Deploying models with AWS Lambda

- This tutorial is based on https://github.com/alexeygrigorev/serverless-deep-learning
- 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)
- You can find all the code here: https://github.com/alexeygrigorev/aws-lambda-model-deployment-workshop

Plan:

- Create the needed resources in AWS (optional)
- Convert the model from Keras to TF Lite
- Extract all the pre-processing logic
- Prepare the code for lambda
- Package the dependencies along with the code into a single archive

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)
- You need to have an account in AWS

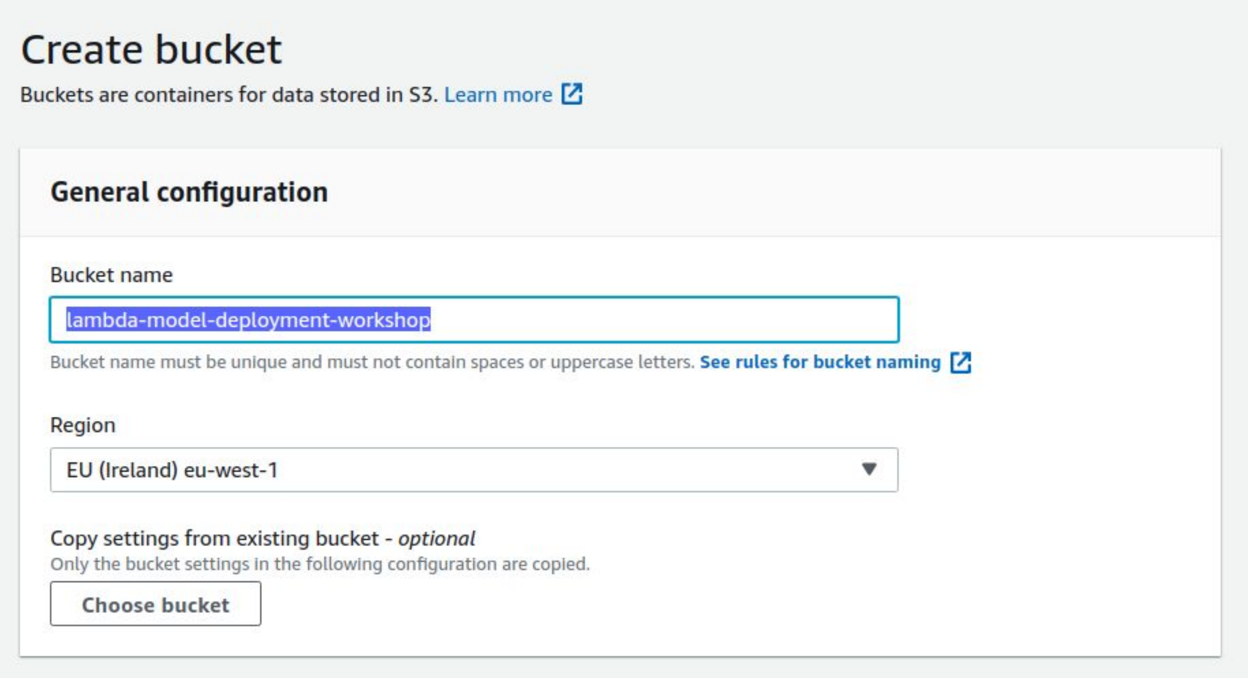
AWS Preparation work

First, we need to do some prep work:

- Create a bucket for storing the code of our lambda function and the model
- Create the lambda function
- Make sure the lambda function can read from the bucket

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.



Create bucket

Buckets are containers for data stored in S3. Learn more

General configuration

Bucket name

lambda-model-deployment-workshop

Bucket name must be unique and must not contain spaces or uppercase letters. See rules for bucket naming

Region

EU (Ireland) eu-west-1

Copy settings from existing bucket - *optional*

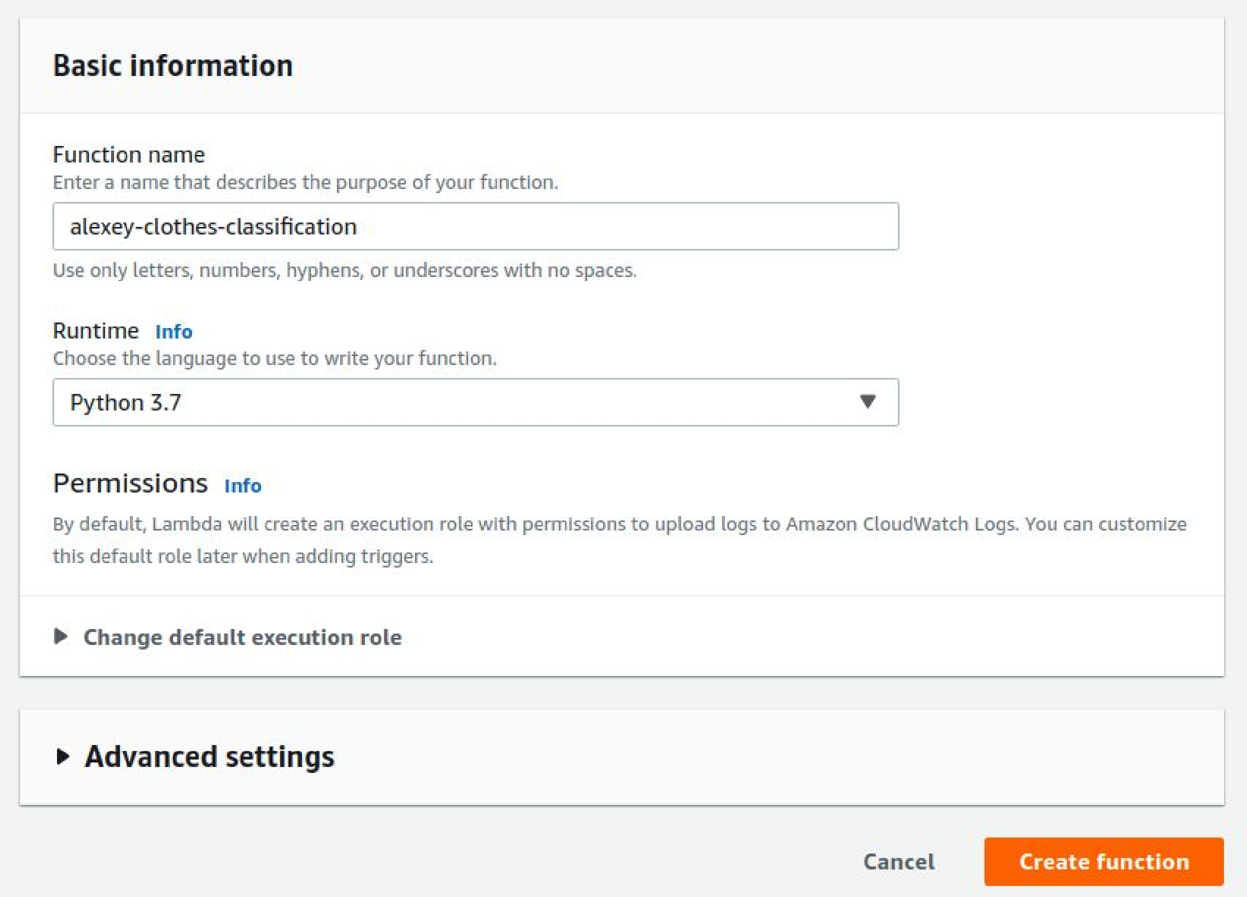
Only the bucket settings in the following configuration are copied.

Press “create bucket” (at the end)

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

Select “create from scratch”, and fill in the basic information: function name, runtime (Python 3.7). The rest can stay as is.

Click “create function”.



Basic information

Function name
Enter a name that describes the purpose of your function.
alexey-clothes-classification
Use only letters, numbers, hyphens, or underscores with no spaces.

Runtime Info
Choose the language to use to write your function.
Python 3.7 ▼

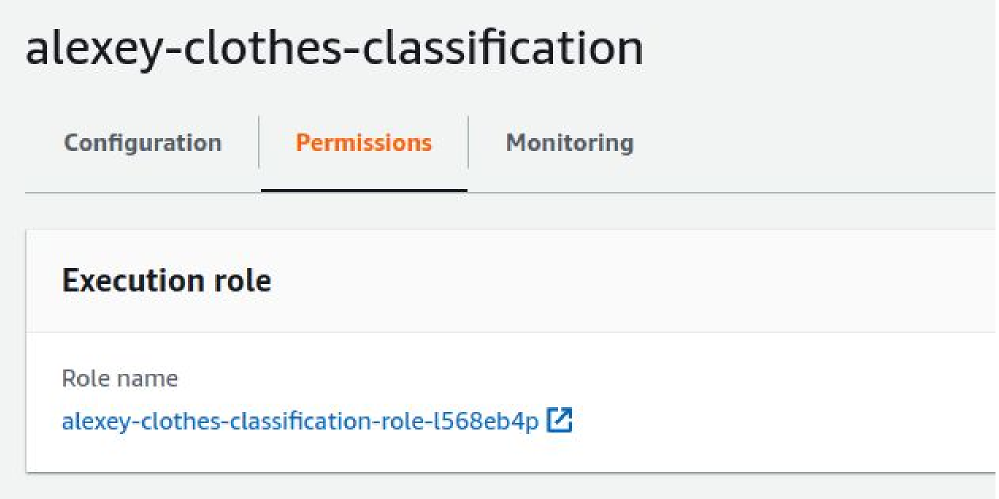
Permissions Info
By default, Lambda will create an execution role with permissions to upload logs to Amazon CloudWatch Logs. You can customize this default role later when adding triggers.

▶ Change default execution role

▶ Advanced settings

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

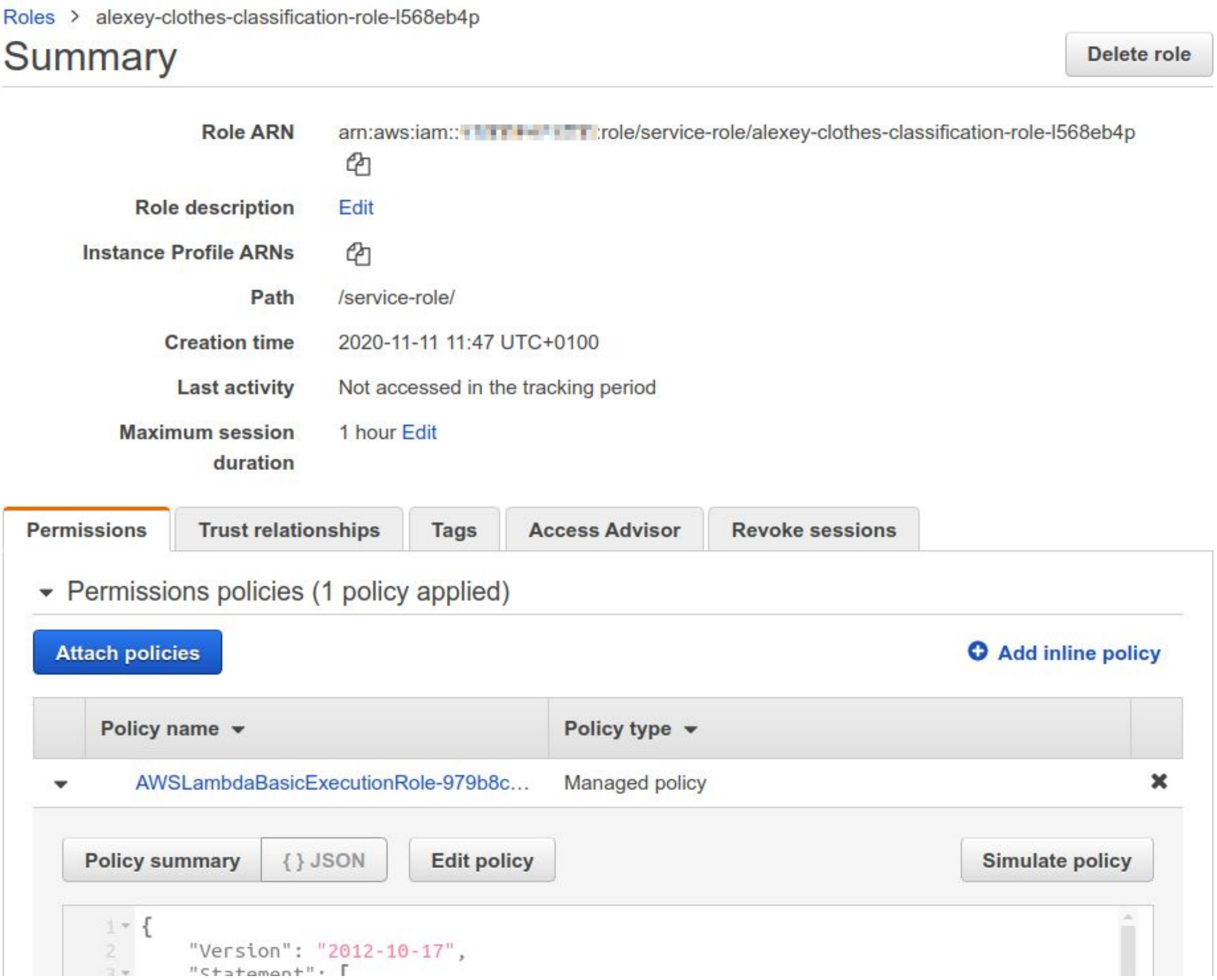


alexey-clothes-classification

Execution role

Role name
alexey-clothes-classification-role-l568eb4p

Select the policy, click on “edit policy”



Roles > alexey-clothes-classification-role-l568eb4p

Summary

Delete role

Role ARN: arn:aws:iam::[redacted]:role/service-role/alexey-clothes-classification-role-l568eb4p

Role description: Edit

Instance Profile ARNs:

Path: /service-role/

Creation time: 2020-11-11 11:47 UTC+0100

Last activity: Not accessed in the tracking period

Maximum session duration: 1 hour Edit

Permissions policies (1 policy applied)

Policy name	Policy type
AWSLambdaBasicExecutionRole-979b8c…	Managed policy

```
{
    "Version": "2012-10-17",
    "Statement": [
```

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.

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/chapter7-
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 package

To deploy something to AWS Lambda, we need to prepare a zip archive that contains everything: the code itself and all the dependencies.

AWS Lambda uses Amazon Linux, so we need to make sure we use the proper binaries when installing the dependencies. The best way to do this is to package everything in Docker.

So, let’s create a Dockerfile for that. We can name it build.dockerfile:

```
FROM amazonlinux
```

```
WORKDIR /tflite

RUN yum groupinstall -y development
RUN yum install -y python3.7
RUN yum install -y python3-devel

RUN pip3 install wheel

WORKDIR /app

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

# here we don’t use requirements.txt/Pipenv for simplicity
RUN pip3 install \
    numpy==1.16.5 \
    Pillow==6.2.1 \
    tflite_runtime-2.2.0-cp37-cp37m-linux_x86_64.whl \
    -t build

WORKDIR /app/build

RUN (find . -name "tests" | xargs -n1 rm -rf) && \
    (find . -name \*.pyc | xargs -n1 rm -rf) && \
    (find . -name "__pycache__" | xargs -n1 rm -rf) && \
    (find . -name "*.dist-info" | xargs -n1 rm -rf)

COPY lambda_function.py lambda_function.py

RUN zip -r ../build.zip * > /dev/null

WORKDIR /app

ENTRYPOINT [ "cp", "build.zip", "results/build.zip" ]
```

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

```
wget
https://github.com/alexeygrigorev/serverless-deep-learning/raw/master/tflite/
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 an image:

```
BUILDER_IMAGE_NAME=tflite_build_lambda
docker build -t ${BUILDER_IMAGE_NAME} -f build.dockerfile .
```

And get the archive out of it:

```
docker run --rm \
    -v $(pwd):/app/results \
    ${BUILDER_IMAGE_NAME}
```

After executing it, you’ll have a zip archive. You can use it now to upload it to AWS Lambda

Testing the package

Next, we need to check that the package can be unpacked correctly and no dependencies are missing.

Let’s create a test.docker file with the following content:

```
FROM amazonlinux

RUN yum groupinstall -y development
RUN yum install -y python3.7 python3-devel
RUN pip3 install boto3

WORKDIR /app

COPY build.zip .
RUN unzip build.zip

COPY test.py test.py

ENV AWS_DEFAULT_REGION="local"
ENV AWS_ACCESS_KEY_ID="KEY"
ENV AWS_SECRET_ACCESS_KEY="SECRET"

ENTRYPOINT [ "python3", "test.py" ]
```

Now, create test.py:

```
import lambda_function
```

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

print(lambda_function.handler(event, None))
```

We're ready to build the image:

```
TEST_IMAGE_NAME=tflite_test_lambda
docker build -t ${TEST_IMAGE_NAME} -f test.dockerfile .
```

And run it to test that our flow works correctly

```
docker run --rm \
    -v $(pwd)/clothing-model-v4.tflite:/tmp/clothing-model-v4.tflite \
    ${TEST_IMAGE_NAME}
```

We should get:

```
{'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}
```

So our model is working!

Updating the Lambda function

For that we'll need two things:

- Upload the package to S3
- Update the actual lambda function

Let's do it:

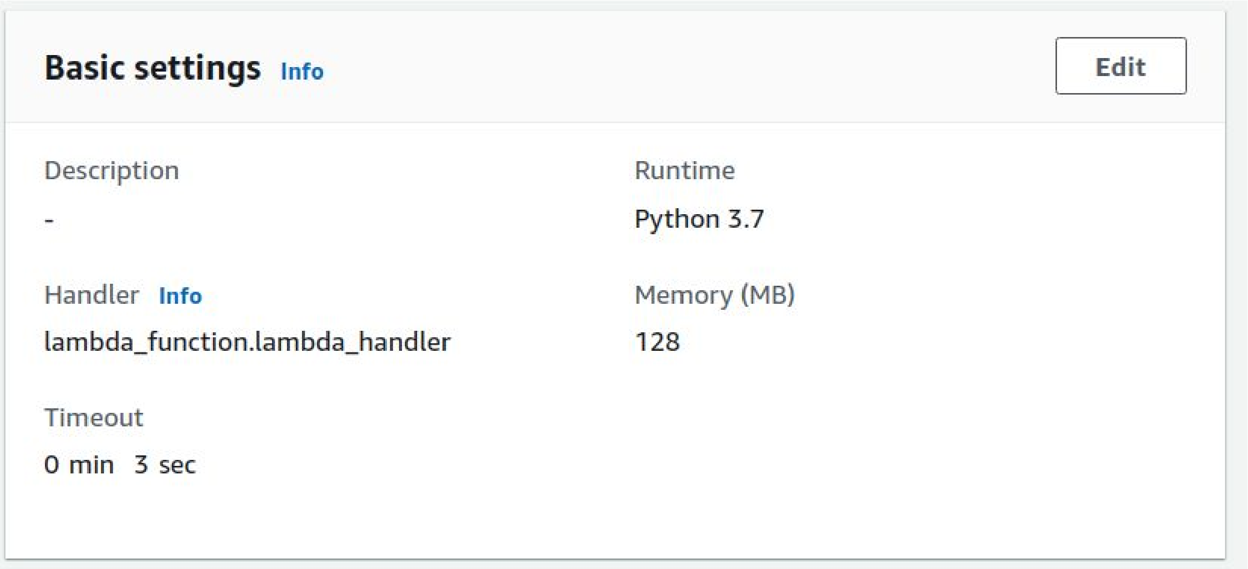
```
ZIP_FILE="build.zip"
S3_BUCKET="lambda-model-deployment-workshop"
S3_KEY="alexey/clothes-classification-package.zip"
FUNCTION_NAME="alexey-clothes-classification"

aws s3 cp "${ZIP_FILE}" "s3://${S3_BUCKET}/${S3_KEY}"
aws lambda update-function-code \
    --function-name ${FUNCTION_NAME} \
    --s3-bucket ${S3_BUCKET} \
    --s3-key ${S3_KEY}
```

Testing the lambda function

Go to the Lambda function.

First, adjust the basic settings. Click edit:



Basic settings

Description
-

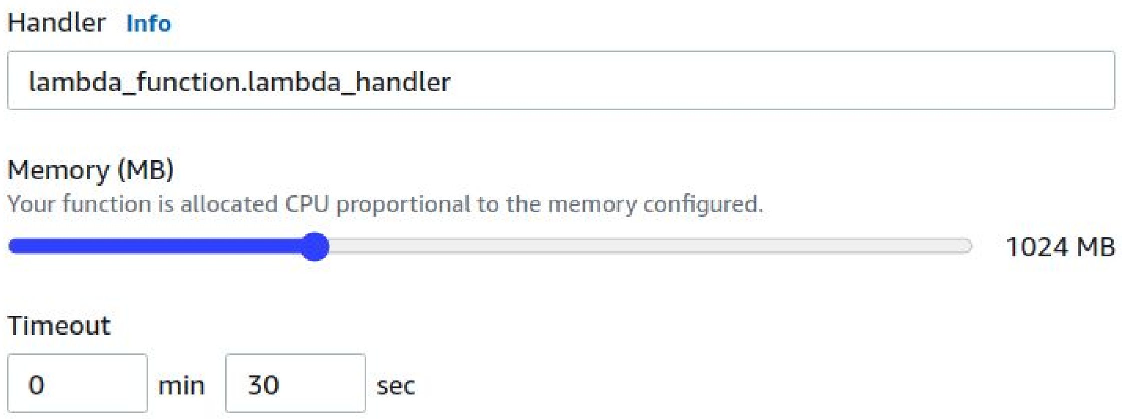
Runtime
Python 3.7

Handler
lambda_function.lambda_handler

Memory (MB)
128

Timeout
0 min 3 sec

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



Handler
lambda_function.lambda_handler

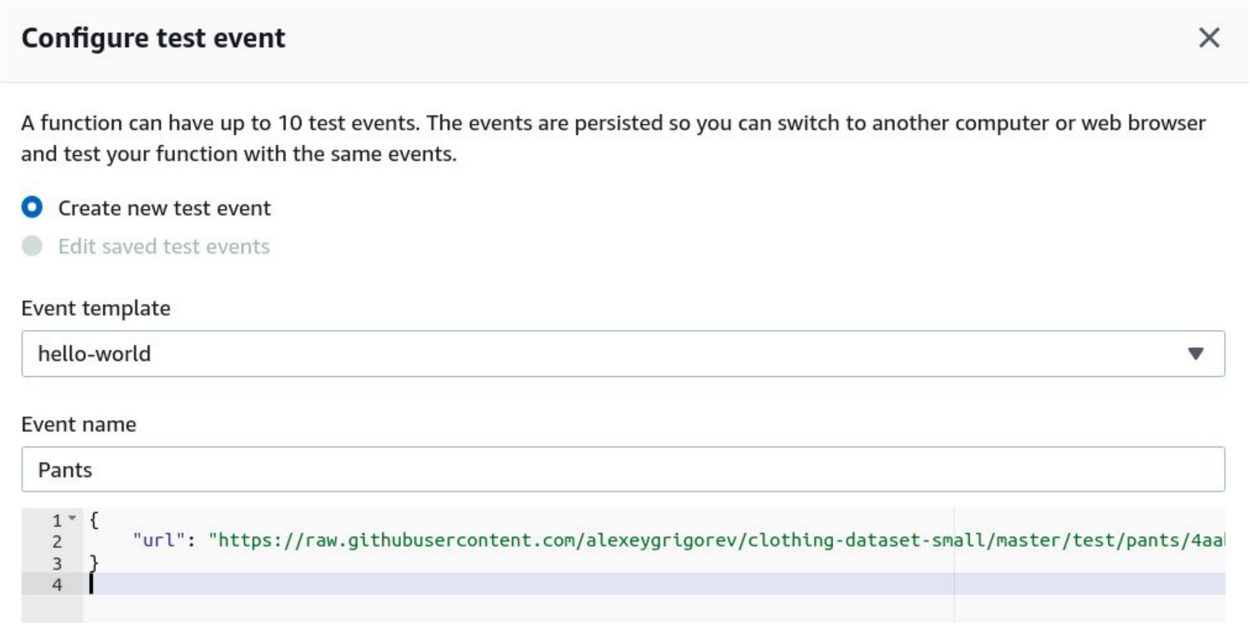
Memory (MB)
Your function is allocated CPU proportional to the memory configured.
1024 MB

Timeout
0 min 30 sec

Save it.

Next, create a test with this request:

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



Configure test event

A function can have up to 10 test events. The events are persisted so you can switch to another computer or web browser and test your function with the same events.

● Create new test event

○ Edit saved test events

Event template

hello-world ▼

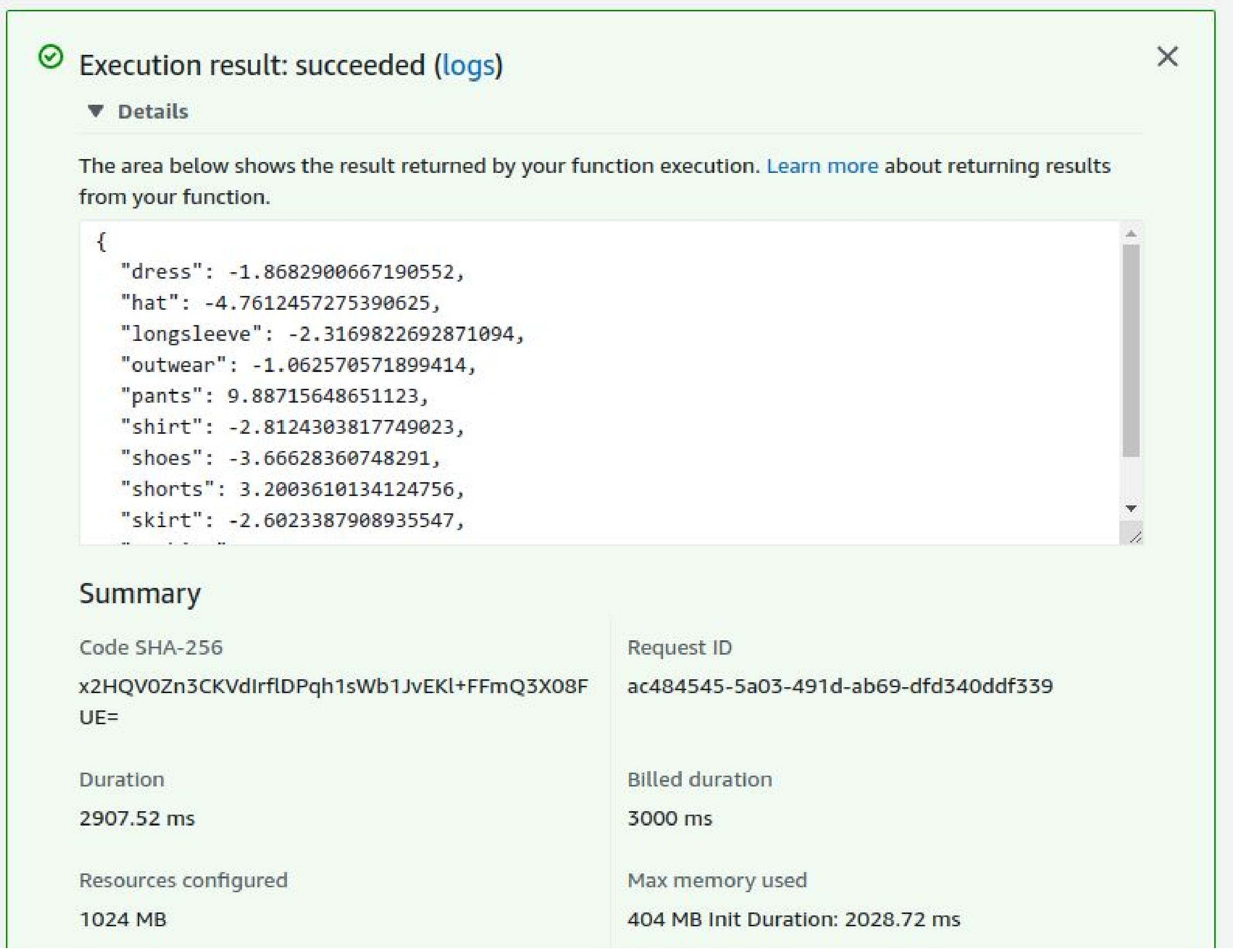
Event name

Pants

```
{
    "url": "https://raw.githubusercontent.com/alexeygrigorev/clothing-dataset-small/master/test/pants/4aa
}
```

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

You should see “Execution results: succeeded”:



Execution result: succeeded (logs)

▼ Details

The area below shows the result returned by your function execution. Learn more about returning results from your function.

```
{
  "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,
```

Summary

Code SHA-256

x2HQV0Zn3CKVdIrflDPqh1sWb1JvEKl+FFmQ3X08FUE=

Request ID

ac484545-5a03-491d-ab69-dfd340ddf339

Duration

2907.52 ms

Billed duration

3000 ms

Resources configured

1024 MB

Max memory used

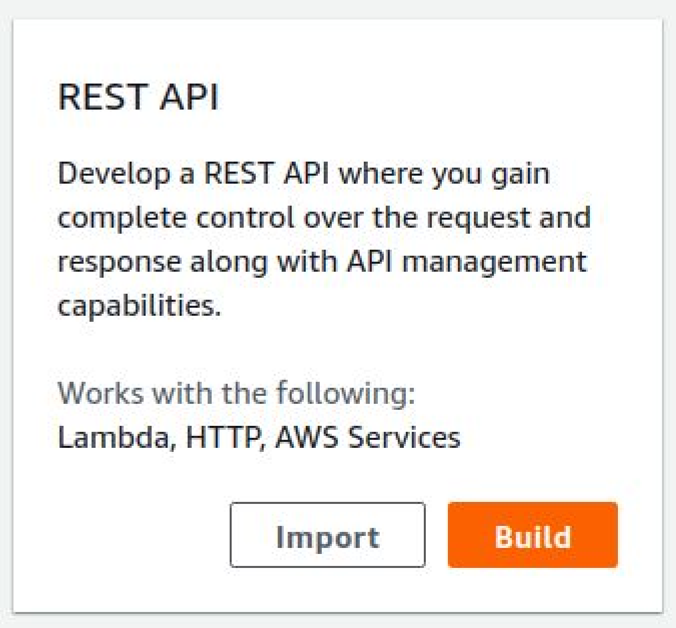
404 MB Init Duration: 2028.72 ms

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:

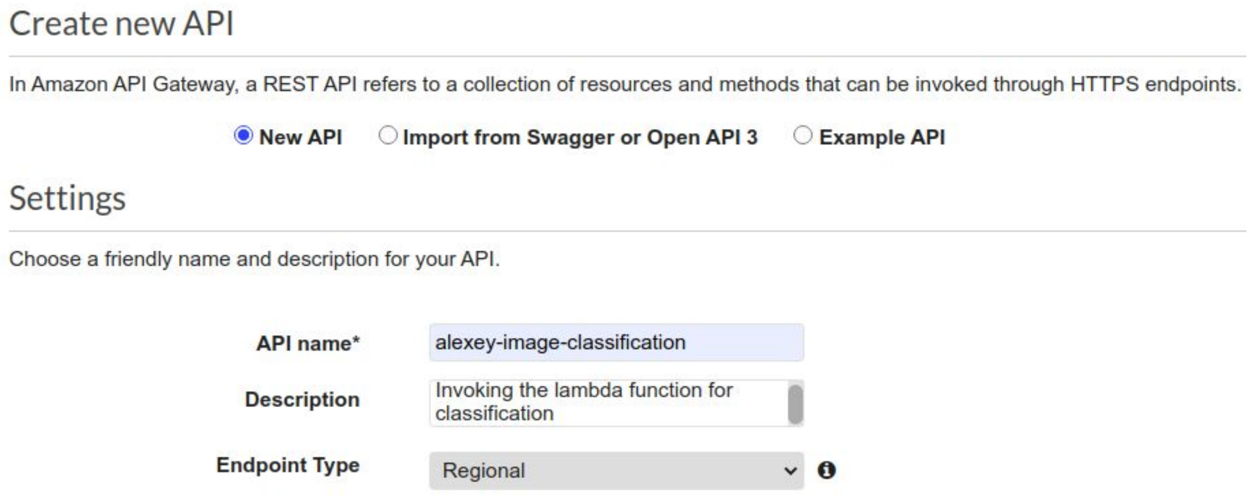


REST API

Develop a REST API where you gain complete control over the request and response along with API management capabilities.

Works with the following:
Lambda, HTTP, AWS Services

Call it “alexey-image-classification”



Create new API

In Amazon API Gateway, a REST API refers to a collection of resources and methods that can be invoked through HTTPS endpoints.

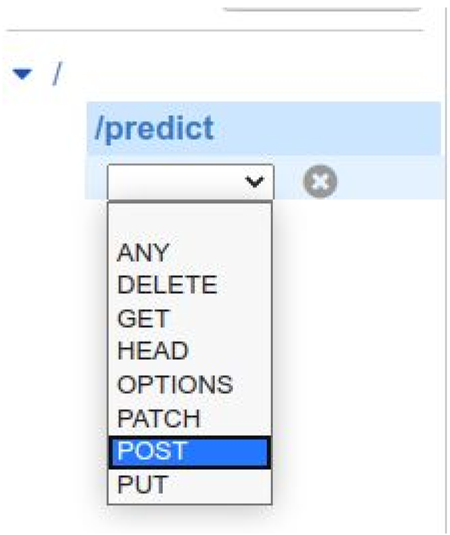
● New API ○ Import from Swagger or Open API 3 ○ Example API

Settings

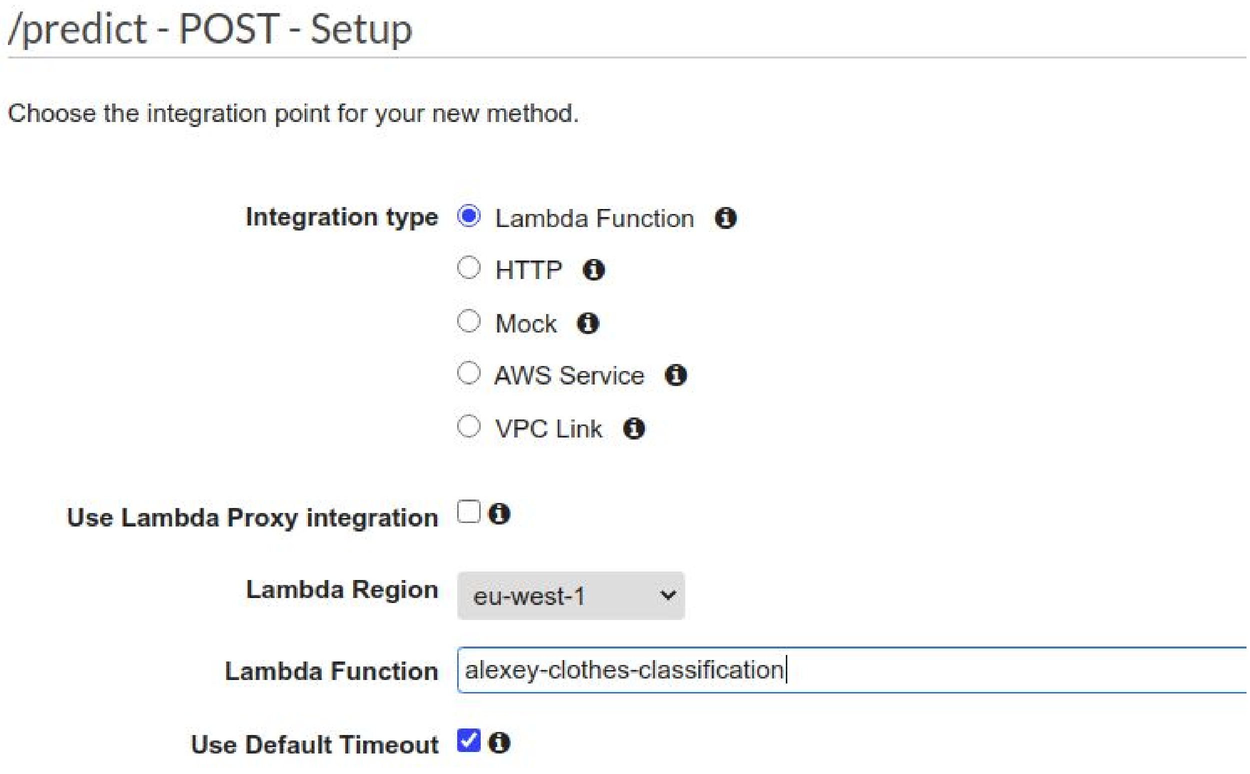
Choose a friendly name and description for your API.

API name*	alexey-image-classification
Description	Invoking the lambda function for classification
Endpoint Type	Regional

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



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



/predict - POST - Setup

Choose the integration point for your new method.

Integration type ● Lambda Function ○ HTTP ○ Mock ○ AWS Service ○ VPC Link

Use Lambda Proxy integration □

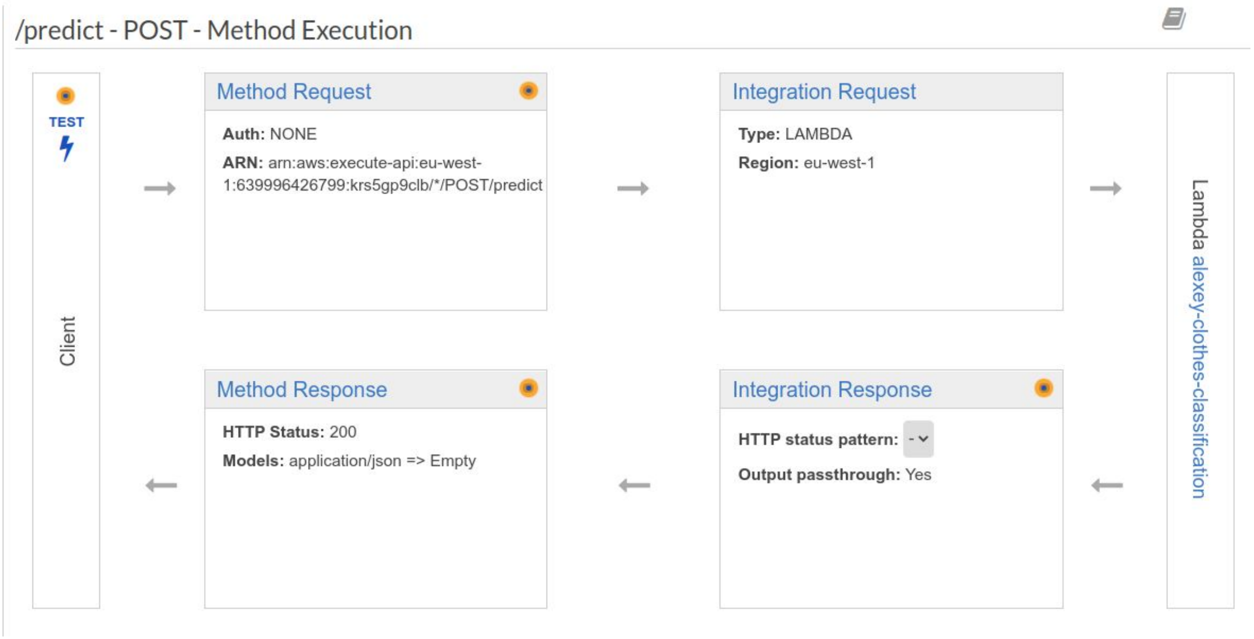
Lambda Region: eu-west-1

Lambda Function: alexey-clothes-classification

Use Default Timeout ☑

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

Now you should see the integration:



/predict - POST - Method Execution

Method Request

Auth: NONE

ARN: arn:aws:execute-api:eu-west-1:639996426799:krs5gp9clb/*/POST/predict

Integration Request

Type: LAMBDA

Region: eu-west-1

Method Response

HTTP Status: 200

Models: application/json => Empty

Integration Response

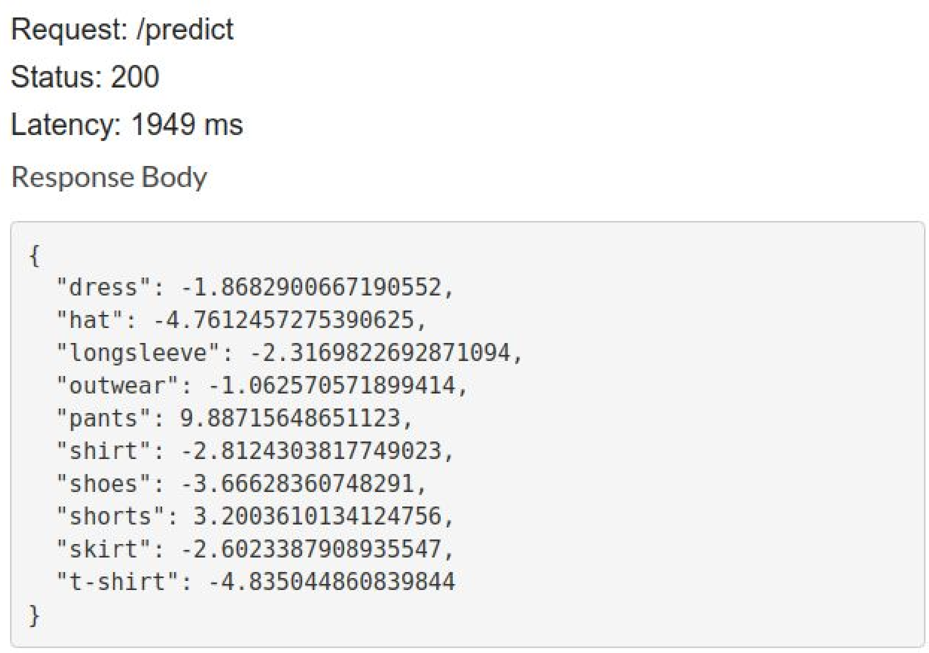
HTTP status pattern: -

Output passthrough: Yes

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

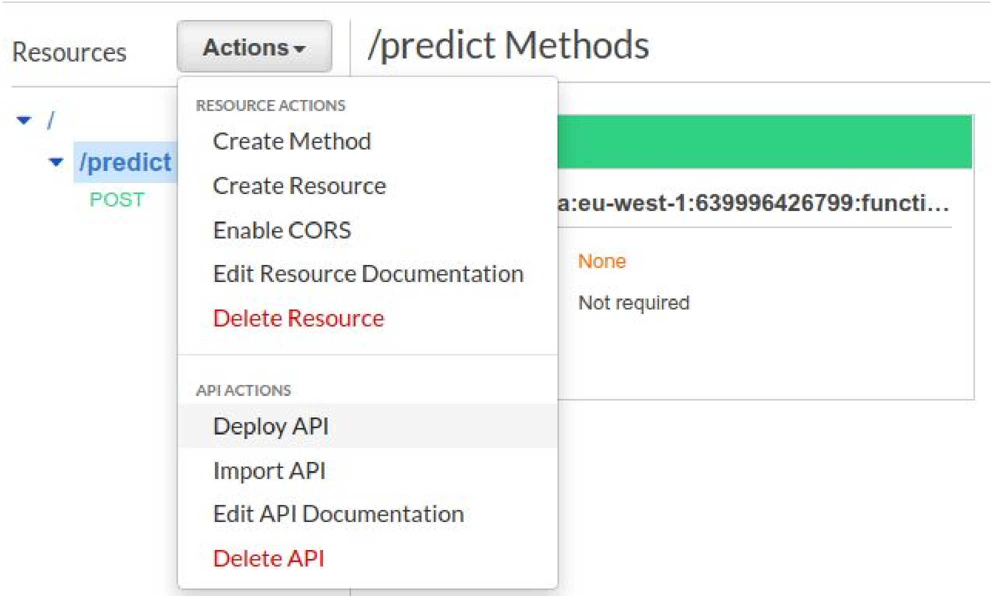
Status: 200

Latency: 1949 ms

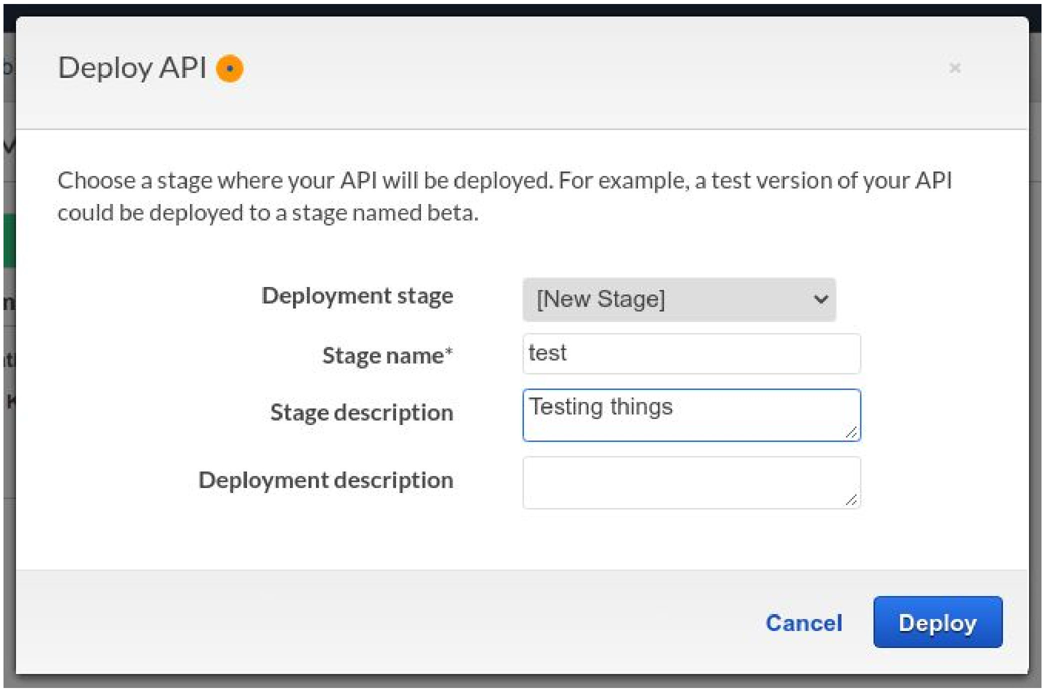
Response Body

```
{
  "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”:



Deploy API

Choose a stage where your API will be deployed. For example, a test version of your API could be deployed to a stage named beta.

Deployment stage: [New Stage]

Stage name*: test

Stage description: Testing things

Deployment description:

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/mast
er/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!