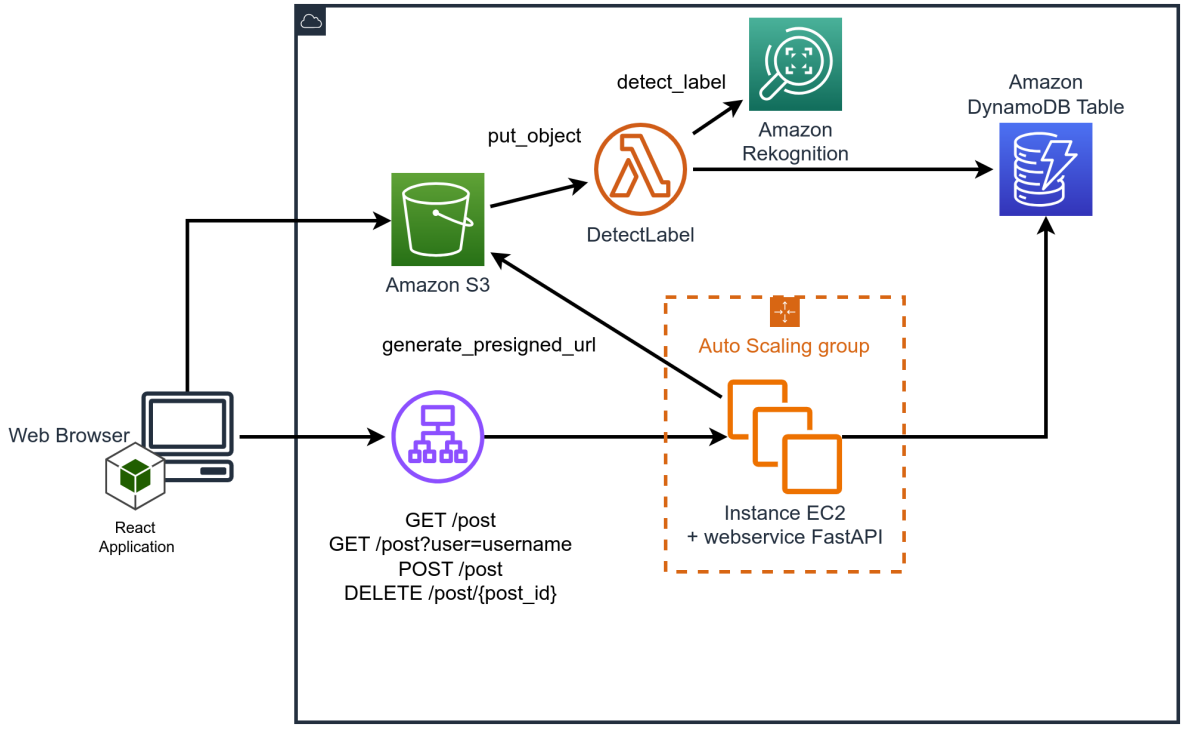
Devoir noté 2 : un web service serverless

Ce sujet peut paraître imposant et impossible à terminer mais ce n'est pas le cas. Il ne contient que peu de code à écrire (max 100 lignes de python, et une infrastructure déjà vue). Résultat avancez pas à pas, il n'est pas demandé de le rendre à la fin du TP. Le rendu final est attendu pour le 19 mai.

📃Sujet

Dans ce devoir noté vous allez concevoir la partie backend du nouveau réseau social de partage de photo, **Postagram**. Voici le diagramme d'architecture de l'application.



Le service sera hébergé par une flotte d'instance **EC2** (1 par défaut, 4 en cas de grosse charge) géré par un **Auto Scaling Group** derrière un Load Balancer. Il sera écrit en python en utilisant la librairie FastApi et devra répondre à plusieurs endpoints :

- `POST /posts` pour créer une publication
- `DELETE /posts/{id}` pour supprimer une publication
- `GET /posts` pour lire toutes les publications stockées en base
- `GET /posts?user=username` pour lire toutes les publications d'un utilisateur
- `GET /getSignedUrlPut` qui va retourner une url signée pour pouvoir uploader une image (déjà fourni)

Vous utiliserez en plus :

- Un **bucket S3** pour stocker des images
- Une base **DynamoDB** pour stocker les post
- Une **fonction lambda** qui se déclenchera à chaque dépôt de fichier dans le bucket S3 et appellera le service **Amazon Rekognition** pour détecter les labels et les stocker dans la base DynamoDB

📦Attendu

Votre but n'est pas de réaliser l'intégralité de l'application, mais seulement la partie création d'une publication et détection des labels via le service **Amazon Rekognition**, récupération des publications et suppression des publications. Le reste vous sera déjà donnée. Ainsi vous avez à votre disposition:

- Un application web écrite en Reactjs (le dossier webapp) qui va communiquer avec votre application. Aucune connaissance en Reactjs n'est attendue, ce code est uniquement là pour requêter votre webservice. Vous avez néanmoins à modifier la ligne 12 du fichier index.js pour mettre à la place l'adresse de votre Load Balancer quand vous testerez votre code votre code sur AWS. Attention l'url ne doit pas avoir de / à la fin !! Pour lancer cette application placez vous dans le dossier webapp et faite **npm install** la première fois puis un npm start ensuite.
- Une base de webservice avec tous les endpoints de définis. Vous allez devoir définir les fonctions vides. Ce web service contient la fonction qui permet de générer une URL présignée pour S3. Ne la touchez pas ! Pour lancer le webservice faite un `python3 app.py`. Le webservice se lance sur le port 8080.
- Une base de projet Terraform à compléter

Le code est à récupérer avec un `git clone https://github.com/HealerMikado/postagram_ensai.git`

Cet exercice est à faire par groupe de 3 max. Vous pouvez ainsi le faire seul à deux ou à trois. Vous noterez les membres du groupe dans un fichier `groupe.md` et ce même si vous êtes seul ! Vous rendrez une Moodle une archive .zip contenant tout le code du projet (sciprts terraform et le code webservice). A la différence de l'exercice précédent, votre code ne peux pas fonctionner tel quel. Il n'est pas possible d'injecter dans les instances EC2 le nom du bucket et de la table dynamoDB que vous aller créer. Vous aller ainsi devoir réaliser 2 scripts terraform :

- Le premier avec l'architecture *serverless* : bucket S3, lambda et DynamoDB qui devra avoir 2 terraform output avec le bucket S3 généré et la table dynamoDB
- Le second avec l'architecture avec serveur : EC2, auto scalling group et load balancer. Vous mettrez à jours les variables `bucket` et `dynamo_table` avec les variables qui proviendront du premier script. Ces variables seront injectées comme variable d'environnement dans les instances EC2 pour être accessible avec `os.getenv()`. Pendant la phase de développement sur votre machine, mettez à jour le fichier `.env`

Pour exécuter un fichier spécifique faites `cdktf deploy -a python mon_script.py`

Si vous faites ce projet en groupe, je vous encourage à rapidement mettre en place un dépôt git et à travailler en parallèle.

Voici le macro barème qui sera appliqué si vous êtes 3 :

- Vous avez tout qui fonctionne correctement : 20
- Vous avez les fonctionnalités de base qui fonctionnent sans la génération url signées pour l'affichage des images et la détection des labels : 16
- Vous avez la possibilité de poster et afficher des publications : 14
- Vous avez seulement la partir création de publication : 10

Je pars du principe que le code python est propre à chaque fois et que le template Terraform fonctionne. Je n'attends pas des commentaires, mais un code lisible.

Si vous faites ce travail seul ou à deux cela sera pris en compte. Considérez que si vous êtes seul, faire la fonctionnalité de création de publications et leur récupération vaudra un 20. Si vous êtes à deux le 20 il faut ajouter la partie détection des labels.

✨Aides

Ce projet contient des choses que vous avez déjà vu, ainsi que des choses nouvelles. Voici pour vous aider de nombreux exemples de code. N'hésitez pas retourner dans le cours ou aller sur internet pour vous aider. Bien entendu ce ne sont que des aides, et pas la solution à l'exercice.

💣Comment attaquer le problème

Vous avez trois choses à faire :

1. Un webservice python qui communique avec divers services AWS
2. Une lambda qui se déclenche quand un fichier est déposé sur S3
3. Le code de l'infra à déployer

Je vous conseille de ce travail dans cet ordre :

1. Créer le code terraform pour créer le bucket s3 et la base dynamoDB
2. Faites la partie ok sur la création de post et leur récupération
3. Mettez à jour le code Terraform pour ajouter une lambda qui se déclenche quand un fichier est déposé dans le bucket
4. Implémentez la lambda via l'interface graphique d'AWS
5. Récupérer le code et mettez votre script Terraform à jour
6. Finissez le code Terraform pour déployer un webservice python comme dans le TP 2

Faire fonctionner votre code avec l'IMH peut s'avérer complexe car l'IHM attend les données dans le format des posts données ci-dessous. Si vous n'arrivez pas à faire fonctionner votre code avec l'interface, **ce n'est pas grave** ! Faite le fonctionner avec Insomnia, Postman ou des requêtes http en python via `request`.

💬 Les posts

La donnée au coeur de votre application est un post. Un post pourra avoir les données suivantes :

- Un `id` unique (obligatoire). Cet id sera généré par l'application quand elle recevra un post. Vous aller utiliser la bibliothèque `uuid` pour ça.

  ```
  import uuid

  str_id = f'{uuid.uuid4()}'
  ```

- Un titre donnée par l'utilisateur (obligatoire)
- Un contenu donnée par l'utilisateur (obligatoire)
- Un auteur (obligatoire)
- S'il y a une image associé son nom et ses labels

📚Base Dynamodb

Votre base Dynamodb aura comme clé de partition les utilisateurs, et comme clé de tri l'id des posts. Pour éviter tout chevauchement entre les concepts, je valoriserai les groupes qui préfixent ses deux attributs comme dans le TP 4. Exemple post = `POST#....`, PK= `USER#....`

Pour les attributs des objets que vous allez stocker, je vous conseiller de reprendre ceux du json des posts.

Voici un rappel des méthodes qui vous seront utiles :

```
dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table(table_name)

data = table.put_item(
    Item={...})

data = table.delete_item(
    Key={'name' : 'jon',
    "lastname" : 'doe'}
)

table.update_item(
Key={
    "name": "jon",
    "lastname": "doe"
},
AttributeUpdates={
    "tel": {
        "Value": "12345678",
        "Action": "PUT"
    }
},
ReturnValues='UPDATED_NEW'
)
data = table.scan()

data = table.query(
    KeyConditionExpression=Key('user').eq('jon' & Key("lastname".eq("doe")))
)
```

✅Les retours des fonctions

Quand vous retourner des posts, pour que l'interface fonctionne correctement, un post doit être un json avec les attributs suivants :

```
{
    'user' : string,
    'title' : string,
    'body' : string,
    'image' : string //contient l'url de l'image,
    'label' : liste de string
}
```

L'id, le titre et le body ne demandent pas d'explication. Image est une url vers l'image S3. Comme votre bucket sera privé, pour récupérer une image il vous faut une url présignée. Voici un exemple de code :

```
def create_presigned_url(bucket_name, object_name, expiration=3600):
    """Generate a presigned URL to share an S3 object

    :param bucket_name: string
    :param object_name: string
    :param expiration: Time in seconds for the presigned URL to remain valid
    :return: Presigned URL as string. If error, returns None.
    """

    # Generate a presigned URL for the S3 object
    s3_client = boto3.client('s3')
    try:
        response = s3_client.generate_presigned_url('get_object',
                                                    Params={'Bucket':
bucket_name,
                                                            'Key':
object_name},
                                                    ExpiresIn=expiration)
    except ClientError as e:
        logging.error(e)
        return None

    # The response contains the presigned URL
    return response
```

Vous devrez générer vous même cette url pour que l'image s'affiche.

Les labels ne seront récupéré qu'après la labélisation de l'image

🤖Récupérer le username

Pour simuler une vraie application, le nom de l'utilisateur sera, sauf mention contraire, récupéré dans le header de la requête. En effet, les utilisateurs authentifiés envoient à chaque requête un jeton avec diverses informations, dont leur username. Dans notre cas pour simplifier, ce n'est pas un jeton qui va être envoyé, mais simplement le username dans un header de la requête.

Le user name est le contenu de la variable `authorization`.

Cela sera utile pour :

- poster une publication
- supprimer une publication

📮 Poster une publication

Voici le corps de la requête pour poster une publication que vous enverra l'application Reactjs :

```
{
    'title' : string,
    'body' : string,
}
```

Il vous faut mettre cela en base en calculant un id pour la publication. Le username est à récupérer dans le header comme dit ci-dessus.

Pour le retour attendu :

```
data = table.put_item(...)
return data
```

🧺 Bucket S3 et détection des labels

Dans le template fourni, un bucket S3 est déjà défini. La détection des labels sera exécutée dés qu'un objet sera uploadé sur S3. Pour faire cela, la lambda ne doit pas être déclenchée par une file SQS comme dans le travail précédent, mais par la création d'un objet sur S3. Voici un exemple pour vous aider :

```
from cdktf_cdktf_provider_aws.s3_bucket_notification import
S3BucketNotification, S3BucketNotificationLambdaFunction
from cdktf_cdktf_provider_aws.lambda_permission import LambdaPermission

bucket = S3Bucket(
    self, "bucket"
)
lambda_function = LambdaFunction(self,"lambda")
permission = LambdaPermission(
    self, "lambda_permission",
    action="lambda:InvokeFunction",
    statement_id="AllowExecutionFromS3Bucket",
    function_name=lambda_function.arn,
    principal="s3.amazonaws.com",
    source_arn=bucket.arn,
    source_account=account_id
)
notification = S3BucketNotification(
    self, "notification",
    lambda_function=[S3BucketNotificationLambdaFunction(
        lambda_function_arn=lambda_function.arn,
        events=["s3:ObjectCreated:*"]
    )],
    bucket=bucket.id
)
```

Le code qui gère l'uploade des fichiers les met dans votre bucket à l'adresse : `user/id_publication/image_name`. Comme votre fonction est déclenchée par l'ajout dans objet dans un bucket, l'event va être différent de celui d'une file SQS.

```
{
  "Records": [
    {
      "eventVersion": "2.0",
      "eventSource": "aws:s3",
      "awsRegion": "us-east-1",
      "eventTime": "1970-01-01T00:00:00.000Z",
      "eventName": "ObjectCreated:Put",
      "userIdentity": {
        "principalId": "EXAMPLE"
      },
      "requestParameters": {
        "sourceIPAddress": "127.0.0.1"
      },
      "responseElements": {
        "x-amz-request-id": "EXAMPLE123456789",
        "x-amz-id-2":
"EXAMPLE123/5678abcdefghijklambdaisawesome/mnopqrstuvwxyzABCDEFGH"
      },
      "s3": {
        "s3SchemaVersion": "1.0",
        "configurationId": "testConfigRule",
        "bucket": {
          "name": "example-bucket",
          "ownerIdentity": {
            "principalId": "EXAMPLE"
          },
          "arn": "arn:aws:s3:::example-bucket"
        },
        "object": {
          "key": "test%2Fkey",
          "size": 1024,
          "eTag": "0123456789abcdef0123456789abcdef",
          "sequencer": "0A1B2C3D4E5F678901"
        }
      }
    }
  ]
}
```


La partie qui vous intéresse est la clef `s3`

```
import boto3
import os
import logging
import json
from datetime import datetime
from urllib.parse import unquote_plus

table_name = os.getenv("TASKS_TABLE")
dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table(table_name)
reckognition = boto3.client('rekognition')
logger = logging.getLogger()
```

```
logger.setLevel(logging.INFO)

def lambda_handler(event, context):
    # Pour logger
    logger.info(json.dumps(event, indent=2))
    # Récupération du nom du bucket
    bucket = event["Records"][0]["s3"]["bucket"]["name"]
    # Récupération du nom de l'objet
    key = unquote_plus(event["Records"][0]["s3"]["object"]["key"])
    # extration de l'utilisateur et de l'id de la tâche
    user, task_id = key.split('/')[:2]
```

Il vous faut maintenant appeler le service Rekognition pour obtenir les labels de l'image. Voici un exemple de code et la documentation

```
# Appel au service, en passant l'image à analyser (bucket et key)
# On souhaite au maximum 5 labels et uniquement les labels avec un taux de
confiance > 0.75
# Vous pouvez faire varier ces valeurs.
label_data = reckognition.detect_labels(
        Image={
            "S3Object": {
                "Bucket": bucket,
                "Name": key
            }
        },
        MaxLabels=5,
        MinConfidence=0.75
    )
logger.info(f"Labels data : {label_data}")
# On extrait les labels du résultat
labels = [label["Name"] for label in label_data["Labels"]]
logger.info(f"Labels detected : {labels}")
```

Puis il vous faut enfin insérer les labels et la localisation de l'image dans s3 dans la base Dynamodb avec la méthode `update_item`. Vous stockerez en base la contenu de la variable `key` qui est en quelque sort le chemin de l'image dans votre bucket.

📑Récupérer des posts

Cette fonctionnalité demande simplement d'aller récupérer les données stockées dans la base. Attention néanmoins car un *query parameter* peut être utilisé pour récupérer les posts d'un utilisateur en particulier. On peut soit appeler `base_url/posts` ou `base_url/posts?id_user=XXX`

```
@app.get("/posts")
async def get_all_posts(id_user: Union[str, None] = None):
```

Votre code va devoir gérer les deux cas. Je vous conseil pour une meilleur lisibilité de faire deux méthodes qui seront appelées dans **le endpoint du webservice** en fonction du cas dans lequel vous vous trouvez. L'application web attend une réponse qui contient une liste de post, chaque poste ayant les informations suivantes :

```
{
    'user' : string,
    'title' : string,
    'body' : string,
    'image' : string //contient l'url de l'image,
    'label' : liste de string
}
```

Pour obtenir l'url de l'image vous allez utiliser une url dite signée. En effet comme votre bucket est privé, pour rendre accessible les images, il faut créer une url spéciale. Voici le code pour générer cette url :

```
bucket = os.getenv("S3_BUCKET")
s3_client = boto3.client('s3')

url = s3_client.generate_presigned_url(
    Params={
    "Bucket": bucket,
    "Key": object_name,
},
    ClientMethod='get_object'
)
```

Pour une publication donnée vous trouverez l'image dans le bucket de votre application au chemin suivant : `user/id_publication/image_name`

❌Supprimer des posts

Enfin la suppression des posts va appeler la méthode `delete_item` présente dans le CM4. Cette fois-ci on va utiliser un *path parameter*.

```
@app.delete("/posts/{post_id}")
async def get_post_user_id(post_id: int):
    # Doit retourner le résultat de la requête la table dynamodb
    return []
```

La subtilité de la suppression est que vous allez devoir supprimer la publication de la base de donnée, mais aussi supprimer son image dans le bucket S3 si elle en a une.

Pour le retour attendu :

```
return table.delete_item()
```

💻 Auto scaling group, Load Balancer, instances EC2, webservice

Cette architecture est gérée par le fichier `main-server.py`. Le script contient déjà de quoi créer les users data pour installer le code du webservice sur une instance EC2. Vous devez remplir la configuration des différents éléments (globalement la même que dans le TP2). Les différences :

- Une seule machine de base au lieu de 2

- Le webservice écoutera sur le port 8080 et pas 80 comme dans le TP. Vous allez devoir changer le port de du `LbTargetGroup`
- Pour que vos instances EC2 aient le droit d'interagir avec S3 et DynamoDB il faut leur donner les droits. Cela passe par l'attribut iam_instance_profile de la classe `LaunchTemplate`. Cet attributs attend un dictionnaire en paramètre `{"arn" : "arn_du_role"}`. Le rôle que vous passerez sera le même rôle que pour la lambda.