Deploying models with AWS Lambda + Docker

We'll use AWS Lambda and Docker to deploy a Keras model

- You can find all the code here: https://github.com/alexeygrigorev/aws-lambda-docker
- This tutorial is based on https://github.com/alexeygrigorev/serverless-deep-learning and https://github.com/alexeygrigorev/aws-lambda-model-deployment-workshop
- We will deploy a model for predicting the types of clothes (trained here: https://github.com/alexeygrigorev/mlbookcamp-code/blob/master/chapter-07-neural-nets/07-neural-nets-train.ipynb)
- Join Datatalks.Club to talk about this tutorial

Plan:

- Create the needed resources in AWS
- Convert the model from Keras to TF Lite
- Extract all the pre-processing logic
- Prepare the code for lambda
- Package everything into a Docker image
- Create an API Gateway

Prerequisites

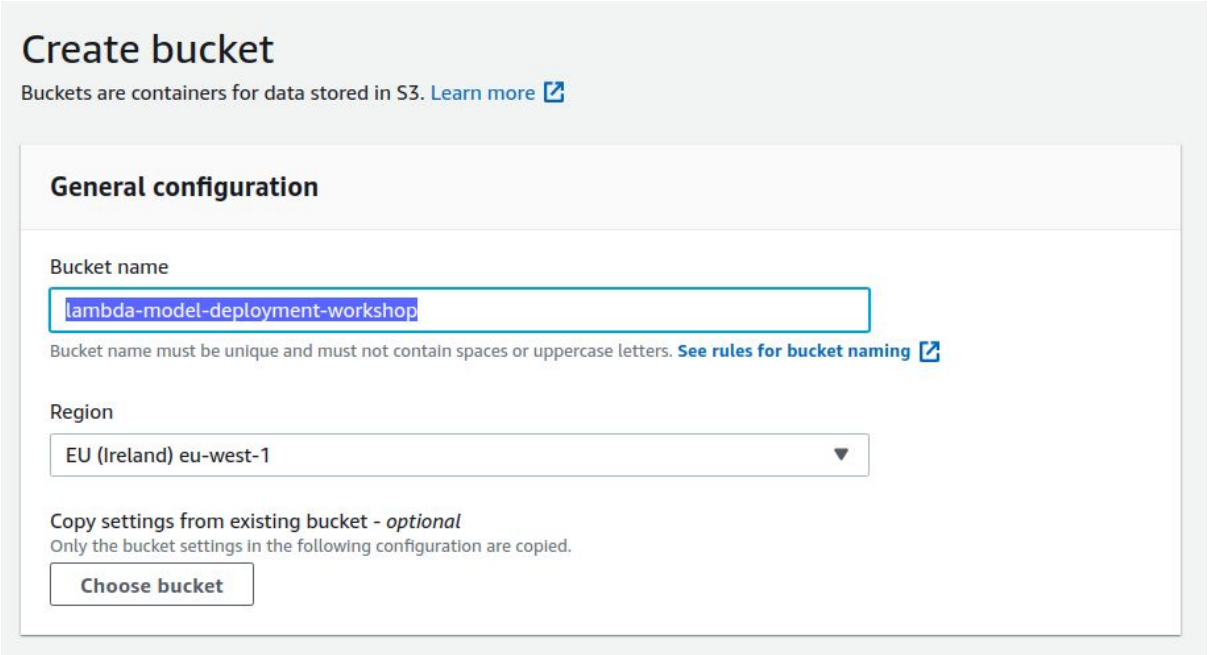
- You need to have Python 3.7 (or Python 3.8). The easiest way to install it — use Anaconda (https://www.anaconda.com/products/individual)
- Install TensorFlow (pip install tensorflow should be sufficient)
- Make sure you have Docker
- You need to have an account in AWS and AWS CLI installed and configured

Preparation work

First, we need to do some prep work. Create a bucket for storing the model

Log in to AWS console

Create an S3 bucket — we will use it for storing the model and the code of the lambda function.
Go to Services ⇒ S3. Click "Create bucket". Write a name ("lambda-model-deployment-workshop"). For this workshop, we'll use the same bucket, so you can skip this step.



Press “create bucket” (at the end)

Preparing the model

Suppose we already trained a model using Keras. Now we want to serve it with AWS Lambda. We need to do a few things for that:

- Convert the model to TF-lite format
- Upload the result to the S3 bucket

We’ll do that in a Jupyter notebook (add link)

Get the model:

```
wget
https://github.com/alexeygrigorev/mlbookcamp-code/releases/download/chapte
r7-model/xception_v4_large_08_0.894.h5
```

Open a Jupyter notebook (or create a simple python script). Start with the imports:

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
```

Load the model:

```
model = keras.models.load_model('xception_v4_large_08_0.894.h5')
```

Convert it to TF-Lite:

```
converter = tf.lite.TFLiteConverter.from_keras_model(model)

tflite_model = converter.convert()

with tf.io.gfile.GFile('clothing-model-v4.tflite', 'wb') as f:
    f.write(tflite_model)
```

The model is ready and we can upload it to S3. Put it to s3://lambda-model-deployment-workshop/clothing-model-v4.tflite:

```
aws s3 cp clothing-model-v4.tflite
s3://lambda-model-deployment-workshop/clothing-model-v4.tflite
```

Preprocessing functions

To apply the model, we need to do the following steps:

- Get the image (as a PIL Image)
- Prepare the image (resize, etc)
- Convert the image to a tensor, apply the pre-processing function (normalization, etc)
- Put the tensor in the model, get the predictions and post-process the predictions

In Keras, the logic for doing most of these operations is in the keras-preprocessing module. We can't use this module inside AWS Lambda (it's too heavy), so we need to write this code ourselves.

Let's do it! For reference, check the notebook here. Later, we'll put this code to our lambda function.

```
from io import BytesIO
from urllib import request

import numpy as np
from PIL import Image

def download_image(url):
    with request.urlopen(url) as resp:
        buffer = resp.read()
    stream = BytesIO(buffer)
    img = Image.open(stream)
    return img

def prepare_image(img, target_size=(224, 224)):
    if img.mode != 'RGB':
```

```
        img = img.convert('RGB')
    img = img.resize(target_size, Image.NEAREST)
    return img

def image_to_array(img):
    return np.array(img, dtype='float32')

def tf_preprocessing(x):
    x /= 127.5
    x -= 1.0
    return x

def convert_to_tensor(img):
    x = image_to_array(img)
    batch = np.expand_dims(x, axis=0)
    return tf_preprocessing(batch)
```

Note: for some models (resnet, vgg), we need to use caffe preprocessing instead of tf preprocessing:

```
mean = [103.939, 116.779, 123.68]

def caffe_preprocessing(x):
    # 'RGB'->'BGR'
    x = x[..., ::-1]

    x[..., 0] -= mean[0]
    x[..., 1] -= mean[1]
    x[..., 2] -= mean[2]

    return x
```

This is how we can use this code to get a tensor:

```
img = download_image(url)
img = prepare_image(img, target_size=(299, 299))
X = convert_to_tensor(img)
```

Now let's use this code in a model!

Loading the model

Load the model:

- Download it form s3
- Load the actual model from disk

Downloading the model is easy: we just use boto3 for that:

```
import os
import boto3

s3_client = boto3.client('s3')

model_bucket = 'lambda-model-deployment-workshop'
model_key = 'clothing-model-v4.tflite'
model_local_path = '/tmp/clothing-model-v4.tflite'

if not os.path.exists(model_local_path):
    s3_client.download_file(model_bucket, model_key, model_local_path)
```

To use the model, we first need to load it with TF lite:

```
# import tensorflow.lite as tflite # if testing locally
import tflite_runtime.interpreter as tflite

interpreter = tflite.Interpreter(model_path=model_local_path)
interpreter.allocate_tensors()

input_details = interpreter.get_input_details()
input_index = input_details[0]['index']

output_details = interpreter.get_output_details()
output_index = output_details[0]['index']
```

Now we can use it:

```
interpreter.set_tensor(input_index, X)
interpreter.invoke()

preds = interpreter.get_tensor(output_index)
```

The `preds` array contains the predictions

Code for Lambda

Each lambda function should have an entrypoint. Let's create it:

```
def lambda_handler(event, context):
    img = download_image(event['url'])
    pred = predict(img)
    result = decode_predictions(pred)
    return result
```

The `predict` function is just the code from the previous sections put together

```
def predict(img):
    img = prepare_image(img, target_size=(299, 299))
    X = convert_to_tensor(img)

    interpreter.set_tensor(input_index, X)
    interpreter.invoke()

    preds = interpreter.get_tensor(output_index)

    return preds[0]
```

The `decode_prediction` function turn the raw output into the final result:

```
labels = [
    'dress',
    'hat',
    'longsleeve',
    'outwear',
    'pants',
    'shirt',
    'shoes',
    'shorts',
    'skirt',
    't-shirt'
]

def decode_predictions(pred):
    result = {c: float(p) for c, p in zip(labels, pred)}
    return result
```

We put all this code in lambda_function.py (see the full example).

Preparing the Docker image

Now we need to prepare a Docker image with all the dependencies. Let's create it:

```
FROM public.ecr.aws/lambda/python:3.7

COPY tflite_runtime-2.2.0-cp37-cp37m-linux_x86_64.whl
tflite_runtime-2.2.0-cp37-cp37m-linux_x86_64.whl

RUN pip3 install --upgrade pip

RUN pip3 install \
    numpy==1.16.5 \
    Pillow==6.2.1 \
    tflite_runtime-2.2.0-cp37-cp37m-linux_x86_64.whl \
    --no-cache-dir

COPY lambda_function.py lambda_function.py

CMD [ "lambda_function.lambda_handler" ]
```

For that you'll need a version of TF-Lite compiled for AWS Lambda:

```
wget
https://github.com/alexeygrigorev/serverless-deep-learning/raw/master/tfli
te/tflite_runtime-2.2.0-cp37-cp37m-linux_x86_64.whl
```

(Use instructions from https://github.com/alexeygrigorev/serverless-deep-learning to compile it yourself for other versions of Python)

Let's build this image:

```
docker build -t tf-lite-lambda .
```

Next, we need to check that the lambda function works.

Let's run the image:

```
docker run --rm \
    -p 8080:8080 \
    -v $(pwd)/clothing-model-v4.tflite:/tmp/clothing-model-v4.tflite \
    tf-lite-lambda
```

And test it with curl:

```
URL="http://localhost:9000/2015-03-31/functions/function/invocations"
REQUEST='{
```

```
    "url":
"https://raw.githubusercontent.com/alexeygrigorev/clothing-dataset-small/m
aster/test/pants/4aabd82c-82e1-4181-a84d-d0c6e550d26d.jpg"
}'

curl -X POST \
    -H "Content-Type: application/json" \
    --data "${REQUEST}" \
    "${URL}" | jq
```

We should see the predictions:

```
{
  "dress": -1.8682900667190552,
  "hat": -4.7612457275390625,
  "longsleeve": -2.3169822692871094,
  "outwear": -1.062570571899414,
  "pants": 9.88715648651123,
  "shirt": -2.8124303817749023,
  "shoes": -3.66628360748291,
  "shorts": 3.2003610134124756,
  "skirt": -2.6023387908935547,
  "t-shirt": -4.835044860839844
}
```

Now create an ECR:

```
aws ecr create-repository --repository-name lambda-images
```

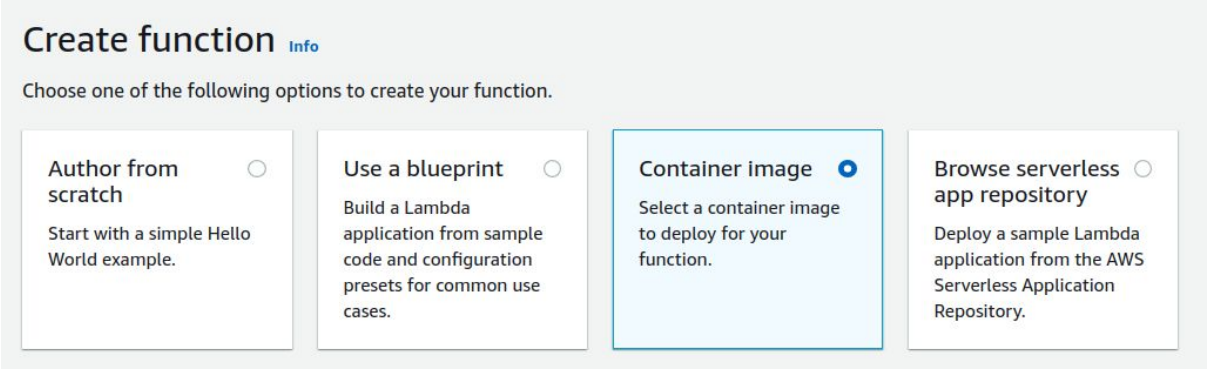
And push the image there:

```
ACCOUNT=XXXXXXXXXXXX

docker tag tf-lite-lambda
${ACCOUNT}.dkr.ecr.eu-west-1.amazonaws.com/lambda-images:tf-lite-lambda

$(aws ecr get-login --no-include-email)

docker push
${ACCOUNT}.dkr.ecr.eu-west-1.amazonaws.com/lambda-images:tf-lite-lambda
```

It’s pushed! Now we can use it to create a lambda

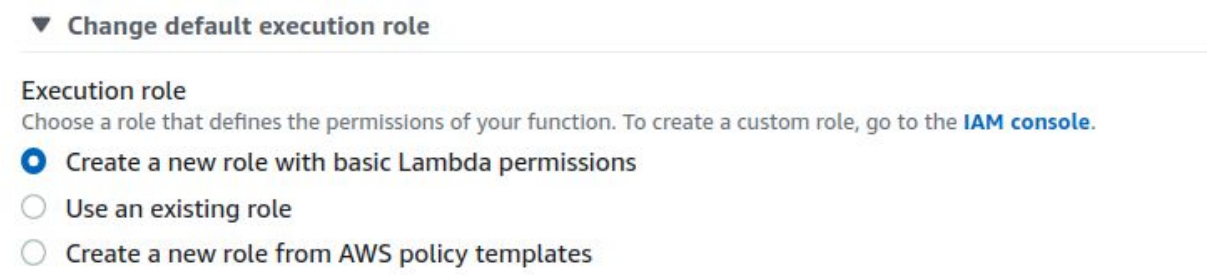
Creating the Lambda function

Create a lambda function. Go to services, select “Lambda”. Click “Create function”. Select “Container image”.



Fill in the details. Container image URI should be the image we created earlier and pushed to ECR: “<ACCOUNT>.dkr.ecr.eu-west-1.amazonaws.com/lambda-images:tf-lite-lambda”. You can also use “Browse images” to find it.

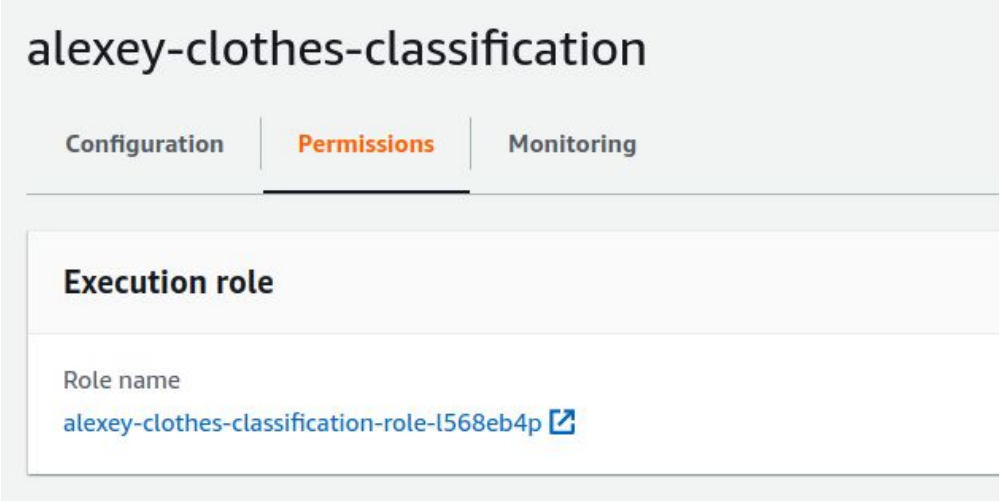
In execution role, choose “Create a new role with basic Lambda permissions”



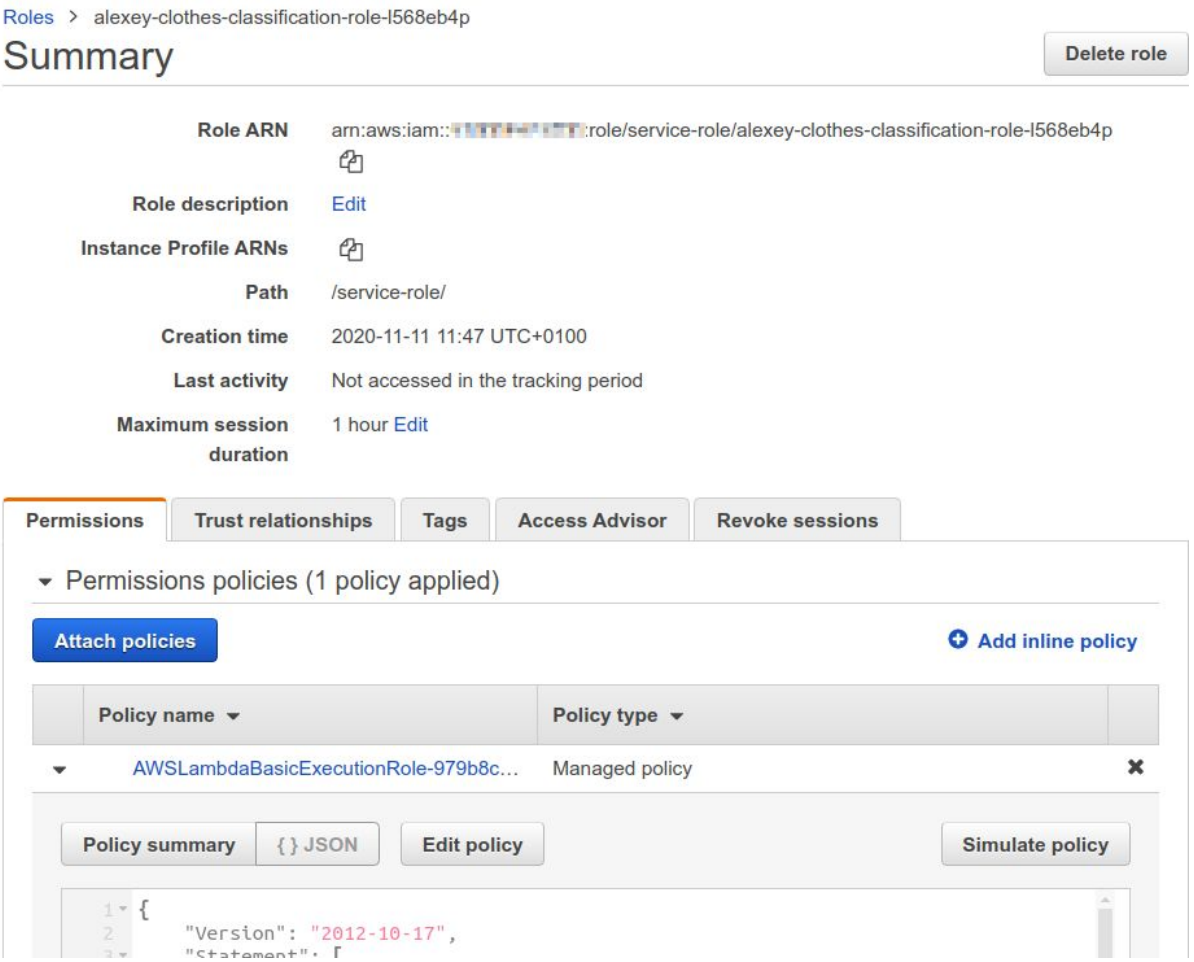
Click “create function”.

Now we have a function! We need to make sure this function can read from the S3 bucket we just created — it will load the model from there.

Go to the “permissions” tab, click on the role name to edit it



Select the policy, click on “edit policy”



Select the tab with “JSON” and add the following statement:

```
{
    "Effect": "Allow",
    "Action": [
        "s3:Get*"
    ],
    "Resource": [
        "arn:aws:s3:::lambda-model-deployment-workshop",
        "arn:aws:s3:::lambda-model-deployment-workshop/*"
    ]
}
```

Where “lambda-model-deployment-workshop” is the name of the bucket we just created — replace it if your bucket is different.

The full policy should look similar to that:

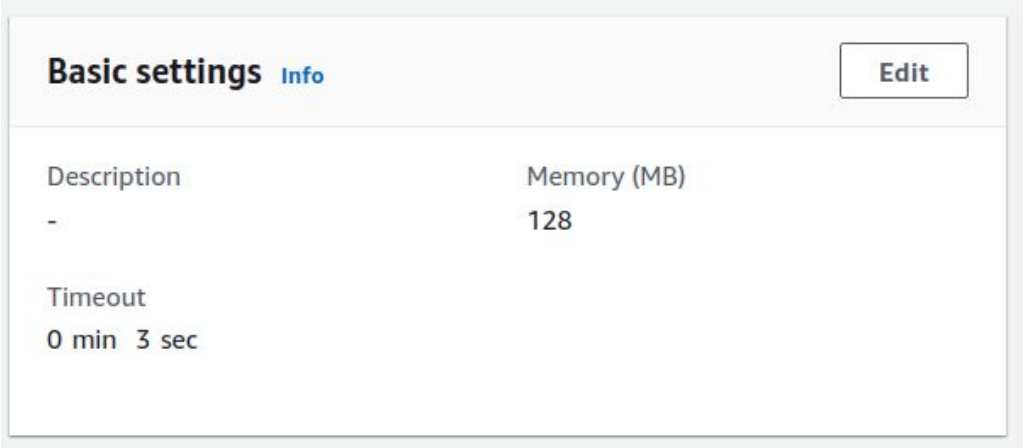
```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": "logs:CreateLogGroup",
            "Resource": "arn:aws:logs:eu-west-1:XXXXXXXXXXXX:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": [
"arn:aws:logs:eu-west-1:XXXXXXXXXXXX:log-group:/aws/lambda/alexey-clothes-classification:*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:Get*"
            ],
            "Resource": [
                "arn:aws:s3:::lambda-model-deployment-workshop",
                "arn:aws:s3:::lambda-model-deployment-workshop/*"
            ]
        }
```

```
    ]
}
```

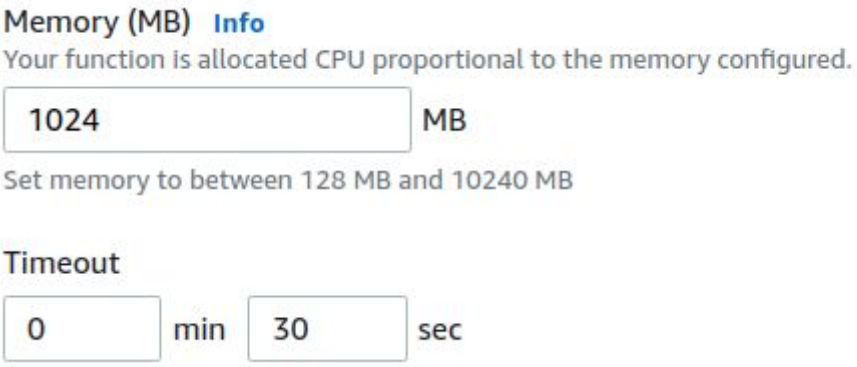
Then review it and save it.

Using the lambda function

Go to the Lambda function. Adjust the basic settings. Click edit:



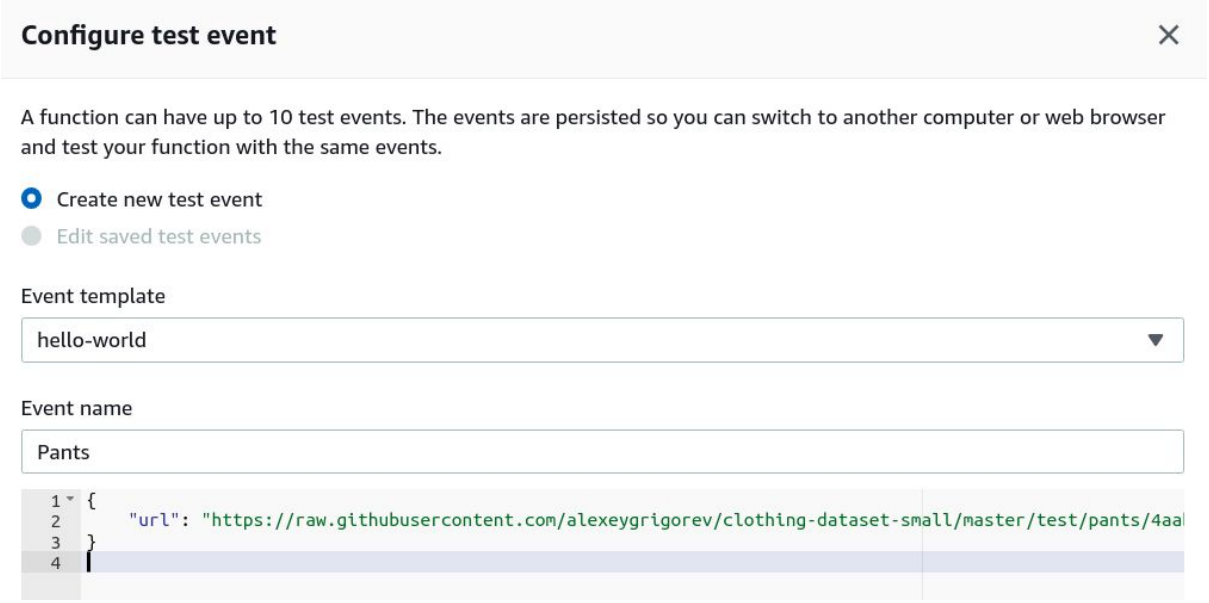
Give it 512MB or 1024MB of RAM and set timeout to 30 sec:



Save it.

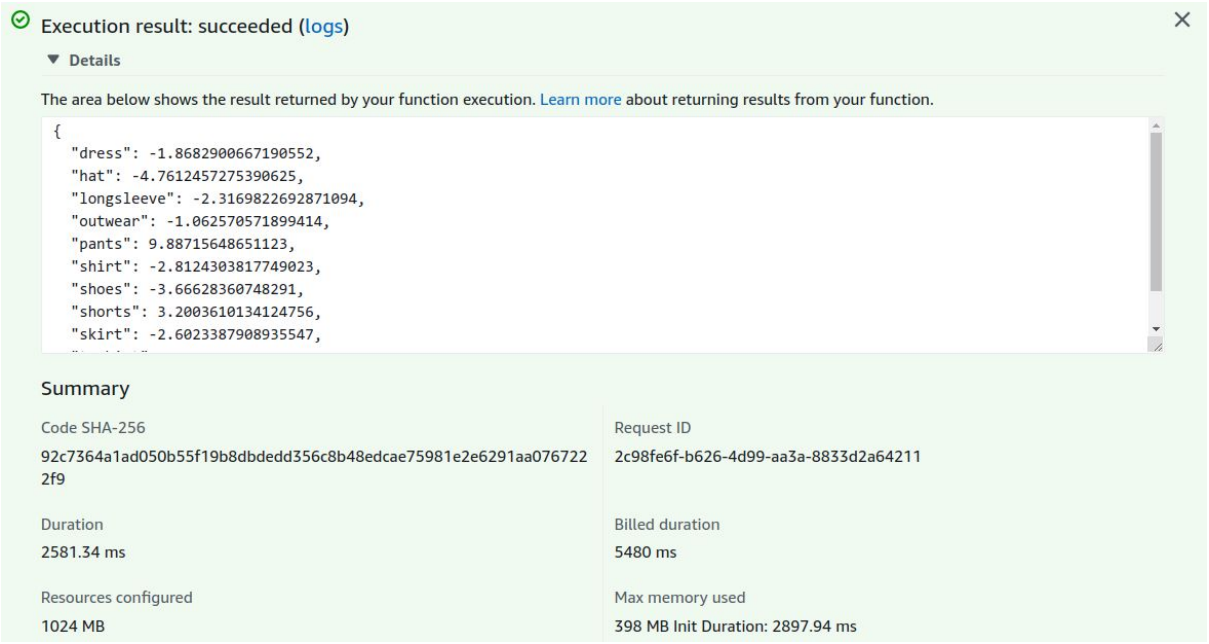
Next, create a test with this request:

```
{
    "url":
"https://raw.githubusercontent.com/alexeygrigorev/clothing-dataset-small/master/test/pants/4aabd82c-82e1-4181-a84d-d0c6e550d26d.jpg"
}
```



Save and test it: click the “test” button.

You should see “Execution results: succeeded”:

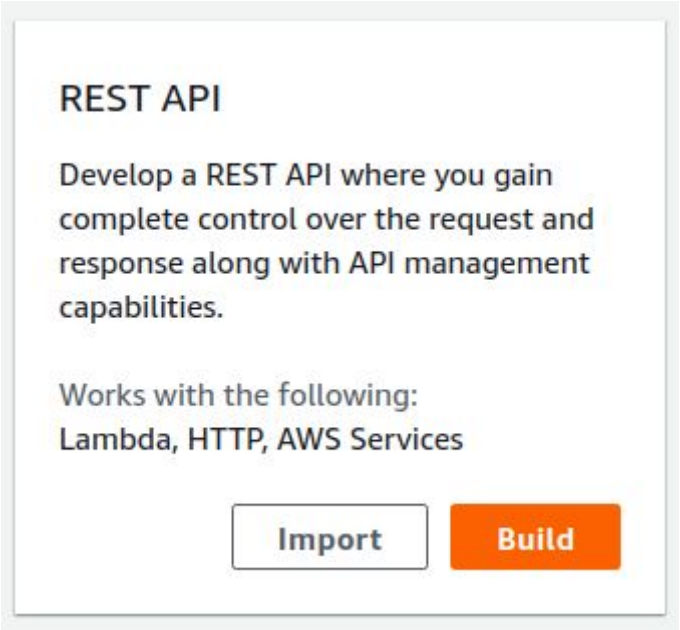


To be able to use it from outside, we need to create an API. We do it with API Gateway.

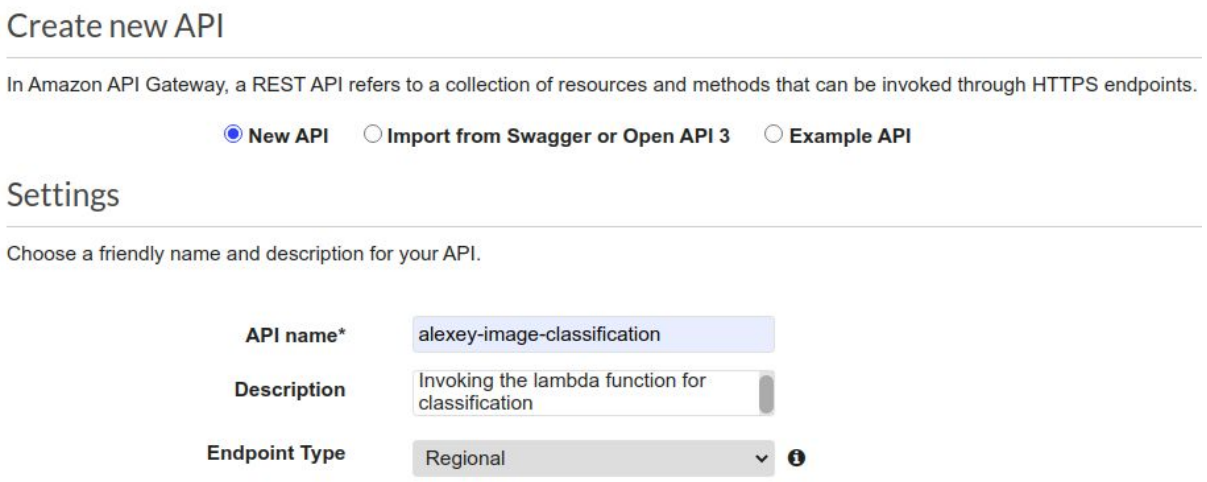
Creating the API Gateway

Go to services ⇒ API Gateway

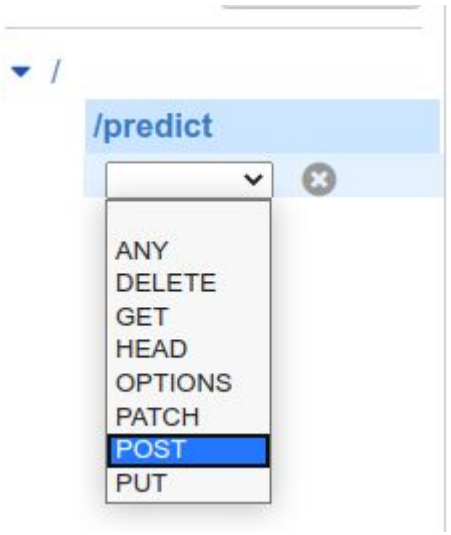
Create a new HTTP API:



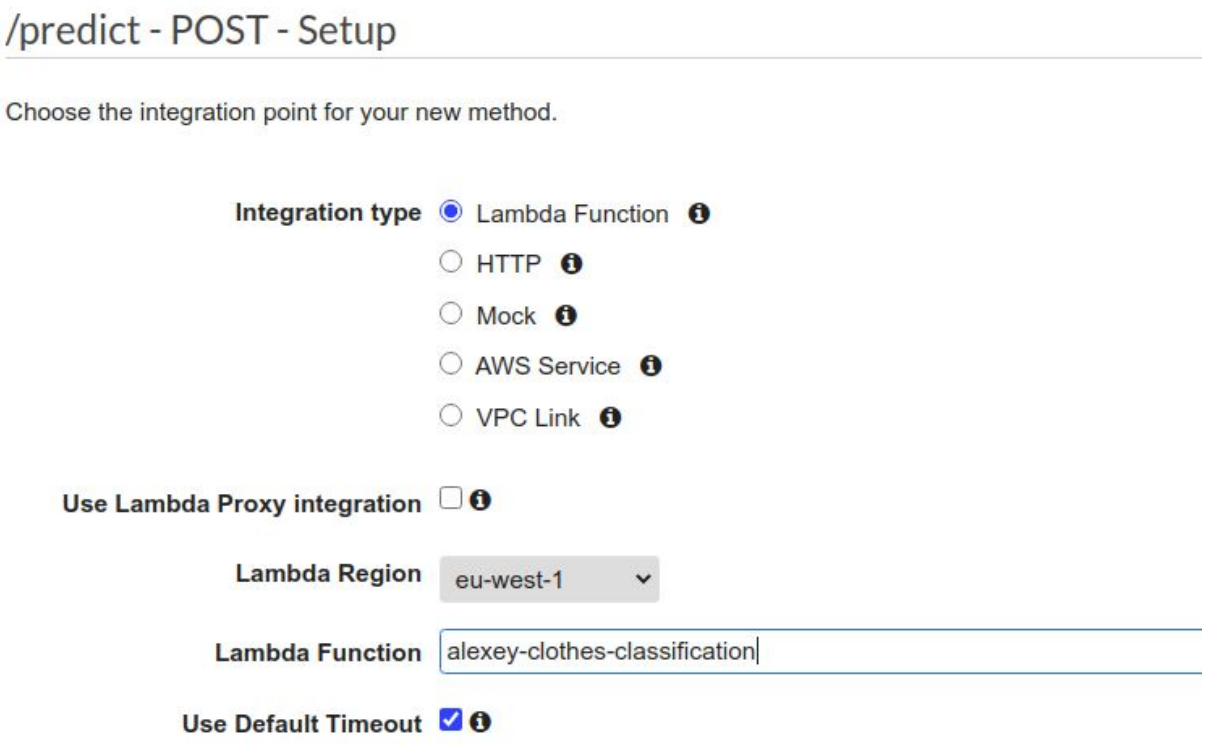
Call it “alexey-image-classification”



Then, create a resource “predict”, and create a method POST in this resource:

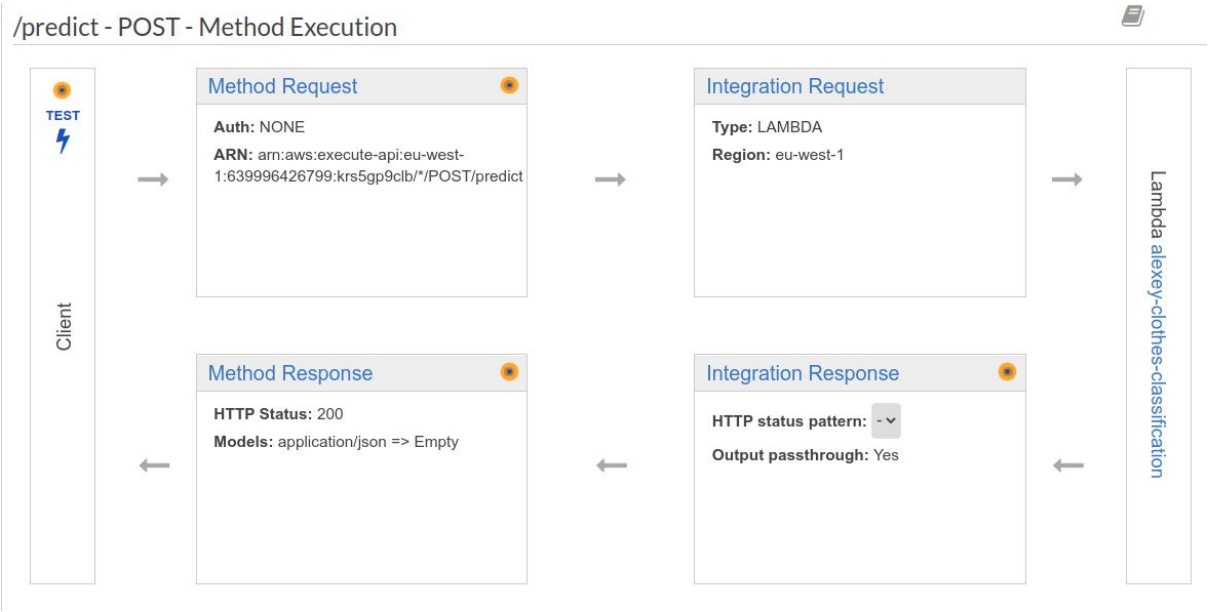


Select “Lambda” and enter the details of your lambda function:



Make sure you don’t select “proxy integration” — this box should remain unchecked.

Now you should see the integration:



To test it, click on “test” and put this request to request body:

```
{
    "url":
"https://raw.githubusercontent.com/alexeygrigorev/clothing-dataset-small/master/test/pants/4aabd82c-82e1-4181-a84d-d0c6e550d26d.jpg"
}
```

You should get the response:

Request: /predict

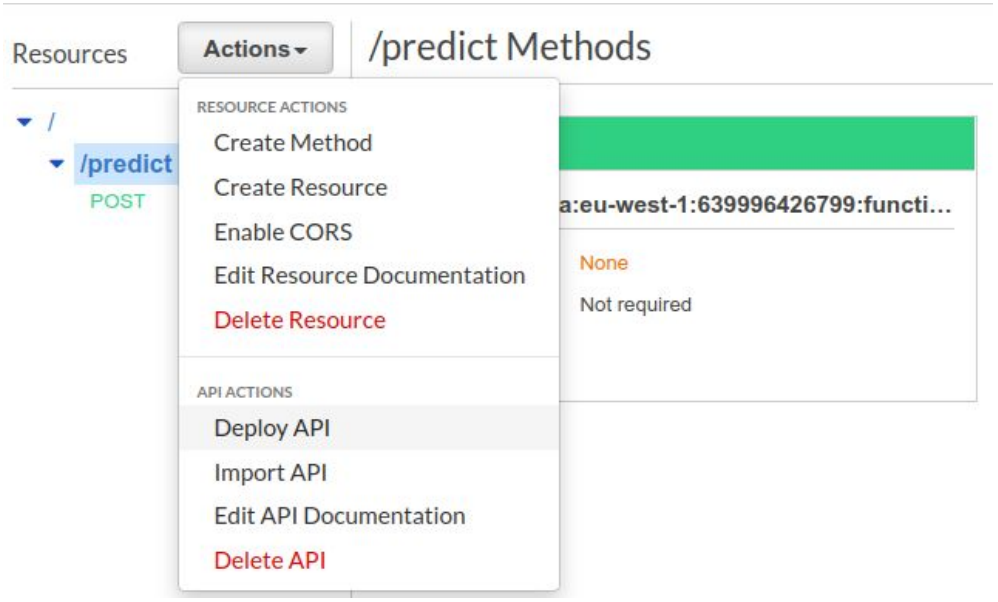
Status: 200

Latency: 1949 ms

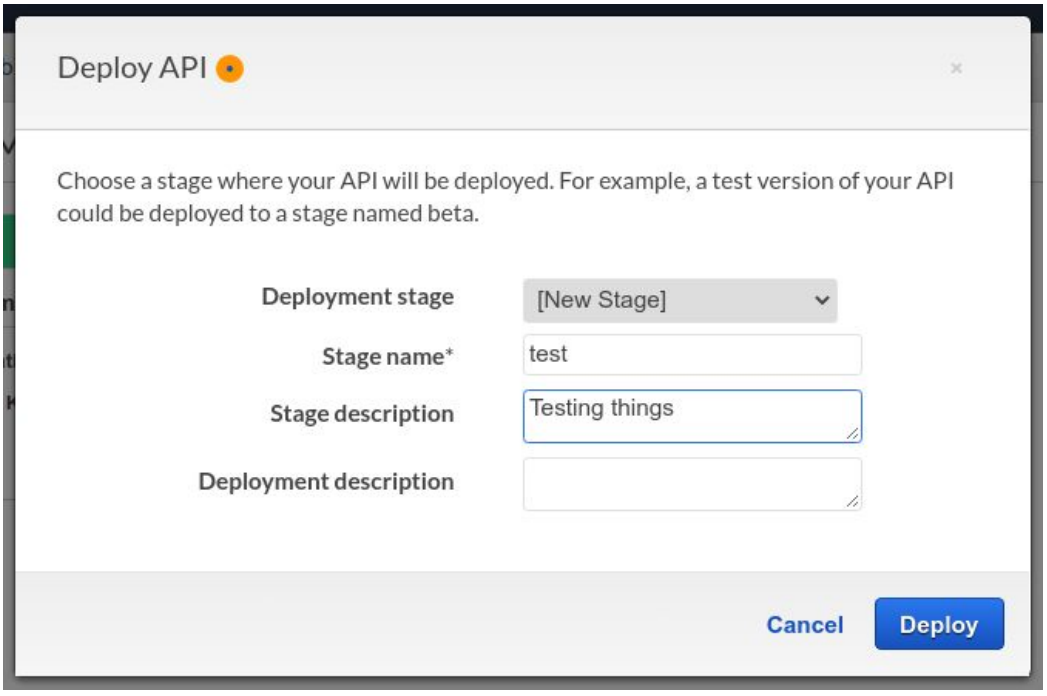
Response Body


```
{
  "dress": -1.8682900667190552,
  "hat": -4.7612457275390625,
  "longsleeve": -2.3169822692871094,
  "outwear": -1.062570571899414,
  "pants": 9.88715648651123,
  "shirt": -2.8124303817749023,
  "shoes": -3.66628360748291,
  "shorts": 3.2003610134124756,
  "skirt": -2.6023387908935547,
  "t-shirt": -4.835044860839844
}
```


To use it, we need to deploy the API. Click on “Deploy API” from Actions.



Create a new stage “test”:



And get the url in from the “Invoke URL” field. For us, it’s
https://krs5gp9clb.execute-api.eu-west-1.amazonaws.com/test

Now we can test it from the terminal:

```
URL="https://krs5gp9clb.execute-api.eu-west-1.amazonaws.com/test"
REQUEST='{
    "url":
"https://raw.githubusercontent.com/alexeygrigorev/clothing-dataset-small/master/test/pants/4aabd82c-82e1-4181-a84d-d0c6e550d26d.jpg"
```

```
}'

curl -X POST \
    -H "Content-Type: application/json" \
    --data "${REQUEST}" \
    "${URL}"/predict | jq
```

The response:

```
{
  "dress": -1.8682900667190552,
  "hat": -4.7612457275390625,
  "longsleeve": -2.3169822692871094,
  "outwear": -1.062570571899414,
  "pants": 9.88715648651123,
  "shirt": -2.8124303817749023,
  "shoes": -3.66628360748291,
  "shorts": 3.2003610134124756,
  "skirt": -2.6023387908935547,
  "t-shirt": -4.835044860839844
}
```

Now it’s working!