Práctica 1: Micro-servicios

Parte 2: REST API en cloud

1. Servicios Cloud

En producción, los microservicios han encontrado un aliado fundamental en los servicios cloud, algunos orientados a contenedores, como Kubernetes o Docker Swarm, y otros con una granilaridad más fina aún, tales como AWS Lambda o Google Cloud Functions, los cuales habitualmente también usan contenedores, pero son gestionados por la infraestructura directamente.

El uso de Docker Swarm o Kubernetes no difiere demasiado del que hemos visto en la primera parte, salvo por la forma de configurar los contenedores, ya que se realiza mediante archivos de configuración, que detallan las relaciones entre los servicios, redes internas, volúmenes de datos, etc. en lugar de ejecutar los comandos docker run directamente.

Al igual que haríamos en un proyecto real, primero escogeremos el proveedor cloud más adecuado, para evitar incurrir en los costes extra de desarrollo y mantenimiento que supondría usar distintos proveedores. En este caso usaremos los servicios Amazon Web Services (AWS), gracias a que podemos usar un entorno de desarrollo local sin ningún coste y de código abierto, compatible con AWS (LocalStack).

2. LocalStack

LocalStack ya lo hemos instalado en la primera parte de la práctica como parte del entorno python. Los servicios de LocalStack normalmente usan contenedores, por lo que será necesario que el servicio docker esté disponible, bien mediante docker Desktop o Docker en modo rootless.

Para iniciar los servicios ejecutaremos en un terminal:

$> source ./venv/si1p1/bin/activate

$> localstack start

Para detenerlo simplemente pulsaremos CTRL+C. En ese terminal también podremos ver la información con los logs que producen los servicios.

LocalStack emula más de 30 servicios AWS, en la versión Open Source, sin embargo, en esta práctica nos centraremos en:

- AWS S3: Proporciona un servicio de archivos
- AWS Lambda: Permite la ejecución de funciones escalables, para implementar los microservicios
- API Gateway: Aporta seguridad a nuestro API, que conectaremos a AWS Lambda.

- DynamoDB: Es una base de datos clave/valor en la nube, que nos permite guardar datos JSON de forma sencilla.

También tendremos instalados un nuevo comando, awslocal, que nos permitirá usar los mismos comandos del cliente de AWS (aws), pero usando el endpoint http://localhost:4566, aunque también podríamos usar el comando aws directamente, usando:

$> AWS_ACCESS_KEY_ID=test AWS_SECRET_ACCESS_KEY=test AWS_DEFAULT_REGION=${DEFAULT_REGION:-$AWS_DEFAULT_REGION} aws --endpoint-url=http://${LOCALSTACK_HOST:-localhost}:4566 … resto de comandos y opciones aws a ejecutar

Puesto que localStack ejecutará nuestro código mediante contenedores, esto podría ocasionar algunos problemas de conectividad, ya que los servicios de LocalStack están escuchando en localhost:4566, pero el contenedor no puede acceder directamente al localhost de la máquina anfitrión (su "localhost" es local al contenedor).

Hay varias formas de solucionar este problema, como por ejemplo usar la IP externa del anfitrión y configurar LocalStack para que escuche en esa IP, pero lo más sencillo y seguro es utilizar las variables de entorno LOCALSTACK_HOSTNAME y EDGE_PORT, que estarán disponibles en nuestros contenedores.

Puedes consultar la documentación en la bibliografía, o el manual de referencia en consola, añadiendo help al comando awslocal. Por ejemplo "awslocal s3 help "o awslocal s3 ls help"

3. AWS S3

El Servicio S3 nos permite crear, modificar o borrar archivos en la nube. Los archivos están organizados en buckets, que almacenan los archivos, cada uno con una ruta dentro del bucket.

Tarea: AWS S3

Ejecuta en un terminal:

```
# create bucket
awslocal s3 mb s3://si1p1
# create dir & copy
awslocal s3 cp src/user_rest.py s3://si1p1/src/user_rest.py
awslocal s3 ls s3://si1p1
# ... dir
awslocal s3 ls s3://si1p1/src
# ... dir contents
awslocal s3 ls s3://si1p1/src/
```

Pregunta: AWS S3

1. ¿Qué hacen los distintos subcomandos de awslocal s3?
2. ¿Qué comando podemos usar para descargar el archivo user_rest.py desde el bucket que hemos creado, dándole otro nombre para evitar sobreescribirlo ?

4. AWS Lambda

Otra forma de ejecutar nuestros microservicios, es mediante la ejecución de funciones en la nube, que en el caso de AWS se llama AWS Lambda.

Estas funciones normalmente se ejecutan bajo demanda, cuando ocurre algún evento que desencadene su invocación. En esta práctica el evento que la desencadenará será peticiones http con una ruta y método http concreto. De esta forma, podemos asociar una función a cada uno de los métodos de nuestro API.

Comparado con el uso de contenedores, las funciones lambda solo están en ejecución el tiempo necesario para atender la petición, reduciendo los costes al mínimo. Este tiempo además suele estar limitado, para evitar consumir demasiados recursos si una función lambda por error entrase en un bucle infinito.

Las funciones lambda pueden estar escritas en distintos lenguajes de programación, incluso para el mismo proyecto, y hacer uso de librerías. Todo lo necesario para la ejecución se ha de empaquetar y subir al servicio, aunque normalmente hay una serie de librerías que ya están incluidas. Por ejemplo, AWS Lambda ya incluye, en el caso de Python, la librería boto3, para conectar a AWS. En nuestro caso usaremos un fichero zip, con el fichero .py en la raíz del archivo comprimido.

Tarea: Lambdas

Ejecuta en un terminal:

```
$> LAMBDA_NAME=user_lambda
$> [[ -d build ]] || mkdir build ; \
   cd src ; zip ../build/${LAMBDA_NAME}.zip ${LAMBDA_NAME}.py; cd -
$> echo "create: newlambda"; \
   awslocal lambda create-function \
    --function-name ${LAMBDA_NAME} \
    --runtime python3.9 \
    --zip-file fileb://build/${LAMBDA_NAME}.zip \
 --handler ${LAMBDA_NAME}.handler \
 --role arn:aws:iam::000000000000:role/si1 ; \
 awslocal lambda create-function-url-config \
 --function-name ${LAMBDA_NAME} \
 --auth-type NONE ;
$> # wait
$> awslocal lambda wait function-active-v2 --function-name ${LAMBDA_NAME}
```

```
$> awslocal lambda list-functions
$> curl -X POST \
  $(awslocal lambda get-function-url-config --function-name ${LAMBDA_NAME} |
   jq -r '.FunctionUrl') \
  -H 'Content-Type: application/json' \
  -d '{"username": "pepe", "data": "10"}'
```

Pregunta: Lambdas

1. ¿Qué runtime podríamos usar si por ejemplo queremos implementar una función Lambda en JavaScript ?
2. ¿Cómo recibe la función lambda el usuario los datos que hemos enviado con el comando curl?
3. Comprueba que ha funcionado descargando del bucket el fichero creado por la función. ¿Qué comandos has utilizado para ello?

Tarea: depuración lambdas

Ejecuta en un terminal:

```
$> docker logs localstack_main
$> awslocal logs describe-log-groups
$> awslocal logs describe-log-streams --log-group-name  /aws/lambda/${LAMBDA_NAME}
$> # last logs
$> awslocal logs describe-log-streams --log-group-name  /aws/lambda/${LAMBDA_NAME} |
  jq '.logStreams[].firstEventTimestamp' | sort -n
$> LLOGSTREAMS=$(awslocal logs describe-log-streams \
     --log-group-name /aws/lambda/${LAMBDA_NAME} |
     jq -r '.logStreams[].logStreamName')
$> for LOGSTREAM in ${LLOGSTREAMS} ; do
     awslocal logs get-log-events \
      --log-group-name "/aws/lambda/${LAMBDA_NAME}" \
      --log-stream-name "${LOGSTREAM}" ; done |
   jq '.events[].message'
```

Pregunta: depuración lambdas

1. ¿Describe a qué corresponde la salida de cada uno de los comandos anteriores?

Tarea: Borrado de funciones

Ejecuta en un terminal:

```
$> echo "clean: dellambda" ; \
$> awslocal lambda delete-function-url-config \
    --function-name user_lambda ; \
$> awslocal lambda delete-function --function-name user_lambda
$> echo "clean: md zip" \
   awslocal s3 rb --force s3://si1p1 ; \
  rm build/user_lambda.*
```

Pregunta: Borrado de funciones

1. ¿Qué ocurre si en lugar de borrar las funciones y archivos en S3, simplemente reiniciamos localStack? ¿Cómo podríamos evitarlo?

5. API Gateway

Aunque hemos visto que podemos ejecutar las funciones lambda mediante una URL asociada, esto presenta muchas algunas limitaciones que podemos evitar gracias al servicio API Gateway.

Este servicio nos permite diseñar el API completo de la aplicación, y conectar distintas URLs (rutas), definiendo sus parámetros, etc., con distintos elementos de la aplicación, y en particular con funciones Lambda.

Por ejemplo, algo habitual es añadir seguridad, mediante por ejemplo roles, para limitar las acciones que cada tipo de usuario puede realizar con nuestro API.

También podemos transformar tanto la entrada a las funciones lambda, u otros servicios invocados desde API Gateway, o transformar el resultado de los mismos.

En nuestro caso utilizaremos este servicio para crear nuestro API Restful, aunque ignoraremos por ahora los aspectos de seguridad, ya que en esta práctica no hemos creado un servicio de autenticación de usuarios, que sería necesario para poder reconocer los roles de los usuarios que se conectan al API.

El servicio API Gateway se configura habitualmente desde la consola web de AWS. Sin embargo, puesto que estamos usando LocalStack, y la versión Open Source (community) no dispone de interfaz de usuario gráfica, tendremos que usar los comandos aws (awslocal) directamente para crear el API.

Necesitaremos realizar los siguientes pasos:

- Crear el API, dándole un nombre, y asociándolo a una región AWS desde donde se ejecutará. Se creará un identificador único, que guardaremos en una variable local, ya que lo necesitaremos para los siguientes pasos.

- Necesitaremos crear los recursos del API. Estos son las distintas rutas que necesitaremos. En nuestro caso todas comenzarán por "user_lambda" (variable API_NAME). Los recursos se crean de forma jerárquica, siguiendo el mismo esquema que las URLs http, creando cada una de las partes que están separadas por / en una URL, algunas de las cuales pueden corresponder a parámetros en el API.
- Una vez creada una ruta completa, con los recursos de cada parte, se pueden asociar métodos http a ella (PUT, POST, etc.), indicando también el tipo de autenticación necesario para llamar a dichos métodos (NONE en este caso, al no usar autenticación).
- Cada método y ruta se puede integrar con otros servicios AWS. En nuestro caso lo integraremos con funciones lambda.
- Por último, realizaremos el despliegue del API.

En una ruta podemos configurar más de un método y asociarlo con la misma u otra función. Las rutas también pueden tener parámetros, cuyo valor podemos leer desde las funciones lambda cuando estas sean invocadas.

Tarea: Creación del API

Ejecuta en un terminal:

```
$> export REGION="us-east-1" ; export API_NAME="user_lambda"
$> awslocal apigateway create-rest-api \
    --region ${REGION} \
    --name ${API_NAME}
$> [ $? == 0 ] || echo "Failed: AWS / apigateway / create-rest-api"
$> awslocal apigateway get-rest-apis
```

Pregunta: Creación del API

1. ¿Qué ID se ha asignado al API ?
2. ¿Qué ocurre si volvemos a crear otro API con el mismo nombre ?

Tarea: Creación de las rutas

Antes de continuar asegúrate que solo tenemos un api con el nombre "user_lambda".

Si hubiera más de uno, puedes borrarlos ejecutando:

$> awslocal apigateway delete-rest-api --rest-api-id id_del_api_a_borrar

Ejecuta en un terminal:

```
$> API_ID=$(awslocal apigateway get-rest-apis \
    --query "items[?name=='${API_NAME}'].id" \
    --output text --region ${REGION})
$> PATH_PART="user" ; PARENT_PATH="/"
$> PARENT_RESOURCE_ID=$(awslocal apigateway get-resources \
  --rest-api-id ${API_ID} --query "items[?path=='${PARENT_PATH}'].id" \
  --output text --region ${REGION})
$> awslocal apigateway create-resource \
    --region ${REGION} \
    --rest-api-id ${API_ID} \
    --parent-id ${PARENT_RESOURCE_ID} \
    --path-part "${PATH_PART}"
$> PATH_PART="{username}" ; PARENT_PATH="/user"
$> PARENT_RESOURCE_ID=$(awslocal apigateway get-resources \
  --rest-api-id ${API_ID} --query "items[?path=='${PARENT_PATH}'].id" \
  --output text --region ${REGION})
$> awslocal apigateway create-resource \
    --region ${REGION} \
    --rest-api-id ${API_ID} \
    --parent-id ${PARENT_RESOURCE_ID} \
    --path-part "${PATH_PART}"
$> [ $? == 0 ] || echo "Failed: AWS / apigateway / create-resource ${PATH_PART}"
$> awslocal apigateway get-resources --rest-api-id ${API_ID}
```

Pregunta: Creación de las rutas

1. ¿Qué ID se ha asignado al recurso con la ruta /user/{username} ?
2. ¿Qué ocurre si creamos también la ruta /user/{name} ?
3. En el caso de que se pueda crear a la vez la ruta /user/{name} y /user/{username}, ¿ tiene sentido ?

Tarea: Creación de métodos

Antes de continuar asegúrate de borrar la ruta /user/{name}, ya que no la necesitaremos.

Para borrar un recurso ejecuta:

$> awslocal apigateway delete-resource --rest-api-id ${API_ID} --resource-id id_del_recurso

Podemos asociar más de un método HTTP a una misma ruta, para los distintos métodos de nuestro API. En este paso especificaremos también los parámetros, que pueden ser de tres tipos, *path*, querystring y *header. Podemos añadir tantos parámetros como necesitemos, y también darles un valor estático.*

Para crear por ejemplo el método POST en /user/{username}, con un parámetro de ruta, ejecuta:

```
$> PARAM=username ; METHOD_PATH="/user/{username}" ; HTTP_METHOD=POST
$> RESOURCE_ID=$(awslocal apigateway get-resources \
  --rest-api-id ${API_ID} --query "items[?path=='${METHOD_PATH}'].id" \
  --output text --region ${REGION})
$> awslocal apigateway put-method \
    --region ${REGION} \
    --rest-api-id ${API_ID} \
    --resource-id ${RESOURCE_ID} \
    --http-method ${HTTP_METHOD} \
    --request-parameters "method.request.path.${PARAM}=true" \
    --authorization-type "NONE"
$> [ $? == 0 ] || echo "Failed: AWS / apigateway / put-method ${METHOD_PATH}"
$> awslocal apigateway get-resources --rest-api-id ${API_ID}
```

Pregunta: Creación de métodos

1. Crea también el método GET asociado a la misma ruta, ¿qué instrucciones has ejecutado para crearlo?
2. ¿Cómo podemos ver los métodos asociados a las rutas del API?

Tarea: Integración

Nota: Para integrar la función lambda usaremos el tipo AWS_PROXY. El método de integración será siempre POST, independiente del valor de http_method. La URI para invocar la lambda, según la especificación de AWS, tendrá la forma: /2015-03-31/functions/FunctionName/invocations

Ejecuta en un terminal:

```
$> HTTP_METHOD=POST ; LAMBDA_NAME=user_lambda
$> awslocal apigateway put-integration \
    --region ${REGION} \
    --rest-api-id ${API_ID} \
    --resource-id ${RESOURCE_ID} \
```

```
    --http-method ${HTTP_METHOD} \
    --type AWS_PROXY \
    --integration-http-method POST \
    --uri arn:aws:apigateway:${REGION}:lambda:path/2015-03-
31/functions/${LAMBDA_NAME}/invocations \
    --passthrough-behavior WHEN_NO_MATCH
$> [ $? == 0 ] || echo "Failed: AWS / apigateway / put-integration"
$> awslocal apigateway get-resources --rest-api-id ${API_ID}
```

Pregunta: Integración

1. Integra también el método GET asociado a la misma ruta con la misma función lambda, ¿qué instrucciones has ejecutado para crearlo?
2. ¿Cómo podemos ver si se ha integrado cada método del API?
3. Con ayuda de la documentación, por ejemplo ejecutando "awslocal apigateway help", borra la integración del método GET, ¿qué instrucciones has ejecutado para lograrlo?

Tarea: Despliegue

AWS soporta distintas etapas (Stage) desplegadas simultáneamente, cada una asociada a un despliegue.

Esto puede ayudar por ejemplo a realizar pruebas durante el desarrollo, desplegando etapas con distintas configuraciones.

Una limitación importante es que el despliegue no fija la implementación de la integración. Por ejemplo, si modificamos una lambda que ya está integrada, la aplicación ejecutará la nueva lambda incluso sin hacer un nuevo despliegue. Para evitar este problema, se pueden asociar distintos números de versión a las lambdas, e integrar versiones específicas en cada método.

Ejecuta en un terminal:

```
$> STAGE=dev
$> awslocal apigateway create-deployment \
    --region ${REGION} \
    --rest-api-id ${API_ID} \
    --stage-name ${STAGE}
$> [ $? == 0 ] || echo " Failed: AWS / apigateway / create-deployment"
$> awslocal apigateway get-deployments --rest-api-id ${API_ID}
$> awslocal apigateway get-stages --rest-api-id ${API_ID}
```

Pregunta: Despliegue

1. ¿Qué ocurre con la etapa "dev" (valor de STAGE) si repetimos el despliegue?
2. Con ayuda de la documentación, por ejemplo ejecutando "awslocal apigateway help", borra el despliegue que ya no está asociado a la etapa "dev". ¿Qué instrucciones has ejecutado para lograrlo?

Tarea: Pruebas del API

Podremos probar nuestro API usando la siguiente URL como base:

http://localhost:4566/restapis/${API_ID}/${STAGE}/_user_request_/

```
$> STAGE=dev
$> ENDPOINT=http://localhost:4566/restapis/${API_ID}/${STAGE}/_user_request_/user
$> curl -X POST ${ENDPOINT}/pepe \
  -H 'Content-Type: application/json' \
  -d '{"username": "tete", "data": "100"}'
```

Pregunta: Pruebas del API

1. Usando los comandos de S3, lista el contenido del bucket y descarga el archivo que se ha creado para comprobar su contenido. ¿Qué ha ocurrido? ¿Ha cambiado el valor del fichero /users/pepe o se ha creado un archivo distinto?

Tarea: Mejora del API

Crea una nueva función lambda llamada "user_lambda_param" que use realmente el parámetro "username" que le proporciona el API, ya que la lambda anterior lo estaba ignorando.

Para esta tarea copia user_lambda.py en user_lambda_param.py, y modifica el código para, de forma similar a lo que hacíamos en user_rest.py, crear en S3 un objeto con los datos pasados en event.body y guardarlos en el bucket S3 en la ruta /user/{username}/data.json

Para obtener el parámetro username del evento, usa el siguiente código:

```
username=event['pathParameters']['username']
```

Nota: Los valores de "pathParameters" los rellena el servicio API Gateway, por lo que no podrás probar la lambda directamente, y será necesario integrarla en el API para realizar las pruebas.

Pregunta: Mejora del API

1. ¿Qué pasos han sido necesarios para usar la nueva lambda en lugar de la anterior? Indica los comandos que has empleado

6. DynamoDB

El servicio S3 está diseñado para guardar ficheros, por lo que no es el más eficiente si queremos guardar datos. En su lugar podemos usar DynamoDB, que es una base documental gestionada como servicio.

Antes de empezar a usar DynamoDB en nuestro API, primero crearemos una tabla, definiendo el esquema de los datos que contendrá.

El esquema permite definir los campos requeridos para crear las claves y su tipo, sin embargo, como en otras bases de datos documentales, podremos crear documentos con otros atributos, aunque no estén en el esquema. Además, aunque no es lo habitual, los atributos adicionales pueden tener tipos diferentes en distintos objetos.

Tarea: Creación de una tabla

Ejecuta en un terminal:

```
$> awslocal dynamodb create-table \
  --table-name si1p1 \
  --attribute-definitions AttributeName=User,AttributeType=S \
  --key-schema AttributeName=User,KeyType=HASH \
  --region $REGION \
  --provisioned-throughput ReadCapacityUnits=5,WriteCapacityUnits=5
```

Pregunta: Creación de una tabla

1. ¿De qué tipo son los distintos campos de la tabla?

Tarea: Insertando datos manualmente

DynamoDB requiere que al insertar un dato indiquemos su tipo, pero este no se deduce automáticamente, de ahí que para indicar el valor de cada atributo se use la sintaxis { atributo: { Tipo : valor }}

Ejecuta en un terminal:

```
$> awslocal dynamodb put-item \
  --table-name si1p1 \
  --item '{"User": {"S":"pipo"}, "EMail": {"S": "pipo@ss.com"}}'
$> awslocal dynamodb scan --table-name si1p1 --region us-east-1
```

Pregunta: Insertando datos manualmente

1. ¿Qué ocurre si creamos un item sin campo "User" o sin campo "Email"?

Tarea: Lectura de datos

Según el esquema que hemos definido, DynamoDB usará el campo User como clave principal (o de particionado) de la tabla "si1p1".

Ejecuta en un terminal:

```
$> awslocal dynamodb get-item \
  --table-name si1p1 \
  --key '{"User": {"S":"pipo"}, "Email": {"S": "pipo@ss.com"}}'
$> awslocal dynamodb scan --table-name si1p1 --region us-east-1
```

Pregunta: Lectura de datos

2. ¿Qué ocurre si creamos un item sin campo "User"?
3. ¿Qué ocurre si volvemos a insertar el mismo usuario sin el campo "Email"?

Tarea: Lambda con dynamoDB

Crea la lambda correspondiente a user_lambda_dynamo.py, siguiendo las instrucciones anteriores, e intégrala en el método POST de user/{username}, donde antes teníamos user_lambda_param.py.

Tras ello, ejecuta en un terminal:

```
$> curl -X POST ${ENDPOINT}/user2 -H 'Content-Type: application/json' \
    -d '{"email": mimail@dominio.es}'
```

Nota: La nueva lambda ya usa el parámetro username, que guardará en el campo User de la tabla, por lo que no es necesario modificarla. En el cuerpo del evento, espera datos con el atributo "email", que guardará en el campo "Email" de la tabla.

Pregunta: Insertando datos manualmente

1. ¿Qué pasos han sido necesarios para cambiar de función lambda?
2. ¿Qué datos se han añadido a la tabla si1p1 tras ejecutar el comando curl anterior?

Tarea: GET /user/{username} con dynamoDB

Usando como ejemplo la lambda user_lambda_dynamo.py, crea e integra user_get_lambda_dynamo.py. En este caso intégrala en el método GET de user/{username}, el cual no tenía ninguna lambda asociada.

La lambda user_get_lambda_dynamo ha de buscar el usuario y devolver el objeto encontrado en el cuerpo de la respuesta (puedes usar la variable *rest_body* para ello). Si no se encuentra se devolverá un objeto vacío.

Notas de implementación:

- La entrada y salida del método get_item en boto3 es muy similar a la que hemos usado anteriormente en línea de comando. En la entrada se usa Key={'clave': valor}, y la respuesta estará en la clave "Item", si se encuentra. Para más detalles consulta la documentación del método get_item en el manual de referencia de boto3 para dynamoDB.
- Como entrada a esta lambda ya no se usará la clave "body" del evento, que no existirá, por lo que dará error si se intenta acceder a event['body']

Pregunta: GET /user/{username} con dynamoDB

1. ¿Qué pasos han sido necesarios para crear e integrar la nueva función lambda?
2. ¿Qué comando curl podemos usar para probar el método GET /user/{username}?. Da ejemplos también con la respuesta tanto si existe como si no existe el usuario.

7. Documentación a aportar

En esta práctica deberéis entregar todo lo desarrollado en las dos partes:

1. Memoria:

   Respuesta a las preguntas planteadas.

   Listado de archivos empleados junto con una breve descripción.

   Detalles de implementación relevantes.

2. Archivos, al menos:

   src/Dockerfile

   src/user_rest.py

   src/user_lambda_param.py

   src/user_get_lambda_dynamo.py

8. Entrega

La fecha y normas de entrega de las prácticas se encuentran en Moodle.

9. Bibliografía

Bibliografía de los distintos apartados.

Cloud

1. AWS Amazon Webservices: https://aws.amazon.com
2. CNCF Landscape: https://landscape.cncf.io/
3. CNCF Trailmap: https://raw.githubusercontent.com/cncf/trailmap/master/CNCF_TrailMap_latest.pdf

Localstack

1. doc: https://docs.localstack.cloud/overview/

AWS lambda

1. boto3:
    a. https://boto3.amazonaws.com/v1/documentation/api/latest/index.html
    b. https://github.com/ciaranevans/localstack_and_pytest_1
    c. https://medium.com/uk-hydrographic-office/developing-and-testing-lambdas-with-pytest-and-localstack-21a111b7f6e8
    d. https://github.com/localstack/localstack-pro-samples/tree/master/lambda-function-urls
    e. https://boto3.amazonaws.com/v1/documentation/api/latest/guide/s3-example-creating-buckets.html
2. aws lambda: https://docs.localstack.cloud/aws/lambda/
3. aws lambda programming model: https://docs.aws.amazon.com/lambda/latest/dg/foundation-progmodel.html

AWS API gateway

1. https://gist.github.com/crypticmind/c75db15fd774fe8f53282c3ccbe3d7ad

2. https://docs.aws.amazon.com/lambda/latest/dg/services-apigateway-tutorial.html