Scrum Documentation
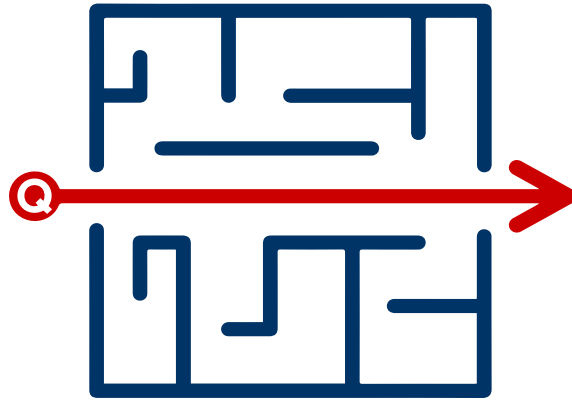

SkipQ DevOps – Proxima Centauri

Author:

Abdullah Zaman Babar   abdullah.zaman.babar.s@skipq.org

Synopsis:

This document is a guide to the techniques and procedures followed for the conclusion of the DevOps training offered at SkipQ.

For the construction and operation of production-grade web applications running across numerous regions, Infrastructure as Code is made use of. RESTful Python Apis are employed on AWS to monitor CloudWatch metrics, along with the application of Continuous Integration and Continuous Deployment for automation.

Table of Contents

1. Orientation Session

1.1. Introduction to Cloud Computing

Cloud computing is the delivery of diverse services through the Internet. It is the on-demand availability of computer system resources. It uses the internet for storing and managing data on remote servers and then access data via the internet.

1.2. Introduction to AWS

AWS (Amazon Web Services) is an extensive and advancing distributed computing platform introduced by Amazon that incorporates a combination of infrastructure as a service (IaaS), platform as a service (PaaS) and packaged software as a service (SaaS).

1.3. Introduction to DevOps

DevOps represents a change in IT culture, focusing on rapid IT service delivery through the adoption of agile, lean practices in the context of a system-oriented approach.
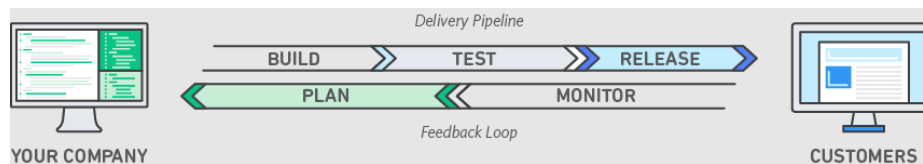
1.4. Introduction to AWS CDK

The AWS CDK is a new software development framework from AWS with the sole purpose of making it fun and easy to define cloud infrastructure in our favorite programming language and deploy it using AWS CloudFormation.
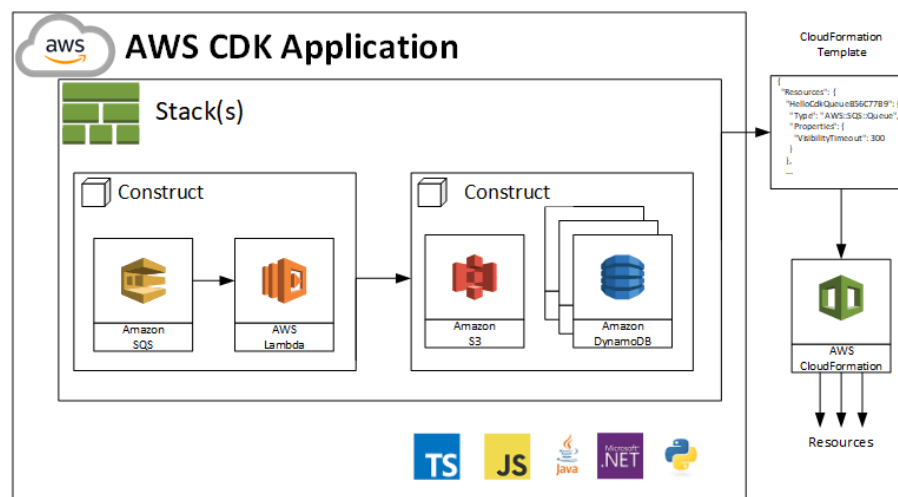
2. Sprint 1

2.1. Objective

Using AWS CDK to measure the availability and latency(delay) of a custom list of websites and monitor the results on a CloudWatch. Then setting up alarms on metrics when the prescribed thresholds are breached. Each alarm is published to SNS notifications which triggers a lambda function that writes the alarm information into DynamoDB.

2.2. Implementation

Creating the Hello World Lambda Function:

AWS Lambda is a serverless compute service that runs your code in response to events and automatically manages the underlying compute resources for you.

Against EC2, we now have Lambda to deploy or scale applications. It is a virtual function; it uploads and scales. It doesn't need to be provisioned, it only needs to provision code which is a function that will run on the cloud, and will auto provision resources file.

We are using the *Function* method of Lambda library. It defines the lambda functions.

```python
from aws_cdk import (
    core as cdk,
    aws_lambda as lambda_
)

# For consistency with other languages, `cdk` is the preferred import name for
# the CDK's core module.  The following line also imports it as `core` for use
# with examples from the CDK Developer's Guide, which are in the process of
# being updated to use `cdk`.  You may delete this import if you don't need it.
from aws_cdk import core


class PcRepoAzbStack(cdk.Stack):

    def __init__(self, scope: cdk.Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # The code that defines your stack goes here

        HWlambda = self.create_lambda("FirstHWLambda", "./resources", "lambda.lambda_handler")

    def create_lambda(self, newid, asset, handler):
        return lambda_.Function(self, id = newid,
                                runtime = lambda_.Runtime.PYTHON_3_6,
                                handler = handler,
                                code = lambda_.Code.asset(asset)
                                )
```

Then we define our lambda handler that takes an *event* and *context* as parameters.

```python
def lambda_handler(event, context):
    return "Hello {} {}".format(event['first_name'], event['last_name'])
```

Now we use the command *cdk synth* to synthesize our code into CloudFormation template and they we use *cdk deploy* to deploy it in our S3 bucket, where an intermediary resource will run our code.

```
(.venv) abdullahzamanskipq:~/environment/PCRepoAZB (master) $ cdk synth
[WARNING] @aws-cdk/aws-lambda.Code#asset is deprecated.
  use `fromAsset`
  This API will be removed in the next major release.
Resources:
  FirstHWLambdaServiceRole73D1B294:
    Type: AWS::IAM::Role
(.venv) abdullahzamanskipq:~/environment/PCRepoAZB (master) $ cdk deploy
[WARNING] @aws-cdk/aws-lambda.Code#asset is deprecated.
  use `fromAsset`
  This API will be removed in the next major release.
```

Now we test our code. The *code source* is our compute resource that runs our code from s3 bucket.
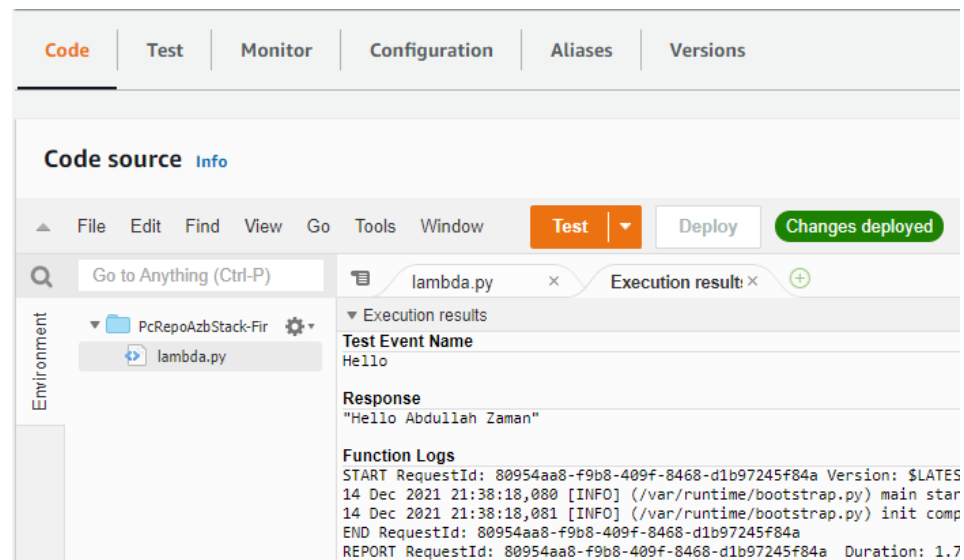
Creating the WebHealth Lambda Function:

In this we are working to achieve two metrics of a url that will determine its availability and latency.

We are using the *urllib3* library that will catch the status of our url and will determine the availability metric.

The latency metric is determined by using the *datetime* module through which we measure the different of the time the url is called and the response we receive.

```python
import datetime
import urllib3

URL_TO_MONITOR = "www.skipq.org"

# AWS: Add Debug Configuration | AWS: Edit Debug Configuration
def lambda_handler(events, context):
    values = dict()
    avail = get_availability()
    latency = get_latency()
    values.update({"availability":avail,"Latency":latency})
    return values

# AWS: Add Debug Configuration | AWS: Edit Debug Configuration
def get_availability():
    http = urllib3.PoolManager()
    response = http.request("GET", URL_TO_MONITOR)
    if response.status==200:
        return 1.0
    else:
        return 0.0

# AWS: Add Debug Configuration | AWS: Edit Debug Configuration
def get_latency():
    http = urllib3.PoolManager()
    start = datetime.datetime.now()
    response = http.request("GET", URL_TO_MONITOR)
    end = datetime.datetime.now()
    delta = end - start
    latencySec = round(delta.microseconds * .000001, 6)
    return latencySec
```

We can see the values returned for our metrics in *code source.*

Periodically monitoring the Metrics:

Currently our WebHealth lambda function is currently a one-time invocation, but being a DevOps engineer we want to periodically monitor our metrics and we want to schedule it to run every one minute.

Amazon EventBridge delivers a near real-time stream of system events that describe changes in AWS resources.

First, we have to schedule an *event*, and then will specify a *target* (i.e., lambda in our case) for our event. Then we specify a *rule* for applying the schedule to the target. We want our lambda to be invoked periodically to show us the availability and latency of our url in order to visualize our metrics graphically.

```python
from aws_cdk import (
    core as cdk,
    aws_lambda as lambda_,
    aws_events as events_,
    aws_events_targets as targets_,
    aws_iam
)

# For consistency with other languages, `cdk` is the preferred import name for
# the CDK's core module.  The following line also imports it as `core` for use
# with examples from the CDK Developer's Guide, which are in the process of
# being updated to use `cdk`.  You may delete this import if you don't need it.
from aws_cdk import core


class PcRepoAzbStack(cdk.Stack):

    def __init__(self, scope: cdk.Construct, construct_id: str, **kwargs) -> None:
        super().__init__(scope, construct_id, **kwargs)

        # The code that defines your stack goes here
        lambda_role = self.create_lambda_role()
        hw_lambda = self.create_lambda("FirstHWLambda", "./resources", "webhealth_lambda.lambda_handler", lambda_role)
        # We define the schedule, target and the rule for our lambda

        lambda_schedule = events_.Schedule.rate(cdk.Duration.minutes(1))
        lambda_target = targets_.LambdaFunction(handler=hw_lambda)
        rule = events_.Rule(self, "WebHealth_Invocation", description = "Periodic Lambda",
                            enabled=True, schedule=lambda_schedule, targets=[lambda_target])

def create_lambda_role(self):
    lambdaRole = aws_iam.Role(self, "lambda-role",
        assumed_by=aws_iam.ServicePrincipal('lambda.amazonaws.com'),
        managed_policies=[
            aws_iam.ManagedPolicy.from_aws_managed_policy_name('service-role/AWSLambdaBasicExecutionRole'),
            aws_iam.ManagedPolicy.from_aws_managed_policy_name('CloudWatchFullAccess'),
            ])
    return lambdaRole

def create_lambda(self, newid, asset, handler, role):
    return lambda_.Function(self, id = newid,
                            runtime = lambda_.Runtime.PYTHON_3_6,
                            handler = handler,
                            code = lambda_.Code.asset(asset),
                            role=role,
                            )
```
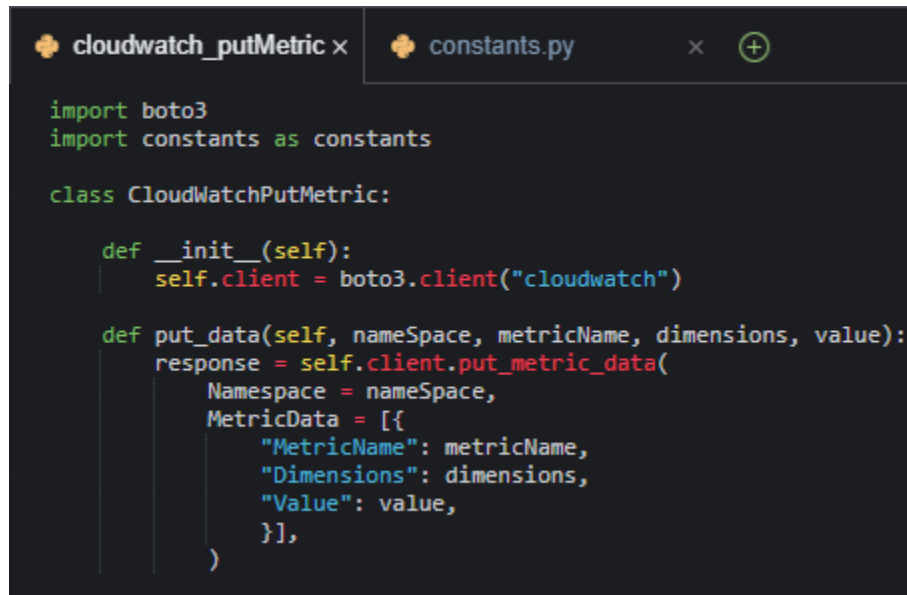
Publishing Metrics on CloudWatch:

CloudWatch is a monitoring service, where we can use functions that will help up set alarms and we can set metrics e.g., availability and latency. We can also use cloudwatch to graphically monitor our metrics and other things

We use *boto3* module to use its *CloudWatch client.*
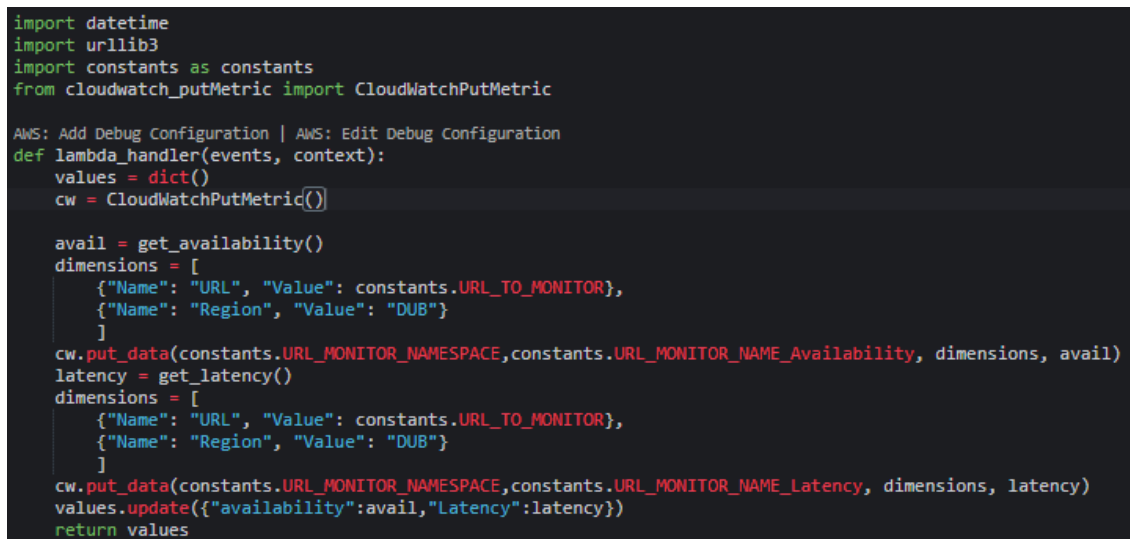
```python
cloudwatch_putMetric ×          constants.py          ×    ⊕

import boto3
import constants as constants

class CloudWatchPutMetric:

    def __init__(self):
        self.client = boto3.client("cloudwatch")

    def put_data(self, nameSpace, metricName, dimensions, value):
        response = self.client.put_metric_data(
            Namespace = nameSpace,
            MetricData = [{
                "MetricName": metricName,
                "Dimensions": dimensions,
                "Value": value,
            }],
        )
```

In the project stack we have an event that will schedule every one minute and every one minute it will call the *webhealth_lambda* function. Now every time *webhealth_lambda* function is called we want to take the metrics and put them on CloudWatch.

```python
import datetime
import urllib3
import constants as constants
from cloudwatch_putMetric import CloudWatchPutMetric

AWS: Add Debug Configuration | AWS: Edit Debug Configuration
def lambda_handler(events, context):
    values = dict()
    cw = CloudWatchPutMetric()

    avail = get_availability()
    dimensions = [
        {"Name": "URL", "Value": constants.URL_TO_MONITOR},
        {"Name": "Region", "Value": "DUB"}
        ]
    cw.put_data(constants.URL_MONITOR_NAMESPACE,constants.URL_MONITOR_NAME_Availability, dimensions, avail)
    latency = get_latency()
    dimensions = [
        {"Name": "URL", "Value": constants.URL_TO_MONITOR},
        {"Name": "Region", "Value": "DUB"}
        ]
    cw.put_data(constants.URL_MONITOR_NAMESPACE,constants.URL_MONITOR_NAME_Latency, dimensions, latency)
    values.update({"availability":avail,"Latency":latency})
    return values
```

The graphical results can be seen as follows in the CloudWatch console.
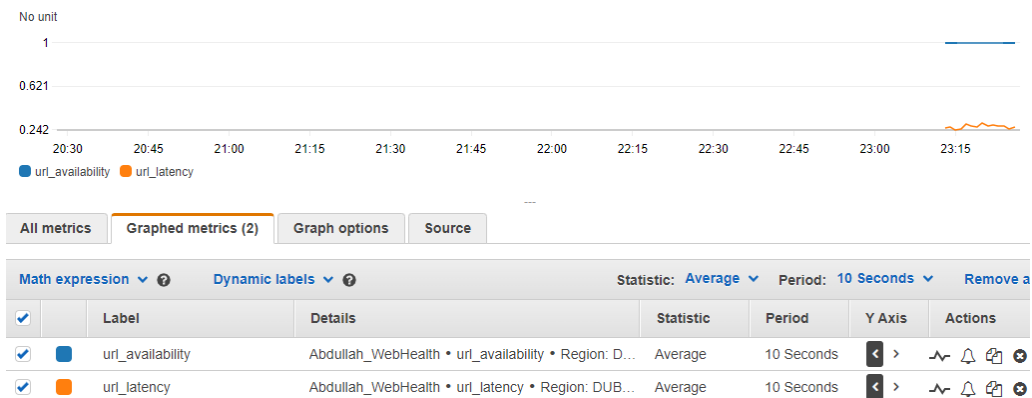
Setting up Alarms:

In our project stack file, we want to directly use CloudWatch, so the metrics we used are able to be imported in our infrastructure.

So, when we invoke lambda, the metrics defined in are automatically published on CloudWatch, but to set alarms on these metrics we need to have these metrics in our infrastructure in stack file.

We can call *Metric* on an existing metric which will be used in our infrastructure to set an alarm on top of it. We can't set an alarm on lambda it has to be done in our infrastructure. Lambda gets invoked every one minute, so every minute our alarm will be created which we do not want. We want our alarm to be continuous so every one minute the metric gets updated, and if the metric goes up the specified threshold alarm will be raised.

The infrastructure that we are setting is done in stack, lambda is the function that gets invoked. So, first the stack will set up and then lambda will be invoked. All the setting is done in stack.

We define our metric and then set an alarm on it in our stack file.

```
cloudwatch_.Metric()
cw.put_data(constants.URL_MONITOR_NAMESPACE,constants.URL_MONITOR_NAME_Availability, dimensions, avail)
cw.put_data(constants.URL_MONITOR_NAMESPACE,constants.URL_MONITOR_NAME_Latency, dimensions, latency)
```

We set up our dimensions as.

```
dimension = {"URL" : constants.URL_TO_MONITOR}
```

We then create the metrics.

```
availability_metric = cloudwatch_.Metric(namespace=constants.URL_MONITOR_NAMESPACE,
                                         metric_name=constants.URL_MONITOR_NAME_Availability,
                                         dimensions_map=dimension,
                                         period=cdk.Duration.minutes(1), label="Availability Meric")

latency_metric = cloudwatch_.Metric(namespace=constants.URL_MONITOR_NAMESPACE,
                                    metric_name=constants.URL_MONITOR_NAME_Latency,
                                    dimensions_map=dimension,
                                    period=cdk.Duration.minutes(1), label="Latency Meric")
```

We can see the results in CloudWatch console.

Alarms (18)	Hide Auto Scaling alarms	Clear selection	C	Create composite alarm	Actions ▼	Create alarm

	Name	▽	State	▽	Last state update	▽	Conditions	Actions
☐	PcRepoAzbStack-LatencyAlarm5394FC57-OT65L8RGAOTX		⊘ OK		2021-12-17 00:55:15		Latency Metric > 0.28 for 1 datapoints within 1 minute	No actions
☐	PcRepoAzbStack-AvailabilityAlarmE6EBAA96-1NKF0D0EGF15G		⊘ OK		2021-12-17 00:54:10		Availability Metric < 1 for 1 datapoints within 1 minute	No actions

SNS Topic – Email Subscription:

SNS is the *simple notification service* and it notifies the subscriber of a breach in threshold.

We will first set the *sns* and then we will add *sns-subscriptions* on it to subscribe to the sns topic.

We define a topic with *id* and the *scope* as parameters.
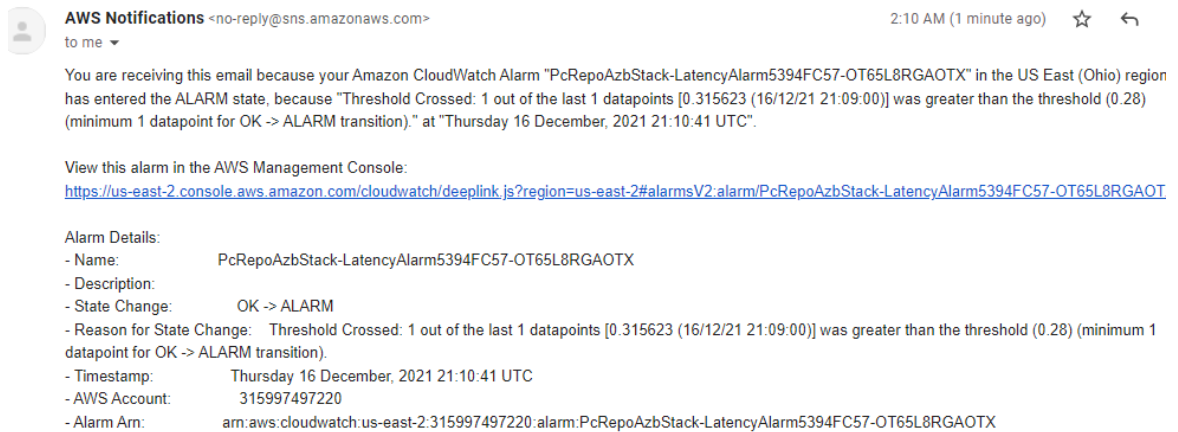
```
topic = sns.Topic(self, "WebHealthTopic")
topic.add_subscription(subscriptions_.EmailSubscription("abdullah.zaman.babar.s@skipq.org"))
```

We then add subscription to our alarms that we specified. These should be under the alarms defined.

```
availability_alarm.add_alarm_action(actions_.SnsAction(topic))
latency_alarm.add_alarm_action(actions_.SnsAction(topic))
```

We receive the alarm notification in our email.

AWS Notifications <no-reply@sns.amazonaws.com>   2:10 AM (1 minute ago)   ☆   ↩
to me ▾

You are receiving this email because your Amazon CloudWatch Alarm "PcRepoAzbStack-LatencyAlarm5394FC57-OT65L8RGAOTX" in the US East (Ohio) region
has entered the ALARM state, because "Threshold Crossed: 1 out of the last 1 datapoints [0.315623 (16/12/21 21:09:00)] was greater than the threshold (0.28)
(minimum 1 datapoint for OK -> ALARM transition)." at "Thursday 16 December, 2021 21:10:41 UTC".

View this alarm in the AWS Management Console:
https://us-east-2.console.aws.amazon.com/cloudwatch/deeplink.js?region=us-east-2#alarmsV2:alarm/PcRepoAzbStack-LatencyAlarm5394FC57-OT65L8RGAOT

Alarm Details:
- Name: PcRepoAzbStack-LatencyAlarm5394FC57-OT65L8RGAOTX
- Description:
- State Change: OK -> ALARM
- Reason for State Change: Threshold Crossed: 1 out of the last 1 datapoints [0.315623 (16/12/21 21:09:00)] was greater than the threshold (0.28) (minimum 1
datapoint for OK -> ALARM transition).
- Timestamp: Thursday 16 December, 2021 21:10:41 UTC
- AWS Account: 315997497220
- Alarm Arn: arn:aws:cloudwatch:us-east-2:315997497220:alarm:PcRepoAzbStack-LatencyAlarm5394FC57-OT65L8RGAOTX

Lambda Subscription:

We want that when we get notified, we want another lambda function to be notified which happens when an alarm is raised.

1. We need to add lambda as a subscriber to the sns topic, and a sns notification which is an event goes to the lambda.
2. When the lambda is triggered, it takes the cloudwatch alarm information as an event.
3. Now parse the alarm event, and write it in DynamoDB. Every time an alarm is raised add a row to the table

First, we create a table in our DynamoDB and make sure that once the table in created it doesn't not try to overwrite it.

```
def create_table(self, t_name):
    try:
        return db.Table(self, id="Table", table_name=t_name,
                partition_key=db.Attribute(name="AlarmDetails", type=db.AttributeType.STRING))
    except:
        pass
```

We the create a lambda function for our database.

```
# Create lambda
db_lambda = self.create_lambda("DynamoLambda", "./resources", "dynamodb_lambda.lambda_handler", lambda_role)
dynamo_table.grant_read_write_data(db_lambda)
# We define the schedule, target and the rule for our lambda
```

Then we create a function to parse through the event.

```python
import boto3
import json

AWS: Add Debug Configuration | AWS: Edit Debug Configuration
def lambda_handler(event, context):
    #write to database
    #resource = boto3.resource('dynamodb')
    client_ = boto3.client('dynamodb')
    info = event['Records'][0]['Sns']['MessageId']
    info = json.loads(info)
    item = {
        'AlarmDetails': {'S': info}
        }
    client_.put_item(TableName="AbdullahTable", Item=item)
```

The entries recorded are as follows.

▸ **AbdullahTable**   View table details
Expand to query or scan items.

tems returned (3)   ⟳   Actions ▾   Create item

                                              ‹ 1 ›   ⚙ ⛶

☐   AlarmDetails   ▽

☐   1271ac12-a445-as12-341d-122acd133a23

☐   81779123-e12a-32a2-171a-c199ad234ad1

☐   23378139-ad24-4132-31e1-ad231a331221

3. Sprint 2

3.1. Objective

Creating a multi-stage pipeline that has a beta, gemma and production stage using AWS CDK. Then adding unit and integration test to the defined stages. The project is concluded by automating a rollback to the previous version if the metric is in alarm.

3.2. Implementation

Continuous Integration and Continuous Delivery:

Continuous Integration is a process where developers regularly merge their changes in code into a central repository, after which automated tests and builds are run. It focuses on smaller commits and smaller code changes to integrate.

Continuous Delivery is a process where code changes are automatically built, tested and prepared for production release.

The point of *continuous delivery* is NOT to apply every change to production immediately, but to ensure that every change is ready to go to production.

Continuous Deployment:

With *Continuous Deployment* revisions are deployed to a production environment automatically without explicit approval from a developer, making the entire software release process automated.

Pipeline:

CI/CD can be pictured as a pipeline, where new code is submitted on one end, tested over a series of stages and then published as production-ready code.
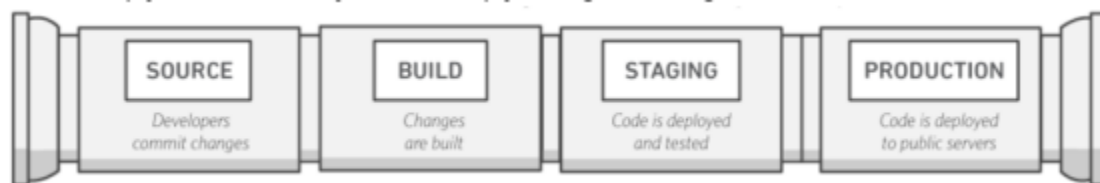
In *source*, as we do on GitHub, our code is always up to date and then commit our changes and our code is ready. This step has to be done manually.

In *build* we automate the cdk synth and cdk deploy process so that it automatically does that itself. The code is synthesized into CloudFormation format.

In *staging* we don't want our code to be put live instantly. There might be a lot of problems in it. So before putting it live, we have to test it. We deploy it in a test environment to better our results.

Once we are satisfied with our code, we can deploy it to public servers which is the *production* stage.

Creating a Pipeline:

The first step is to specify the location of our source code. In our case it's the GitHub's central repository. We have to specify that we need to get our code from GitHub. Then we specify how to build the source code that we got from GitHub. After that we need to deploy in a test environment, and then finally in the production area.

We have to use the *aws_cdk.pipelines* module for our program.

So, in our cloud instance we need to create a file called *piperline_stack.py* where we will first define the source to our code. We will use the *CodePipelineSource* class which has multiple sources but we'll use *git_hub,* we can see that here in aws construct library.

In *git_hub* method we need to specify the *repo_string* which is the location for our GitHub and then the *branch* which will be *main* in our case. We then need to provide *authentication*, so when we are doing it through code, we need to find some way to authenticate unlike doing it manually using the username and token. So, for authentication we have to save GitHub's access token in a service called *Secrets Manager*.

Now we also have to define the *trigger* parameter in our *git_hub* method, which specifies when will our pipeline start working. So, we see that the trigger has a few values. We'll use *POLL* that periodically checks the code for changes, and whenever it detects changes it will start the pipeline.

```python
from aws_cdk import(
    core,
    aws_codepipeline_actions as cpactions,
    pipelines
)
from pc_repo_azb.infra_stage import InfraStage

class PipelineStack(core.Stack):
    def __init__(self, scope:core.Construct, id:str, **kwargs):
        super().__init__(scope, id, **kwargs)


        source = pipelines.CodePipelineSource.git_hub(repo_string="abdullah2021skipq/ProximaCentauri", branch='main',
            authentication=core.SecretValue.secrets_manager('Abdullah_token'),
            trigger=cpactions.GitHubTrigger.POLL
            )
```

Now we have our code so it now needs to build. It basically needs to do *cdk synth* and *cdk deploy*. We'll specify how we'll build our code. We have to tell our code which dependencies are required to run our code, which we specify in our *requirements.txt* file and after that we have to do synthesize and deploy. We have to use the *ShellStep* module from the *pipelines* library. This will help us run the shell commands.

So, we name it e.g., "synth", then we define the *input* which we have defined as source above. The next are the *commands* in a list that we need to execute in our shell that are required to build our source code.

So we need to define where our cdk.out file will be generated in our pipeline which will be the *primary_output_directory* parameter.

Now remember that whenever we do *cdk synth* we get a folder called *cdk.out.* When we compile our code we get some metadata and that is stored in this folder. If we open the *template.json* file we have all the information in it like the name of database we used etc and when we do *cdk deploy* it reads this template file. We don't put this folder on GitHub because whenever we do cdk synth it creates this folder, so we can put this file in our *.gitignore***.**

We define our pipeline using the *CodePipeline* method.

```
synth = pipelines.ShellStep('synth', input=source,
        commands = [
        "cd AbdullahZaman/sprint2/PCRepoAZB", "pip install -r requirements.txt", "npm install -g aws-cdk", "cdk synth"],
    primary_output_directory="AbdullahZaman/sprint2/PCRepoAZB/cdk.out"
        )


pipeline = pipelines.CodePipeline(self, 'Pipeline', synth=synth)
```

The *app.py* file is our entry point. We have imported our stack file and then instantiated it in 3rd line which means that we are creating an instance of that class and we are passing it an object called *core.App()* and then we are synthesizing it.

We can also have other arguments to specify what our environment is in which this code will run. If it is commented out then by default it will use our current environment but we can change it to any region we want.

Now that we are using code pipelines we need to change the entry point. We are not running our code from the cloud9, we have created a pipeline where we have indicated the source code and how to build it. So, we need to change our **app.py** file as well. Now the entry point should be the pipeline. When we deploy our pipeline all the intermediary steps will be taken care of automatically.

Up till now we have worked in out stack file and we use *app.py* as the entry point where we started. So we will use the file *pipeline_stack.py* where we have already

defined our source and synth. So, for this we also need to define a class e.g. PipelineStack and then initialize it.

Pipeline is also a stack.

Now we don't need to call our infrastructure stack in *app.py*, we need to call our pipeline stack here.

And then we instantiate it and pass the arguments **app**, the name of the pipeline and we can specify the region where we have to run it.

```python
#!/usr/bin/env python3
import os

from aws_cdk import core as cdk


from aws_cdk import core

#from pc_repo_azb.pc_repo_azb_stack import PcRepoAzbStack1
from pc_repo_azb.pipeline_stack import PipelineStack


app = core.App()


PipelineStack(app, 'PipelineStackAbdullah3', env=core.Environment(account='315997497220', region='us-east-2'))
app.synth()
```

Staging:

We have to specify to deploy it in the region. Now we need to instantiate our infrastructure stack in our pipeline stack and we tell it to run our instantiated object in this environment.

We can run our program in multiple regions and we have to tell our pipeline these details.

So let's say running our program in one region is seen as one stage. So our test and production environment are different stages.

For this we'll create another file called *infra_stage.py* and similarly as before we define a class and now it uses *cdk.stage* as its super class.

```python
from aws_cdk import core as cdk

from aws_cdk import (core,
                     aws_events,
                     aws_events_targets,
                     aws_iam, aws_sns as sns,
                     aws_cloudwatch as cw,
                     aws_sns_subscriptions as subs,
                     aws_cloudwatch_actions as cw_actions,
                     aws_dynamodb as dynamodb,
                     aws_lambda as lambda_
                     )

from pc_repo_azb.pc_repo_azb_stack import PcRepoAzbStack1

class InfraStage(cdk.Stage):
    def __init__(self, scope: cdk.Construct, construct_id: str, **kwargs):
        super().__init__(scope, construct_id, **kwargs)

        infra_stack = PcRepoAzbStack1(self, 'infraStackAZB')
```

We then specify our test and production stages in our *pipeline_stack.py* file

```python
beta = InfraStage(self, "AbdullahZBeta", env={
'account': '315997497220',
'region': 'us-east-2'
})

gemma = InfraStage(self, "AbdullahZGemma", env={
'account': '315997497220',
'region': 'us-east-2'
})

production = InfraStage(self, "AbdullahZProduction", env={
'account': '315997497220',
'region': 'us-east-2'
})
```

Then we use the *add_stage* method to deploy our stages.

```python
pipeline.add_stage(beta)
pipeline.add_stage(gemma)
pipeline.add_stage(production)
```

Bootstrapping:

Before we use any resources, we need to provision them which is called *Bootstrapping*. You have to tell aws that you need these resources.

So if we use the *cdk bootstrap* command only, by default it will bootstrap the region in which we are working. We can also specify a different region as well.

To do Bootstrapping we need to push our code to GitHub and then run the following command. We can also specify a *qualifier* name and a *toolkit* name to overwrite the default and in the end specify the region at which we are going to bootstrap it.

```
abdullahzamanskipq:~/environment/ProximaCentauri/AbdullahZaman/sprint2/PCRepoAZB (main) $ cdk bootstrap --qualifier abdullah --toolkit-stack
-name abdullahtoolkit14 aws://315997497220/us-east-2
```

Then we use the following command to deploy our pipeline.

```
abdullahzamanskipq:~/environment/ProximaCentauri/AbdullahZaman/sprint2/PCRepoAZB (main) $ cdk deploy PipelineStackAbdullah3
```

We can see our pipeline in the *CodePipeline* service.

PipelineStac
kAbdullah3-
Pipeline985
0B417-
138HNX74K
QIPJ

⊘ Succeeded

abdullah2021skipq_ProximaCentauri
– d035674d ☑: Merge branch 'main' of
https://github.com/abdullah2021skipq
/ProximaCentauri into…

2 hours ago

Manual Approval Step:

We have added the test stages and now we can add the production stage but before that we need to add a manual approval step which will allow the user to either approve or reject the deployment of the code to the production stage.

We can do that by using the *ManualApprovalStep* method of the *pipelines* class.

The *add_stage* has two arguments *post* and *pre*. If we don't add it, it means that it doesn't need to do anything before or after adding the stage.

Let's say before adding the production stage we want to add the manual approval step, so we add the *pre* argument. When our code will go from gemma to production stage it will request a manual approval on our pipeline and it will deploy only if we specify manually.

```
pipeline.add_stage(production, pre=[pipelines.ManualApprovalStep("Approve Changes and deploy to Production")])
```

Unit and Integration Test:

We can add various tests to check if the infrastructure we designed is working as intended or not.

Here we've added a unit test in the *unit_tests* directory to check if we are getting the two lambda functions that we created. We'll use *pytest* library for our tests.

```python
import pytest
from aws_cdk import core
from pc_repo_azb.pc_repo_azb_stack import PcRepoAzbStack1


def test_lambda_count():

    app=core.App()
    PcRepoAzbStack1(app,'Unittest')
    template=app.synth().get_stack_by_name('Unittest').template
    fun=[resource for resource in template['Resources'].values() if resource['Type']=='AWS::Lambda::Function']

    assert len(fun)==2
```

We also need to make the following changes to the test file in the *unit* directory for our unit test to function.

```python
import aws_cdk.assertions as assertions

from pc_repo_azb.pc_repo_azb_stack import PcRepoAzbStack1


# example tests. To run these tests, uncomment this file along with the example
# resource in pc_repo_azb/pc_repo_azb_stack.py
def test_sqs_queue_created():
    app = core.App()
    stack = PcRepoAzbStack1(app, "pc-repo-azb")
    template = assertions.Template.from_stack(stack)

#     template.has_resource_properties("AWS::SQS::Queue", {
#         "VisibilityTimeout": 300
#     })
#  pass
```

Now in our pipeline stack we add the tests that we have defined. So, our tests are not going to run in our local environment and it needs to install all the requirements in the environment where it's going to run and then it needs to do the unit and integration test. We can add the tests either as pre or post deploying.

```python
unitTest = pipelines.CodeBuildStep(
    'unit_tests',input=source,
    commands=["cd AbdullahZaman/sprint2/PCRepoAZB/","pip install -r requirements.txt", "npm install -g aws-cdk", "pytest unit_tests"]
    )

integrationTest = pipelines.CodeBuildStep(
    'integration_tests',input=source,
    commands=["cd AbdullahZaman/sprint2/PCRepoAZB/","pip install -r requirements.txt", "npm install -g aws-cdk", "pytest integration_tests"]
    )

pipeline.add_stage(beta, pre=[unitTest])
pipeline.add_stage(gemma, pre=[integrationTest])
pipeline.add_stage(production, pre=[pipelines.ManualApprovalStep("Approve Changes and deploy to Production")])
```

We can also run our tests locally to check if our code is working. We only specify the folder for *pytest* and it runs all the tests inside it.

```
(main) $ pytest unit_tests/
```

Rollback:

Let's say we make changes to our lambda function.

Now if you visit *Cloudwatch* we can see that apart from the custom metrics we defined, we have access to the pre-defined metrics that we can use.

AWS Namespaces		
CodeBuild 7,817 Metrics	DocDB 142 Metrics	DynamoDB 639 Metrics
EBS 747 Metrics	EC2 1,410 Metrics	ES/OpenSearchService 254 Metrics

If we visit the *Lambda* metric we say the following, then we select *By Function Name* and we find the following. We can see the *Duration* metric which tells how much time is being taken by our function when running. So we can set an alarm if our lambda function is taking more time than set in the threshold and we can set a rollback on that which can roll back to the previous version of our lambda before the changes were made.

So we have to read these metrics and we'll do that in our infrastructure stack file. To read the metric we'll have to define the name of the metric in the AWS Namespace as *AWS/Lambda.*

To get the name of the lambda we can use the *function_name* attribute of the *Function* method of the *aws_lambda* class.

Now we want to set an alarm on our metric if it crosses a threshold. And when the alarm is generated, we need to do a rollback to a previous version. To do that we use *aws_codedeploy* class and the *LambaDeploymentGroup* method of this class. This method will help us in deployment and it helps us with shifting traffic. Let's say we deploy a new lambda, so with the use of this function aws is not going to start acting on the new lambda. It will divert 10-20% of the traffic to the lambda that we have changed and the rest will remain in the previous lambda and on that small portion of the traffic we can have certain tests if the alarm is generated or not and if it's happening then do a rollback.

The default *deployment_config* is 10% with 5 minutes and if it works fine it will move to the new version. We can see that in *LambdaDeploymentConfig*.

In *auto_rollback* parameter we are going to define how to perform the rollback. In the *alias* parameter we will have two versions, old and new.

This *alias* can be created using the *aws_lambda* class.

This portion of code will be placed in our infrastructure stack file.

```python
# ROLLBACK to the previous version if an alarm is raised
metric_roll = cloudwatch_.Metric(namespace='AWS/Lambda', metric_name='Duration',
                        dimensions_map={'FunctionName':hw_lambda.function_name},
                        period= cdk.Duration.minutes(1))

alarm_roll = cloudwatch_.Alarm(self, id="RollBackAlarm", metric= metric_roll,
                        comparison_operator=cloudwatch_.ComparisonOperator.GREATER_THAN_THRESHOLD,
                        datapoints_to_alarm=1,
                        evaluation_periods=1,
                        threshold=3000)          # 3000 ms = 3 sec

alarm_roll.add_alarm_action(actions_.SnsAction(topic))
alias = lambda_.Alias(self, "LambdaAlias",alias_name="Lambda",version=hw_lambda.current_version)

codedeploy.LambdaDeploymentGroup(self, "WebHealth Lambda", alias=alias,
                        deployment_config=codedeploy.LambdaDeploymentConfig.LINEAR_10_PERCENT_EVERY_1_MINUTE,
                        alarms=[alarm_roll]
)
```

4. Sprint 3

4.1. Objective

Building a public CRUD API gateway endpoint to perform the create, update, delete and read the list of websites. To achieve this task first move the json file containing the predefined URLs from the S3 bucket to a DynamoDB database. Then implement the CRUD REST commands on the DynamoDB entries. After that extend the tests in each stage to cover the CRUD operations.

4.2. Implementation

API Gateway:

An *API gateway* is an API management tool between a collection of backend services and a client. It acts as a reverse proxy that accepts all application programming interface (API) calls aggregating the various services required to fulfill them, and returns a useful result.

Amazon API Gateway is a completely managed service that makes it comfortable for developers to create and secure APIs at any scale. APIs act as the front door for applications to access data from our backend services.

Using API Gateway, we can build RESTful and WebSocket APIs that enable real-time communication applications. API Gateway supports containerized and serverless workloads and web applications.

REST API:

A REST API (also known as RESTful API) is an application programming interface (API or web API) that conforms to the constraints of REST architectural style and allows for interaction with RESTful web services.

When a client request is made though a RESTful API, it transfers a state of the resource to the endpoint. This information is delivered in one of several formats via HTTP: JSON (JavaScript Object Notation), HTML, XLT, Python, PHP, or plain text. JSON is the most generally popular file format to use because, despite its name, it's language-agnostic, as well as readable by both humans and machines.

Writing Json data to DynamoDB:

First, we create a table using *boto3* to store our entries from the Json file that is present in our S3 bucket. To create a table we use the *create_table* method as follows:

```python
def create_sprint3_table():
    client_ = boto3.resource('dynamodb')
    try:
        table = client_.create_table(
            TableName='AbdullahSprint3',
            KeySchema=[
                {
                    'AttributeName': 'URL_ADDRESS',
                    'KeyType': 'HASH'   # Partition key
                }
            ],
            AttributeDefinitions=[
                {
                    'AttributeName': 'URL_ADDRESS',
                    'AttributeType': 'S'
                }

            ],
            ProvisionedThroughput={
                'ReadCapacityUnits': 10,
                'WriteCapacityUnits': 10
            }
        )
        time.sleep(5)
        #table.grant_read_write_data()
    except:
        pass
```

Note that we use the *sleep* method because it takes some time to create the table in the DynamoDB.

After that we write a function to write our data to the table we have just created.

```python
def putting_sprint3_data():
    URLs = s3bucket.read_file("abdullahzamanbucket", "urlsList.json")
    client_ = boto3.client('dynamodb')
    for U in URLs:
        item = {
            'URL_ADDRESS': {'S': U}
        }
        print(item)
        client_.put_item(TableName="AbdullahSprint3", Item=item)
```

Now we call these functions into our infrastructure stack.

```python
sprint3_dynamo.create_sprint3_table()
sprint3_dynamo.putting_sprint3_data()
```

We can see the items in our database table.

|	URL_ADDRESS
☐	www.dawn.com
☐	www.espn.com
☐	www.skipq.org

Creating the API Gateway:

We use the *LambdaRestApi* method provided by the *aws_apigateway* class to create our API gateway.

```python
def create_gateway(self, name, handler):
    return gateway.LambdaRestApi(self, id=name, handler=handler,
                                 proxy=False
    )
```

We then add different resources and methods to our API gateway.

```python
api = self.create_gateway('AzbApi',sprint3_lambda)
api_resource1 = api.root.add_resource("health")
api_resource1.add_method("GET") # GET /health
api_resource2 = api.root.add_resource("url")
api_resource2.add_method("GET")
api_resource2.add_method("POST")
api_resource2.add_method("DELETE")
api_resource2.add_method("PATCH")
api_resource3 = api.root.add_resource("urls")
api_resource3.add_method("GET")
```

We can see the new API built in our console.
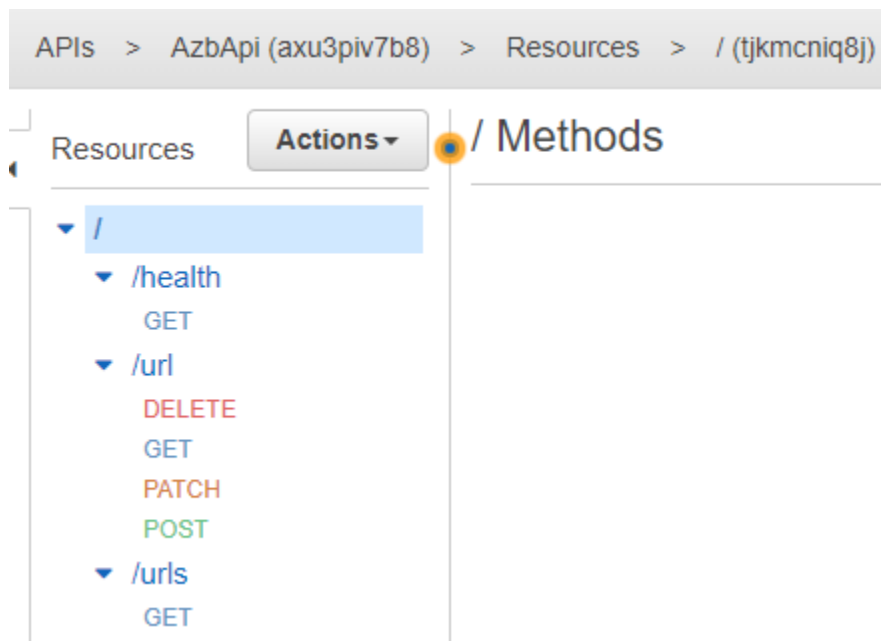
Creating the Lambda Function:

We now define some global variables to be used by our lambda hander function. These variables include the methods of the API Gateway and the path to which they are intended.

```python
import boto3
import json
import logging
from custom_encoder import CustomEncoder


logger = logging.getLogger()
logger.setLevel(logging.INFO)

dynamodbTableName = 'AbdullahSprint3'
dynamodb = boto3.resource('dynamodb')
table = dynamodb.Table(dynamodbTableName)

getMethod = 'GET'
postMethod = 'POST'
patchMethod = 'PATCH'
deleteMethod = 'DELETE'
healthPath = '/health'
urlPath = '/url'
urlsPath = '/urls'
```

Now we use the conditional statements to return the responses for our CRUD functionality in our lambda handler function.

```python
def lambda_handler(event, context):
    logger.info(event)
    httpMethod = event['httpMethod']
    path = event['path']
    if httpMethod == getMethod and path == healthPath:
        response = buildResponse(200)
    elif httpMethod == getMethod and path == urlPath:
        response = getItem(event['queryStringParameters']['URL_ADDRESS'])
    elif httpMethod == getMethod and path == urlsPath:
        response = getItems()
    elif httpMethod == postMethod and path == urlPath:
        response = saveItem(json.loads(event['body']))
    elif httpMethod == patchMethod and path == urlPath:
        requestBody = json.loads(event['body'])
        response = modifyItem(requestBody['URL_ADDRESS'],requestBody['updateValue'])
    elif httpMethod == deleteMethod and path == urlPath:
        requestBody = json.loads(event['body'])
        response = deleteItem(requestBody['URL_ADDRESS'])
    else:
        response = buildResponse(404, 'Not Found')

    return response
```

We then define the *getItem* function to output the URL address present in our database.

```python
def getItem(URL_ADDRESS):
    try:
        response = table.get_item(
                Key={
                    'URL_ADDRESS': URL_ADDRESS
                }
        )
        if 'Item' in response:
            return buildResponse(200, response['Item'])
        else:
            return buildResponse(404, {'Message': 'URL_ADDRESS: %s not found' % URL_ADDRESS})
    except:
        logger.exception('Sprint3 getItem() Exception')
```

Then we define the *getItems* function to output all the items present in our database.

```python
def getItems():
    try:
        response = table.scan()
        result = response['Items']

        while 'LastEvaluatedKey' in response:
            response = table.scan(ExclusiveStartKey=response['LastEvaluatedKey'])
            result.extend(response['Item'])

        body = {
            'urls': result
        }
        return buildResponse(200, body)

    except:
        logger.exception('Sprint3 getItems() Exception')
```

After that we define a *saveItem* function to store a new URL address in our database.

```python
def saveItem(requestBody):
    table.put_item(Item=requestBody)
    try:
        body = {
            'Operation': 'SAVE',
            'Message': 'SUCCESS',
            'Item': requestBody
        }
        return buildResponse(200, body)
    except:
        logger.exception('Sprint3 saveItem() Exception')
```

To update the database, we define a function called *modifyItem*. It will delete the desired item and store a new item in its place.

```python
def modifyItem(URL_ADDRESS, updateValue):
    try:
        table.put_item(Item={"URL_ADDRESS": updateValue})
        response = table.delete_item(
                Key={
                    'URL_ADDRESS': URL_ADDRESS
                },
                ReturnValues='ALL_OLD'
        )
        body = {
            'Operation': 'DELETE',
            'Message': 'SUCCESS',
            'deletedItem': response
        }
        return buildResponse(200, body)
    except:
        logger.exception('Sprint3 modifyItem() Exception')
```

We also define a *deleteItem* function to remove the desired URL from our database.

```python
def deleteItem(URL_ADDRESS):
    try:
        response = table.delete_item(
                Key={
                    'URL_ADDRESS': URL_ADDRESS
                },
                ReturnValues='ALL_OLD'
        )
        body = {
            'Operation': 'DELETE',
            'Message': 'SUCCESS',
            'deletedItem': response
        }
        return buildResponse(200, body)
    except:
        logger.exception('Sprint3 modifyItem() Exception')
```

The last function we define is called *buildResponse* and it will return a status code of 200 if the CRUD operations are successful or not.

```python
def buildResponse(statusCode, body=None):
    response = {
        'statusCode': statusCode,
        'headers': {
            'Content-Type': 'application/json',
            'Access-Control-Allow-Origin': '*'
        }
    }
    if body is not None:
        response['body'] = json.dumps(body, cls=CustomEncoder)
    return response
```

Testing the API Gateway:

We can test our API using the console. To do that go to your created API and select the method to test. In this case let's say we want to test the *GET* method of our *urls* resource. Click the *Test* button and that will give us all the urls present in our DynamoDB database.

No client certificates have been generated.

Request Body

Request Body is not supported for GET methods.

⚡ Test

Request: /urls

Status: 200

Latency: 870 ms

Response Body

```
{
  "urls": [
    {
      "URL_ADDRESS": "www.dawn.com"
    },
    {
      "URL_ADDRESS": "www.espn.com"
    },
    {
      "URL_ADDRESS": "www.skipq.org"
    }
  ]
}
```

Unit Test:

Unit testing means testing individual modules of an application in isolation (without any interaction with dependencies) to confirm that the code is doing things right.

We can define a unit test to test our resources. For example, we want to test if our stack creates the three lambda functions or not, for that we can define a method as follows.

```python
def test_lambda_count():

    app=core.App()
    PcRepoAzbStack1(app,'Unittest')
    template=app.synth().get_stack_by_name('Unittest').template
    fun=[resource for resource in template['Resources'].values() if resource['Type']=='AWS::Lambda::Function']

    assert len(fun)==3
```

We can also check for the number of alarms being created by our stack as follows.

```python
def test_alarm_count():

    app=core.App()
    PcRepoAzbStack1(app,'Unittest')
    template=app.synth().get_stack_by_name('Unittest').template
    fun=[resource for resource in template['Resources'].values() if resource['Type']=='AWS::CloudWatch::Alarm']

    assert len(fun)==3
```

Integration Test:

Integration testing means checking if different modules are working fine when combined together as a group.

First, we declare some global variables that we are going to need in our test functions.

```python
import requests
import pytest

url = "https://uy6n152jga.execute-api.us-east-2.amazonaws.com/prod"
item = {"URL_ADDRESS": "www.test123.com"}
item1 = {
        "URL_ADDRESS": "www.test123.com",
        "updateValue": "www.test123456.com"
        }
```

Then we define the test function to test if the API Gateway address provided is functional. If the status code returned is 200 then the response is successful.

```
def test_health():
    response = requests.get(url+"/health")
    statusCode = response.status_code
    assert statusCode == 200
```

We can also define separate test functions to check the POST, DELETE, GET and PATCH methods of our API Gateway.

```
def test_url_post():
    response = requests.post(url+"/url",json=item)
    statusCode = response.status_code
    assert statusCode == 200

def test_url_delete():
    response = requests.delete(url+"/url",json=item)
    statusCode = response.status_code
    assert statusCode == 200

def test_url_patch():
    requests.post(url+"/url",json=item)
    response = requests.patch(url+"/url",json=item1)
    statusCode = response.status_code
    assert statusCode == 200
```

5. Sprint 4

5.1. Objective

Building a Front-end user interface for the CRUD API using REACTJS. The user interface will allow users to see and search the DynamoDB database and will load Urls with pagination.

The login system will be enabled through React with authentication using "AWS Cognito".

5.2. Implementation

Initializing our React App:

To start working on our web application in React we need to download Nodejs. To do that we use the following command:

```
sudo apt-get install nodejs
```

Then we initialize the React app using the following lines of code:

```
npm init react-app reactjs
▶ npm run start
```

Searching Urls:

First, we need to define a function that will fetch the data from the API Gateway. Let's call the function *clickHandler2;* it will use the *axios* module to store the response. If the response is collected without any errors that a success statement is generated. In case of error an error statement will be generated.

In the return call we create a button and an input box that will store any text input in a variable.

To download the *axios* dependency we run the following command:

```
npm install axios
```

So, we start by initializing a React class as:

```
import React, { Component } from 'react';
import axios from 'axios';

class SearchUrl extends Component {
    constructor(props) {
        super(props);

        this.state = {
            value : '',
            result : '',
        }
    }

    getValue = (event) => { // it raises event on every letter entered so we want the last event which is the complete word
        this.setState({value : event.target.value});
    };
```

Then we define our function.

```
clickHandler2 = () => { //On click we store the last event in a variable
    const value = this.state.value;
    console.log(value)
    axios.get('https://l2qs966u0c.execute-api.us-east-2.amazonaws.com/prod/url?URL_ADDRESS='+value)
    .then((response) => {
        this.setState({result: 'The Url '+value+' Exists in the DataBase'});
        //console.log(this.state.result)
    })
    .catch((error) => {
        this.setState({result: value+' NOT FOUND'});
        //console.log(this.state.result)
    })
};
```

The return call will look similar to the following:

```
        render() {
            return (
                <div>
                    <input type="text" onChange={this.getValue}/>
                    <button onClick={this.clickHandler2}>Search URL</button>
                    {
                        <h1>{this.state.result}</h1>
                    }
                </div>
            );
        }
    }
    SearchUrl.propTypes = {

    };

    export default SearchUrl;
```

Calling the Database with Pagination:

In our *clickHandler* function we will call the API Gateway using the *fetch* method and we will use the *paginate* method to paginate the data.

First, we need to install the *paginate-array* module.

```
npm install paginate-array
```

We define our class as:

```
import React, { Component } from 'react';
import paginate from 'paginate-array';

class AllUrls extends Component {
  constructor(props) {
    super(props);

    this.state = {
      posts: [],
      size: 3,
      page: 1,
      currPage: null
    }

    this.previousPage = this.previousPage.bind(this);
    this.nextPage = this.nextPage.bind(this);
  }
}
```

Then we define our *clickHandler* function.

```
clickHandler1 = () => {
  fetch(`https://l2qs966u0c.execute-api.us-east-2.amazonaws.com/prod/urls`)
    .then(response => response.json())
    .then(posts => {
      const { page, size } = this.state;

      const currPage = paginate(posts, page, size);

      this.setState({
        ...this.state,
        posts,
        currPage
      });
    });
}
```

Now we need to show the functionality of our *Next Page* and *Previous Page* buttons.

```
previousPage() {
  const { currPage, page, size, posts } = this.state;

  if (page > 1) {
    const newPage = page - 1;
    const newCurrPage = paginate(posts, newPage, size);

    this.setState({
      ...this.state,
      page: newPage,
      currPage: newCurrPage
    });
  }
}

nextPage() {
  const { currPage, page, size, posts } = this.state;

  if (page < currPage.totalPages) {
    const newPage = page + 1;
    const newCurrPage = paginate(posts, newPage, size);
    this.setState({ ...this.state, page: newPage, currPage: newCurrPage });
  }
}
```

We then use the *map* method in the return call to parse through all the items present in the database and presents three items as specified on each page.

```
render() {
  const { page, size, currPage } = this.state;

  return (
    <div>
      <button onClick={this.clickHandler1}>GET URLs</button>
      <div>page: {page}</div>
      <div>size: {size}</div>
      {currPage &&
        <ol>
          {currPage.data.map(post => <li style={{padding: "10px"}}><a href={"https://"+post.URL_ADDRESS}>{post.URL_ADDRESS}</a></li>)}
        </ol>
      }
      <button onClick={this.previousPage}>Previous Page</button>
      <button onClick={this.nextPage}>Next Page</button>
    </div>
  )
}
}

export default AllUrls;
```

Calling the components in App.js file:

App.js is the main file for our React application. Here we call the two classes that we have created with some styling to our page.

We import the required libraries in the file.

```
import React, { Component } from 'react';
import './App.css';
import SearchUrl from './components/SearchUrl';
import AllUrls from './components/AllUrls';

document.body.style = 'background: #e7feff;';
```

After that we can define style components for our application.

```
class App extends Component{
  render() {
    const mystyle1 = {
        position: "absolute",
        top: "30%",
        marginTop: "-50px",
        width: "200px",
        height: "200px",
    };

    const mystyle2 = {
      position: "absolute",
      bottom: "30%",
      left: "75%",
      color: "#1A5276",
      fontSize: "20px",
    };
```

In the return call we call the classes that we have created to call the database.

```
    return (
        <div className="App">
          <h1 style = {{color: '#3C565B'}}>SPRINT 4: User Interface for CRUD API</h1>
          <div style={mystyle1}>
            <AllUrls></AllUrls>
          </div>
          <p></p>
          <SearchUrl></SearchUrl>
          <p></p>
          <div style={mystyle2}>
            <h4>Designed by Abdullah Zaman Babar</h4>
            <p style={{fontSize: "17px"}}>Email: <a href="mailto:abdullah.zaman.babar.s@skipq.org">abdullah.zaman.babar.s@skipq.org</a></p>
          </div>
        </div>
    );
  }
}

export default App;
```

Aws Amplify:

AWS Amplify is a set of tools that lets frontend web and mobile developers quickly and easily build full-stack applications on AWS.

With Amplify, you can configure a web or mobile app backend, connect your app in minutes, visually build a web frontend UI, and easily manage app content outside the AWS console.

So first we need to push our code to the remote repository.
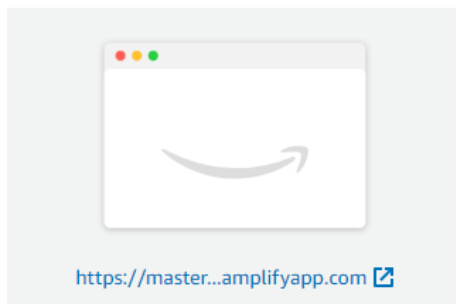
```
git add .
```

```
git commit -m "Deploy"
```

```
git push
```

Now we go to the Aws Amplify console to link our app from GitHub, and as soon as we link it, the application starts to build and gets deployed.

master
Continuous deploys set up (Edit)

https://master...amplifyapp.com

Provision — Build — Deploy — Verify

Last deployment	Last commit	Previews
1/10/2022, 11:30:52 PM	2	78ca562	GitHub - master	Disabled

We can visit the provided link to see our web application in action.

SPRINT 4: User Interface for CRUD API

GET URLs
page: 1
size: 3
Previous Page | Next Page

Designed by Abdullah Zaman Babar

Email: abdullah.zaman.babar.s@skipq.org

Testing:

We can test our application now.

If we click the *GET URLs* button, we can get our results and on pression the *Next* and *Previous Page* buttons we can navigate through our database.

We can also check the availability of an item in our database as follows.

Troubleshooting:

The users may face a common problem when fetching data from the DynamoDB database. This can occur due to the format the data is in.

For this sprint, I have collected my data as a list of dictionaries which makes it possible parsing through the data.

[{"URL_ADDRESS": "www.dawn.com"}, {"URL_ADDRESS": "www.test1.com"}, {"URL_ADDRESS": "www.youtube.com"}, {"URL_ADDRESS": "www.test123456.com"}, {"URL_ADDRESS": {"URL_ADDRESS": "www.espn.com"}, {"URL_ADDRESS": "www.skipq.org"}]

6. Sprint 5

6.1. Objective

Using Docker to build API test clients using pyresttest and Syntribos. The tests will exercise the CRUD endpoints. The built images will be pushed to ECR. The API test clients will be deployed to the pipeline.

6.2. Implementation

Using Docker to build a pyresttest image:

First, we create a *Dockerfile* where we define all the steps required to setup our image to be built.

We pull an *ubuntu* image and install the dependencies. The dependencies we need are *python2*, *pip*, *git*, *pycurl* and *yaml*. Then we clone the open-source *pyresttest* project and install it on our ubuntu environment. We copy a yaml file where we have defined our API tests.

```
FROM ubuntu:20.04

RUN apt-get update
RUN apt-get upgrade -y
RUN apt-get install -y python2
RUN apt-get install -y pip
RUN apt-get install -y git
RUN apt-get install -y python-pycurl
RUN apt-get install -y python-yaml
RUN git clone https://github.com/svanoort/pyresttest.git
RUN cd pyresttest && python2 setup.py install && cd ../
COPY apiTest.yaml /apiTest.yaml
RUN pyresttest https://s7rt0vfv66.execute-api.us-east-2.amazonaws.com apiTest.yaml
COPY script.sh /script.sh

CMD ["/bin/bash","script.sh"]
```

The tests defined in our yaml file are as follows:

```
---
- config:
    - testset: "Basic tests"
    - timeout: 100  # Increase timeout from the default 10 seconds
- test:
    - name: "Checking Health of API"
    - url: "/prod/health"
    - method: 'GET'
- test:
    - name: "Get all Urls"
    - url: "/prod/urls"
    - method: 'GET'
- test:
    - name: "Get one Url"
    - url: "/prod/url?URL_ADDRESS=www.dawn.com"
    - method: 'GET'
- test:
    - name: "Post Url"
    - url: "/prod/url"
    - method: 'POST'
    - body: '{"URL_ADDRESS":"www.pyresttest.com"}'
```

In our *scripts.sh* file we define the commands that we want to run when we run the container.

We use the following docker commands to build and push our image.

```
docker build -t azb-pyresttest .

docker tag azb-pyresttest:latest public.ecr.aws/m0u6n8i0/azb-pyresttest:latest

docker push public.ecr.aws/m0u6n8i0/azb-pyresttest:latest
```

We can use the *docker* run command to see the result of our tests.

```
abdullahzamanskipq:~/environment $ docker run public.ecr.aws/m0u6n8i0/azb-pyresttest:pyresttest
Test Group Default SUCCEEDED: : 7/7 Tests Passed!
```

Using Docker to build a Syntribos image:

First, we create a *Dockerfile* where we define all the steps required to setup our image to be built.

We then pull an *ubuntu* image and install the dependencies. The dependencies we need are *python3*, *pip*, and *git*. Then we clone the open-source *syntribos* project and install it on our ubuntu environment. We install the syntribos project on our environment and then update our endpoint in the syntribos configuration file. We also create two template file that will be automatically tested by syntribos.

```
FROM ubuntu:20.04

RUN apt-get update
RUN apt-get upgrade -y
RUN apt-get install -y python3
RUN apt-get install -y python3-pip
RUN apt-get install -y git
RUN git clone https://github.com/abdullahzamanbabar/syntribos.git
RUN python3 -m pip install /syntribos
RUN yes | syntribos init
RUN sed -i 's|endpoint=.*|endpoint=https://s7rt0vfv66.execute-api.us-east-2.amazonaws.com|' $(syntribos root)/syntribos.conf
RUN printf "GET /prod/health HTTP/1.1\nContent-Type: application/json" > $(syntribos root)/templates/health_check.template
RUN printf "GET /prod/urls HTTP/1.1\nContent-Type: application/json" > $(syntribos root)/templates/urls_check.template
RUN syntribos --config-file $(syntribos root)/syntribos.conf run

COPY script.sh /script.sh

CMD ["/bin/bash", "script.sh"]
```

We can test the image by running the following:

```
abdullahzamanskipq:~/environment $ docker run public.ecr.aws/m0u6n8i0/azb-syntribos:syntribos
```

We get the following results:

```
==================================================================================================
Template File...: urls_check.template
==================================================================================================

   ID            Test Name                                    Progress
[1010]  :  Buffer overflow headers              |####################|  100 %  : 0 Failure(s), 0 Error(s)
[1030]  :  Command injection headers            |####################|  100 %  : 0 Failure(s), 0 Error(s)
[1045]  :  Cors wildcard headers                |####################|  100 %  : 1 Failure(s), 0 Error(s)
[1055]  :  Integer overflow headers             |####################|  100 %  : 0 Failure(s), 0 Error(s)
[1080]  :  Ldap injection headers               |####################|  100 %  : 0 Failure(s), 0 Error(s)
[1100]  :  Redos headers                        |####################|  100 %  : 0 Failure(s), 0 Error(s)
[1120]  :  Sql injection headers                |####################|  100 %  : 0 Failure(s), 0 Error(s)
[1135]  :  Ssl endpoint body                    |####################|  100 %  : 0 Failure(s), 0 Error(s)
[1145]  :  String validation headers            |####################|  100 %  : 0 Failure(s), 0 Error(s)
[1190]  :  Xst headers                          |####################|  100 %  : 0 Failure(s), 0 Error(s)

Ran 132 test(s) in 5.419 s
```

```
==================================================================================================
Template File...: health_check.template
==================================================================================================

   ID            Test Name                                    Progress
[1205]  :  Buffer overflow headers              |####################|  100 %  : 1 Failure(s), 0 Error(s)
[1225]  :  Command injection headers            |####################|  100 %  : 0 Failure(s), 0 Error(s)
[1240]  :  Cors wildcard headers                |####################|  100 %  : 1 Failure(s), 0 Error(s)
[1250]  :  Integer overflow headers             |####################|  100 %  : 0 Failure(s), 0 Error(s)
[1275]  :  Ldap injection headers               |####################|  100 %  : 0 Failure(s), 0 Error(s)
[1295]  :  Redos headers                        |####################|  100 %  : 0 Failure(s), 0 Error(s)
[1315]  :  Sql injection headers                |####################|  100 %  : 0 Failure(s), 0 Error(s)
[1330]  :  Ssl endpoint body                    |####################|  100 %  : 0 Failure(s), 0 Error(s)
[1340]  :  String validation headers            |####################|  100 %  : 0 Failure(s), 0 Error(s)
[1385]  :  Xst headers                          |####################|  100 %  : 0 Failure(s), 0 Error(s)

Ran 132 test(s) in 3.229 s
```

Now we push the image to our repository.

```
docker push public.ecr.aws/m0u6n8i0/azb-syntribos:latest
```

ECR Repository:

We can see the pushed images in our ECR repository.

| ○ | azb-pyresttest | 🗐 public.ecr.aws/m0u6n8i0/azb-pyresttest |
| ○ | azb-syntribos | 🗐 public.ecr.aws/m0u6n8i0/azb-syntribos |

Deploying images through pipeline on AWS Fargate:

We use the cdk construct library to define our cluster as follows:

```python
vpc = ec2.Vpc.from_lookup(self, "Vpc",
                          is_default=True)

cluster = ecs.Cluster(self, "AZBStackCluster2",
                      vpc=vpc)

# Add capacity to it
cluster.add_capacity("DefaultAutoScalingGroupCapacity",
    instance_type=ec2.InstanceType("t2.micro"),
    desired_capacity=3
)
```

We then define our task for pyresttest.

```python
##### Defining a task for pyresttest
task_definition = ecs.FargateTaskDefinition(self, "azbPYRESTTESTtask1",
                memory_limit_mib=512, cpu=256)

repo = ecr.Repository.from_repository_name(self, "AZBRepo", "azb-pyresttest")

task_definition.add_container("AZBpyresttestContainer",
    image=ecs.ContainerImage.from_ecr_repository(repo,tag="pyresttest"),
    memory_limit_mib=512
)
```
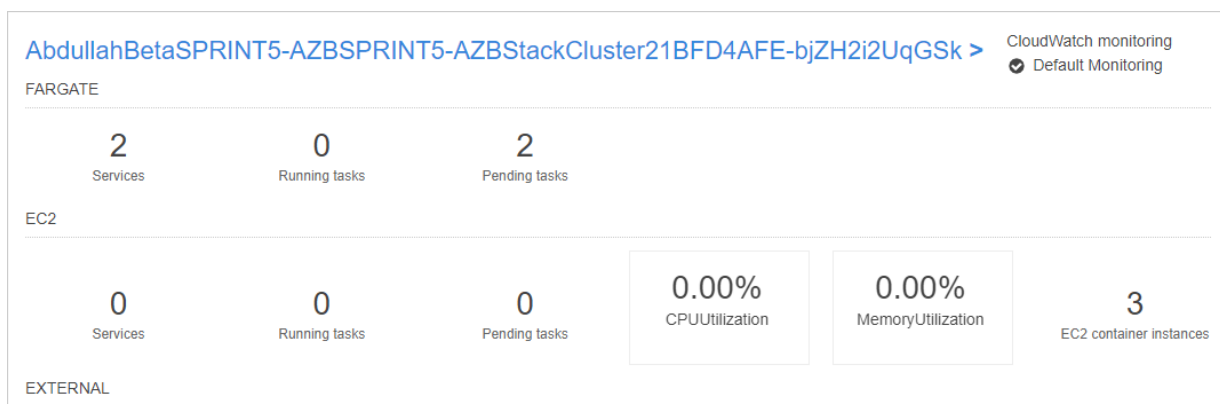
We then define a service for our task.

```python
## Instantiate an Amazon ECS Service
ecs_service = ecs.FargateService(self, "azbPYRESTTESTservice",
    cluster=cluster,
    task_definition=task_definition
)
```

Similarly, we define the service and task definition for syntribos.

```python
######### Defining Task for Syntribos
task_definition1 = ecs.FargateTaskDefinition(self, "azbSYNTRIBOStask1",
                memory_limit_mib=512, cpu=256)

repo1 = ecr.Repository.from_repository_name(self, "AZBRepo1", "azb-syntribos")

task_definition1.add_container("AZBsyntribosContainer",
    image=ecs.ContainerImage.from_ecr_repository(repo1,tag="syntribos"),
    memory_limit_mib=512
)

## Instantiate an Amazon ECS Service
ecs_service1 = ecs.FargateService(self, "azbSYNTRIBOSservice",
    cluster=cluster,
    task_definition=task_definition1
)
```

Results:

We can see our cluster in ECS as follows:

AbdullahBetaSPRINT5-AZBSPRINT5-AZBStackCluster21BFD4AFE-bjZH2i2UqGSk >   CloudWatch monitoring
                                                                          ● Default Monitoring
FARGATE

| 2 | 0 | 2 |
| Services | Running tasks | Pending tasks |

EC2

| 0 | 0 | 0 | 0.00% | 0.00% | 3 |
| Services | Running tasks | Pending tasks | CPUUtilization | MemoryUtilization | EC2 container instances |

EXTERNAL

Our services appear as follows:

We can see the log files for our tasks as follows:

Troubleshooting:

We can get a timeout error while pushing our image if we are building the image

using our local computer.

```
7905481a41b0: Retrying in 1 second
7432377d20f9: Retrying in 1 second
9011fa7415f2: Pushed
9b3efe998fdd: Pushing [==================================================>]  1.823MB
31ec0acf28d9: Pushed
0eba131dffd0: Pushed
net/http: request canceled while waiting for connection (Client.Timeout exceeded while awaiting headers)
```

To fix it, we just need to logout and then login with our credentials.