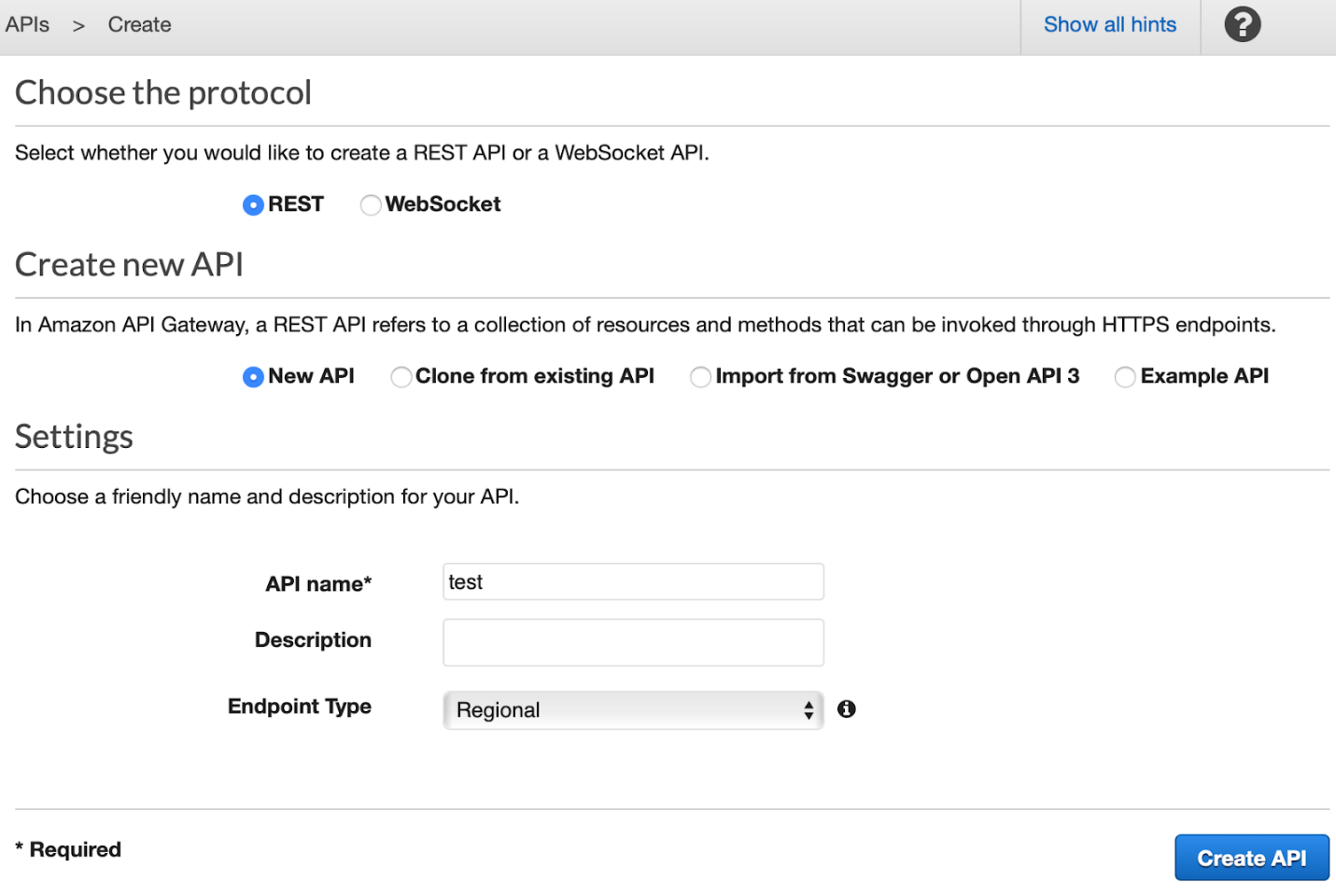
**Setting up the API Gateway**

Next, we need to create the AWS API Gateway endpoint, and have our Lambda function invoked when a request is sent.

**Creating the API Gateway endpoint**

Open the Services menu and select API Gateway.

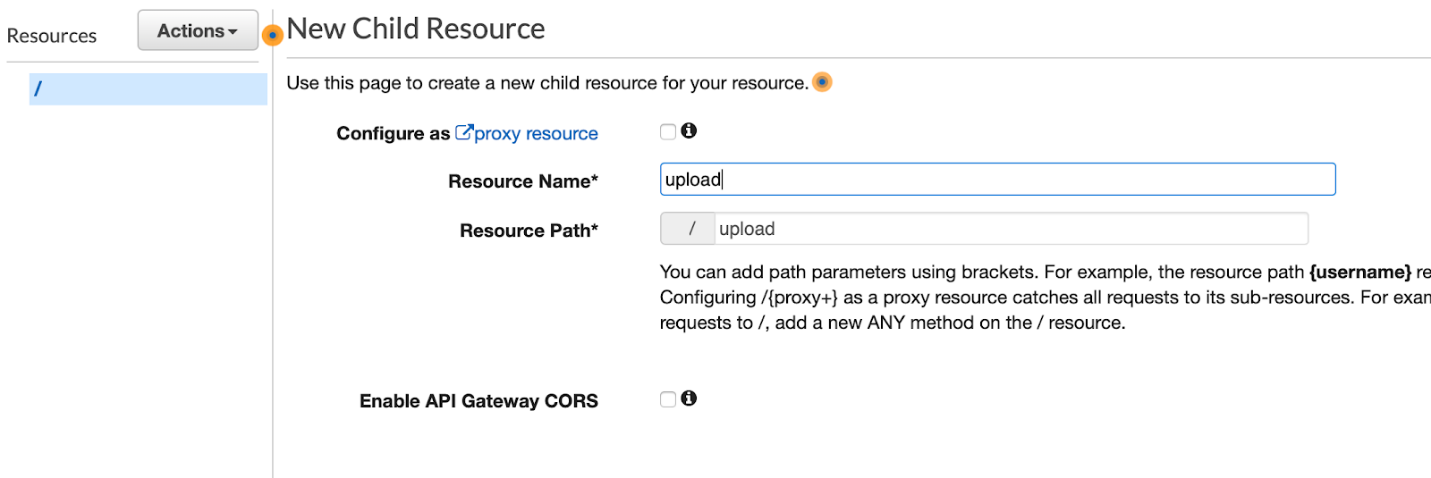
Click on *Create API*, choose *REST*, select *New API*andtype a name. Click on *Create API*to finish.



You will be redirected to the resources page. Here you can define your endpoint paths (defined here as resources), and the methods behavior for each path.

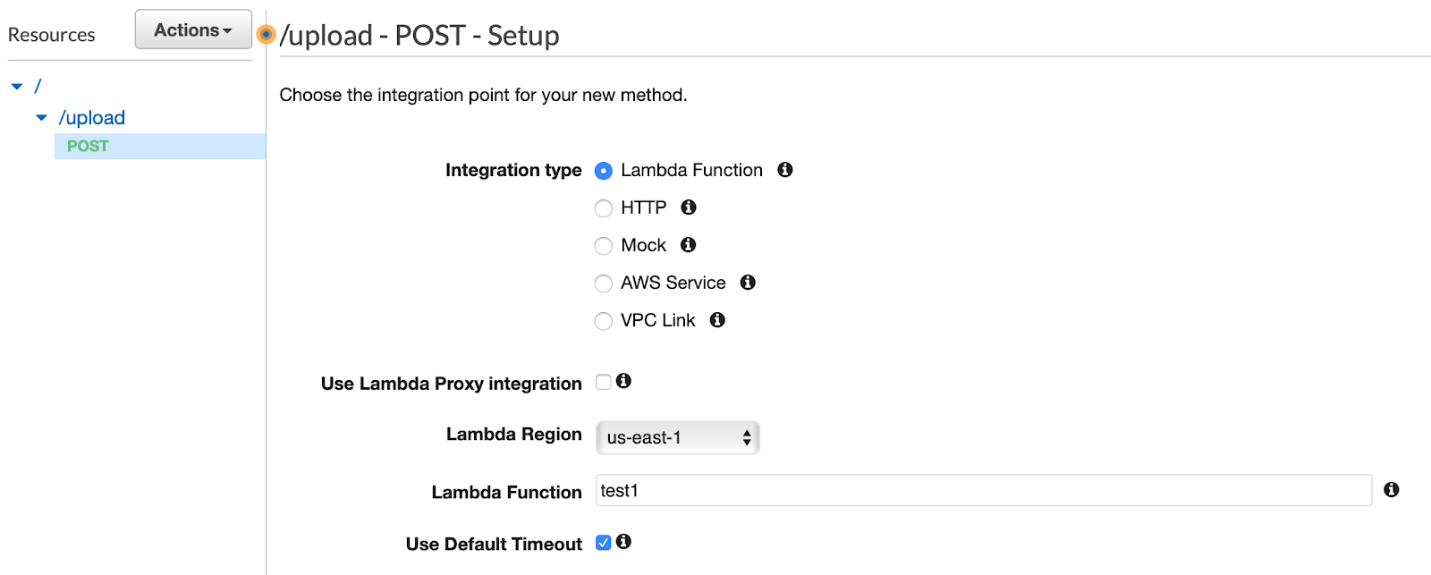
**Setting up a POST method endpoint**

In the *Resources* page, open the *Actions* menu and choose *Create Resource*. Type a resource name and path (for example ‘upload’), and click on *Create Resource*.



Then, open the *Actions* menu again, and this time choose *Create Method*. In the opened dropdown select *POST*, and click on the small *V* icon to confirm.

In the opened form, choose *Lambda Function* as the integration type, and select your newly created Lambda function by typing its name or ARN identifier. Click on *Save*, and confirm that you allow API Gateway to invoke your Lambda function.



You will then be redirected to the *Method Execution* configuration page of your newly created method, where you can handle the flow of the request to the gateway and from the gateway to your environment (in this case, Lambda function), and the response from your environment to the gateway and back to the requester.

You can also test your method execution here, but since we handle binary content, this is not really feasible.

**Accepting binary data of certain types**

We want to configure our endpoint to accept binary content of certain types through our newly created method. We will take as an example PDF files, but the following guide applies for any type of content, and for multiple types of content as well.

Open your method *Integration Request*, and expand the *Mapping Templates* section.

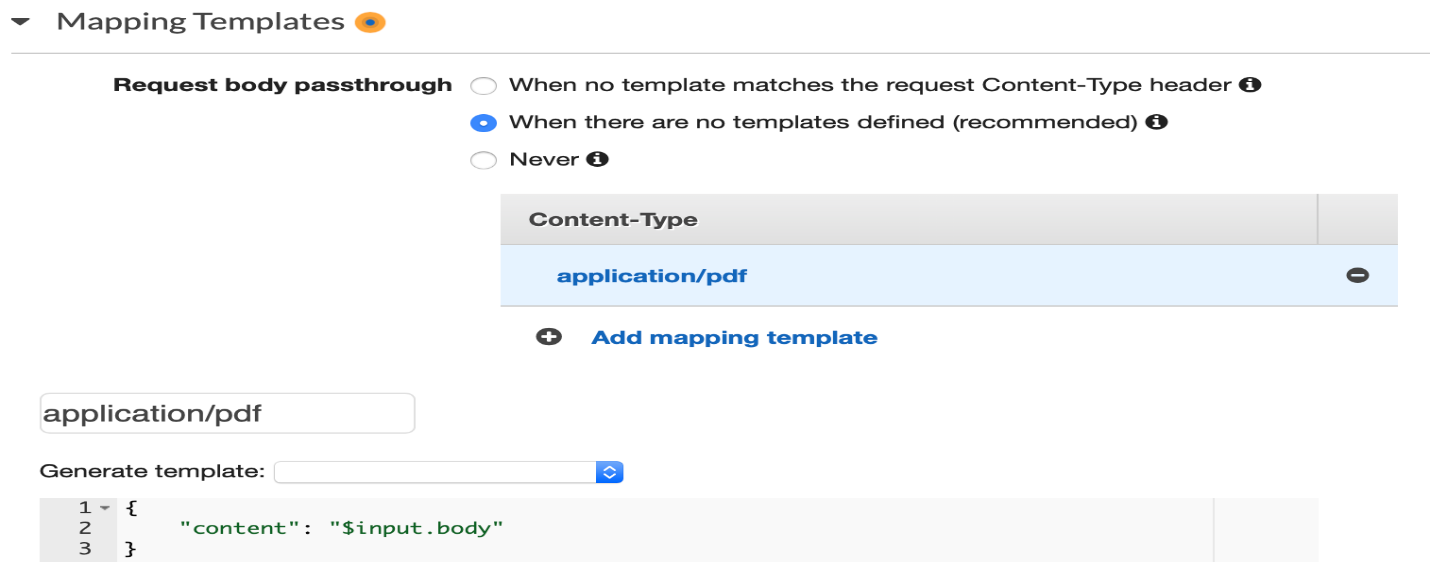
In *Request body passthrough*choose *When there are no templates defined*.

Then, for each of your media types click on *Add mapping template*. In the *Content-Type* box type your media type identifier (for example, for PDF type *application/pdf*), and click on the small *V* icon.

In the template box type the following, then click *Save*:

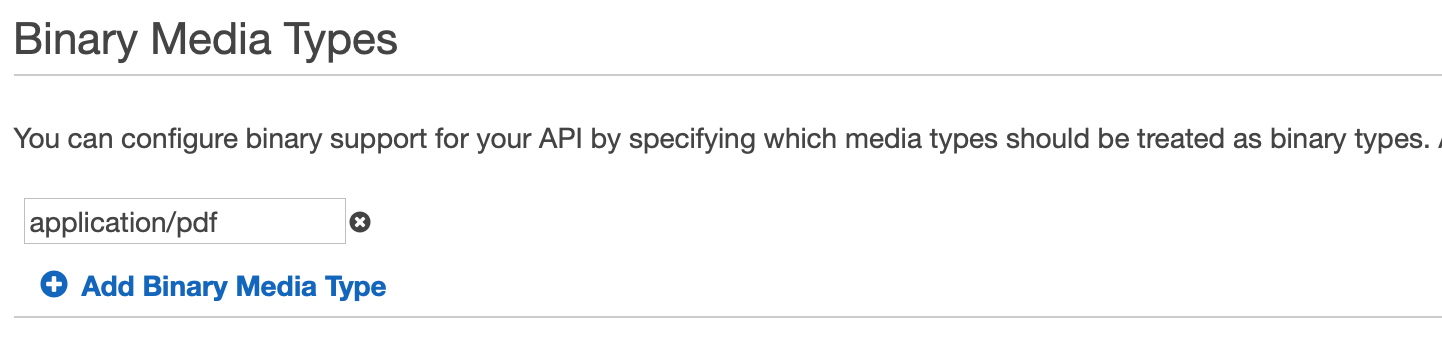
{  
 "content": "$input.body"  
}

This will tell your endpoint to pass the body of your request (the binary content) to the Lambda function inside the *content* property. The content will be passed encoded as base64. Recall that in our Lambda function we got the file content by accessing the event *content* property and decoding it as base64.



Next we need to add binary media types support to our endpoint. This will determine which media types will be treated as binary (and encoded as base64 to be processed in our Lambda function).

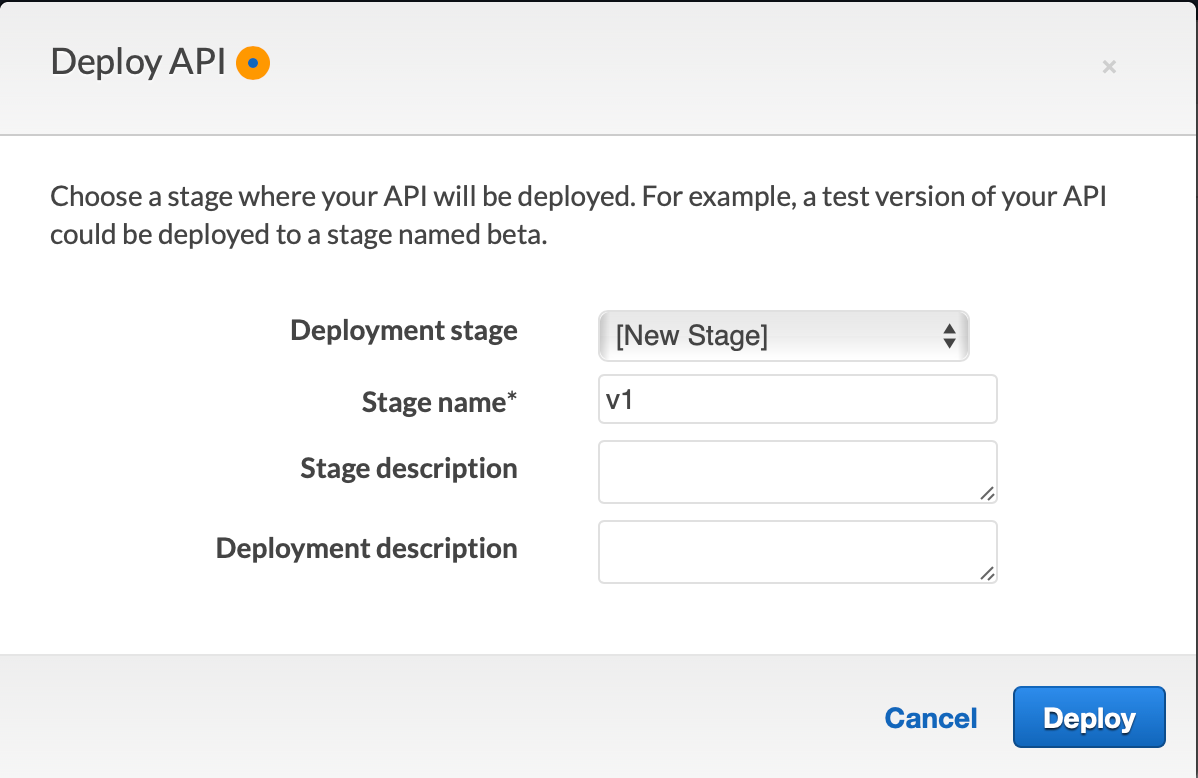
In the endpoint left menu click on *Settings*, then in the *Binary Media Types*section, for each of your content types click on *Add Binary Media Type* and type your content identifier. Click on *Save Changes* to finish.



**Deploying the endpoint**

From the *Actions* menu choose *Deploy API*.

Select *[New Stage]* as the deployment stage, type a stage name (e.g. v1), and click *Deploy*.



You will be redirected to the *Stages* page, and at the top you will see your endpoint *Invoke URL*.

It’s important to note that while your endpoint is managed by AWS, it is not secured out-of-the-box. Once you have deployed your endpoint it becomes publicly accessible, and any request to the invocation URL with the correct method and data will invoke your Lambda function. I am planning to cover endpoint security in my next post.

**Testing the API Gateway**

You can test your newly deployed API Gateway endpoint by sending a POST request with a binary file data to the endpoint invocation URL, using the *cURL*command, or a request generator application such as Postman.

For a cURL command, run something like the following example:

curl --request POST -H "Content-Type: application/pdf" --data-binary "@/path/to/your/file.pdf" https://YOUR\_API\_GATEWAY\_ID.execute-api.us-east-1.amazonaws.com/v1/upload

A successful execution should return the response sent by the Lambda function:

{  
 "statusCode": 200,  
 "body": {  
 "file\_path": "sample.txt"  
 }  
}