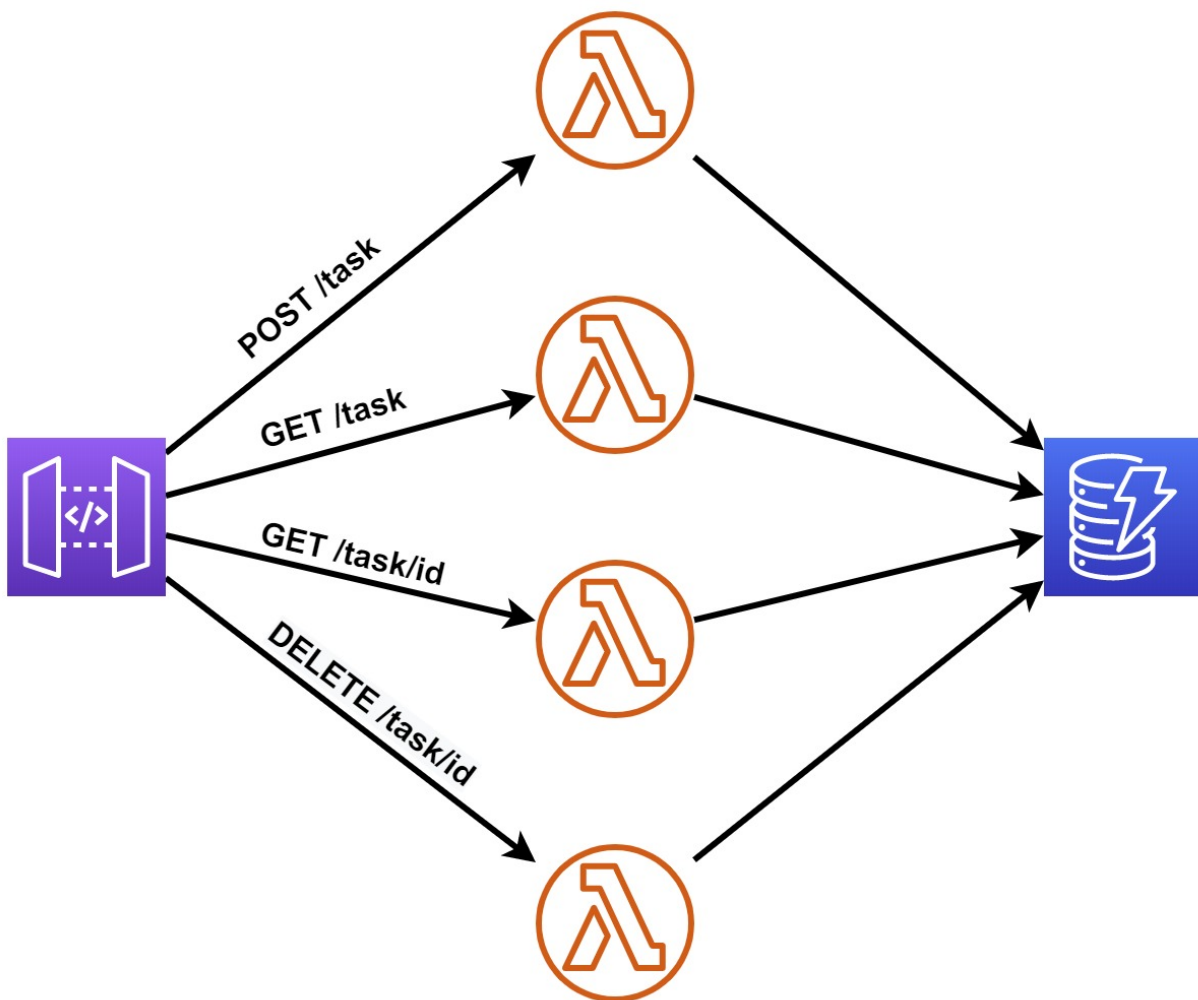


TP5 - Faire une API REST 100% serverless

Le but de ce TP est de faire une API REST en utilisant uniquement des services serverless proposés par AWS. Cela ne va pas être plus compliqué pour vous (voir même cela risque d'être plus facile). Comme votre application sera 100% serverless, vous n'allez plus utiliser Terraform, mais AWS SAM (Serverless Application Model). Comme Terraform, SAM est une solution IaC, et va permettre de définir l'architecture de votre application comme du code. Cette fois-ci ce ne sera pas du code python, mais un simple yaml (YAML Ain't Markup Language), un format clef valeur proche du json, mais où l'indentation est importante.

Le but du TP est de mettre en place une API REST pour gérer des tâches d'une to-do liste. Vous n'allez faire pendant la séance que la partie création d'une tâche. À vous de terminer si vous le souhaitez.



Un hello world avec SAM

Placez-vous dans le répertoire où vous souhaitez faire le TP puis exécutez la commande `sam init`. Sélectionnez l'option `AWS Quick Start Templates` puis sélectionnez le premier template. Pour la question suivante validez que vous souhaitez utiliser python, refusez X-ray puis donnez un nom à votre projet. Une fois le projet téléchargé un dossier sera apparu avec arborescences suivante

(certaines parties sont omises) :

```
1 test
2   └─ events
3     └─ event.json
4   └─ hello_world
5     └─ app.py
6     └─ requirements.txt
7     └─ __init__.py
8   └─ template.yaml
9     └─ __init__.py
```

Le fichier `template.yaml` contient l'infrastructure de votre code, le dossier `hello_world` va contenir le code de votre lambda (ce dossier peut être renommé) et `event.json` va contenir un évènement pour tester votre application.

Le fichier qui nous intéresse particulièrement en ce moment est le fichier `template.yaml`. Voici son contenu (certaines parties sont omises):

```
1  AWSTemplateFormatVersion: '2010-09-09'
2  Transform: AWS::Serverless-2016-10-31
3
4  Globals:
5    Function:
6      Timeout: 3
7
8  Resources:
9    HelloWorldFunction:
10     Type: AWS::Serverless::Function
11     Properties:
12       CodeUri: hello_world/
13       Handler: app.lambda_handler
14       Runtime: python3.9
15       Architectures:
16         - x86_64
17       Events:
18         HelloWorld:
19           Type: Api
20           Properties:
21             Path: /hello
22             Method: get
23
24  Outputs:
25    HelloWorldApi:
26     Description: "API Gateway endpoint URL for Prod stage for Hello world
27     function"
28     Value: !Sub "https://${ServerlessRestApi}.execute-
29     api.${AWS::Region}.amazonaws.com/Prod/hello/"
30    HelloWorldFunction:
31     Description: "Hello world Lambda Function ARN"
32     Value: !GetAtt HelloWorldFunction.Arn
```

```

31 | HelloWorldFunctionIamRole:
32 |   Description: "Implicit IAM Role created for Hello world function"
33 |   Value: !GetAtt HelloWorldFunctionRole.Arn

```

Vous constatez qu'il y a 3 parties dans votre fichier et évidemment chacune à son rôle:

- **Globals** permet de définir des configurations de manière globale. Ici le timeout des fonctions lambda du template est de 3 secondes
- **Ressources** est la partie où infrastructure est défini. Actuellement ce template définit :
 - Une fonction lambda, c'est ce que veut dire la ligne 10. Son code est dans le dossier hello-world, le handler est la fonction lambda_handler du fichier app.py, et la version de python de la fonction est 3.9
 - Une API Gateway. C'est moins clair mais c'est ce que veut dire la ligne 19. Comme la lambda est déclenché par une API Gateway, elle va être créée même si elle n'est pas formellement définie. Le code de la lambda sera déclenché par un appel GET sur le chemin /hello
- **Outputs** permet de récupérer facilement des valeurs qui ne sont connues qu'une fois le template déployé. Ici l'URL pour déclencher la fonction, l'identifiant AWS de la lambda et de son rôle.

Comme vous utilisez des labs AWS academy, ce code ne fonctionne pas tel quel. En effet, ce code crée automatiquement un rôle pour la lambda, sauf que c'est impossible sur un lab. Ajoutez après `Handler` la ligne `Role: !Sub arn:aws:iam::${AWS::AccountId}:role/LabRole` pour mettre le bon rôle à votre lambda.. Supprimez également les 3 dernières lignes du template.

Maintenant cela fait, il est temps de déployer le template ! Faites un `aws sam deploy --guided` et répondez aux questions dans le terminal.

- Stack Name : répondez ce que vous souhaitez
- AWS Region : us-east-1
- Confirm changes before deploy : y
- Allow SAM CLI IAM role creation : n
- Capabilities [CAPABILITY_IAM]: validez avec entrée
- Disable rollback [y/N]: répondez ce que vous souhaitez
- HelloWorldFunction may not have authorization defined, Is this okay? [y/N]: y
- Validez les prompts suivant avec entrée


Après quelques instants, la liste des ressources à déployer va apparaître dans votre terminal

Operation	LogicalResourceId	ResourceType	Replacement
+ Add	HelloWorldFunctionHelloWorldPermissionProd	AWS::Lambda::Permission	N/A
+ Add	HelloWorldFunction	AWS::Lambda::Function	N/A
+ Add	ServerlessRestApiDeployment47fc2d5f9d	AWS::ApiGateway::Deployment	N/A
+ Add	ServerlessRestApiProdStage	AWS::ApiGateway::Stage	N/A
+ Add	ServerlessRestApi	(AWS::ApiGateway::RestApi)	N/A

Comme c'est un premier déploiement vous avez seulement des opérations `Add`. Votre template tout simple est en train de créer :

- Une fonction lambda (AWS::Lambda::Function) et un politique de sécurité associée (AWS::Lambda::Permission)
- Une API Gateway de type Rest (AWS::ApiGateway::RestApi), avec un stage (AWS::ApiGateway::Stage) le tout accessible sur internet (AWS::ApiGateway::Deployment)

Validez les changements et après quelques instants un bloc output va apparaitre dans le terminal. Copiez/collez l'url de l'api et vérifiez qu'elle fonctionne bien.

 Félicitation vous venez de créer votre premier API Rest 100% serverless sur AWS ! Avec un peu de travail, vous pourriez redéployer tout votre projet info de 2A.

Ajouter une base de données DynamoDB

DynamoDB est une base de données serverless, il est donc possible de l'ajouter dans le template. Dans la partie Resources du template ajoutez :

```
1  DynamoDBTable:
2    Type: AWS::DynamoDB::Table
3    Properties:
4      AttributeDefinitions:
5        - AttributeName: user
6          AttributeType: S
7        - AttributeName: id
8          AttributeType: S
9      KeySchema:
10       - AttributeName: user
11         KeyType: HASH
12       - AttributeName: id
13         KeyType: RANGE
14      ProvisionedThroughput:
15        ReadCapacityUnits: 5
16        WriteCapacityUnits: 5
```

Et déployez de nouveau votre template. Vous allez voir que seule la nouvelle ressource va être déployée.

Maintenant il faut arriver à faire le lien entre la base DynamoDB et la fonction lambda. Pour rendre le code le plus souple possible, le nom de la table ne va pas être *harcodé* dans la fonction, mais mis dans ses variables d'environnement. Dans les properties de votre fonction ajoutez les lignes :

```
1  Environment:
2    Variables:
3      DYNAMO_TABLE: !Ref DynamoDBTable
```

Ces lignes vont faire que lors du déploiement, une variable d'environnement `DYNAMO_TABLE` va être créée, et qu'elle va avoir pour valeur le nom de la table. Comme le nom est généré dynamiquement, on va passer une référence à la ressource DynamoDB (`!Ref DynamoDBTable`) et SAM va dynamiquement injecter le nom.

Déployez le template, allez sur la page du service AWS Lambda, puis sur la lambda créée par le template, puis `configuration` et `Environment variables` et vérifier que tout est bon.

Le code de la lambda

Pour le moment vous avez seulement fait la partie infrastructure de l'application, il est temps de regarder un peu le code. Quand AWS SAM a généré le template de base, il a créé un dossier `hello_world`. Ce dossier contient le code de la fonction lambda. Quand AWS SAM va déployer le code de la lambda, il cherche le contenu de la variable `codeuri` et déploie l'intégralité du dossier. Si le dossier contient un fichier `requirements.txt`, SAM va installer les dépendances pour la lambda. Si vous regardez le fichier `app.py`, il contient la méthode `lambda_handler()`, qui pour le moment ne renvoie qu'une réponse prédéfinie.

1. Dans le `template.yml` changez le nom de la ressource `HelloWorldFunction` en `PostTaskFunction` (attention vous devez mettre à jour la dernière ligne du template également)
2. Changez dans l'Events de la fonction le nom de l'API (HelloWorld -> Task), et de sa propriété (/hello -> task, get -> post)
3. Créez un dossier `PostTask` et faites que votre lambda pointe vers ce dossier.
4. Implémentez la fonction `lambda_handler()` qui va poster un commentaire dans la base DynamoDB. Une requête http post permet de créer une ressource, les éléments de la ressource seront dans le `body` de la requête. Voici un exemple de body

```
1 {
2   "user" : "Rémi",
3   "taskTitle" : "Corriger le TP noté",
4   "taskBody" : "Tout est dans le titre",
5   "taskDueDate" : "18/05/2023"
6 }
```

Pour vous aider :

- Pour récupérer le contenu du corps de la requête, utilisez `body = json.loads(event['body'])` (pensez à importer le package json, ce package est dans les packages de base de python, pas besoin de l'ajouter dans le requirements.txt). La variable body est un dictionnaire python.
- Chaque task va avoir un identifiant unique générée par avec la fonction `uuid.uuid4()` (il faut importer uuid, c'est un package dans la distribution python de base)
- Voici un code exemple pour écrire un objet dans DynamoDB, ce code n'est pas à reprendre tel quel ! Vous trouverez le nom de la table dans les variables d'environnement de votre lambda.

```

1  import boto3
2  # Get the service resource.
3  dynamodb = boto3.resource('dynamodb')
4  # Get the table.
5  table = dynamodb.Table('users')
6  # Put item to the table.
7  table.put_item(
8      Item={
9          'username': 'janedoe',
10         'first_name': 'Jane',
11         'last_name': 'Doe',
12         'age': 25,
13         'account_type': 'standard_user',
14     }
15 )

```

🧑🏻‍💻 Bien que n'étant pas par défaut dans python, boto3 est par défaut dans les lambdas

- Une fois l'ajout fait, faites retourner à votre lambda un dictionnaire s'inspirant de celui ci :

```

1  {
2      "statusCode": 200,
3      "body": votre objet task
4  }

```