TP4 - Stocker des données dans le Cloud 

 Mise en place

Allez sur la plateforme AWS academy et accédez au cours AWS Academy Learner Lab [43226]. Puis cliquez sur `Modules` > `Learner Lab`. Lancez votre environnement en cliquant sur `Start Lab`. Une fois le cercle passé au vert, cliquer sur `AWS Details` et `AWS CLI`. Les clefs que vous voyez vont permettre un accès programmatique à votre compte. Chercher le dossier `.aws` sur votre machine puis remplacez le contenu du fichier `credentials` par les clefs que vous venez de récupérer

 Manipulation de S3

Création d'un bucket S3 

Sur la console AWS cherchez le service `S3`. Normalement vous ne devez pas avoir de bucket associé à votre compte. En utilisant le CDK de Terraform créez un bucket. La classe à utiliser est la classe `S3Bucket`. Voici un petit bout de code pour vous aider. Attention ce code ne fonctionne pas ! Il doit être mis dans une classe qui hérite de `TerraformStack` comme dans les TP précédents.

```
from cdktf_cdktf_provider_aws.s3_bucket import S3Bucket

bucket = S3Bucket(
    self, "s3_bucket",
    bucket_prefix = "my-cdtf-test-bucket",
    acl="private",
    force_destroy=True
)
```

Déployez votre architecture et regardez si votre bucket est bien crée.

Manipulation d'objets 

Maintenant vous allez ajouter des objets dans votre bucket. Vous trouverez sur Moodle différents fichiers à uploader, mais vous pouvez utiliser les votre si vous le souhaitez. Une fois vos fichiers poussés, essayez de les récupérer, les mettre à jouer et les supprimer. Voici des exemples de code pour vous aider :

```python
import boto3
# Create an S3 resource
s3 = boto3.resource('s3')
# Create a bucket
s3.create_bucket(Bucket='mybucket')
# Upload file
with open('/tmp/hello.txt', 'rb') as file:
    s3.Object('mybucket', 'hello_s3.txt').put(Body=file)
# Download file
s3.Bucket('mybucket').download_file('hello_s3.txt', 'hello.txt')
# Delete file
s3 = boto3.resource('s3')
s3.Object('mybucket', 'hello_s3.txt').delete()
```

Ajout du versionnage 🔴🟠🟡

Un bucket S3 peut versionner ses objets et ainsi conserver les différentes versions d'un même fichier. Cette fonctionnalité est utile pour ne pas perdre des données, mais elle va augmenter les coûts, car toutes les versions vont compter dans le volume facturé. Activez le versionnage en ajoutant un attribut `versionning` valant `True` à l'objet S3Bucket. Redéployez votre infrastructure.

Maintenant, avec votre code python, uploadez un fichier qui aura le même nom qu'un objet déjà présent dans votre bucket. Aller sur la console AWS, cherchez le service `S3`, cliquez sur votre bucket puis sur l'objet réuploadé. Dans l'onglet version vous devrez voir les différentes versions de votre objet.

> 🤷Une fois que l'option des versionnage est activée sur un bucket S3 elle ne peut plus être désactivée, mais seulement suspendue. Cela veut dire que les nouveaux objets ne seront pas versionnés, mais que les anciens garderont leurs versions.

Manipulation de DynamoDB

> Cet exercice s'inspire du workshop DynamoDB d'AWS : https://amazon-dynamodb-labs.com/game-player-data.html

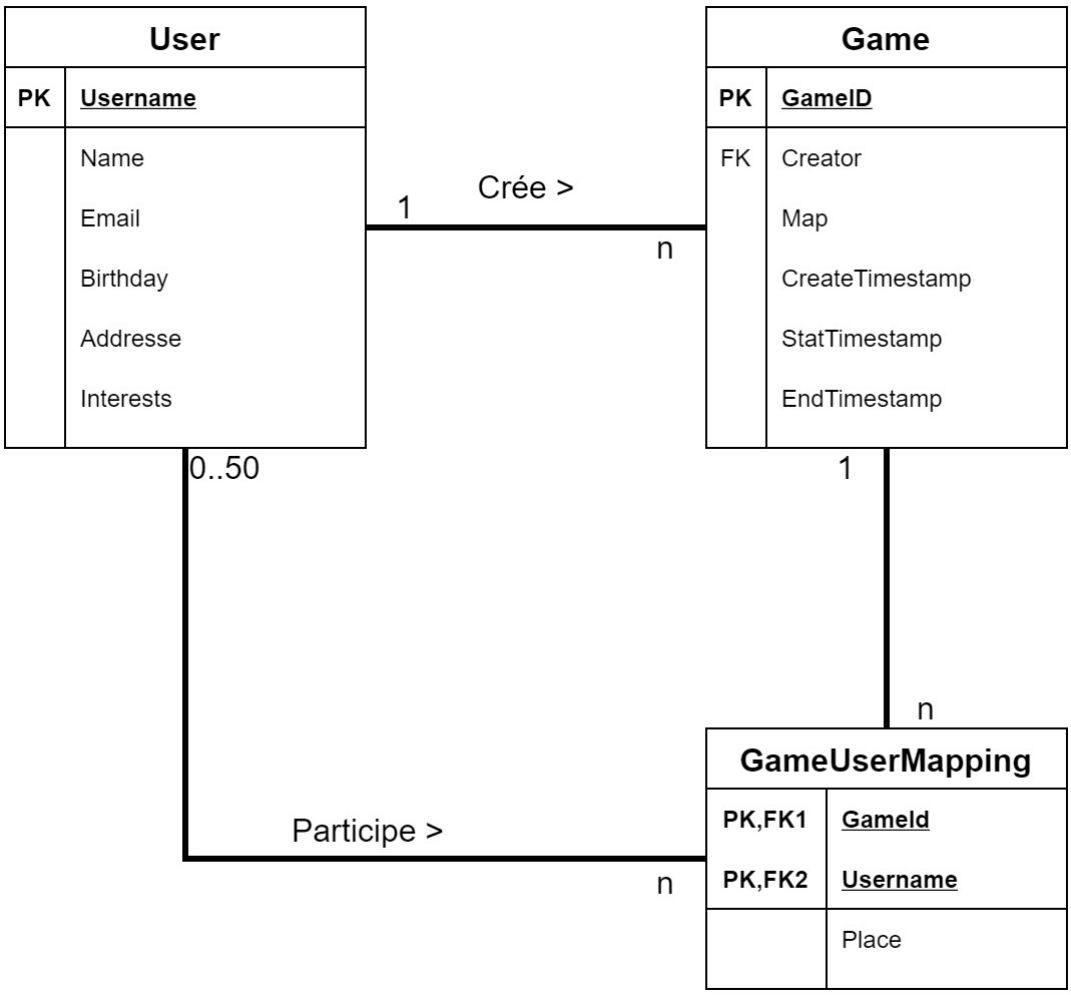
Cette partie du TP consiste à mettre en place une base de données pour stocker des données d'un jeu de type *battle royal*. Chaque partie regroupe 50 joueurs qui s'affrontent pour une trentaine de minutes. Notre base devra stocker en temps réel le temps joué par chaque joueur, leur score, et quel joueur l'a emporté. Chaque joueur devra pouvoir accéder à ses parties passées, et la revisionner.

Modèle de données 🧩

Conceptuellement, notre jeu va mobiliser 2 concepts :

- Les joueurs (User)
- Les parties (Game)

Et une table pour associer les deux. Un joueur va pouvoir créer une partie (il en devient le `Creator`), mais il peut rejoindre une partie et crée une ligne dans la table `GameUserMapping`.



DynamoDB est une base de données No-SQL et ne dispose pas de moteur de jointure, et ne peux pas faire d'aggrégation type GROUP BY. Elle offre par contre de excellentes performances quelque soit la volumétrie. Choisir une base No-SQL contre une base SQL est un choix qui va considérablement changer les outils à disposition.

Dans notre cas, avec 3 entités, faire plusieurs tables n'a pas d'intérêt. À la place nous allons faire une seule grosse table dans cette exercice qui va contenir toutes les données. Néanmoins il nous faut dans notre table pouvoir identifier de manière unique les différentes informations de notre base à savoir `User`, `Game` et `GameUserMapping`. Un User est identifié de manière unique par son `USERNAME`, une Game pas son `GAME_ID` et une ligne de `GameUserMapping` par le couple `GAME_ID` et `USERNAME`.

Une table DynamoDB est définie par aa clé primaire, qui est soit sa partition (hash) key, ou le couple partition key, sort (range) key. Au vu de notre modèle de données (association many to many), la bonne solution est de prendre une clé composite. Donc avec une partition key qui sera un string, et une sort key qui sera un string aussi.

Ainsi la clé de la table que nous allons faire sera de la forme suivante :

Entity	Partition Key	Sort Key
User	USER#<USERNAME>	#METADATA#<USERNAME>
Game	GAME#<GAME_ID>	#METADATA#<GAME_ID>
UserGameMapping	GAME#<GAME_ID>	USER#<USERNAME>

Pour rappel, nous allons faire **une seule table**, mais les lignes vont pouvoir définir plusieurs concepts en fonction de leur couple partition key/sort key. Pour éviter tout chevauchement des `USENAME` et des `GAME_ID` nous allons préfixer ses valeurs par le concept qu'elle représente. Comme l'entité `UserGameMapping`. L'avantage de faire une table sera qu'on pourra ajouter des indexes secondaires à notre table pour faire des requêtes complexes, choses impossibles si nous avions eu plusieurs tables. Par contre à la différence d'un modèle relationnel qui peut répondre à presque toutes les questions avec une seule requête, ici, il faut savoir les questions que l'on souhaite poser à la base et la créer en fonction. La phase d'analyse du besoin est donc particulièrement importante !

Création et peuplement d'une table 🎮

Créez une table DynamoDB en utilisant le CDK Terraform. Votre table appellera `battle-royale` aura comme partition key `PK` qui sera un String, et la sort `SK` qui sera une String aussi. Voici un code exemple pour créer votre table

```
 from cdktf_cdktf_provider_aws.dynamodb_table import DynamodbTable, DynamodbTableAttribute


bucket = DynamodbTable(
    self, "DynamodDB-table",
    name= "user_score",
    hash_key="username",
    range_key="lastename",
    attribute=[
        DynamodbTableAttribute(name="username",type="S" ),
        DynamodbTableAttribute(name="lastename",type="S" )
    ],
    billing_mode="PROVISIONED",
    read_capacity=5,
    write_capacity=5
)
```

> 🤷 Les trois derniers paramètres sont liés à la facturation de votre table. Laissez les tels quels.

Une fois la table créée, créez un script python "classique" (= pas lié à Terraform), et chargez les données contenues dans le fichier `items.json`. Chaque ligne de ce fichier est un json qui contient une ligne de notre table. Comme il y a beaucoup de données, faite un upload en batch. Voici des codes pour vous aider. L'idée est d'ouvrir le fichier et un batch_writer, et quand vous lisez une ligne vous l'ajoutez au batch_writer.

```python
# Read file
import json
with open('myfile.json', 'r') as f:
    for row in f:
        items.append(json.loads(row))

# Batch upload
import boto3
# Get the service resource.
dynamodb = boto3.resource('dynamodb')
# Get the table.
table = dynamodb.Table('users')
# Batch writing item. Only one big query, cost less ans it's quicker
with table.batch_writer() as batch:
for i in range(50):
    batch.put_item(
        Item={
            'account_type': 'anonymous',
            'username': 'user' + str(i),
            'first_name': 'unknown',
            'last_name': 'unknown'
        }
    )
```

Si tous à l'air de s'être bien passé, requêtez la table pour compter le nombre de ligne. Voici le code à exécuter :

```python
# Get the service resource.
dynamodb = boto3.resource('dynamodb')
# Get the table.
table = dynamodb.Table('battle-royal')

response = table.scan(
    Select='COUNT',
    ReturnConsumedCapacity='TOTAL',
)
```

Vous devrez obtenir 835 lignes et 14.5 capacityUnits d'utilisée.

Lire les données 👀

Récupérer les données d'un joueur

Récupérez les données associées au joueur avec le username `johnsonscott`. Voici un code pour vous aider

```python
import boto3
# Get the service resource.
dynamodb = boto3.resource('dynamodb')
# Get the table.
table = dynamodb.Table('item')
```

```
item="apple_pie"
resp = table.query(
    Select='ALL_ATTRIBUTES',
    KeyConditionExpression="PK = :pk AND SK = :name",
    ExpressionAttributeValues={
        ":pk": f"PRODUCT#{item}",
        ":name": f"#NAME#{item}",
    },
)
```

Récupérer les informations sur une partie

En vous inspirant du code précédant récupérez les information correspondantes à la partie : `c9c3917e-30f3-4ba4-82c4-2e9a0e4d1cfd`

Récupérer la liste des joueurs pour une partie

Si vous regardez plus en détails le contenu de la clé `Items` du résultat précédent vous allez vous apercevoir que vous avez récupérez une ligne liée de l'entité `Game` et 50 autres de l'entité `UserGameMapping`. Réalisez une requête qui ne vous retournera que les joueurs d'une partie données. Pour ce faire vous pouvez utiliser la condition `begins_with(col, val)` dans la condition de votre requête.

Ajout d'indexes secondaires 

Les indexes secondaire sont une fonctionnalité importante de DynamoDB. Ils permettent définir une nouvelle clé primaire, ce qui permet de requêter la table différemment Chaque index secondaire doit permettre de réaliser de nouveaux types de requêtes et doit être placé judicieusement. En d'autres termes, si vous avez besoin d'indexes secondaires créez-en, sinon vous pouvez vous en dispenser !

Index inversé

Actuellement notre base nous permet à partir d'une partie de récupérer la liste des joueurs, mais pas l'historique des parties d'un joueur. Cela provient du choix de la `partition key` et de la `sort key` pour `UserGameMapping`. Comme la partition key est `GAME#<GAME_ID>` on ne peut faire une recherche qu'à partir d'une partie. Pour permettre la recherche dans les deux sens nous allons mettre en place un index inversé, qui nous permettra de chercher sur la sort key.

Ajoutez l'attribut suivante à l'objet DynamodbTable dans votre code cdktf pour créer un index global dans votre table.

```
global_secondary_index=[
    DynamodbTableGlobalSecondaryIndex(
        name="InvertedIndex",
        hash_key="SK",
        range_key="PK",
        projection_type="ALL",
        write_capacity=5,
        read_capacity=5
    )
]
```

Une fois l'index crée, implémentez le code pour récupérer la liste des parties jouées par un joueur. Ce code va utiliser la méthode `query`, mais vous allez devoir ajouter un paramètre `IndexName` avec le nom de l'index pour réaliser votre requête.

Index secondaire "creux"

Il est également possible de poser un index sur attribut de la table. Cela permettra de faire des requêtes sur cet attribut comme s'il était une clé primaire. Il n'y a pas besoin que l'attribut en question soit présent dans tous les éléments de la table. Seuls les éléments avec l'attribut utilisé seront indexés (d'où le nom d'index creux)

Poser un tel index permet de faire de nouvelles requêtes, impossible à faire précédemment. Par exemple, il est actuellement difficile de faire une recherche pour obtenir les parties encore ouverte sur une carte donnée. Il nous faudrait récupérer toutes les parties, puis filtrer sur les parties avec un `open_timestamp`. Sauf que DynamoDB va devoir scanner toutes la table pour pouvoir faire cela, ce qui va faire exploser les coûts. La solution est de créer un index secondaire avec comme hash key `map` et sort key `open_timestamp`.

En vous inspirant du code précédent mettez en place ce nouvel index et cherchez les parties ouvertes sur la carte `Green Grasslands`