AWS Machine Learning Blog

Build Your Own Face Recognition Service Using Amazon Rekognition

by Christian Petters | on 14 AUG 2017 | in Amazon Rekognition | Permalink | Comments | Share

Amazon Rekognition is a service that makes it easy to add image analysis to your applications. It's based on the same proven, highly scalable, deep learning technology developed by Amazon's computer vision scientists to analyze billions of images daily for Amazon Prime Photos. Facial recognition enables you to find similar faces in a large collection of images.

In this post, I'll show you how to build your own face recognition service by combining the capabilities of Amazon Rekognition and other AWS services, like Amazon DynamoDB and AWS Lambda. This enables you to build a solution to create, maintain, and query your own collections of faces, be it for the automated detection of people within an image library, building access control, or any other use case you can think of.

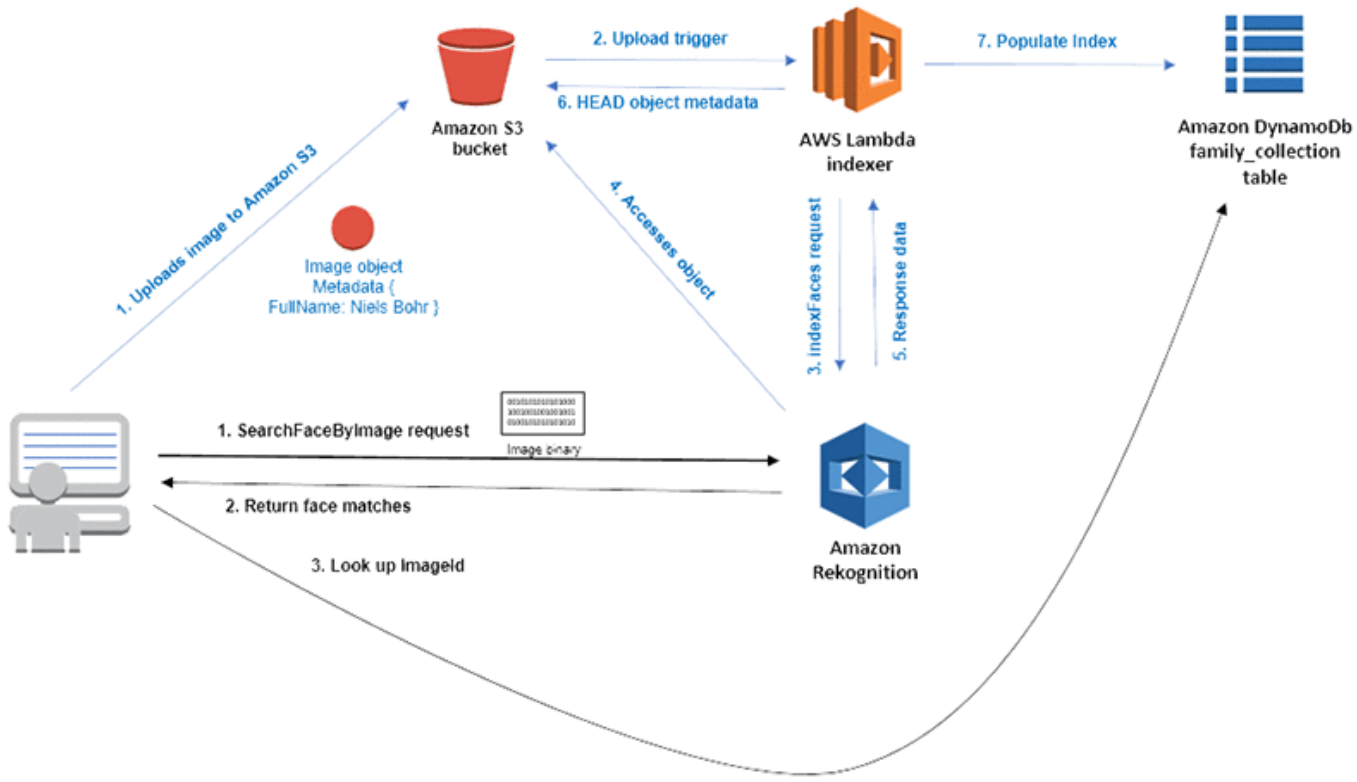
If you want to get started quickly, launch this Cloudformation template to get started now. For the manual walkthrough, please ensure that you replace resource names with your own values.

Launch Stack

How it works

The following figure shows the application workflow. It's separated into two main parts:

- **Indexing** (blue flow) is the process of importing images of faces into the collection for later analysis.
- **Analysis** (black flow) is the process of querying the collection of faces for matches within the index.



Implementation

Before we can start to index the faces of our existing images, we need to prepare a couple of resources.

We start by creating a collection within Amazon Rekognition. A collection is a container for persisting faces detected by the IndexFaces API. You might choose to create one container to store all faces or create multiple containers to store faces in groups.

Your use case will determine the indexing strategy for your collection, as follows:

- ***Face match.*** *You might want to find a match for a face within a collection of faces (as in our current example). Face match can support a variety of use cases. For example, whitelisting a group of people for a VIP experience, blacklisting to identify bad actors, or supporting logging scenarios. In those cases, you would create a single collection that contains a large number of faces or, in the case of the logging scenario, one collection for a certain time period, such as a day.*

- ***Face verification.*** *In cases where a person claims to be of a certain identity, and you are using face recognition to verify the identity (for example, for access control or authentication), you would actually create one collection per person. You would store a variety of face samples per person to improve the match rate. This also enables you to extend the recognition model with samples of different appearances, for example, where a person has grown a beard.*

- ***Social tagging.*** *In cases where you might like to automatically tag friends within a social network, you would employ one collection per application user.*

You can find more information about use cases and indexing strategies in the Amazon Rekognition Developer Guide*.*

Amazon Rekognition doesn't store copies of the analyzed images. Instead, it stores face feature vectors as the mathematic representation of a face within the collection. This is often referred to as a *thumbprint* or

faceprint.

You can manage collection containers through the API. Or if you installed and configured the AWS CLI, you can use the following command.

```
aws rekognition create-collection --collection-id family_collection --region eu-
```

The user or role that executes the commands must have permissions in AWS Identity and Access Management (IAM) to perform those actions. AWS provides a set of managed policies that help you get started quickly. For our example, you need to apply the following minimum managed policies to your user or role:

- AmazonRekognitionFullAccess
- AmazonDynamoDBFullAccess
- AmazonS3FullAccess
- IAMFullAccess

Be aware that we recommend you follow AWS IAM best practices for production implementations, which is out of scope for this blog post.

Next, we create an Amazon DynamoDB table. DynamoDB is a fully managed cloud database that supports both document and key-value store models. In our example, we'll create a DynamoDB table and use it as a simple key-value store to maintain a reference of the FaceId returned from Amazon Rekognition and the full name of the person.

You can use either the AWS Management Console, the API, or the AWS CLI to create the table. For the AWS CLI, use the following command, which is documented in the CLI Command Reference.

```
aws dynamodb create-table --table-name family_collection \
--attribute-definitions AttributeName=RekognitionId,AttributeType=S \
--key-schema AttributeName=RekognitionId,KeyType=HASH \
--provisioned-throughput ReadCapacityUnits=1,WriteCapacityUnits=1 \
--region eu-west-1
```

For the IndexFaces operation, you can provide the images as bytes or make them available to Amazon Rekognition inside an Amazon S3 bucket. In our example, we upload the images to an Amazon S3 bucket.

Again, you can create a bucket either from the AWS Management Console or from the AWS CLI. Use the following command, which is documented in the CLI Command Reference.

```
aws s3 mb s3://bucket-name --region eu-west-1
```

As shown earlier in the architecture diagram, we have separated the processes of image upload and face detection into two parts. Although all the preparation steps were performed from the AWS CLI, we use an

AWS Lambda function to process the images that we uploaded to Amazon S3.

For this, we need to create an IAM role that grants our function the rights to access the objects from Amazon S3, initiate the IndexFaces function of Amazon Rekognition, and create multiple entries within our Amazon DynamoDB key-value store for a mapping between the FaceId and the person's full name.

By now, you will have noticed that I do favor the AWS CLI over the use of the console. You can find detailed instructions for creating service roles using the AWS CLI in the documentation. To create the service role for Lambda, we need two JSON files that describe the trust and access policies: **trust-policy.json** and **access-policy.json**.

trust-policy.json

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "lambda.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

access-policy.json

```
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:PutItem"
            ],
            "Resource": [
                "arn:aws:dynamodb:aws-region:account-id:table/family_collectio
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "rekognition:IndexFaces"
            ],
            "Resource": "*"
        }
    ]
```

For the access policy, ensure you replace aws-region, account-id, and the actual name of the resources (e.g., bucket-name and family_collection) with the name of the resources in your environment.

Now we can use the AWS CLI again to create the service role that our indexing Lambda function will use to retrieve temporary credentials for authentication.

As described in the documentation, you first need to create the role that includes the trust policy.

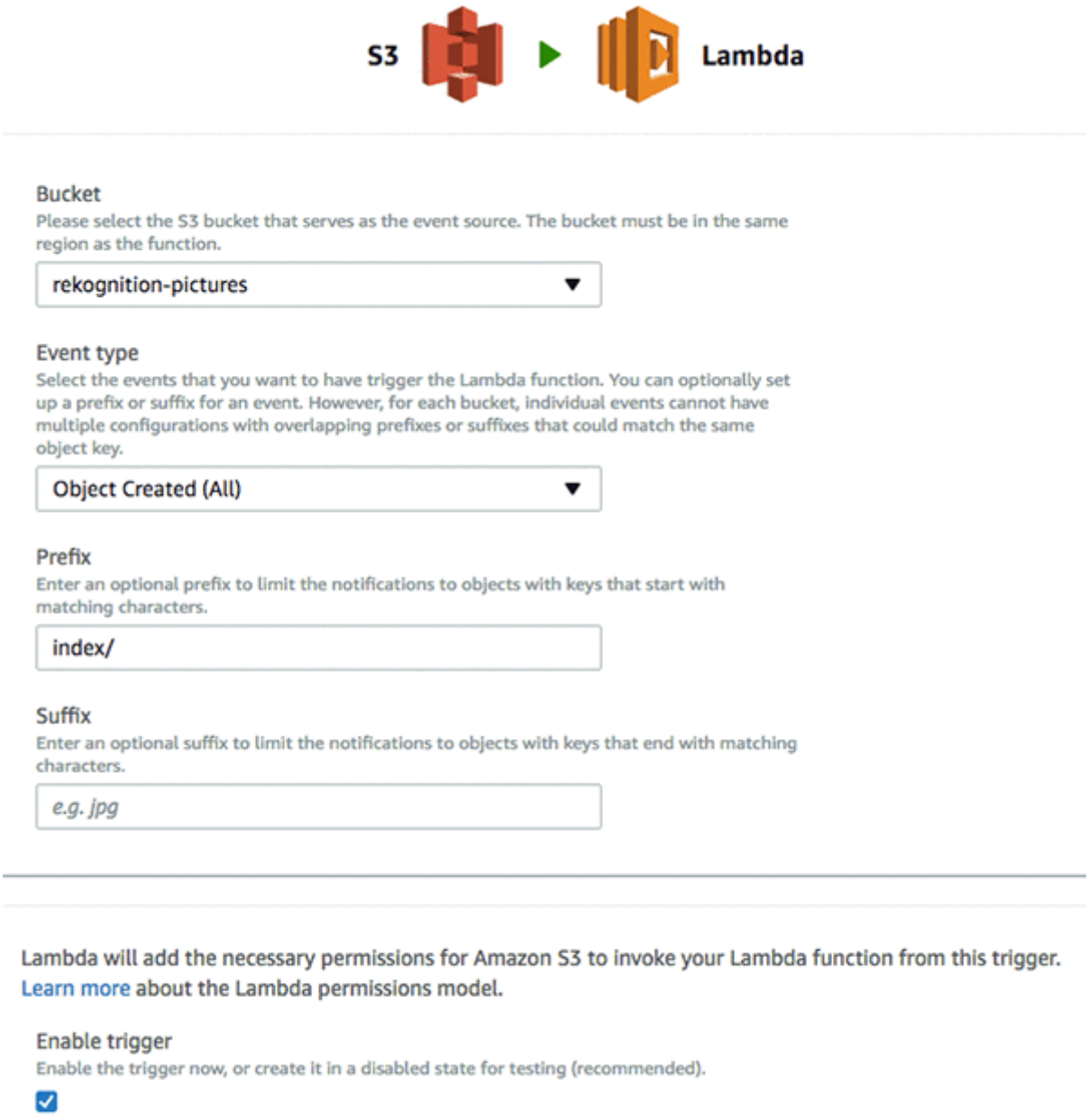
```
aws iam create-role --role-name LambdaRekognitionRole --assume-role-policy-docum
```

This is followed by attaching the actual access policy to the role.

```
aws iam put-role-policy --role-name LambdaRekognitionRole --policy-name LambdaPe
```

As a last step, we need to create the Lambda function that is triggered every time a new picture is uploaded to Amazon S3. To create the function using the **Author from scratch** option, follow the instructions in the AWS Lambda documentation.

On the configure triggers page, select S3, and the name of your bucket as the trigger. Then configure the **Event type** and **Prefix** as shown in the following example. This ensures that your Lambda function is triggered only when new objects that start with a key matching the **index/** pattern are created within the bucket.



On the next page, you give your Lambda function a name and description, choose the Python 2.7 runtime, and paste the Python code below into the editor.

In a nutshell, the script performs two main activities:

1. It uses the Amazon Rekognition IndexFaces API to detect the face in the input image and adds it to the specified collection.
2. If successful, it retrieves the full name of the person from the metadata of the object in Amazon S3. Then it stores this as a key-value tuple with the FaceId in the DynamoDB table for later reference.

```
from __future__ import print_function

import boto3
from decimal import Decimal
import json
import urllib
```

```
print('Loading function')

dynamodb = boto3.client('dynamodb')
s3 = boto3.client('s3')
rekognition = boto3.client('rekognition')


# --------------- Helper Functions ------------------

def index_faces(bucket, key):
```

Before you click **Next**, find the Lambda function handler and role section at the end of the page. In the **Role** field, select **Choose an existing role**, and then select the name of the **Role** we created earlier.

Now you can click **Next**, and then choose **Create function** in the summary page.

We can now upload our images to Amazon S3 to seed the face collection. For this example, we again use a small piece of Python code that iterates through a list of items that contain the file location and the name of the person within the image.

Look closely at line 16 in the following code example. Here we add additional metadata to the objects in Amazon S3. The Lambda function uses this metadata to extract the full name of the person within the image.

Depending on your needs, you can alter this process to include additional metadata or to load the metadata definition from another source, like a database or a metadata file.

```

s3 = boto3.resource('s3')

# Get list of objects for indexing
images=[('image01.jpeg','Albert Einstein'),
      ('image02.jpeg','Albert Einstein'),
      ('image03.jpeg','Albert Einstein'),
      ('image04.jpeg','Niels Bohr'),
      ('image05.jpeg','Niels Bohr'),
      ('image06.jpeg','Niels Bohr')
      ]

# Iterate through list to upload objects to S3
for image in images:
    file = open(image[0],'rb')
    object = s3.Object('rekognition-pictures','index/'+ image[0])
    ret = object.put(Body=file,
                     Metadata={'FullName':image[1]}
                     )
```

The reason I'm adding multiple references for a single person to the image collection is because adding multiple reference images per person greatly enhances the potential match rate for a person. It also provides additional matching logic to further enhance the results.

Analysis

Once the collection is populated, we can query it by passing in other images that contain faces. Using the SearchFacesByImage API, you need to provide at least two parameters: the name of the collection to query, and the reference to the image to analyze. You can provide a reference to the Amazon S3 bucket name and object key of the image, or provide the image itself as a bytestream.

In the following example, I show how to submit the image as a bytestream. In response, Amazon Rekognition returns a JSON object containing the FaceIds of the matches. This object includes a confidence score and the coordinates of the face within the image, among other metadata as documented.

```
response = rekognition.search_faces_by_image(
        CollectionId='family_collection',
        Image={'Bytes':image_binary}
        )

for match in response['FaceMatches']:
    print (match['Face']['FaceId'],match['Face']['Confidence'])

    face = dynamodb.get_item(
        TableName='family_collection',
        Key={'RekognitionId': {'S': match['Face']['FaceId']}}
        )

    if 'Item' in face:
        print (face['Item']['FullName']['S'])
    else:
        print ('no match found in person lookup')
```

We could extend this further by providing a secondary match logic.

For example, if we have received three matches for Person A and a confidence score of more than 90 percent each, and one match for Person B with a confidence of 85 percent or below, we can reasonably assume that the person was indeed Person A. Based on your business requirements, you could also return "fuzzy" matches like this to a human process step for validation.

Enhancements

For the analysis part of the process, you need to understand that Amazon Rekognition tries to find a match for only the most prominent face within an image. If your image contains multiple people, you first need to

use the DetectFaces API to retrieve the individual bounding boxes of the faces within the image. You can then use the retrieved x,y coordinates to cut out the faces from the image and submit them individually to the SearchFacesByImage API.

In the following code example, notice that I'm extending the boundaries of the face boxes by moving them 10 percent in either direction. I do this to simplify the definition of the box to crop. Ideally, I would have to adjust the location and orientation of the box to reflect any tilting of the head. Instead, I simply extend the size of the crop-out. This works because Amazon Rekognition tries to detect a person for only the largest face within an image. Slightly extending the size of the crop-out by up to 50 percent won't unveil another face that is larger than the original detected face.

```
            CollectionId='family_collection',
            Image={'Bytes':image_crop_binary}
            )

    if len(response['FaceMatches']) > 0:
        # Return results
        print ('Coordinates ', box)
        for match in response['FaceMatches']:

            face = dynamodb.get_item(
                TableName='family_collection',
                Key={'RekognitionId': {'S': match['Face']['FaceId']}}
                )

            if 'Item' in face:
                person = face['Item']['FullName']['S']
            else:
                person = 'no match found'
```

In addition to the manual approach I described above, you can also create a Lambda function that contains the face detection code. Instead of merely displaying the detected faces on the console, you would write the names of the persons that are detected to a database, like DynamoDB. This allows you to detect faces within a large collection of images at scale.

Conclusion

In this blog post, I provided you with an example you can use to design and build your own face recognition service. I have given you pointers that enable you to decide on your strategy for using collections, depending on your use case. I also provided guidance on how to integrate Amazon Rekognition with other AWS services such as AWS Lambda, Amazon S3, Amazon DynamoDB, or IAM. With this in hand, you can build your own solution that detects, indexes, and recognizes faces, whether that's from a collection of family photos, a large image archive, or a simple access control use case.

Additional Reading

Extend you knowledge even further. Learn how to find distinct people in a video with Amazon Rekognition.

About the Author

Christian Petters is a Solutions Architect for Amazon Web Service in Germany. He has a background in the design, implementation and operation of large scale web and groupware applications. At AWS he is helping our customers to assemble the right building blocks to address their business challenges.

TAGS: Amazon Rekognition

AWS Podcast

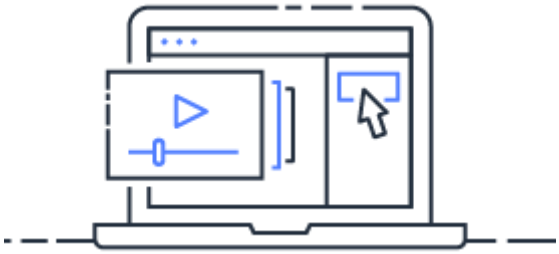
Subscribe for weekly AWS news and interviews

Learn more »

AWS Partner Network

Find an APN member to support your cloud business needs

Learn more »



AWS Training & Certifications

Free digital courses to help you develop your skills

Learn more »

Resources

Getting Started
What's New
Top Posts
Official AWS Podcast
AWS Case Studies

Follow

Twitter
Facebook
LinkedIn
Twitch
RSS Feed
Email Updates

AWS Events
Discover the latest AWS events in your region
Learn more »

Related Posts

Creating a searchable enterprise document repository

Learn More: M&E Chalk Talk Presentations from AWS re:Invent 2019

Managing Hybrid Video Processing Workflows with Apache Airflow

AWS re:Invent 2019: Key announcements for the Advertising and Marketing Industry

Amplify CLI announces new GraphQL transform feature for orchestrating multiple AI/ML use cases

Announcing Amazon Rekognition Custom Labels

Exploring images on social media using Amazon Rekognition and Amazon Athena

Adding AI to your applications with ready-to-use models from AWS Marketplace