

TP4 - Stocker des données dans le Cloud

Mise en place

Allez sur la plateforme AWS Academy et accédez au cours AWS Academy Learner Lab. Puis cliquez sur `Modules` > `Learner Lab`. Lancez votre environnement en cliquant sur `Start Lab`. Une fois le cercle passé au vert, cliquez sur `AWS Details` et `AWS CLI`. Les clés que vous voyez vont permettre un accès programmatique à votre compte. Cherchez le dossier `.aws` sur votre machine puis remplacez le contenu du fichier `credentials` par les clés 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 tel quel!** 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",
    force_destroy=True
)
```

Déployez votre architecture et vérifiez si votre bucket est bien créé.

Manipulation d'objets


Maintenant, vous allez ajouter des objets dans votre bucket. Vous trouverez sur Moodle différents fichiers à téléverser, mais vous pouvez utiliser les vôtres si vous le souhaitez. Une fois vos fichiers téléversés, essayez de les récupérer, les lire et les supprimer depuis Python. Voici des exemples de code pour vous aider. Attention, ces codes doivent être mis dans un fichier Python différent que celui de votre stack terraform. Il vous faut en plus installer la bibliothèque Python boto3 avec `pip install boto3`

```
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 versioning valant `{"enabled":True}` à l'objet S3Bucket. Redéployez votre infrastructure.

Maintenant, avec votre code Python, téléversez un fichier qui aura le même nom qu'un objet déjà présent dans votre bucket. Allez sur la console AWS, cherchez le service S3, cliquez sur votre bucket puis sur l'objet réuploadé. Dans l'onglet Version, vous devriez voir les différentes versions de votre objet.

 Une fois que l'option de versionnage est activée sur un bucket S3, elle ne peut plus être désactivée, mais seulement suspendue. Cela signifie 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>

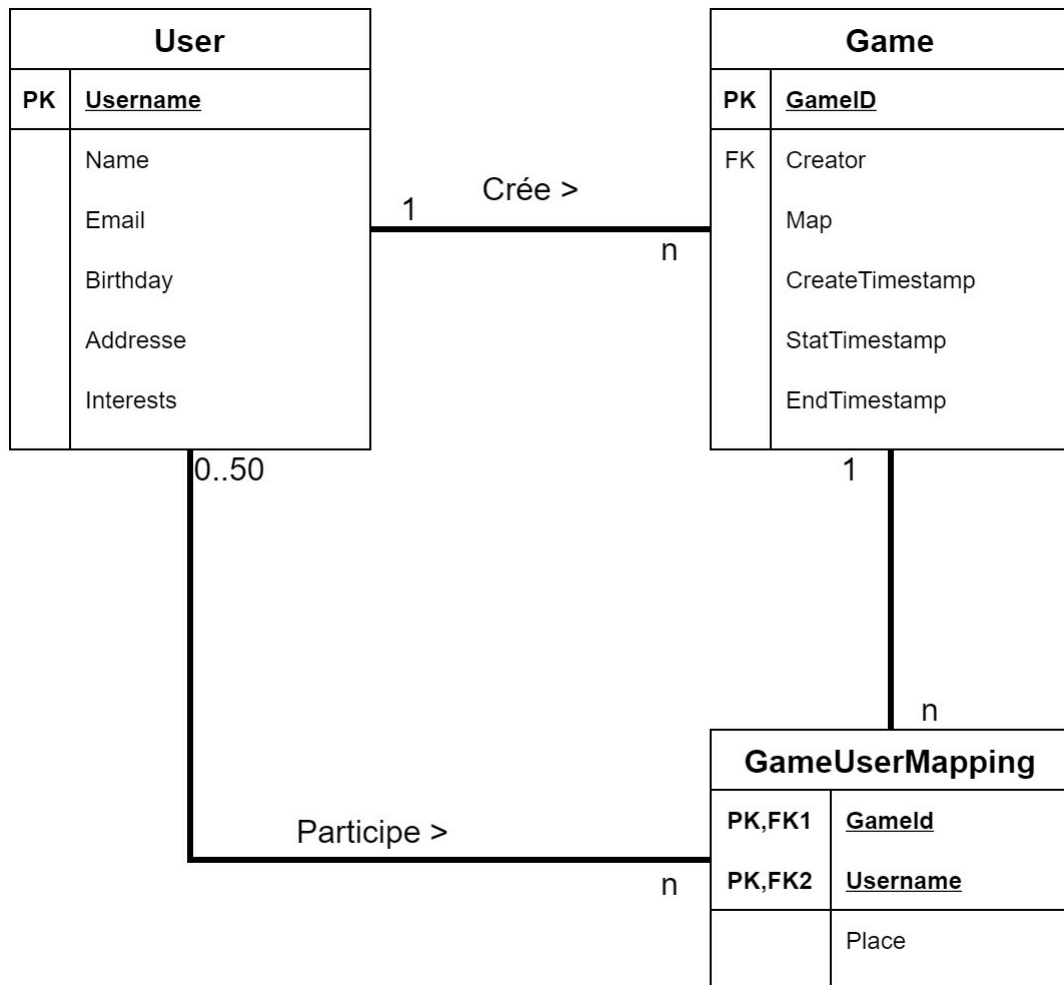
Cette partie du TP consiste à mettre en place une base de données pour stocker des données d'un jeu de type battle royal. Chaque partie regroupe 50 joueurs qui s'affrontent pendant une trentaine de minutes. Notre base devra stocker en temps réel le temps joué par chaque joueur, leur score, et quel joueur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

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 créer une ligne dans la table `GameUserMapping`.



DynamoDB est une base de données NoSQL qui ne dispose pas de moteur de jointure et ne peut pas effectuer d'agrégation de type GROUP BY. En revanche, elle offre d'excellentes performances, quel que soit le volume de données. Choisir une base de données NoSQL plutôt qu'une base SQL est un choix qui entraîne des différences significatives dans les outils disponibles.

Dans notre cas, avec trois entités, il n'est pas nécessaire de créer plusieurs tables. À la place, nous allons utiliser une seule grande table dans cet exercice qui contiendra toutes les données. Cependant, nous devons pouvoir identifier de manière unique les différentes informations de notre base, à savoir `User`, `Game` et `GameUserMapping`. Un utilisateur est identifié de manière unique par son `USERNAME`, un jeu par son `GAME_ID` et une ligne de `GameUserMapping` par le couple `GAME_ID` et `USERNAME`.

Une table DynamoDB est définie par une clé primaire, qui peut être soit sa clé de partition (hash key), soit le couple clé de partition, clé de tri (sort key). Au vu de notre modèle de données (association many-to-many), la meilleure solution est d'utiliser une clé composite. Ainsi, la clé de notre table DynamoDB aura 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 utiliser **une seule table**, mais les lignes pourront représenter plusieurs concepts en fonction de leur combinaison de clé de partition/clé de tri. Pour éviter toute confusion entre les `USERNAME` et les `GAME_ID`, nous allons préfixer ces valeurs par le concept qu'elles représentent, comme l'entité `UserGameMapping`. L'avantage de cette approche est que nous pourrions ajouter des index secondaires à notre table pour effectuer des requêtes complexes, ce qui serait impossible avec plusieurs tables distinctes. Cependant, contrairement à un modèle relationnel qui peut répondre à presque toutes les questions avec une seule requête, ici, il est essentiel de connaître les questions que l'on souhaite poser à la base de données et de la concevoir en conséquence. La phase d'analyse des besoins est donc particulièrement importante !

Création et peuplement d'une table 🎮

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

```
from cdktf_cdktf_provider_aws.dynamodb_table import DynamodbTable, DynamodbTableAttribute

bucket = DynamodbTable(
    self, "DynamoDB-table",
    name= "user_score",
    hash_key="username",
    range_key="lastname",
    attribute=[
        DynamodbTableAttribute(name="username", type="S" ),
        DynamodbTableAttribute(name="lastname", 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, faites un envoi 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`.

```
# 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 and 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 tout a l'air de s'être bien passé, requêtez la table pour compter le nombre de lignes. Voici le code à exécuter :

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

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

Vous devriez 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

```
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 précédent, récupérez les informations 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 remarquerez que vous avez récupéré 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 donnée. Pour ce faire, vous pouvez utiliser la condition `begins_with(col, val)` dans la condition de votre requête.

Ajout d'index secondaires

Les index secondaires sont une fonctionnalité importante de DynamoDB. Ils permettent de 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'index 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 suivant à l'objet `DynamodbTable` dans votre code CDK 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 créé, 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 un 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 toute la table, 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`