Sprint Retrospective Documentation

SkipQ

Authored by: Sikandar Bakht Kiani

Contents

Sprint 1: Documentation

<u>Building a Web Crawler using AWS CDK</u>

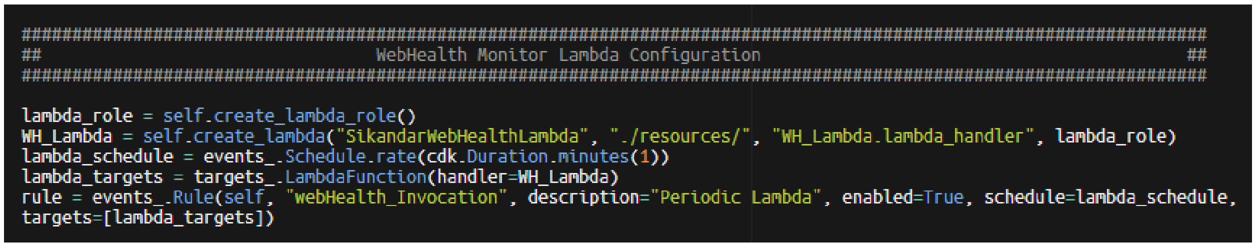
Authored by: Sikandar Bakht Kiani

Sprint Objective:

In this sprint, we have to build a canary in lambda function that will measure the availability and latency of a custom list of websites. The crawler will run periodically on a 5-minute cadence and write the web health metrics on CloudWatch. In the CloudWatch dashboard, we will then set some alarms on the metrics and employ some SNS notification services to receive alarm alerts via email.

Implementation:

AWS Lambda:



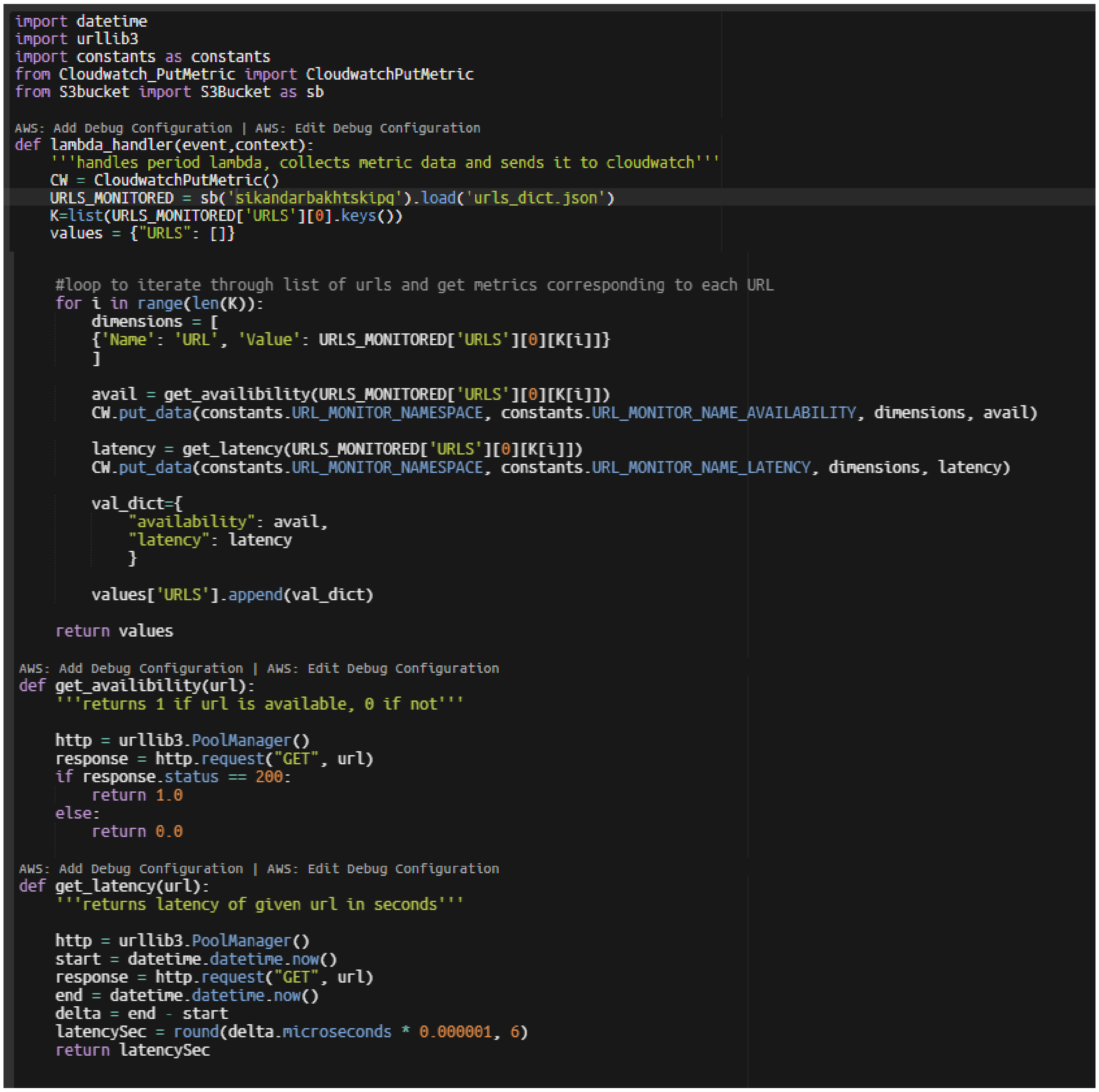
```
####################################################################################################################
##                                 WebHealth Monitor Lambda Configuration                                         ##
####################################################################################################################

lambda_role = self.create_lambda_role()
WH_Lambda = self.create_lambda("SikandarWebHealthLambda", "./resources/", "WH_Lambda.lambda_handler", lambda_role)
lambda_schedule = events_.Schedule.rate(cdk.Duration.minutes(1))
lambda_targets = targets_.LambdaFunction(handler=WH_Lambda)
rule = events_.Rule(self, "webHealth_Invocation", description="Periodic Lambda", enabled=True, schedule=lambda_schedule,
targets=[lambda_targets])
```

This is the Lambda Function that is periodically invoked. Here, it is scheduled to invoke every 1 minute.

The events library is used to set the condition for invoking the Lambda Function. We define a lambda handler for our Lambda function in the resources folder. Whenever the specific lambda function is invoked, the handler is called to perform the required functionality.

In the WH Lambda handler, we send requests to certain URLs and measure metrics such as availability (whether the URL is working) and latency (time for URL to respond)

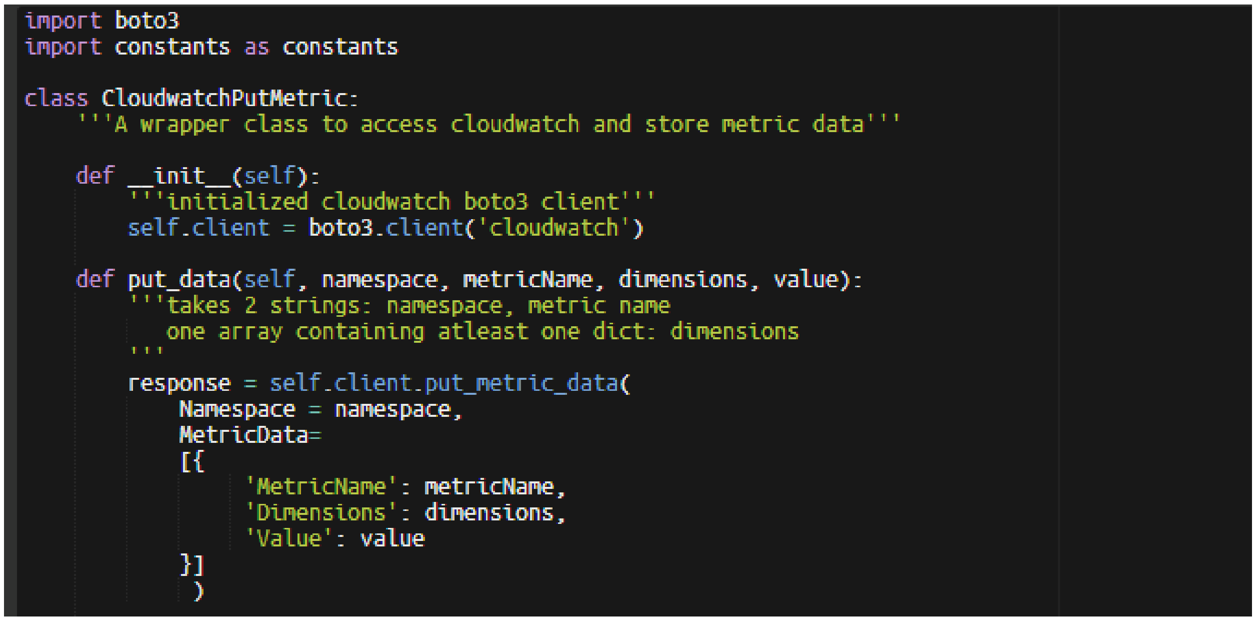
Lambda Handler:



```
import datetime
import urllib3
import constants as constants
from Cloudwatch_PutMetric import CloudwatchPutMetric
from S3bucket import S3Bucket as sb

AWS: Add Debug Configuration | AWS: Edit Debug Configuration
def lambda_handler(event,context):
    '''handles period lambda, collects metric data and sends it to cloudwatch'''
    CW = CloudwatchPutMetric()
    URLS_MONITORED = sb('sikandarbakhtskipq').load('urls_dict.json')
    K=list(URLS_MONITORED['URLS'][0].keys())
    values = {"URLS": []}

    #loop to iterate through list of urls and get metrics corresponding to each URL
    for i in range(len(K)):
        dimensions = [
        {'Name': 'URL', 'Value': URLS_MONITORED['URLS'][0][K[i]]}
        ]

        avail = get_availibility(URLS_MONITORED['URLS'][0][K[i]])
        CW.put_data(constants.URL_MONITOR_NAMESPACE, constants.URL_MONITOR_NAME_AVAILABILITY, dimensions, avail)

        latency = get_latency(URLS_MONITORED['URLS'][0][K[i]])
        CW.put_data(constants.URL_MONITOR_NAMESPACE, constants.URL_MONITOR_NAME_LATENCY, dimensions, latency)

        val_dict={
            "availability": avail,
            "latency": latency
            }

        values['URLS'].append(val_dict)

    return values

AWS: Add Debug Configuration | AWS: Edit Debug Configuration
def get_availibility(url):
    '''returns 1 if url is available, 0 if not'''

    http = urllib3.PoolManager()
    response = http.request("GET", url)
    if response.status == 200:
        return 1.0
    else:
        return 0.0

AWS: Add Debug Configuration | AWS: Edit Debug Configuration
def get_latency(url):
    '''returns latency of given url in seconds'''

    http = urllib3.PoolManager()
    start = datetime.datetime.now()
    response = http.request("GET", url)
    end = datetime.datetime.now()
    delta = end - start
    latencySec = round(delta.microseconds * 0.000001, 6)
    return latencySec
```

We can see two support functions i.e. get_availability and get_latency. They use the urllib3 and request library to send HTTP requests to URLs passed as parameter and return availability and latency in seconds.

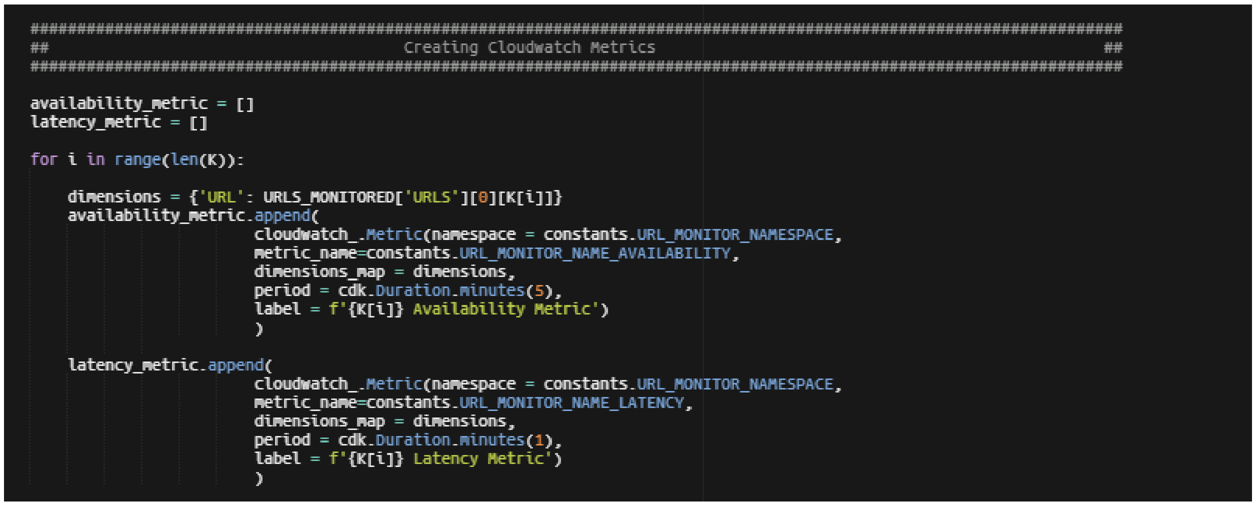
There is a Cloudwatch_PutMetric module imported as well. It is a class that has a single method: put_metric_data(). It takes a namespace, metric name, dimensions and value as input and publishes the metric data to CloudWatch.



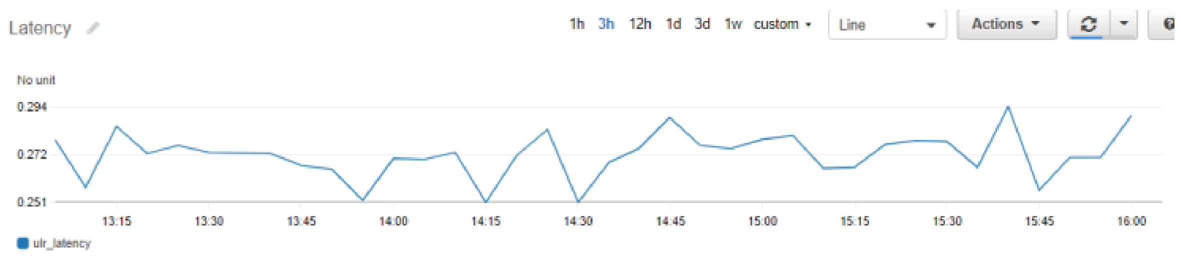
```python
import boto3
import constants as constants

class CloudwatchPutMetric:
    '''A wrapper class to access cloudwatch and store metric data'''

    def __init__(self):
        '''initialized cloudwatch boto3 client'''
        self.client = boto3.client('cloudwatch')

    def put_data(self, namespace, metricName, dimensions, value):
        '''takes 2 strings: namespace, metric name
           one array containing atleast one dict: dimensions
        '''
        response = self.client.put_metric_data(
            Namespace = namespace,
            MetricData=
            [{
                 'MetricName': metricName,
                 'Dimensions': dimensions,
                 'Value': value
            }]
             )
```

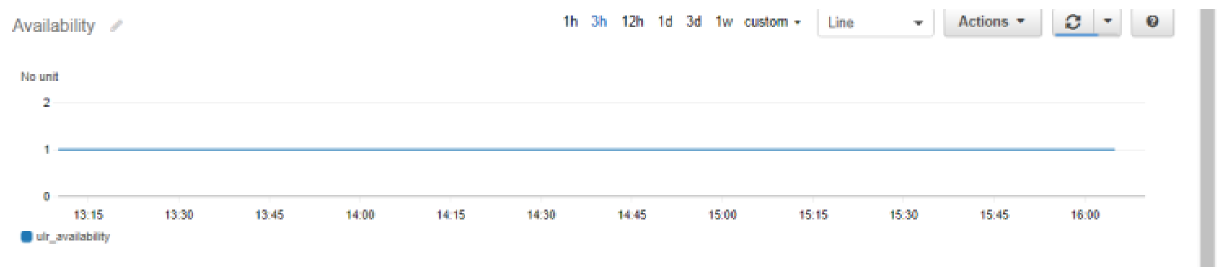
Accessing CloudWatch published metrics:

We can access our metrics through code. They can then be passed to Alarm service to raise alarm if the monitored values passes a threshold.



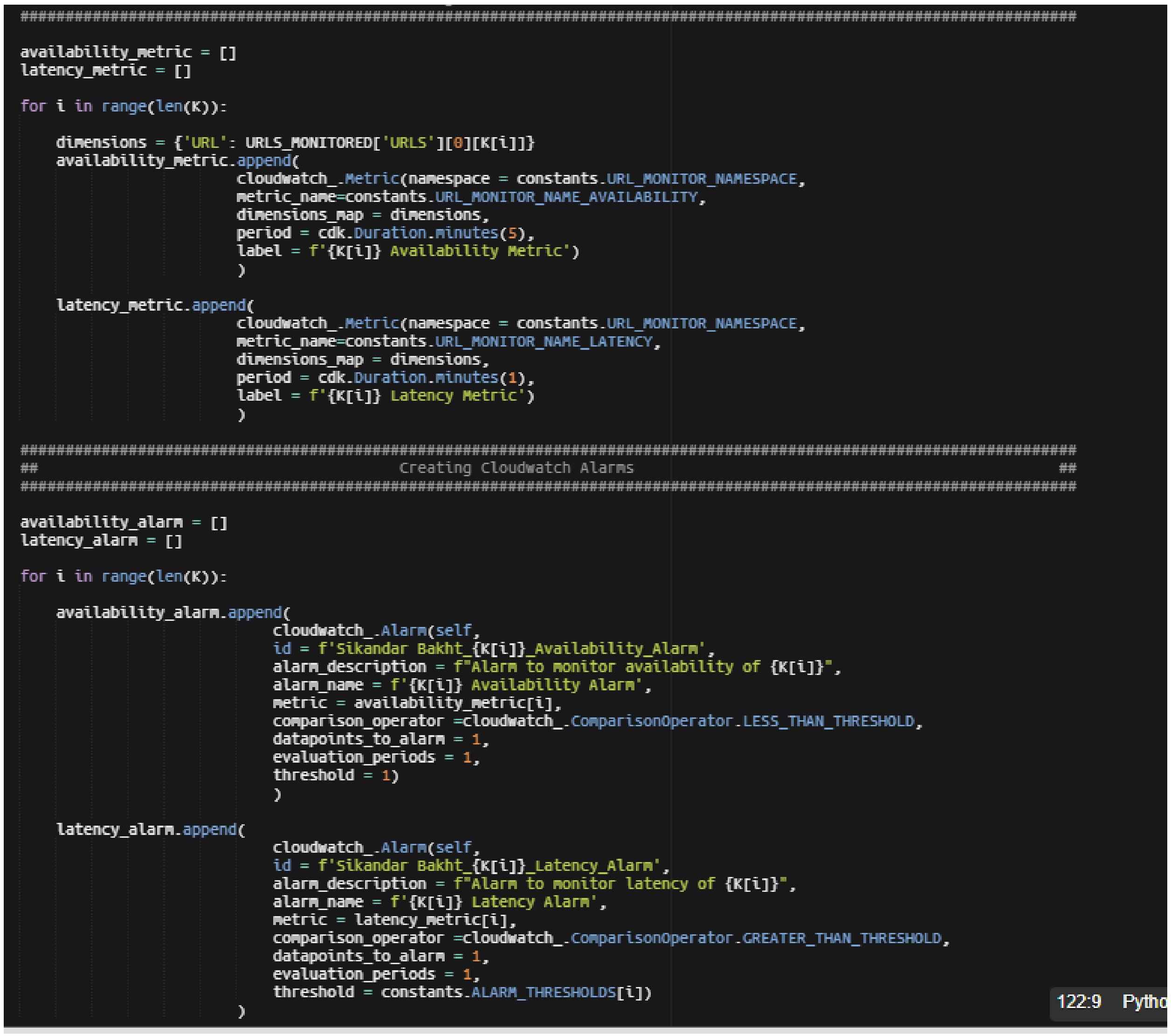
```python
####################################################################################################################
##                                              Creating Cloudwatch Metrics                                       ##
####################################################################################################################

availability_metric = []
latency_metric = []

for i in range(len(K)):

    dimensions = {'URL': URLS_MONITORED['URLS'][0][K[i]]}
    availability_metric.append(
                         cloudwatch_.Metric(namespace = constants.URL_MONITOR_NAMESPACE,
                         metric_name=constants.URL_MONITOR_NAME_AVAILABILITY,
                         dimensions_map = dimensions,
                         period = cdk.Duration.minutes(5),
                         label = f'{K[i]} Availability Metric')
                         )

    latency_metric.append(
                         cloudwatch_.Metric(namespace = constants.URL_MONITOR_NAMESPACE,
                         metric_name=constants.URL_MONITOR_NAME_LATENCY,
                         dimensions_map = dimensions,
                         period = cdk.Duration.minutes(1),
                         label = f'{K[i]} Latency Metric')
                         )
```





CloudWatch Alarms:

After publishing the metric data to Cloudwatch, we want to monitor it and raise alarm if it crosses a threshold. To define alarms, we write the following code.

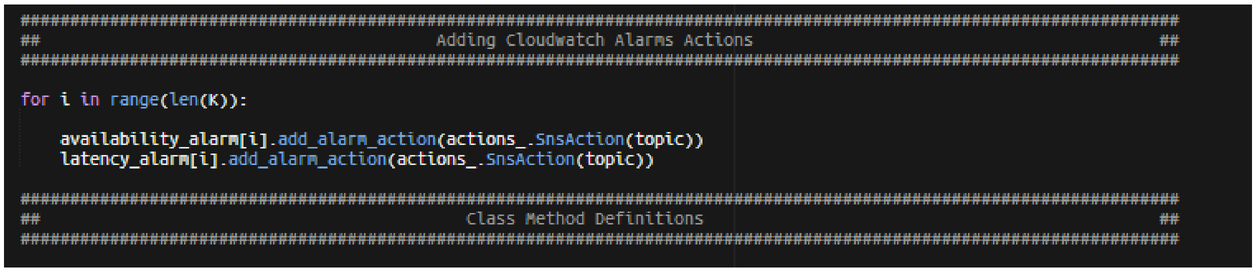


```
#####################################################################################################################

availability_metric = []
latency_metric = []

for i in range(len(K)):

    dimensions = {'URL': URLS_MONITORED['URLS'][0][K[i]]}
    availability_metric.append(
                        cloudwatch_.Metric(namespace = constants.URL_MONITOR_NAMESPACE,
                        metric_name=constants.URL_MONITOR_NAME_AVAILABILITY,
                        dimensions_map = dimensions,
                        period = cdk.Duration.minutes(5),
                        label = f'{K[i]} Availability Metric')
                        )

    latency_metric.append(
                        cloudwatch_.Metric(namespace = constants.URL_MONITOR_NAMESPACE,
                        metric_name=constants.URL_MONITOR_NAME_LATENCY,
                        dimensions_map = dimensions,
                        period = cdk.Duration.minutes(1),
                        label = f'{K[i]} Latency Metric')
                        )

#####################################################################################################################
##                                        Creating Cloudwatch Alarms                                              ##
#####################################################################################################################

availability_alarm = []
latency_alarm = []

for i in range(len(K)):

    availability_alarm.append(
                            cloudwatch_.Alarm(self,
                            id = f'Sikandar Bakht_{K[i]}_Availability_Alarm',
                            alarm_description = f"Alarm to monitor availability of {K[i]}",
                            alarm_name = f'{K[i]} Availability Alarm',
                            metric = availability_metric[i],
                            comparison_operator =cloudwatch_.ComparisonOperator.LESS_THAN_THRESHOLD,
                            datapoints_to_alarm = 1,
                            evaluation_periods = 1,
                            threshold = 1)
                            )

    latency_alarm.append(
                            cloudwatch_.Alarm(self,
                            id = f'Sikandar Bakht_{K[i]}_Latency_Alarm',
                            alarm_description = f"Alarm to monitor latency of {K[i]}",
                            alarm_name = f'{K[i]} Latency Alarm',
                            metric = latency_metric[i],
                            comparison_operator =cloudwatch_.ComparisonOperator.GREATER_THAN_THRESHOLD,
                            datapoints_to_alarm = 1,
                            evaluation_periods = 1,
                            threshold = constants.ALARM_THRESHOLDS[i])
                        )
```

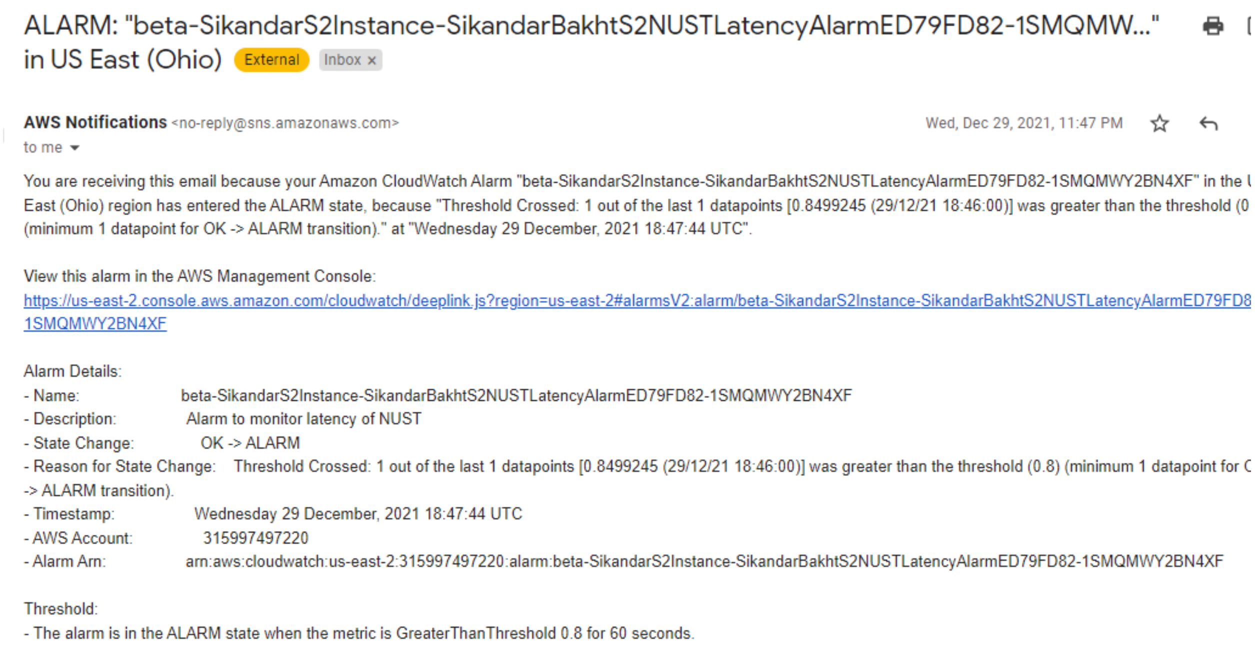
Here availability and latency alarms are set for 4 URLs whose thresholds are stored in a constants file. If the availability falls below 1, alarm is raised. Similarly, if latency is greater than a maximum value, alarm is raised.

SNS Subscription:

We want to be notified via SMS or Email if one of our URLs is in alarm state. This can be done by adding an action to alarm to send an sns notification to the subscriber.



```
###########################################################################################
##                              Adding Cloudwatch Alarms Actions                         ##
###########################################################################################

for i in range(len(K)):

    availability_alarm[i].add_alarm_action(actions_.SnsAction(topic))
    latency_alarm[i].add_alarm_action(actions_.SnsAction(topic))

###########################################################################################
##                                 Class Method Definitions                              ##
###########################################################################################
```



ALARM: "beta-SikandarS2Instance-SikandarBakhtS2NUSTLatencyAlarmED79FD82-1SMQMW..." in US East (Ohio) External Inbox ×

AWS Notifications <no-reply@sns.amazonaws.com> Wed, Dec 29, 2021, 11:47 PM
to me

You are receiving this email because your Amazon CloudWatch Alarm "beta-SikandarS2Instance-SikandarBakhtS2NUSTLatencyAlarmED79FD82-1SMQMWY2BN4XF" in the U
East (Ohio) region has entered the ALARM state, because "Threshold Crossed: 1 out of the last 1 datapoints [0.8499245 (29/12/21 18:46:00)] was greater than the threshold (0
(minimum 1 datapoint for OK -> ALARM transition)." at "Wednesday 29 December, 2021 18:47:44 UTC".

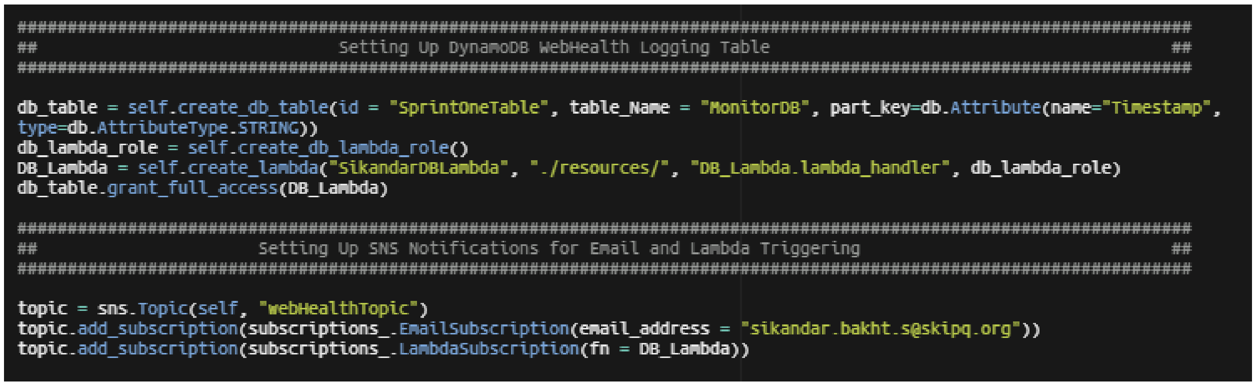
View this alarm in the AWS Management Console:
https://us-east-2.console.aws.amazon.com/cloudwatch/deeplink.js?region=us-east-2#alarmsV2:alarm/beta-SikandarS2Instance-SikandarBakhtS2NUSTLatencyAlarmED79FD8
1SMQMWY2BN4XF

Alarm Details:
- Name: beta-SikandarS2Instance-SikandarBakhtS2NUSTLatencyAlarmED79FD82-1SMQMWY2BN4XF
- Description: Alarm to monitor latency of NUST
- State Change: OK -> ALARM
- Reason for State Change: Threshold Crossed: 1 out of the last 1 datapoints [0.8499245 (29/12/21 18:46:00)] was greater than the threshold (0.8) (minimum 1 datapoint for C
-> ALARM transition).
- Timestamp: Wednesday 29 December, 2021 18:47:44 UTC
- AWS Account: 315997497220
- Alarm Arn: arn:aws:cloudwatch:us-east-2:315997497220:alarm:beta-SikandarS2Instance-SikandarBakhtS2NUSTLatencyAlarmED79FD82-1SMQMWY2BN4XF

Threshold:
- The alarm is in the ALARM state when the metric is GreaterThanThreshold 0.8 for 60 seconds.

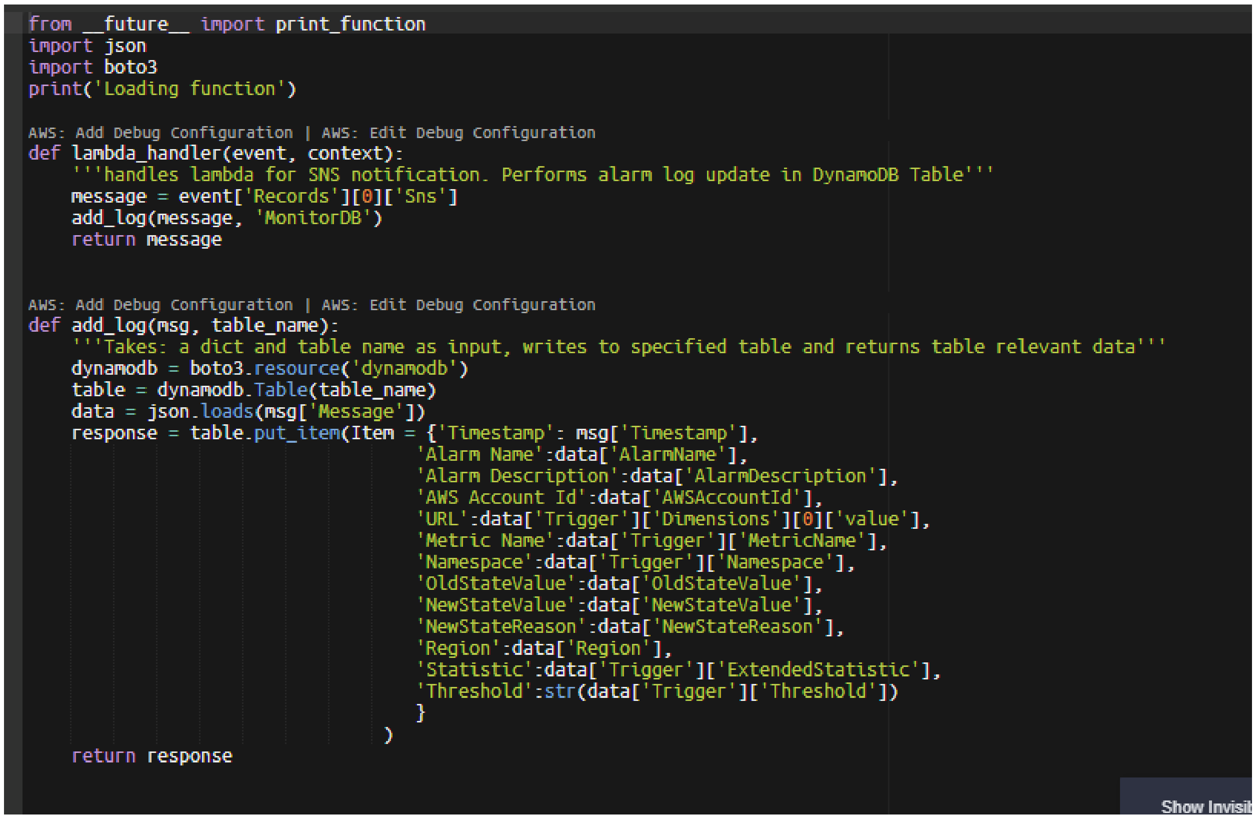
DynamoDB:

We can also log our data to a DynamoDB table whenever there is an alarm. To do this, we create another lambda function that is invoked by an SNS subscription. The idea is that whenever an Alarm is raised, an SNS notification will be sent, and the notification can be sent to a Lambda Function as well. The Lambda function will then perform the data logging into our table.



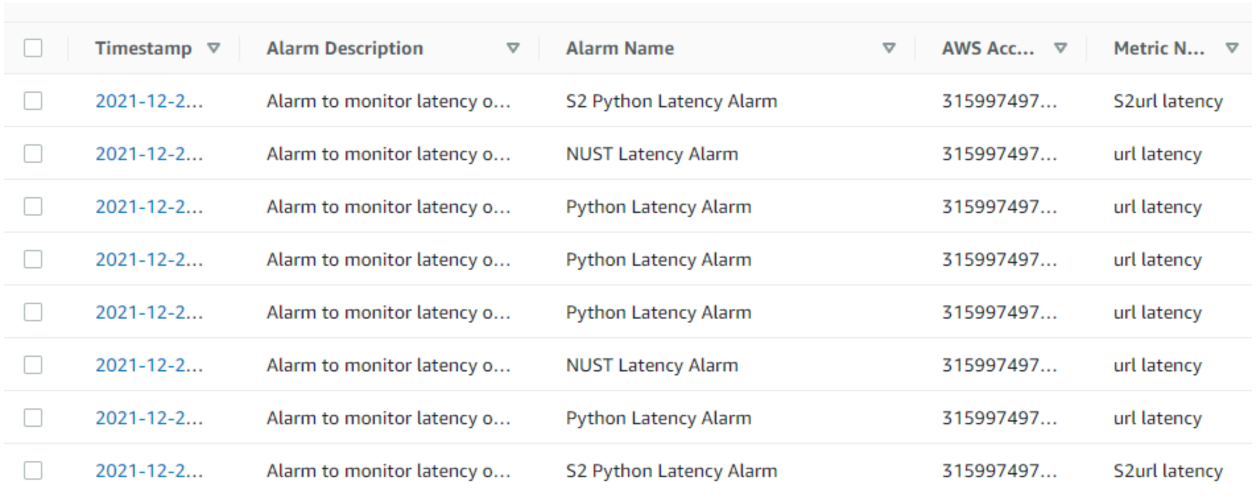
```
###################################################################################################
##                         Setting Up DynamoDB WebHealth Logging Table                          ##
###################################################################################################

db_table = self.create_db_table(id = "SprintOneTable", table_Name = "MonitorDB", part_key=db.Attribute(name="Timestamp",
type=db.AttributeType.STRING))
db_lambda_role = self.create_db_lambda_role()
DB_Lambda = self.create_lambda("SikandarDBLambda", "./resources/", "DB_Lambda.lambda_handler", db_lambda_role)
db_table.grant_full_access(DB_Lambda)

###################################################################################################
##                  Setting Up SNS Notifications for Email and Lambda Triggering                ##
###################################################################################################

topic = sns.Topic(self, "webHealthTopic")
topic.add_subscription(subscriptions_.EmailSubscription(email_address = "sikandar.bakht.s@skipq.org"))
topic.add_subscription(subscriptions_.LambdaSubscription(fn = DB_Lambda))
```

By adding a subscription topic to both user email and a Lambda Function, we can an automatic logging system. The lambda function parses through the notification and stores it in an easy to look up manner in the table.



```
from __future__ import print_function
import json
import boto3
print('Loading function')

AWS: Add Debug Configuration | AWS: Edit Debug Configuration
def lambda_handler(event, context):
    '''handles lambda for SNS notification. Performs alarm log update in DynamoDB Table'''
    message = event['Records'][0]['Sns']
    add_log(message, 'MonitorDB')
    return message


AWS: Add Debug Configuration | AWS: Edit Debug Configuration
def add_log(msg, table_name):
    '''Takes: a dict and table name as input, writes to specified table and returns table relevant data'''
    dynamodb = boto3.resource('dynamodb')
    table = dynamodb.Table(table_name)
    data = json.loads(msg['Message'])
    response = table.put_item(Item = {'Timestamp': msg['Timestamp'],
                                      'Alarm Name':data['AlarmName'],
                                      'Alarm Description':data['AlarmDescription'],
                                      'AWS Account Id':data['AWSAccountId'],
                                      'URL':data['Trigger']['Dimensions'][0]['value'],
                                      'Metric Name':data['Trigger']['MetricName'],
                                      'Namespace':data['Trigger']['Namespace'],
                                      'OldStateValue':data['OldStateValue'],
                                      'NewStateValue':data['NewStateValue'],
                                      'NewStateReason':data['NewStateReason'],
                                      'Region':data['Region'],
                                      'Statistic':data['Trigger']['ExtendedStatistic'],
                                      'Threshold':str(data['Trigger']['Threshold'])
                                      }
                                  )
    return response
```

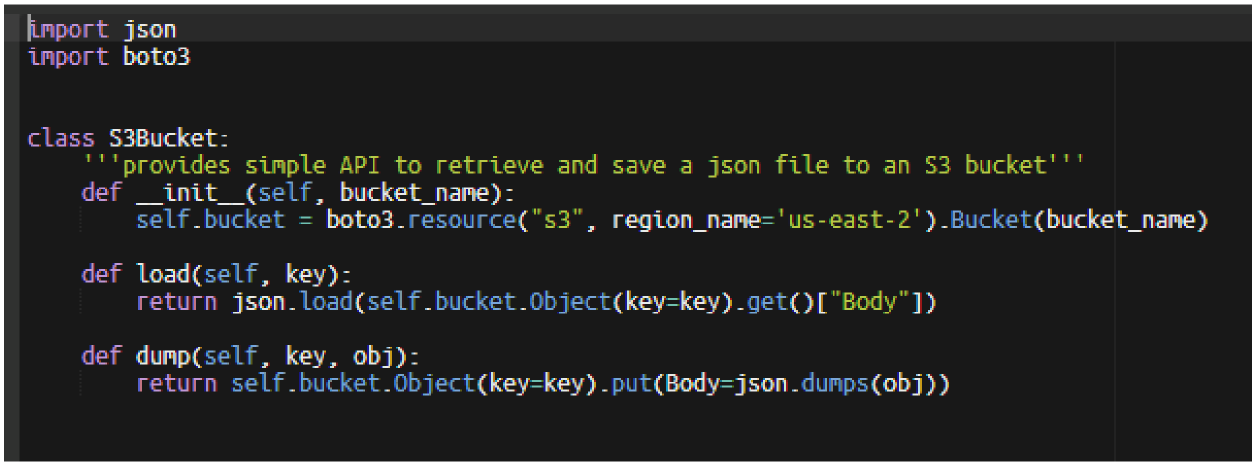
The table stores every notification by its timestamp which must be unique. We can view all the information about the event from the table.



	Timestamp	Alarm Description	Alarm Name	AWS Acc...	Metric N...
☐	2021-12-2...	Alarm to monitor latency o...	S2 Python Latency Alarm	315997497...	S2url latency
☐	2021-12-2...	Alarm to monitor latency o...	NUST Latency Alarm	315997497...	url latency
☐	2021-12-2...	Alarm to monitor latency o...	Python Latency Alarm	315997497...	url latency
☐	2021-12-2...	Alarm to monitor latency o...	Python Latency Alarm	315997497...	url latency
☐	2021-12-2...	Alarm to monitor latency o...	Python Latency Alarm	315997497...	url latency
☐	2021-12-2...	Alarm to monitor latency o...	NUST Latency Alarm	315997497...	url latency
☐	2021-12-2...	Alarm to monitor latency o...	Python Latency Alarm	315997497...	url latency
☐	2021-12-2...	Alarm to monitor latency o...	S2 Python Latency Alarm	315997497...	S2url latency

S3 Bucket:

We need to dynamically add metrics and alarms based on the URLs stored in an S3 bucket. For this reason, we create an S3 bucket manually in the S3 Console. We store a JSON formatted file with 4 URLs in it. Then we use our S3Bucket Class to write to the S3Bucket file or read from it.



```
import json
import boto3


class S3Bucket:
    '''provides simple API to retrieve and save a json file to an S3 bucket'''
    def __init__(self, bucket_name):
        self.bucket = boto3.resource("s3", region_name='us-east-2').Bucket(bucket_name)

    def load(self, key):
        return json.load(self.bucket.Object(key=key).get()["Body"])

    def dump(self, key, obj):
        return self.bucket.Object(key=key).put(Body=json.dumps(obj))
```

Troubleshooting:

Installing dependencies:

There was an error aws-cdk not found despite running pip install -r requirements.txt. It was fixed by running. venv/bin/pip install -r requirements.txt

Sprint 2: Documentation

Multistage pipelined web-crawler using CI/CD features of AWS

Authored by: Sikandar Bakht Kiani

CI/CD in Software Development:

A CI/CD pipeline automates your software delivery process. The pipeline builds code, runs tests (CI), and safely deploys a new version of the application (CD).

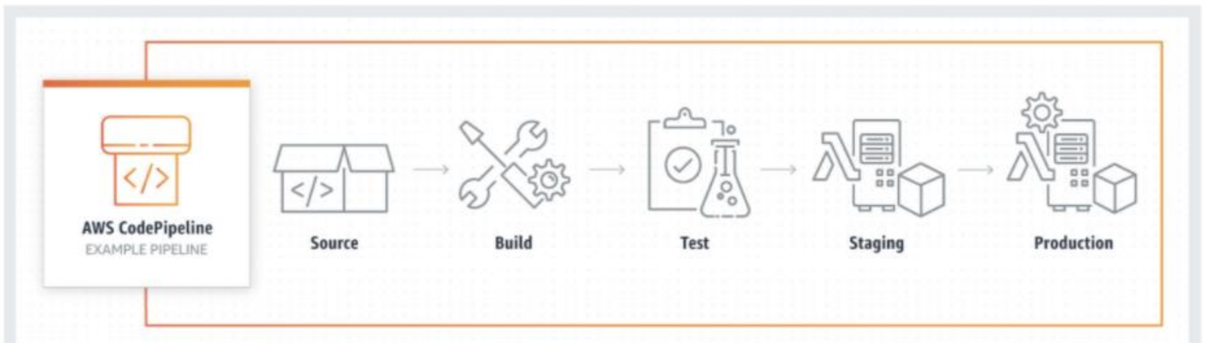
Automated pipelines remove manual errors, provide standardized feedback loops to developers, and enable fast product iterations.

What do CI and CD mean?

CI, short for *Continuous Integration*, is a software development practice in which all developers merge code changes in a central repository multiple times a day. CD stands for **Continuous Delivery**, which on top of Continuous Integration adds the practice of automating the entire software release process.

With CI, each change in code triggers an automated build-and-test sequence for the given project, providing feedback to the developer(s) who made the change. The entire CI feedback loop should run in less than 10 minutes.

Continuous Delivery includes infrastructure provisioning and deployment, which may be manual and consist of multiple stages. What's important is that all these processes are fully automated, with each run fully logged and visible to the entire team.



Typical Pipeline in AWS CodePipeline

Elements of a CI/CD pipeline:

A CI/CD pipeline may sound like overhead, but it isn't. It's essentially a runnable specification of the steps that any developer needs to perform to deliver a new version of a software product. In the absence of an automated pipeline, engineers would still need to perform these steps manually, and hence far less productively.

Most software releases go through a couple of typical stages:

Source stage:

In most cases, a pipeline run is triggered by a source code repository. A change in code triggers a notification to the CI/CD tool, which runs the corresponding pipeline. Other common triggers include automatically scheduled or user-initiated workflows, as well as results of other pipelines.

Build stage

We combine the source code and its dependencies to build a runnable instance of our product that we can potentially ship to our end users. Programs written in languages such as Java, C/C++, or Go need to be compiled, whereas Ruby, Python and JavaScript programs work without this step.

Failure to pass the build stage is an indicator of a fundamental problem in a project's configuration, and it's best to address it immediately.

Test stage

In this phase, we run automated tests to validate our code's correctness and the behavior of our product. The test stage acts as a safety net that prevents easily reproducible bugs from reaching the end-users.

The responsibility of writing tests falls on the developers. The best way to write automated tests is to do so as we write new code in test- or behavior-driven development.

Depending on the size and complexity of the project, this phase can last from seconds to hours. Many large-scale projects run tests in multiple stages, starting with smoke tests that perform quick sanity checks to end-to-end integration tests that test the entire system from the user’s point of view. An extensive test suite is typically parallelized to reduce run time.

Failure during the test stage exposes problems in code that developers didn’t foresee when writing the code. It’s essential for this stage to produce feedback to developers quickly, while the problem space is still fresh in their minds and they can maintain the state of flow.

Deploy stages

Once we have a built a runnable instance of our code that has passed all predefined tests, we’re ready to deploy it. There are usually multiple deploy environments, for example, a “beta” or “staging” environment which is used internally by the product team, and a “production” environment for end-users.

Teams that have embraced the Agile model of development—guided by tests and real-time monitoring—usually deploy work-in-progress manually to a staging environment for additional manual testing and review, and automatically deploy approved changes from the master branch to production.

Deploying pipeline in AWS

Bootstrapping your target environment:

Before deploying a pipeline, we need to provision the environment with the required resources and permission to deploy our stacks and constructs.

These resources include an Amazon S3 bucket for storing files and IAM roles that grant permissions needed to perform deployments. The process of provisioning these initial resources is called bootstrapping. The required resources are defined in a AWS CloudFormation stack, called the bootstrap stack, which is usually named CDKToolkit. Like any AWS CloudFormation stack, it appears in the AWS CloudFormation console once it has been deployed. Environments are independent, so if you want to deploy to multiple environments (different AWS accounts or different regions in the same account), each environment must be bootstrapped separately.

Bootstrapping Procedure:

Bootstrapping is a relatively straightforward process. All we need is to create a clean environment and do the following steps:

- Set up a fresh python environment and install requirements.

```
python -m venv .venv
source .venv/bin/activate
pip install -m requirements.txt
```

- Open the cdk.json file and add the following entries to the context dict:

```
"@aws-cdk/core:newStyleStackSynthesis": true,
"@aws-cdk/core:bootstrapQualifier": "sikandars2"
```

If you are creating your own project, you should supply some value other than “sikandars2”

- Open the terminal and type the following commands:

```
export CDK_NEW_BOOTSTRAP=1
cdk bootstrap --qualifier <qualifier> --toolkit-stack-name <toolkit-stack-name> --cloudformation-execution-policies arn:aws:iam::aws:policy/AdministratorAccess <account id>/<region>
```

This should create your bootstrapped environment in the same region that you supplied in the cdk bootstrap command.

Errors during bootstrapping:

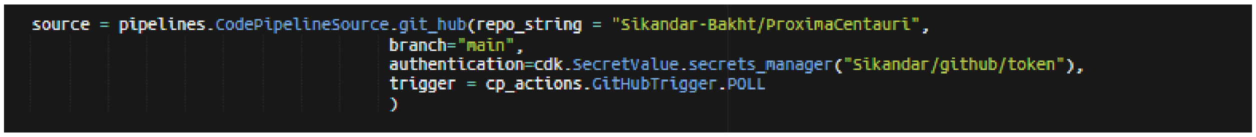
1. Staging bucket already exists:

If you have bootstrapped the environment before, it can give you an error like this. To resolve it, go to AWS S3 and delete any s3 buckets under your name that include keywords “assets” or “linear-artifact”

Defining your Pipeline:

To define the pipeline, we need to create a pipeline stack. The stack contains information about where to get the source code from, how to synthesize e.g. permissions assumed by CloudFormation, the stages that the pipeline has and any steps to take before or after deployment of those stages.

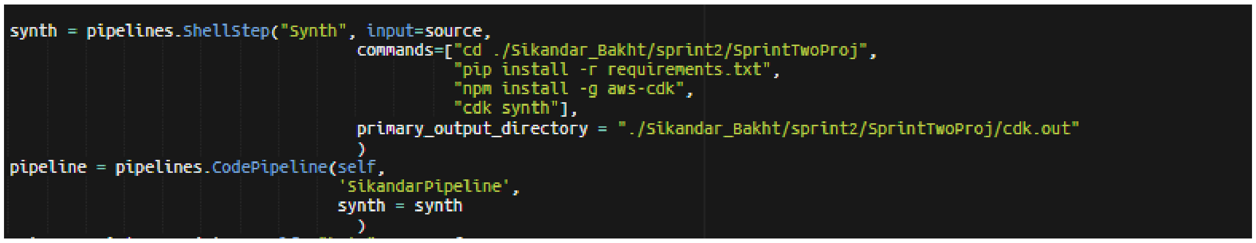
Source:



```
source = pipelines.CodePipelineSource.git_hub(repo_string = "Sikandar-Bakht/ProximaCentauri",
                                              branch="main",
                                              authentication=cdk.SecretValue.secrets_manager("Sikandar/github/token"),
                                              trigger = cp_actions.GitHubTrigger.POLL
                                              )
```

Use the aws_pipelines CodePipelineSource function with the github method to specify your github repo and branch to use as source code for your pipeline. You can add authentication via AWS Secrets Manager, but it is optional. The trigger can be a web hook which can be configured in your github repo or a polling function. This makes the pipeline check for changes in source periodically and redeploy when it finds changes.

Synth:

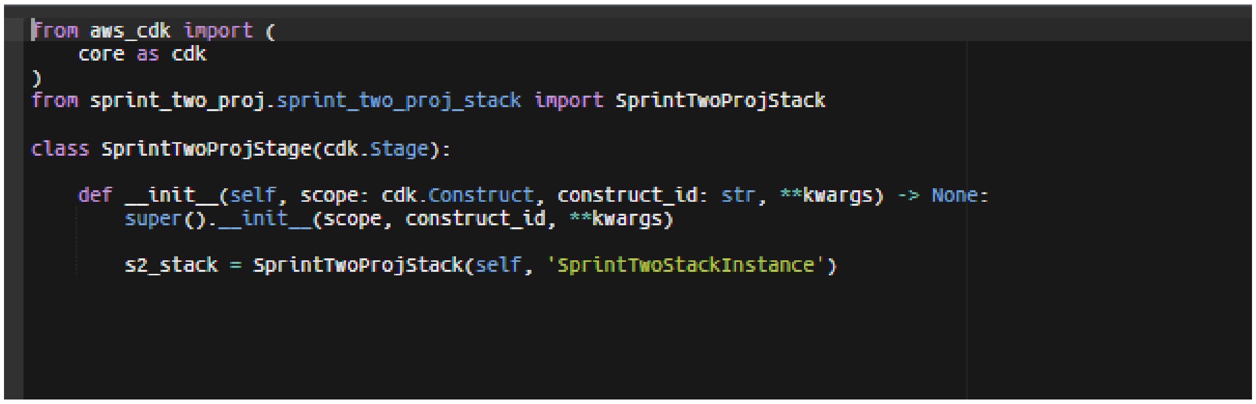


```
synth = pipelines.ShellStep("Synth", input=source,
                            commands=["cd ./Sikandar_Bakht/sprint2/SprintTwoProj",
                                      "pip install -r requirements.txt",
                                      "npm install -g aws-cdk",
                                      "cdk synth"],
                            primary_output_directory = "./Sikandar_Bakht/sprint2/SprintTwoProj/cdk.out"
                            )
pipeline = pipelines.CodePipeline(self,
                           'SikandarPipeline',
                           synth = synth
                            )
```

In this step, we use the ShellStep function to give commands to initialize our cloned repo. The primary output directory receives the logs from the synth function.

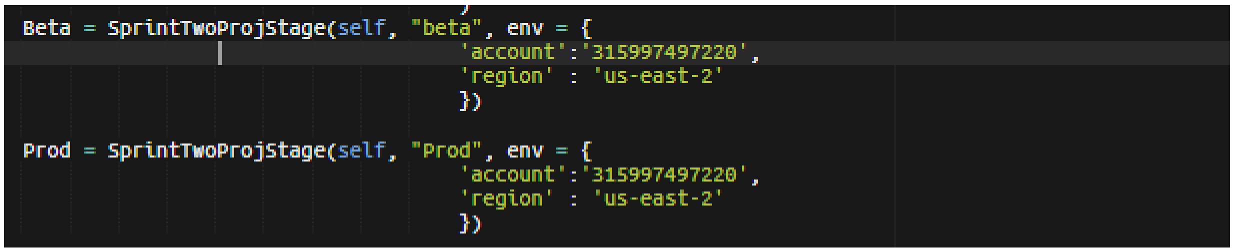
After these two steps, we can define our pipeline construct with CodePipeline, which synthesizes the actual environment, performing the CloudFormation build steps.

Defining and adding Stages:



```
from aws_cdk import (
    core as cdk
)
from sprint_two_proj.sprint_two_proj_stack import SprintTwoProjStack

class SprintTwoProjStage(cdk.Stage):

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

        s2_stack = SprintTwoProjStack(self, 'SprintTwoStackInstance')
```

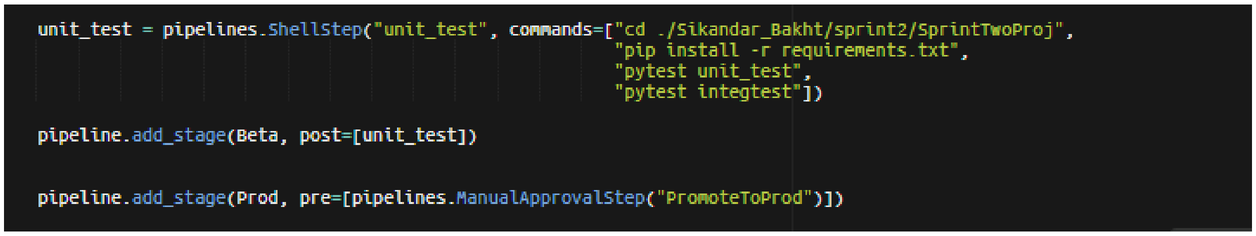
A stage is a construct that can instantiate a stack. We can use it to create separate versions of the same stack. It is useful when you want to separate your test or unstable environment from the stable production deployment



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

Prod = SprintTwoProjStage(self, "Prod", env = {
                                    'account':'315997497220',
                                    'region' : 'us-east-2'
                                    })
```

We can create two different versions of the same stack under the stage names beta and prod. We have the option to deploy different stacks in different regions using the region parameter.

Adding pre and post actions:

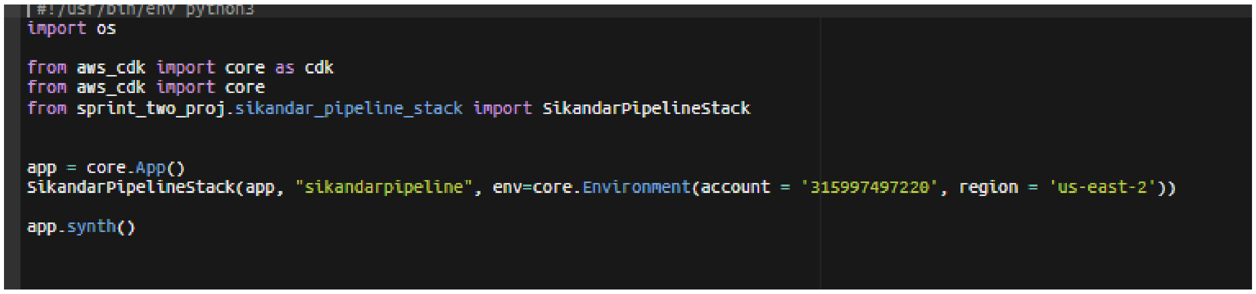


```
unit_test = pipelines.ShellStep("unit_test", commands=["cd ./Sikandar_Bakht/sprint2/SprintTwoProj",
                                                        "pip install -r requirements.txt",
                                                        "pytest unit_test",
                                                        "pytest integtest"])

pipeline.add_stage(Beta, post=[unit_test])


pipeline.add_stage(Prod, pre=[pipelines.ManualApprovalStep("PromoteToProd")])
```

We can do any actions before the deployment of a stage. It maybe running test, installing an additional dependency etc. Here we have added some unit tests to run after deploying beta stage and added a manual approval step before deploying production stage.

Deploying the pipeline:

While deploying the pipeline, instead of instantiating our application stack, we instantiate our pipeline stack in the entry point. Then we need to run the “cdk deploy sikandarpipeline” command for one time to deploy our pipeline. Subsequent changes to our source code are automatically detected and deployed.



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

from aws_cdk import core as cdk
from aws_cdk import core
from sprint_two_proj.sikandar_pipeline_stack import SikandarPipelineStack


app = core.App()
SikandarPipelineStack(app, "sikandarpipeline", env=core.Environment(account = '315997497220', region = 'us-east-2'))

app.synth()
```

Errors during pipeline deployment:

1. Access denied to resource:

This issue rises mainly due to incorrect bootstrapping the environment. It is advised to delete deployed stacks, if any and bootstrap the environment from scratch (delete s3 buckets first). You can also manually assign permissions to the resources by using CodeBuildStep function.

2. Resource already exists:

This error is caused by a resource with the same name existing in the environment. It can be remedied by appending the names of all constructs with a random string. Using AWS default generated names is recommended.

3. Rollback_failed error:

It may be caused by a stack that failed to deploy. You can resolve this by deleting the stack from CloudFormation console and redeploying.

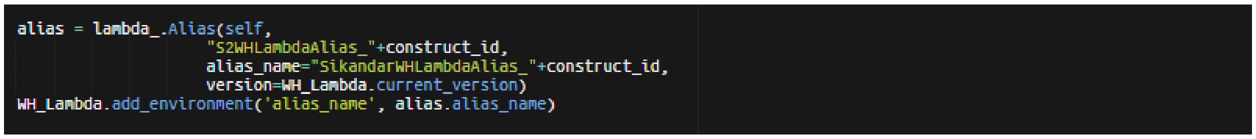
Automatic rollback for Lambda Function:

To enable automatic rollback functionality for a lambda function that you have deployed, we need to define a lambda deployment group that triggers when a change is made to the lambda function in code or an alarm indicating abnormal behavior in lambda function is raised. It allows CodeDeploy to automatically shift traffic to newer version of lambda or rollback to an older version if the alarm is raised.

Setting up deployments:

To setup automatic deployments, we need to perform 3 steps:

1. Define a function alias for your lambda function:

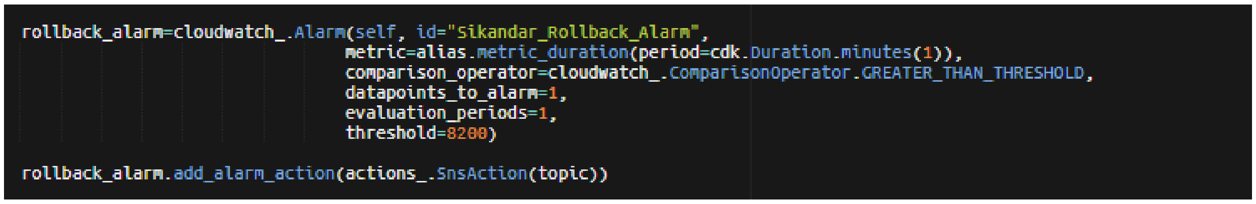


```
alias = lambda_.Alias(self,
                     "S2WHLambdaAlias_"+construct_id,
                     alias_name="SikandarWHLambdaAlias_"+construct_id,
                     version=WH_Lambda.current_version)
WH_Lambda.add_environment('alias_name', alias.alias_name)
```

Alias functions are pointers to a Lambda Function. They are like temporal snapshots where they keep track of different deployed versions of your lambda function

The main parameter to look out for is the version parameter. It can accept different values for your use-case. For example, if you want your lambda to revert to a specific version, you can define the alias with that version. We want our alias to point to the current version of the lambda function so any changes are deployed.

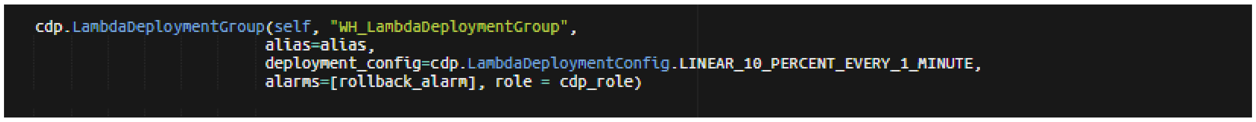
2. (Optional) Setup a CloudWatch alarm on a specific metric in your Lambda Function:



```
rollback_alarm=cloudwatch_.Alarm(self, id="Sikandar_Rollback_Alarm",
                                 metric=alias.metric_duration(period=cdk.Duration.minutes(1)),
                                 comparison_operator=cloudwatch_.ComparisonOperator.GREATER_THAN_THRESHOLD,
                                 datapoints_to_alarm=1,
                                 evaluation_periods=1,
                                 threshold=8200)

rollback_alarm.add_alarm_action(actions_.SnsAction(topic))
```

The CloudWatch alarm monitors the duration metric exposed by the lambda function. You can use other metrics to trigger this alarm like number of times the lambda function has been invoked, errors incurred etc.

3. Define the Lambda Deployment Group:

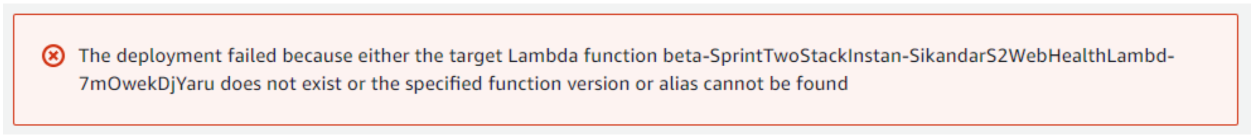


```
cdp.LambdaDeploymentGroup(self, "WH_LambdaDeploymentGroup",
                          alias=alias,
                          deployment_config=cdp.LambdaDeploymentConfig.LINEAR_10_PERCENT_EVERY_1_MINUTE,
                          alarms=[rollback_alarm], role = cdp_role)
```

The LambdaDeploymentGroup is a CodeDeploy construct that defines the settings for your deployment. It takes an alias to a lambda function to deploy, the deployment_config which basically tells it how to shift traffic from the older lambda version to the new version so that there is no downtime in deployment. There is a myriad of options to choose from, the one used in this function shifts 10 percent of traffic every 1 minute.

The alarm, if specified, allows it to rollback a deployment when it is triggered. It is useful in cases where you do not trust your updated lambda function to work as it should and have a working contingency to fall back on.

Errors during deployment of lambda:



The deployment failed because either the target Lambda function beta-SprintTwoStackInstan-SikandarS2WebHealthLambd-7mOwekDjYaru does not exist or the specified function version or alias cannot be found

Sometimes deployments fail because alias fails to update in the stack. In such

Sprint 3: Documentation

Build a CRUD API using AWS APIGateway with AWS Lambda as backend, with CI/CD, multistage pipeline

Authored by: Sikandar Bakht Kiani

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. REST stands for representational state transfer and was created by computer scientist Roy Fielding.

API

An API is a set of definitions and protocols for building and integrating application software. It's sometimes referred to as a contract between an information provider and an information user—establishing the content required from the consumer (the call) and the content required by the producer (the response). For example, the API design for a weather service could specify that the user supply a zip code and that the producer reply with a 2-part answer, the first being the high temperature, and the second being the low.

In other words, if you want to interact with a computer or system to retrieve information or perform a function, an API helps you communicate what you want to that system so it can understand and fulfill the request.

You can think of an API as a mediator between the users or clients and the resources or web services they want to get. It's also a way for an organization to share resources and information while maintaining security, control, and authentication—determining who gets access to what.

Another advantage of an API is that you don't have to know the specifics of caching—how your resource is retrieved or where it comes from.

REST:

REST is a set of architectural constraints, not a protocol or a standard. API developers can implement REST in a variety of ways.

When a client request is made via a RESTful API, it transfers a representation of the state of the resource to the requester or endpoint. This information, or representation, 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.

Something else to keep in mind: Headers and parameters are also important in the HTTP methods of a RESTful API HTTP request, as they contain important identifier information as to the request's metadata, authorization, uniform resource identifier (URI), caching, cookies, and more. There are request headers and response headers, each with their own HTTP connection information and status codes.

In order for an API to be considered RESTful, it has to conform to these criteria:

- A client-server architecture made up of clients, servers, and resources, with requests managed through HTTP.
- Stateless client-server communication, meaning no client information is stored between get requests and each request is separate and unconnected.
- Cacheable data that streamlines client-server interactions.
- A uniform interface between components so that information is transferred in a standard form. This requires that:
    - resources requested are identifiable and separate from the representations sent to the client.
    - resources can be manipulated by the client via the representation they receive because the representation contains enough information to do so.
    - self-descriptive messages returned to the client have enough information to describe how the client should process it.
    - hypertext/hypermedia is available, meaning that after accessing a resource the client should be able to use hyperlinks to find all other currently available actions they can take.
- A layered system that organizes each type of server (those responsible for security, load-balancing, etc.) involved the retrieval of requested information into hierarchies, invisible to the client.
- Code-on-demand (optional): the ability to send executable code from the server to the client when requested, extending client functionality.

CRUD Operations:

Within computer programming, the acronym CRUD stands for create, read, update and delete. These are the four basic functions of persistent storage. Also, each letter in the acronym can refer to all functions executed in relational database applications and mapped to a standard HTTP method, SQL statement or DDS operation.

It can also describe user-interface conventions that allow viewing, searching and modifying information through computer-based forms and reports. In essence, entities are read, created, updated and deleted. Those same entities can be modified by taking the data from a service and changing the setting properties before sending the data back to the service for an update. Plus, CRUD is data-oriented and the standardized use of HTTP action verbs.

Most applications have some form of CRUD functionality. In fact, every programmer has had to deal with CRUD at some point. Not to mention, a CRUD application is one that utilizes forms to retrieve and return data from a database.

The first reference to CRUD operations came from Haim Kilov in 1990 in an article titled, "From semantic to object-oriented data modeling." However, the term was first made popular by James Martin's 1983 book, *Managing the Data-base Environment*. Here's a breakdown:

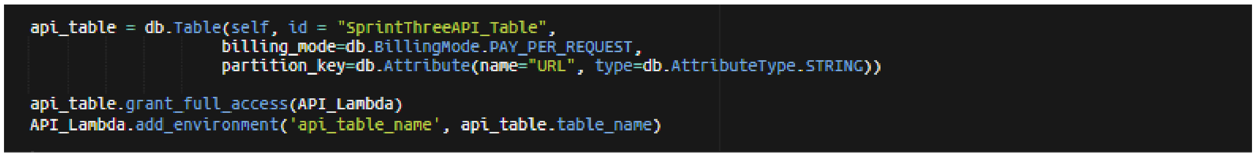
- CREATE procedures: Performs the INSERT statement to create a new record.
- READ procedures: Reads the table records based on the primary keynoted within the input parameter.
- UPDATE procedures: Executes an UPDATE statement on the table based on the specified primary key for a record within the WHERE clause of the statement.
- DELETE procedures: Deletes a specified row in the WHERE clause.

Using a REST API to perform CRUD operations on a DynamoDB Table:

This project is almost identical to Sprint 2, except for the addition of another DynamoDB table, a lambda function and an API gateway.

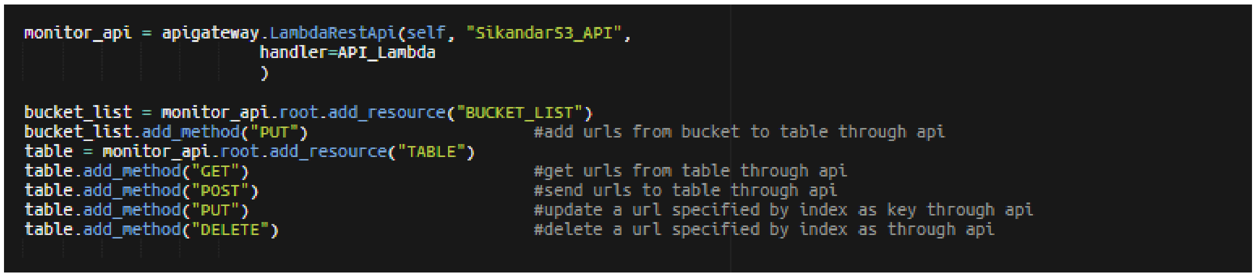
DynamoDB Table to store URLs:

In sprint 3, we want to create an API that implements CRUD operations for user to read, create, update and delete URLs in a DynamoDB table. Hence, we create a table for this purpose and grant permissions to a lambda function that can receive API requests and perform the corresponding operations.



```
api_table = db.Table(self, id = "SprintThreeAPI_Table",
                     billing_mode=db.BillingMode.PAY_PER_REQUEST,
                     partition_key=db.Attribute(name="URL", type=db.AttributeType.STRING))

api_table.grant_full_access(API_Lambda)
API_Lambda.add_environment('api_table_name', api_table.table_name)
```

API gateway with Lambda backend:



```
monitor_api = apigateway.LambdaRestApi(self, "SikandarS3_API",
                        handler=API_Lambda
                        )

bucket_list = monitor_api.root.add_resource("BUCKET_LIST")
bucket_list.add_method("PUT")                    #add urls from bucket to table through api
table = monitor_api.root.add_resource("TABLE")
table.add_method("GET")                          #get urls from table through api
table.add_method("POST")                         #send urls to table through api
table.add_method("PUT")                          #update a url specified by index as key through api
table.add_method("DELETE")                       #delete a url specified by index as through api
```

The AWS API Gateway resource exposes many ways to build Restful API without using any other framework like Flask. By using the LambdaRestApi function, we can define an API which can invoke a lambda function to process the requests received through it.

In any API, we have to define root paths for our methods or requests. The root paths are equivalent of resources i.e. we should make a separate root path for every unique resource we plan to operate on. We have defined a root for BUCKET_LIST wherein we add a method to create URLs based on a file in an S3 bucket. We define a root named TABLE which updates the table with URLs sent via the requests.

Lambda Backend:

This is the driving function for this API. It adds functionality to each of the methods we have exposed in the API. We will examine the results of each of the API methods implemented in the lambda handler:

```python
import os
import boto3
import json
from S3bucket import S3Bucket as sb
import botocore
from boto3.dynamodb.conditions import Attr

TABLE_PATH = '/TABLE'
BUCKET_PATH = '/BUCKET_LIST'

def lambda_handler(event, context):
    path = event['path']
    httpMethod = event['httpMethod']
    table_name = os.getenv("api_table_name")
    response = {}
    if path == BUCKET_PATH and httpMethod == 'PUT':
        bucket_name = event['queryStringParameters']["bucket_name"]
        object_key = event['queryStringParameters']["object_key"]
        response = bucket_to_table(bucket_name, object_key, table_name)

    elif path == TABLE_PATH and httpMethod == 'GET':
        url = event['queryStringParameters']['url']
        response = fetch_url(table_name, url)

    elif path == TABLE_PATH and httpMethod == 'POST':
        url_name = event['queryStringParameters']['url_name']
        new_url = event['queryStringParameters']['url']
        response = add_url(table_name, url_name, new_url)

    elif path == TABLE_PATH and httpMethod == 'PUT':
        url_to_update = event['queryStringParameters']['url']
        url_name = event['queryStringParameters']['url_name']
        body = json.loads(event['body'])
        updated_url_name = body['updated_url_name']
        updated_url = body['updated_url']
        response = update_url(table_name, url_name, url_to_update, updated_url_name,
updated_url)

    elif path == TABLE_PATH and httpMethod == 'DELETE':
        url_name = event['queryStringParameters']['url_name']
        url_del = event['queryStringParameters']['url']
        response = delete_url(table_name, url_name, url_del)

    else:
        response = {
                    "statusCode":200,
                    "body": "request failed"
                    }

    return response
```

1. Construct Response Helper Method:

This is a simple helper function that is called upon successful request operation. It sends a successful request status code and the msg that is passed to it as a parameter back to the client.

```
def construct_response(msg):
    response = {
                "statusCode":200,
                "body": msg
               }
    return response
```

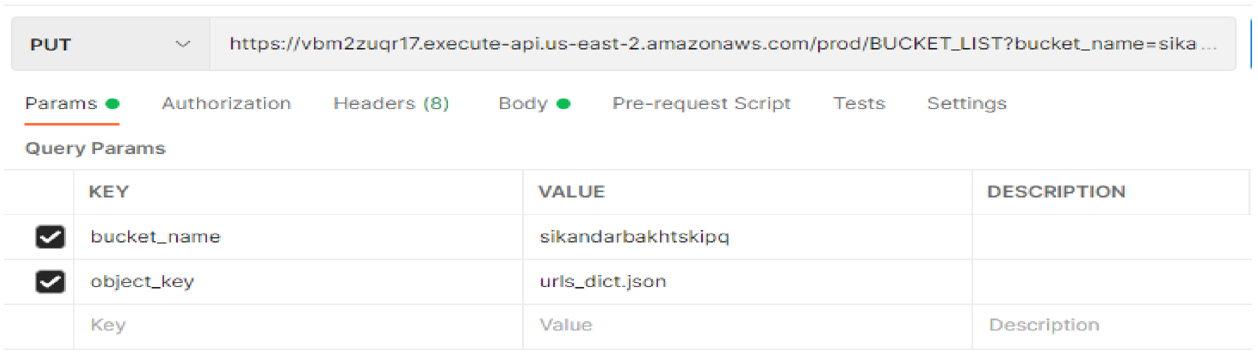
2. Bucket to Table Method:

<u>CRUD: BUCKET_LIST/PUT</u>

This method takes bucket_name as input, the json file as object_key and name of table as table_name. It will parse through the json file and put the URLs found in the table.

```
def bucket_to_table(bucket_name, object_key, table_name):

    URLS = sb(bucket_name).load(object_key)
    K=list(URLS['URLS'][0].keys())

    dynamodb = boto3.resource('dynamodb')
    table = dynamodb.Table(table_name)

    for i in range(len(K)):
        table.put_item(Item = {
                                'URL': URLS['URLS'][0][K[i]],
                                'Name': K[i]})

    msg = f"{len(K)} urls added/updated"
 return construct_response(msg)
```

Request Response

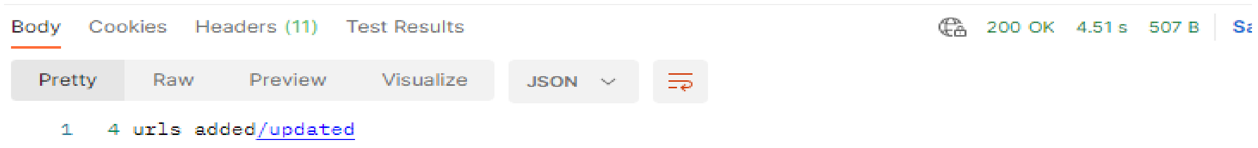


PUT https://vbm2zuqr17.execute-api.us-east-2.amazonaws.com/prod/BUCKET_LIST?bucket_name=sika ...

Params ● Authorization Headers (8) Body ● Pre-request Script Tests Settings

Query Params

	KEY	VALUE	DESCRIPTION
✓	bucket_name	sikandarbakhtskipq	
✓	object_key	urls_dict.json	
	Key	Value	Description

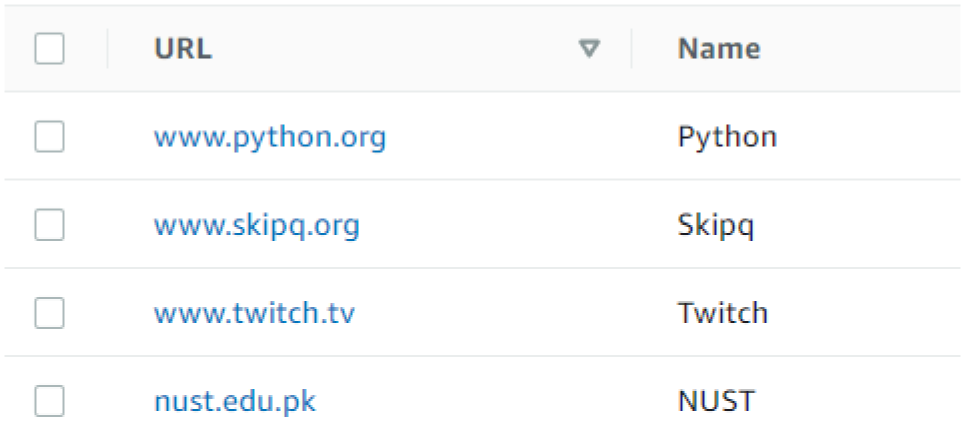


Body Cookies Headers (11) Test Results — 200 OK 4.51 s 507 B

Pretty Raw Preview Visualize JSON

```
4 urls added/updated
```

DynamoDB Table result:

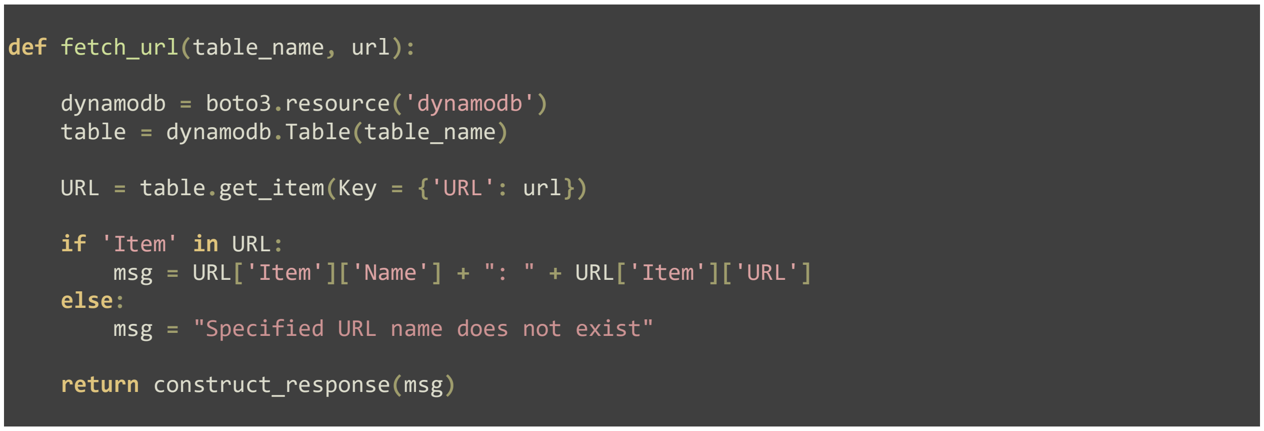


URL	Name
www.python.org	Python
www.skipq.org	Skipq
www.twitch.tv	Twitch
nust.edu.pk	NUST

3. Fetch URL Method:

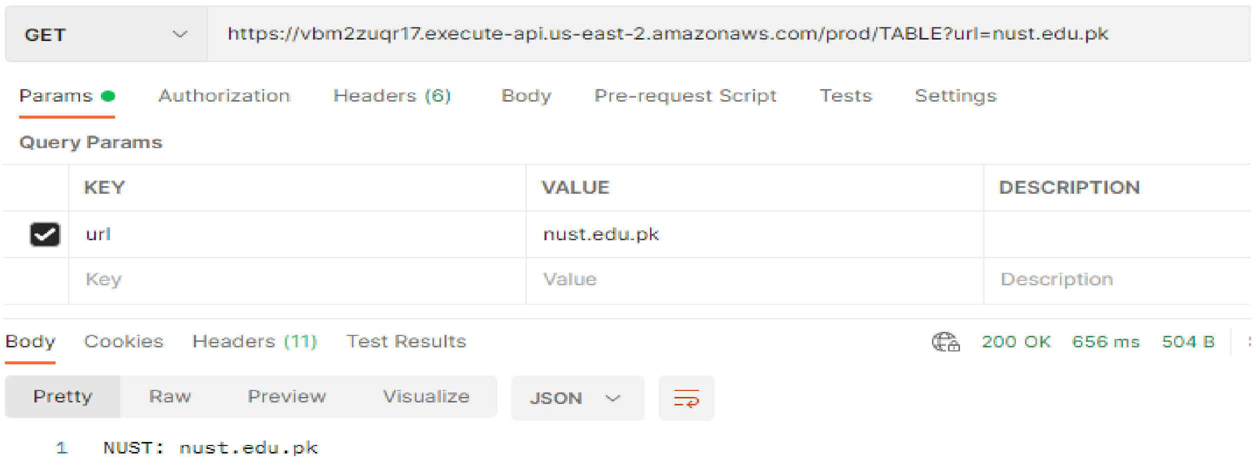
CRUD: TABLE/GET

This method queries the table given by table_name for an item by its key value ‘url’. It returns the URL name and URL if it is found in record, else a message stating that it does not exist is returned.



```python
def fetch_url(table_name, url):

    dynamodb = boto3.resource('dynamodb')
    table = dynamodb.Table(table_name)

    URL = table.get_item(Key = {'URL': url})

    if 'Item' in URL:
        msg = URL['Item']['Name'] + ": " + URL['Item']['URL']
    else:
        msg = "Specified URL name does not exist"

    return construct_response(msg)
```

Request response when URL exists:



GET https://vbm2zuqr17.execute-api.us-east-2.amazonaws.com/prod/TABLE?url=nust.edu.pk

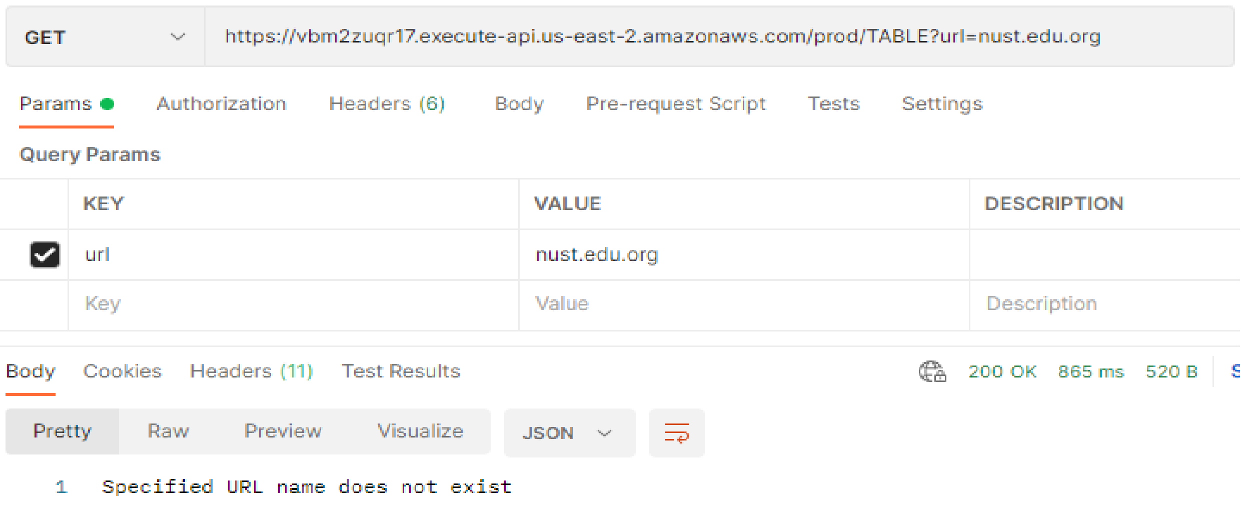
Query Params

	KEY	VALUE	DESCRIPTION
✓	url	nust.edu.pk	
	Key	Value	Description

200 OK 656 ms 504 B

```
NUST: nust.edu.pk
```

Request response when URL does not exist:



GET https://vbm2zuqr17.execute-api.us-east-2.amazonaws.com/prod/TABLE?url=nust.edu.org

Query Params

	KEY	VALUE	DESCRIPTION
✓	url	nust.edu.org	
	Key	Value	Description

200 OK 865 ms 520 B

```
Specified URL name does not exist
```

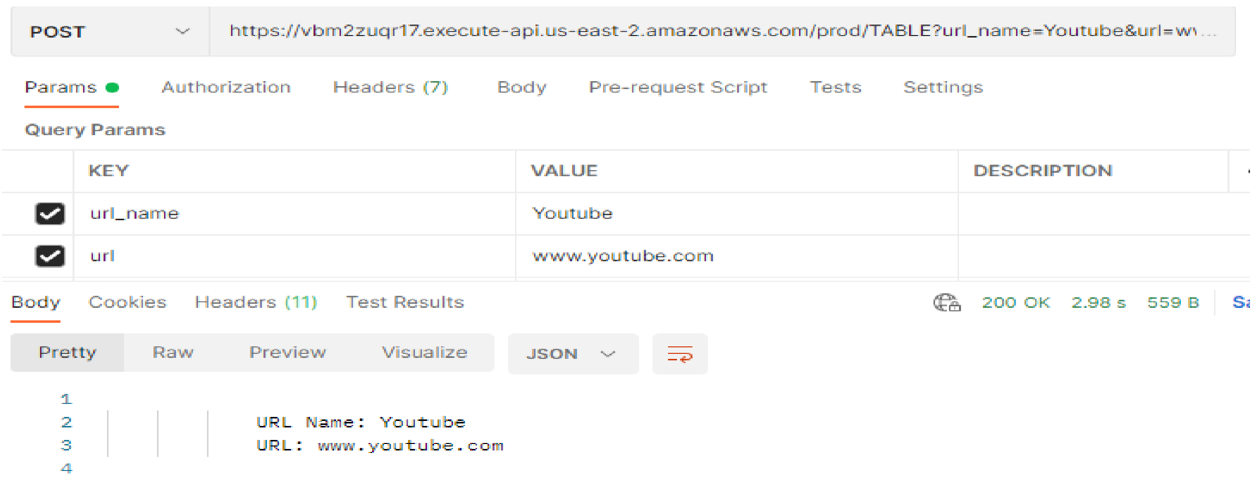
4. Add URL method:

CRUD: TABLE/POST:

This method inserts a new url passed to it via query parameter with additional information such as url name passed as url_name into the table specified by table_name. It adds the url to table if it is unique i.e. does not exist in table already, else it returns the message that URL already exists.

```
def add_url(table_name, url_name, new_url):

    dynamodb = boto3.resource('dynamodb')
    table = dynamodb.Table(table_name)

    try:
        table.put_item (Item = {'URL': new_url,
                                 'Name' : url_name
                               },
                        ConditionExpression=Attr('URL').ne(new_url)
                        )
        msg = f'''
            URL Name: {url_name}
            URL: {new_url}
        '''

    except botocore.exceptions.ClientError as e:

        if e.response['Error']['Code'] == 'ConditionalCheckFailedException':

            msg = f'URL: {new_url} already exists'

    return construct_response(msg)
```

Request response when URL does not exist in table:


POST https://vbm2zuqr17.execute-api.us-east-2.amazonaws.com/prod/TABLE?url_name=Youtube&url=w…

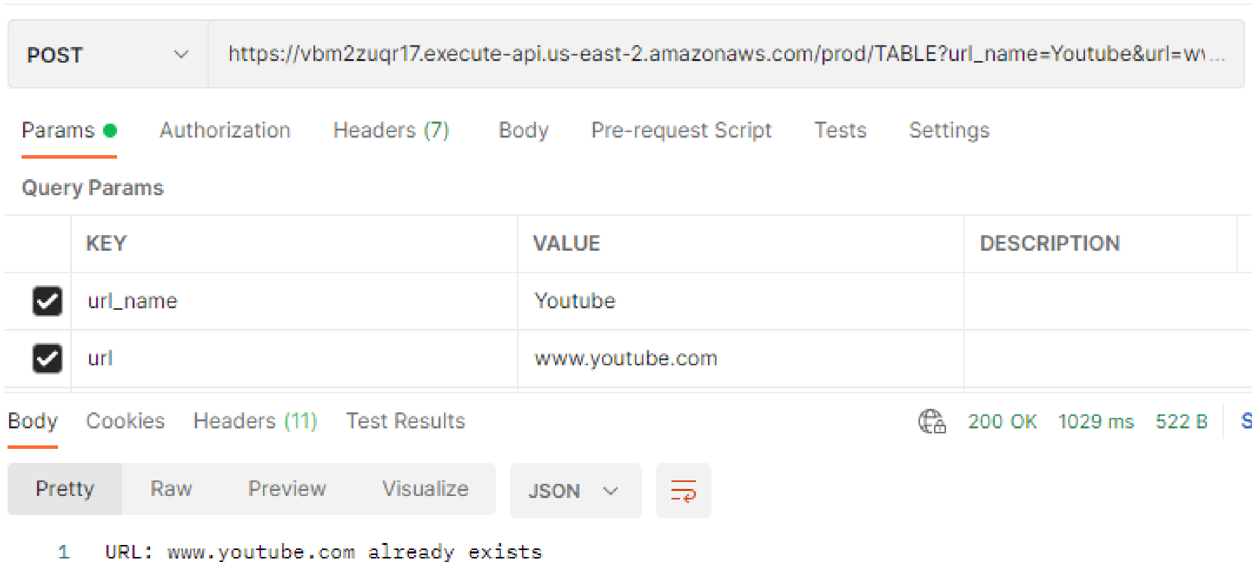
Query Params

	KEY	VALUE	DESCRIPTION
✓	url_name	Youtube	
✓	url	www.youtube.com	

200 OK 2.98 s 559 B

```
URL Name: Youtube
URL: www.youtube.com
```

Request response when URL does exist in table:



POST https://vbm2zuqr17.execute-api.us-east-2.amazonaws.com/prod/TABLE?url_name=Youtube&url=w...

Params Authorization Headers (7) Body Pre-request Script Tests Settings

Query Params

	KEY	VALUE	DESCRIPTION
✓	url_name	Youtube	
✓	url	www.youtube.com	

Body Cookies Headers (11) Test Results — 200 OK 1029 ms 522 B

Pretty Raw Preview Visualize JSON

```
URL: www.youtube.com already exists
```

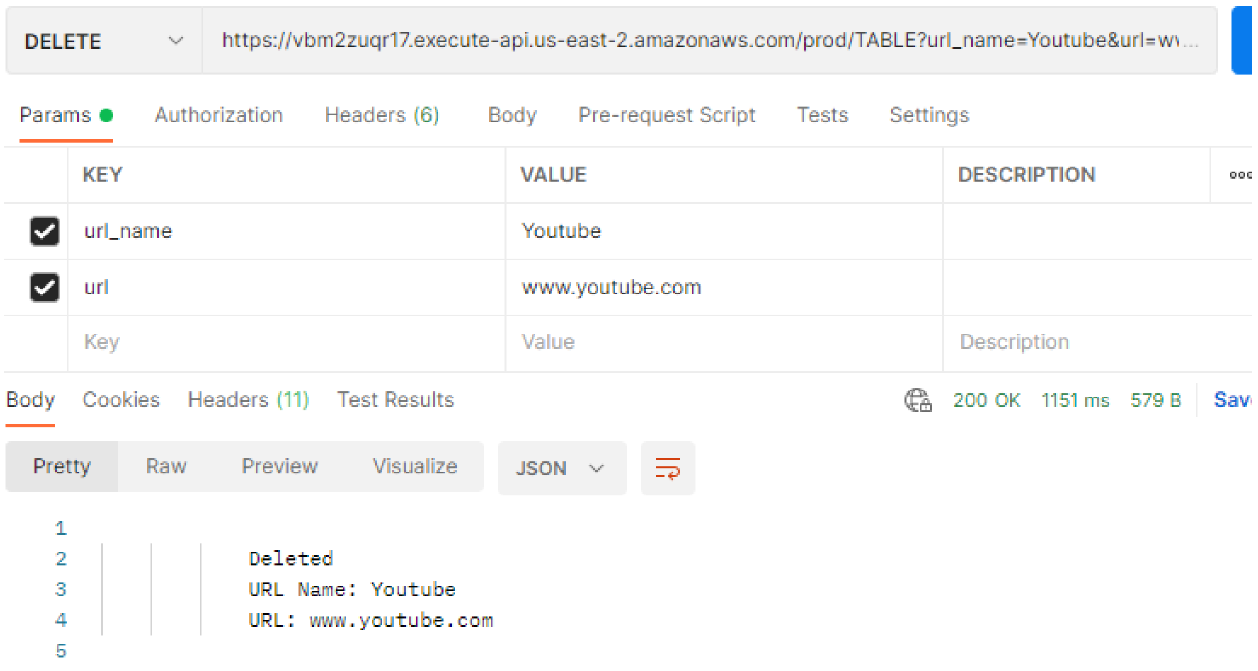
5. Delete URL Method:

CRUD: TABLE/DELETE

This method takes a URL and deletes it if found in table, else passes message stating that URL does not exist.

```python
def delete_url(table_name, url_name, url_del):

    dynamodb = boto3.resource('dynamodb')
    table = dynamodb.Table(table_name)

    try:
        table.delete_item (Key = {'URL': url_del,
                                },
                        ConditionExpression=Attr('URL').eq(url_del))
        msg = f'''
            Deleted
            URL Name: {url_name}
            URL: {url_del}
        '''

    except botocore.exceptions.ClientError as e:

        if e.response['Error']['Code'] == 'ConditionalCheckFailedException':
            msg = f'URL: {url_del} does not exist'

    return construct_response(msg)
```

Request Response when URL exists:



DELETE https://vbm2zuqr17.execute-api.us-east-2.amazonaws.com/prod/TABLE?url_name=Youtube&url=w...

Params ● Authorization Headers (6) Body Pre-request Script Tests Settings

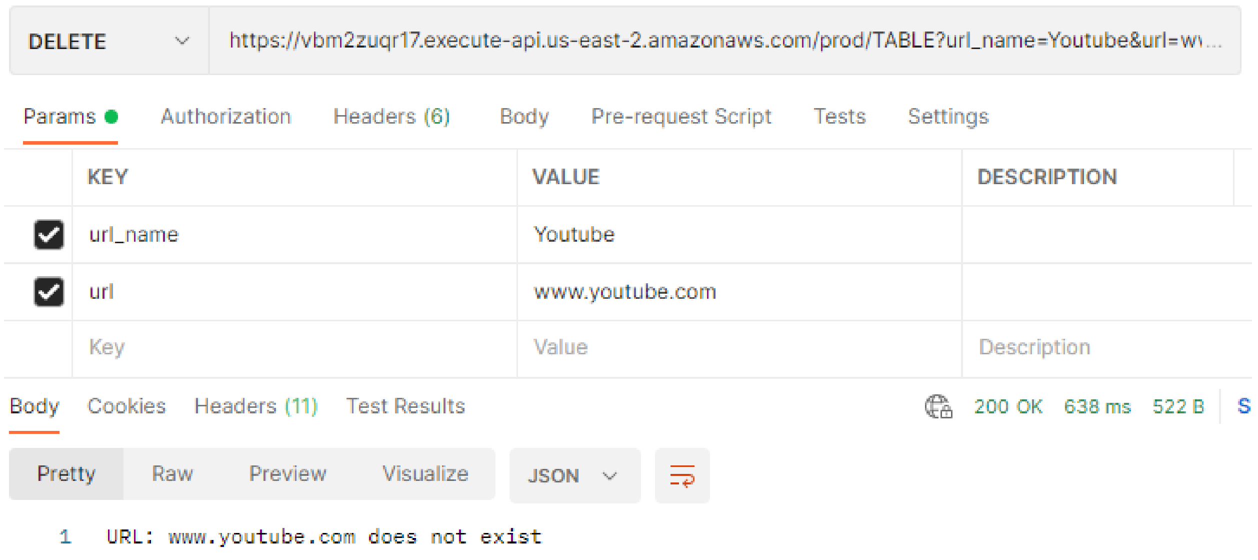
	KEY	VALUE	DESCRIPTION
✓	url_name	Youtube	
✓	url	www.youtube.com	
	Key	Value	Description

Body Cookies Headers (11) Test Results 200 OK 1151 ms 579 B Sav

Pretty Raw Preview Visualize JSON

```

            Deleted
            URL Name: Youtube
            URL: www.youtube.com

```

Request response when URL does not exists:



DELETE https://vbm2zuqr17.execute-api.us-east-2.amazonaws.com/prod/TABLE?url_name=Youtube&url=w...

Params ● Authorization Headers (6) Body Pre-request Script Tests Settings

	KEY	VALUE	DESCRIPTION
✓	url_name	Youtube	
✓	url	www.youtube.com	
	Key	Value	Description

Body Cookies Headers (11) Test Results 200 OK 638 ms 522 B S

Pretty Raw Preview Visualize JSON

```
URL: www.youtube.com does not exist
```

6. Update URL Method:

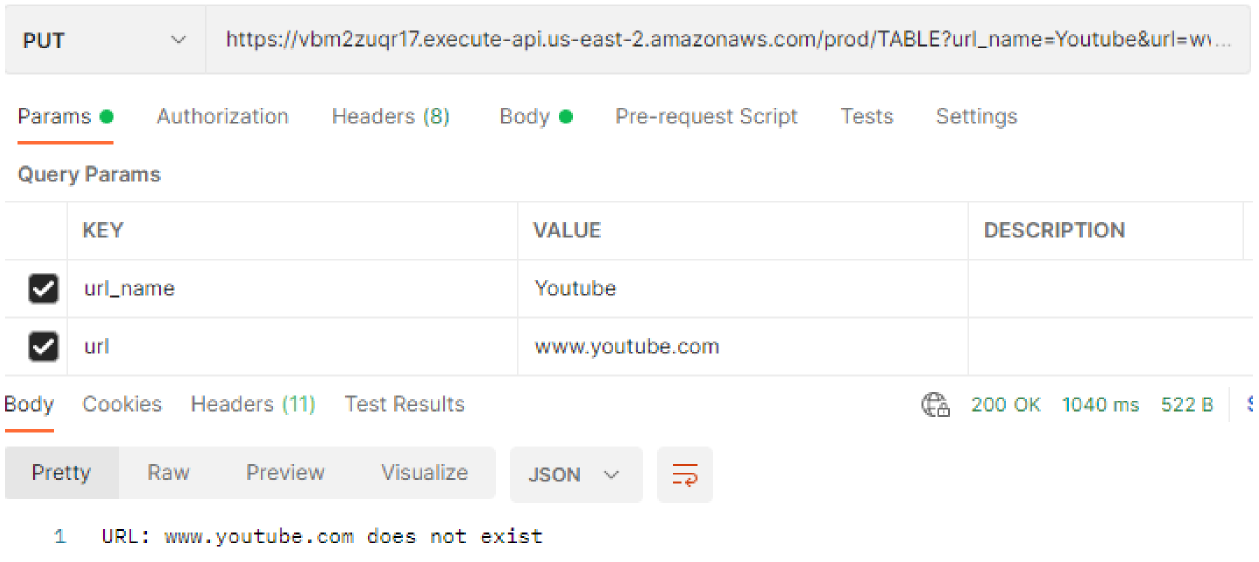
<u>CRUD: TABLE/PUT</u>

This method updates an item if it exists in table. If it does not exist, it will send a message stating it does not exist. If we tried to update an item with the same info, it will state that it already exists.

To pass the requested update, we need to pass the request with a body.

```
def update_url(table_name, url_name, url_to_update, updated_url_name, updated_url):

    dynamodb = boto3.resource('dynamodb')
    table = dynamodb.Table(table_name)
    msg = ''

    try:
        table.delete_item (Key = {'URL': url_to_update},
                        ConditionExpression=Attr('URL').eq(url_to_update))

    except botocore.exceptions.ClientError as e:
        if e.response['Error']['Code'] == 'ConditionalCheckFailedException':
            msg = f'URL: {url_to_update} does not exist'

    if msg != f'URL: {url_to_update} does not exist':
        try:
            table.put_item (Item = {'URL': updated_url,
                                    'Name' : updated_url_name},
                            ConditionExpression=Attr('URL').ne(updated_url))
            msg = f'''
            Update from:
                URL: {url_to_update}
                URL Name: {url_name}
            to:
                URL Name: {updated_url_name}
                URL: {updated_url}
            '''
        except botocore.exceptions.ClientError as e:

            if e.response['Error']['Code'] == 'ConditionalCheckFailedException':
                msg = f'URL: {updated_url} already exists'

    return construct_response(msg)
```

Request response when URL to update does not exist:


PUT https://vbm2zuqr17.execute-api.us-east-2.amazonaws.com/prod/TABLE?url_name=Youtube&url=w...

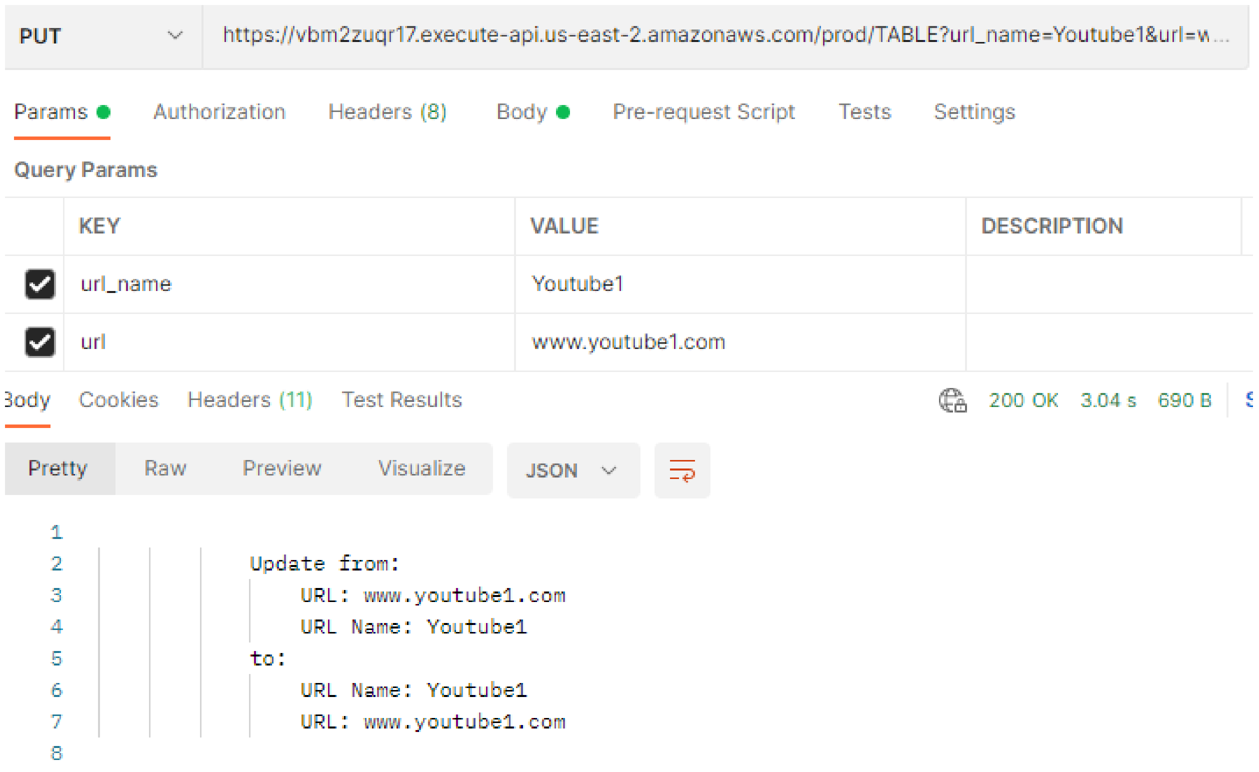
Query Params

	KEY	VALUE	DESCRIPTION
✓	url_name	Youtube	
✓	url	www.youtube.com	

200 OK 1040 ms 522 B

```
URL: www.youtube.com does not exist
```

Request response when URL to update exists:


PUT https://vbm2zuqr17.execute-api.us-east-2.amazonaws.com/prod/TABLE?url_name=Youtube1&url=w...

Query Params

	KEY	VALUE	DESCRIPTION
✓	url_name	Youtube1	
✓	url	www.youtube1.com	

200 OK 3.04 s 690 B

```

Update from:
    URL: www.youtube1.com
    URL Name: Youtube1
to:
    URL Name: Youtube1
    URL: www.youtube1.com

```

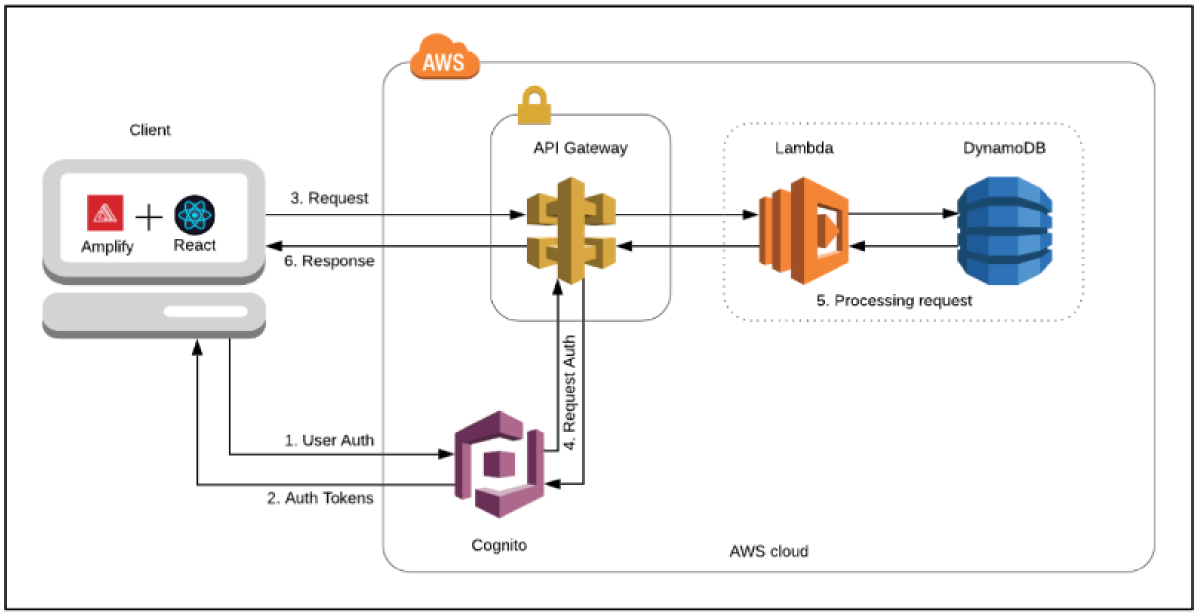
Sprint 4: Documentation

React JS Based Front End for Sprint 3 CRUD API

Authored by: Sikandar Bakht Kiani

Sprint Objective:

In this sprint, we will develop a React JS based static web app frontend for our RESTful CRUD API built in sprint 3. The app will be responsible for making GET requests to read data from our database. It should also be capable of querying a specific URL in our database.



Implementation

Installing Node.js:

We don’t need node.js itself for this project. However, node.js has a very popular package manager named npm which is very useful for setting up react app and installing additional modules.

To install node.js, go to https://nodejs.org/en/ and download the latest release of node.js. Run the installer and follow the simple installation wizard to complete installation.

Installing React and creating our React App:

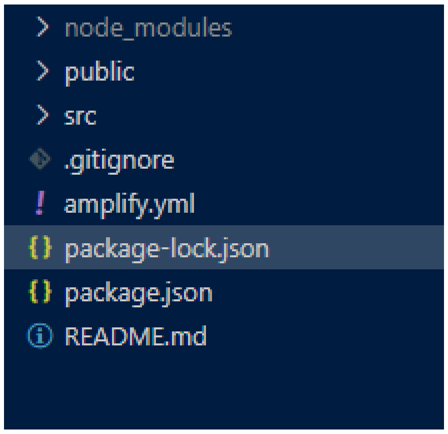
Open a text editor like VS Code. Download a utility called create-react-app by running the following command in a terminal

```
npm install -g create-react-app
```

Now, change directory to your choice and run:

```
npm create-react-app skipq-app
cd skipq-app
```

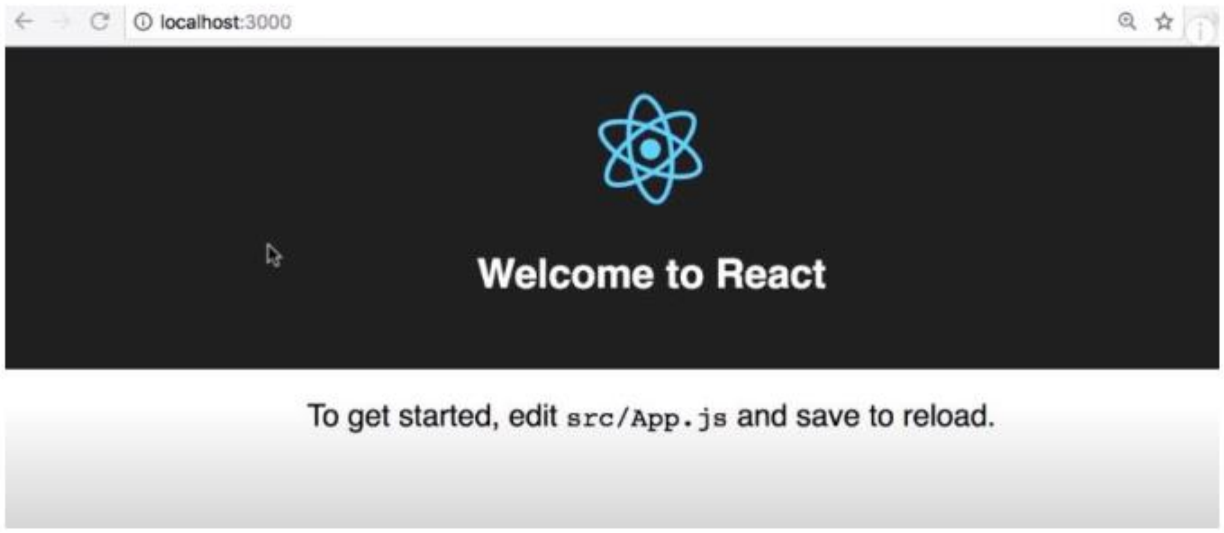
You should see a folder structure like this.



Now, run the command:

```
npm start
```

It should start the basic UI template every react app starts with:



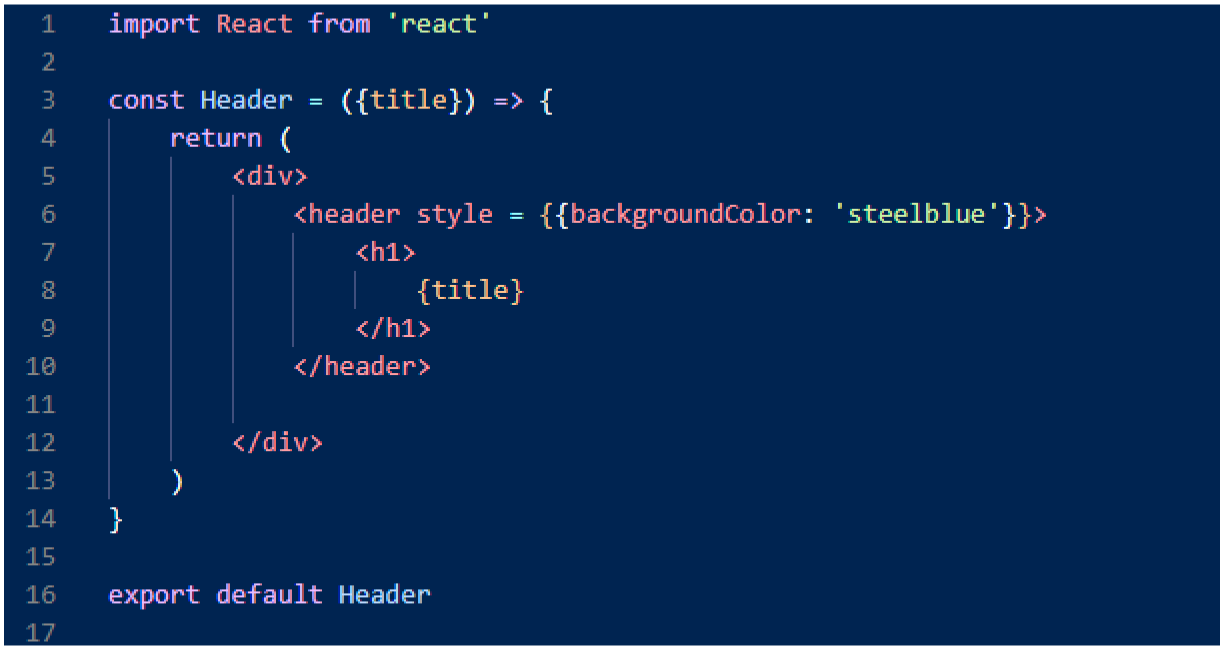
To get started, edit `src/App.js` and save to reload.

Creating support modules:

In addition, the default entry point App.js, we create 3 support modules namely, Header.js, Button.js and URL.js.

- Header.js:

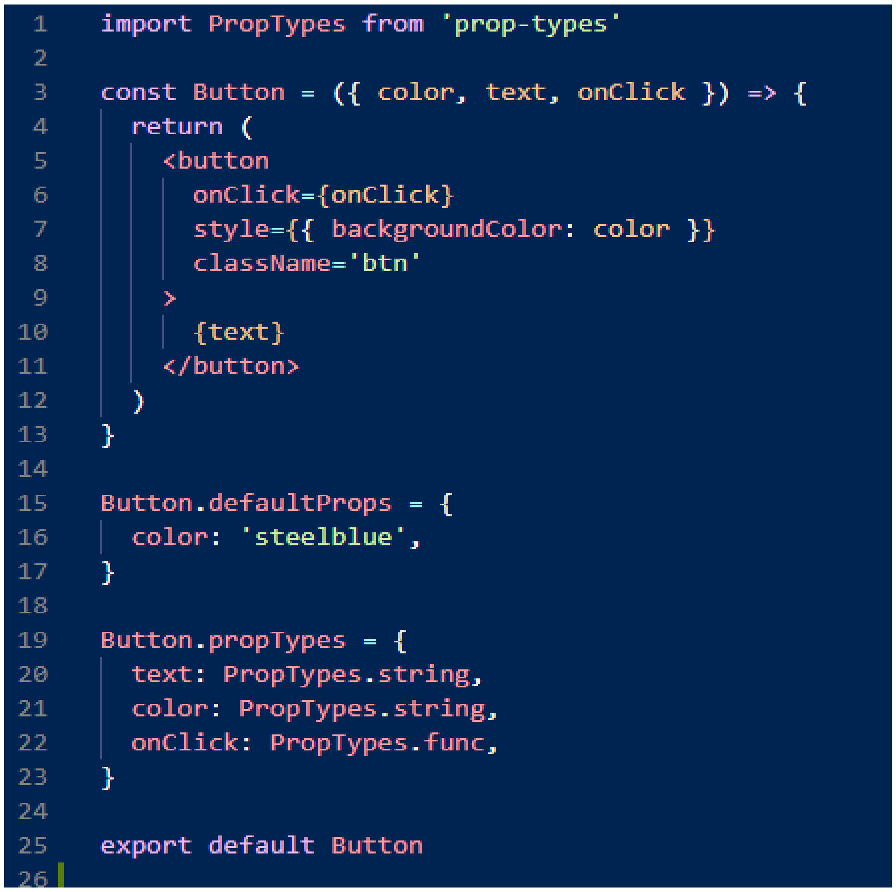
It is simply the title of the react app in a colored bar.



```
import React from 'react'

const Header = ({title}) => {
    return (
        <div>
            <header style = {{backgroundColor: 'steelblue'}}>
                <h1>
                    {title}
                </h1>
            </header>

        </div>
    )
}

export default Header

```

- Button.js:

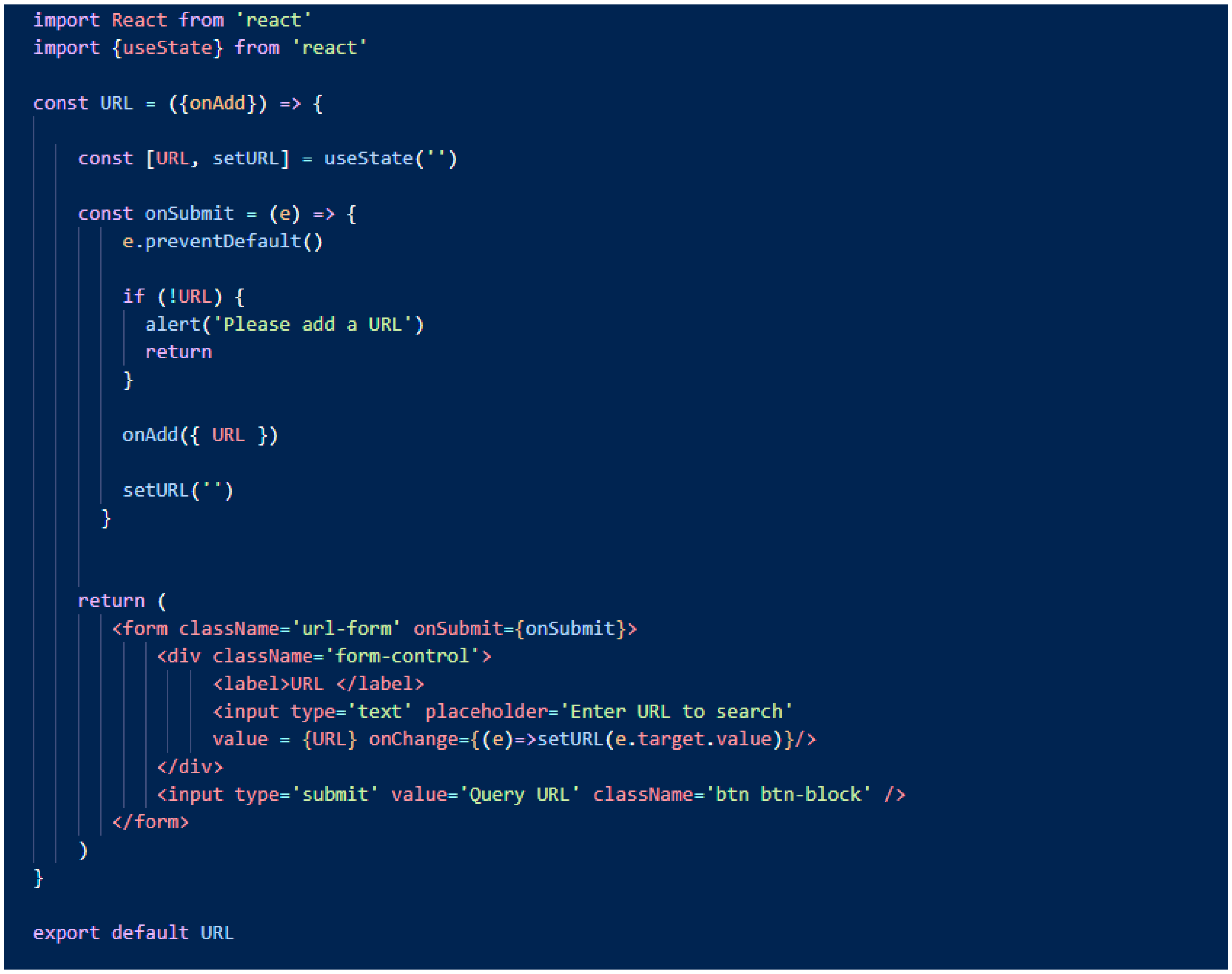
It is a simple interactive button with an onClick prop that can be used to pass a function that carries out certain functionality on click.



```
import PropTypes from 'prop-types'

const Button = ({ color, text, onClick }) => {
  return (
    <button
      onClick={onClick}
      style={{ backgroundColor: color }}
      className='btn'
    >
      {text}
    </button>
  )
}

Button.defaultProps = {
  color: 'steelblue',
}

Button.propTypes = {
  text: PropTypes.string,
  color: PropTypes.string,
  onClick: PropTypes.func,
}

export default Button

```

URL.js:

This module is used to store a function state, "URL" as the value of the input received by the form.



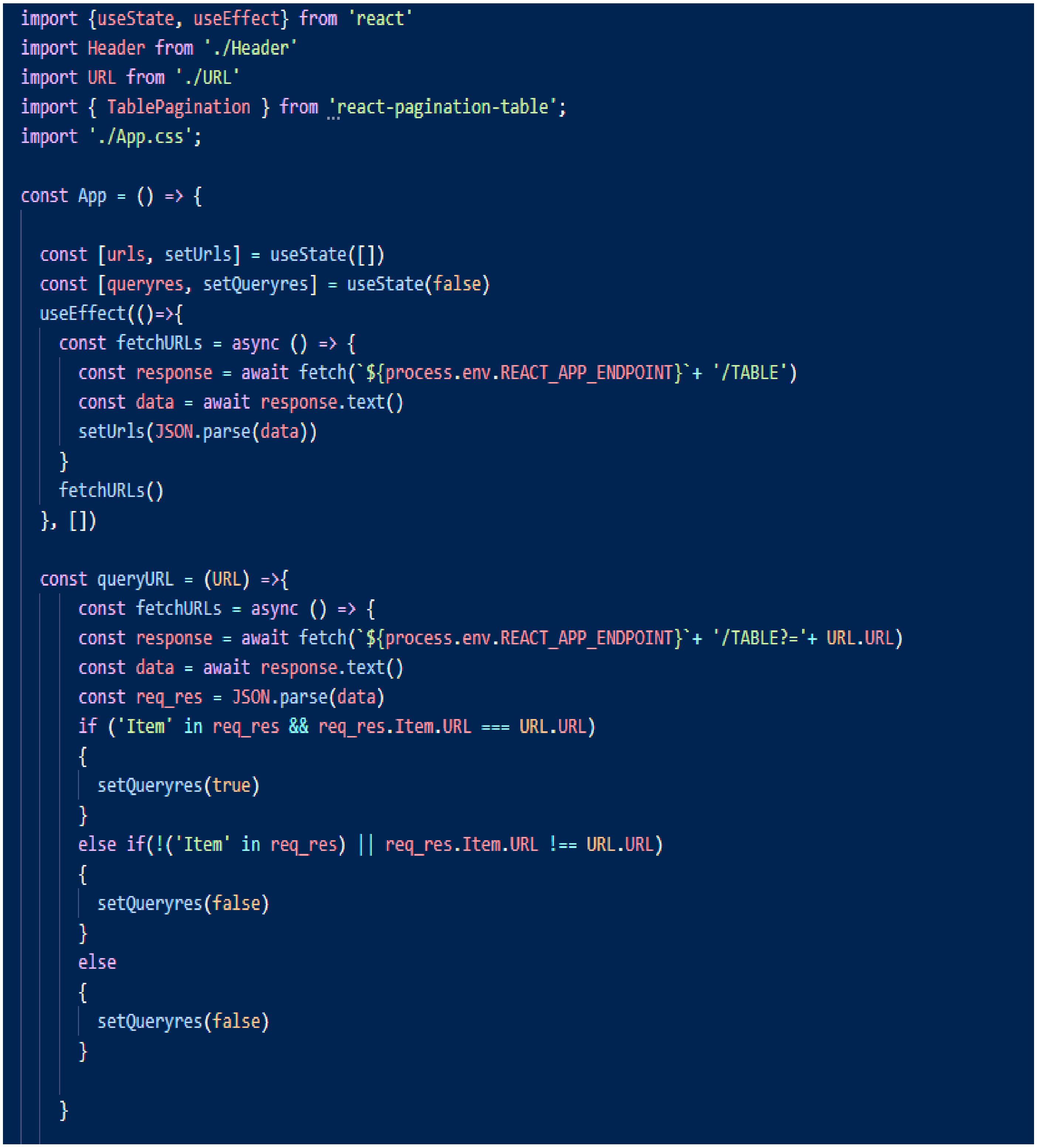
```
import React from 'react'
import {useState} from 'react'

const URL = ({onAdd}) => {

    const [URL, setURL] = useState('')

    const onSubmit = (e) => {
        e.preventDefault()

        if (!URL) {
          alert('Please add a URL')
          return
        }

        onAdd({ URL })

        setURL('')
      }


    return (
      <form className='url-form' onSubmit={onSubmit}>
          <div className='form-control'>
              <label>URL </label>
              <input type='text' placeholder='Enter URL to search'
              value = {URL} onChange={(e)=>setURL(e.target.value)}/>
          </div>
          <input type='submit' value='Query URL' className='btn btn-block' />
      </form>
    )
}

export default URL
```

Table with Pagination:

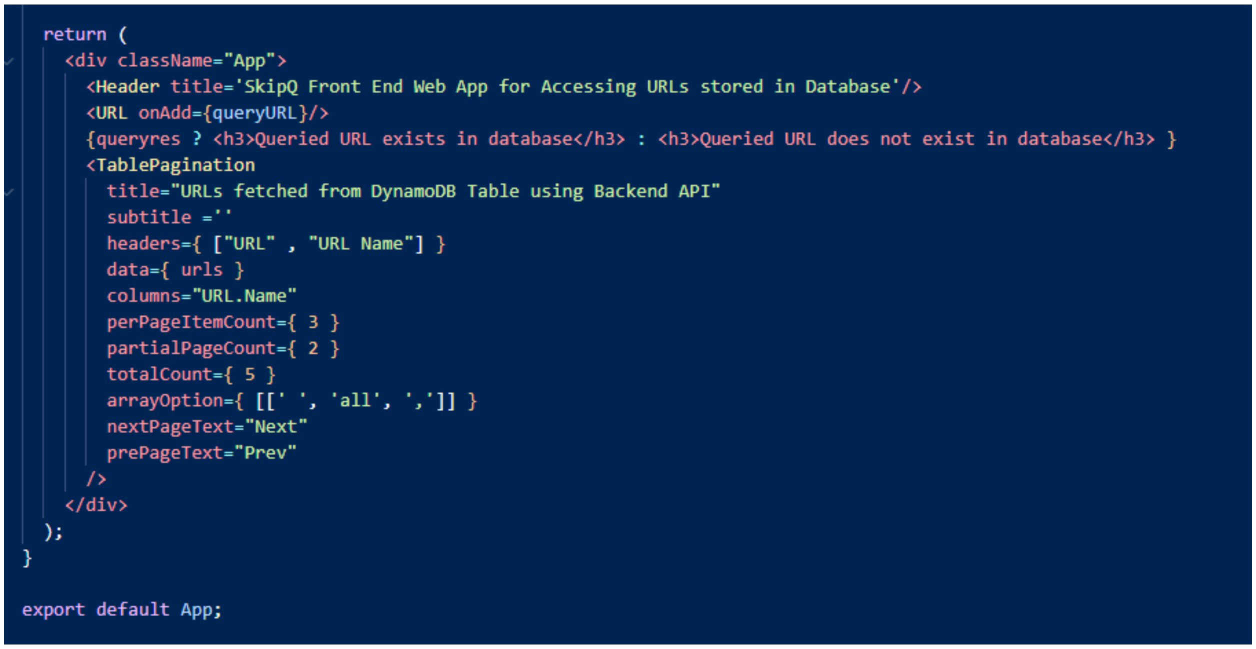
In this project, we need to create a table that displays all items in our database with pagination. To achieve this, we can install a simple module called react-pagination-table with:

```
npm install --save react-pagination-table
```

We also need to create a function to send a GET request to our API and pass our table the relevant data. We use the fetch library with async function to get our results. Here is the complete code for App.js



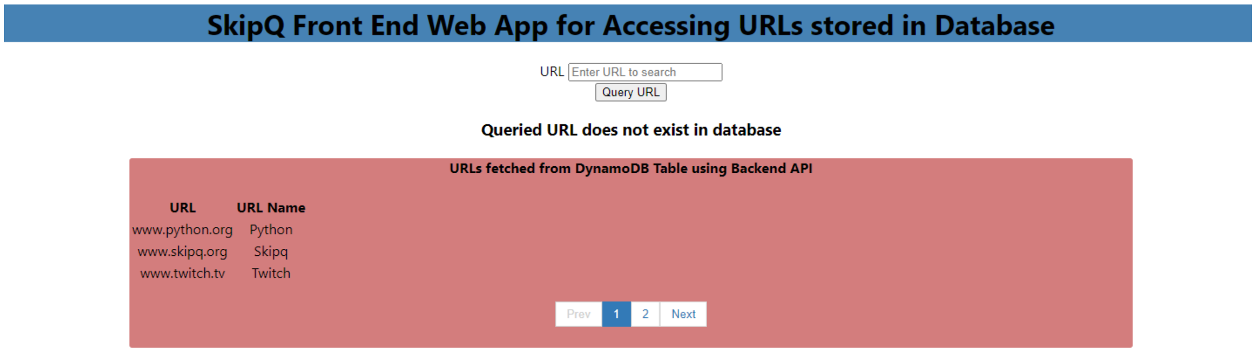
```
import {useState, useEffect} from 'react'
import Header from './Header'
import URL from './URL'
import { TablePagination } from 'react-pagination-table';
import './App.css';

const App = () => {

  const [urls, setUrls] = useState([])
  const [queryres, setQueryres] = useState(false)
  useEffect(()=>{
    const fetchURLs = async () => {
      const response = await fetch(`${process.env.REACT_APP_ENDPOINT}`+ '/TABLE')
      const data = await response.text()
      setUrls(JSON.parse(data))
    }
    fetchURLs()
  }, [])

  const queryURL = (URL) =>{
      const fetchURLs = async () => {
      const response = await fetch(`${process.env.REACT_APP_ENDPOINT}`+ '/TABLE?='+ URL.URL)
      const data = await response.text()
      const req_res = JSON.parse(data)
      if ('Item' in req_res && req_res.Item.URL === URL.URL)
      {
        setQueryres(true)
      }
      else if(!('Item' in req_res) || req_res.Item.URL !== URL.URL)
      {
        setQueryres(false)
      }
      else
      {
        setQueryres(false)
      }

    }
```



```
  return (
    <div className="App">
      <Header title='SkipQ Front End Web App for Accessing URLs stored in Database'/>
      <URL onAdd={queryURL}/>
      {queryres ? <h3>Queried URL exists in database</h3> : <h3>Queried URL does not exist in database</h3> }
      <TablePagination
        title="URLs fetched from DynamoDB Table using Backend API"
        subtitle =''
        headers={ ["URL" , "URL Name"] }
        data={ urls }
        columns="URL.Name"
        perPageItemCount={ 3 }
        partialPageCount={ 2 }
        totalCount={ 5 }
        arrayOption={ [[' ', 'all', ',']] }
        nextPageText="Next"
        prePageText="Prev"
      />
    </div>
  );
}

export default App;
```

Now, we can see that the useEffect hook is used to get all the URLs present in the table whereas the fetchURL async function is used to query specific URLs. The output is shown below:

<u>1st Page:</u>



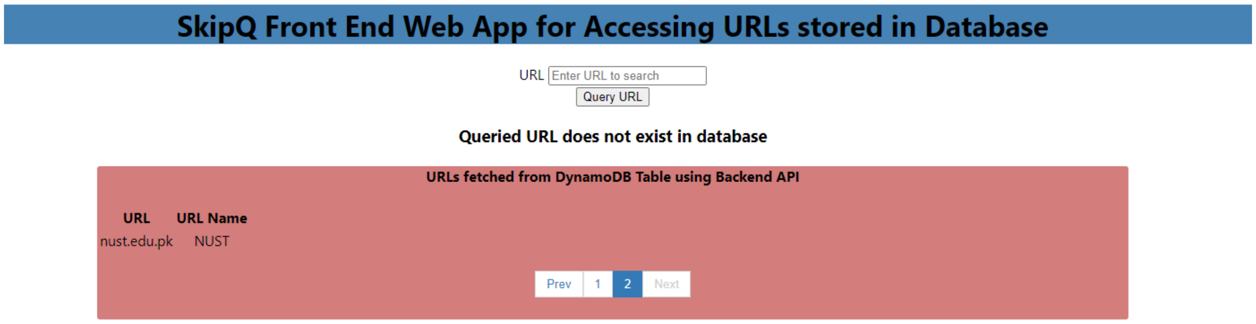
SkipQ Front End Web App for Accessing URLs stored in Database

Queried URL does not exist in database

URLs fetched from DynamoDB Table using Backend API

URL	URL Name
www.python.org	Python
www.skipq.org	Skipq
www.twitch.tv	Twitch

<u>2nd page</u>



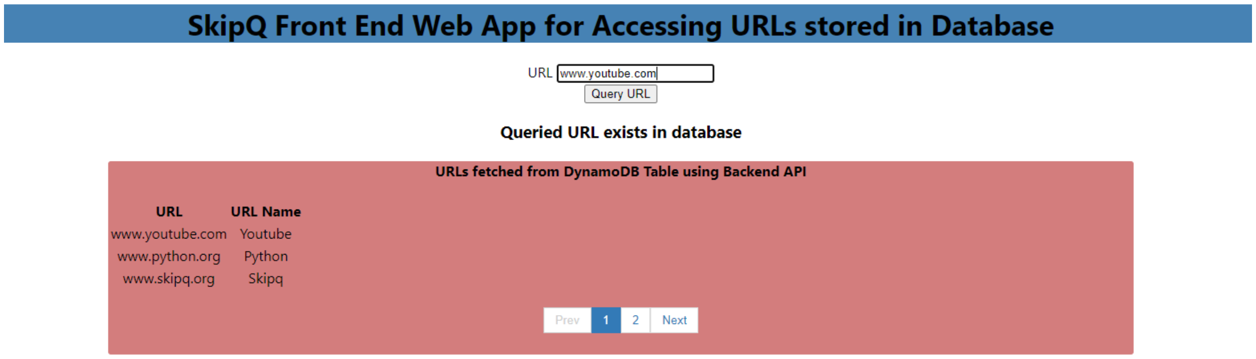
SkipQ Front End Web App for Accessing URLs stored in Database

Queried URL does not exist in database

URLs fetched from DynamoDB Table using Backend API

URL	URL Name
nust.edu.pk	NUST

If queries URL exists in database:



SkipQ Front End Web App for Accessing URLs stored in Database

URL www.youtube.com

Query URL

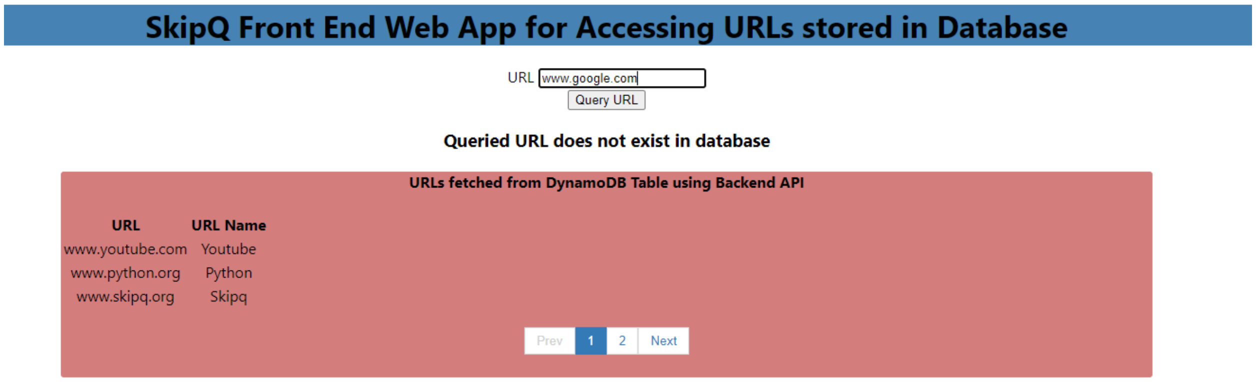
Queried URL exists in database

URLs fetched from DynamoDB Table using Backend API

URL	URL Name
www.youtube.com	Youtube
www.python.org	Python
www.skipq.org	Skipq

Prev 1 2 Next

If queried URL does not exist



SkipQ Front End Web App for Accessing URLs stored in Database

URL www.google.com

Query URL

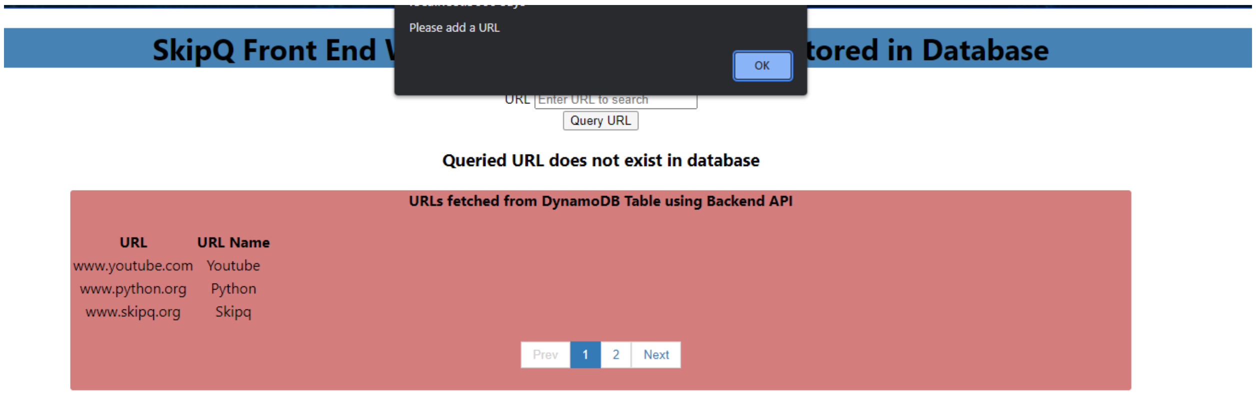
Queried URL does not exist in database

URLs fetched from DynamoDB Table using Backend API

URL	URL Name
www.youtube.com	Youtube
www.python.org	Python
www.skipq.org	Skipq

Prev 1 2 Next

If nothing is entered in the feel before pressing button.



SkipQ Front End Web App for Accessing URLs stored in Database

Please add a URL

OK

URL Enter URL to search

Query URL

Queried URL does not exist in database

URLs fetched from DynamoDB Table using Backend API

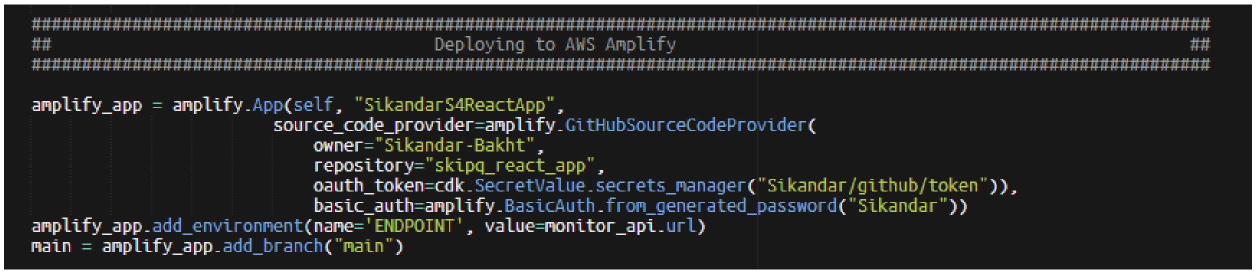
URL	URL Name
www.youtube.com	Youtube
www.python.org	Python
www.skipq.org	Skipq

Prev 1 2 Next

AWS Amplify:

AWS Amplify is a set of purpose-built tools and features that lets frontend web and mobile developers quickly and easily build full-stack applications on AWS, with the flexibility to leverage the breadth of AWS services as your use cases evolve. 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.

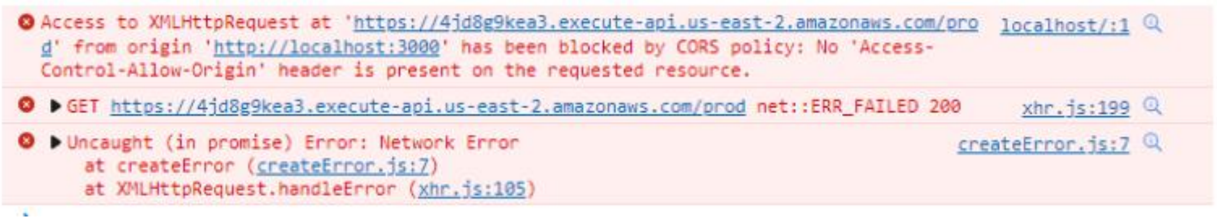
We can add the following code in our stack to deploy our react app to AWS Amplify. We have created a GitHub repository for our react app. We have also added an environment variable that sends the value of API URL to our react app.



```
##################################################################################################
##                                 Deploying to AWS Amplify                                     ##
##################################################################################################

amplify_app = amplify.App(self, "SikandarS4ReactApp",
                    source_code_provider=amplify.GitHubSourceCodeProvider(
                        owner="Sikandar-Bakht",
                        repository="skipq_react_app",
                        oauth_token=cdk.SecretValue.secrets_manager("Sikandar/github/token")),
                        basic_auth=amplify.BasicAuth.from_generated_password("Sikandar"))
amplify_app.add_environment(name='ENDPOINT', value=monitor_api.url)
main = amplify_app.add_branch("main")
```

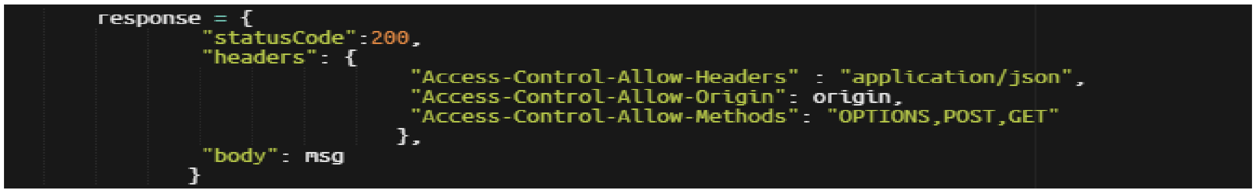
Troubleshooting:

CORS Error:



```
Access to XMLHttpRequest at 'https://4jd8g9kea3.execute-api.us-east-2.amazonaws.com/pro   localhost/:1
d' from origin 'http://localhost:3000' has been blocked by CORS policy: No 'Access-
Control-Allow-Origin' header is present on the requested resource.
▶GET https://4jd8g9kea3.execute-api.us-east-2.amazonaws.com/prod net::ERR_FAILED 200   xhr.js:199
▶Uncaught (in promise) Error: Network Error                                    createError.js:7
    at createError (createError.js:7)
    at XMLHttpRequest.handleError (xhr.js:105)
```

This error is caused by CORS policy not being configured for the API. It can be handled by adding a Access-Control-Allow-Origin: “*” to the response of API but it is a security threat. Instead we can whitelist URLs or take the incoming origin name from the request body and allow it in the response.



```
        response = {
                "statusCode":200,
                "headers": {
                                "Access-Control-Allow-Headers" : "application/json",
                                "Access-Control-Allow-Origin": origin,
                                "Access-Control-Allow-Methods": "OPTIONS,POST,GET"
                              },
                "body": msg
              }
```

Sprint 5: Documentation

Containerized API Test clients running on AWS ECS

Authored by: Sikandar Bakht Kiani

Sprint Objective:

In this sprint, we will develop API testing clients to test functionality and security of our API built in sprints 3 and 4. The clients will be based on popular testing frameworks: pyRestTest and Syntribos.

The caveat is that these frameworks are not very easy to setup on any machine. Their requirements vary from machine to machine, even across different Linux distros. This is where containers come in. We will create Docker containers for these test clients so that our clients can be easily deployed programmatically and without setting up the environments across various builds.

To deploy our containerized test clients, we can use AWS Elastic Container Registry which works similar to DockerHub. After publishing our containers to AWS ECR, we can deploy them as event triggered tasks through AWS ECS and AWS Fargate.

Technology Overview

PyRestTest

PyRestTest is a REST testing and API micro benchmarking tool. Tests are defined in basic YAML or JSON config files. It returns exit codes on failure, to slot into automated configuration management/orchestration tools.

Containers: What are they?

Containers are self-contained software packages that are used to run applications in an isolated environment. A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings.

Docker:

Docker is a container utility that can be used to create containers on both Windows and Linux. It uses the Docker Engine as the layer between the host OS and container. The service allows easily building and running containers via a simple YAML formatted Dockerfile.

Dockerfile:

Dockerfile can be considered a build script for Docker to build container images. It is a YAML formatted document that uses keywords to define what Docker needs to create an image and run the code inside.

Amazon Elastic Container Registry:

ECR is an AWS that allows you to easily store, share, and deploy your container software anywhere. With ECR, you can push container images to Amazon ECR without installing or scaling infrastructure, and pull images using any management tool. You can publish containerized applications with a single command and easily integrate your self-managed environments while accessing and distributing your images faster, reduce download times, and improve availability using a scalable, durable architecture.

Amazon Elastic Container Service:

Amazon ECS is a fully managed container orchestration service that helps you easily deploy, manage, and scale containerized applications. With ECS, you can plan, schedule, and execute batch computing workloads across the full range of AWS services, including Amazon Elastic Compute Cloud (EC2) and Fargate. You can also automatically scale and run web applications in multiple Availability Zones with the performance, scale, reliability, and availability of AWS.

Implementation

PyRestTest:

We start by installing pyRestTest on our machine. Installing it is as simple as running a few commands in the terminal. (Note: It is significantly harder to install it on Windows, this implementation uses Ubuntu 18.04 as the OS)

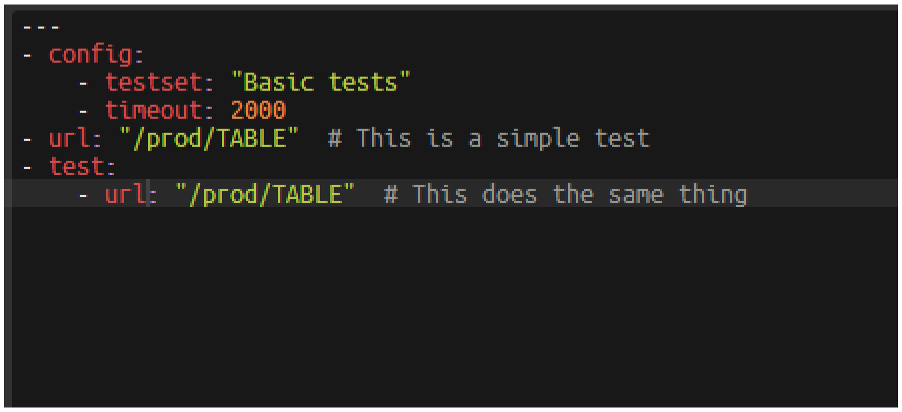
Assuming, Python is setup, the following terminal commands need to be run to install pyRestTest

```
“sudo apt-get install python-pycurl” OR “pip install pycurl”
“pip install future”
“pip install pyresttest” OR

git clone https://github.com/svanoort/pyresttest.git
cd pyresttest
sudo python setup.py install
```

Executing these commands should result in successful installation of PyRestTest

Now, create a new file and name it anything with a. yml extension.

This is a simple test file. You can create more comprehensive test using pyRestTest. See documentation for details: https://github.com/svanoort/pyresttest



```
---
- config:
    - testset: "Basic tests"
    - timeout: 2000
- url: "/prod/TABLE"  # This is a simple test
- test:
    - url: "/prod/TABLE"  # This does the same thing
```

This file sends a basic GET command to the URL passed in the command. Add more tests if you like and save the file.

Now, in the terminal, run the command

```
pyresttest <your-url-endpoint> <test-filename>.yml
```

You should see 2/2 tests passed in the output of the command.

Docker:

To install Docker, you need to run the following commands in a terminal.

```
sudo apt-get update
sudo apt-get install docker-ce docker-ce-cli containerd.io
```

To give non-privileged users access to docker, run the following commands:

```
sudo groupadd docker
sudo usermod -aG docker $USER
newgrp docker
```

To confirm that it works, run the following command:

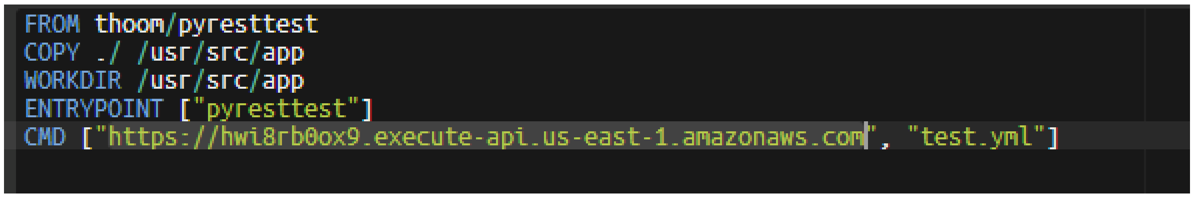
```
docker run hello-world
```

The command should return a message saying Hello from hello-world.

Now, let’s build a container for the pyRestTest test.yml we just created.

In the same directory as test.yml, create a new file with the name “Dockerfile”. Make sure the file has no extensions.

In the file, write these directives:



```
FROM thoom/pyresttest
COPY ./ /usr/src/app
WORKDIR /usr/src/app
ENTRYPOINT ["pyresttest"]
CMD ["https://hwi8rb0ox9.execute-api.us-east-1.amazonaws.com", "test.yml"]
```

Save the file. Now enter the following commands in a terminal:

```
docker build -t <container-name> <directory containing Dockerfile>
docker run <container-name>
```

Congratulations, you have built a new Docker image. You can see all docker images by running:

```
docker images
```

Running the image should return the same message you produced when running the pyresttest command.

Amazon ECR:

Now, we need to push our docker image to a managed registry like ECR. We can do this via the AWS CLI.

First, we need to authenticate to default registry so that docker can login to the ECR repository. To do this, run the following command:

```
aws ecr get-login-password --region <region> | docker login --username
AWS --password-stdin <aws_account_id>.dkr.ecr.<region>.amazonaws.com
```

For example:

```
aws ecr get-login-password --region us-east-1 | docker login --
username AWS --password-stdin 315997497220.dkr.ecr.us-east-
1.amazonaws.com
```

The result should be a message saying docker login successful.

Now, we have to create a repository which can be done via the following command:

```
aws ecr create-repository --repository-name <repo-name> --image-
scanning-configuration scanOnPush=true --region <region>
```

For example,

```
aws ecr create-repository --repository-name <sikandarsprint5> --image-
scanning-configuration scanOnPush=true --region <us-east-1>
```

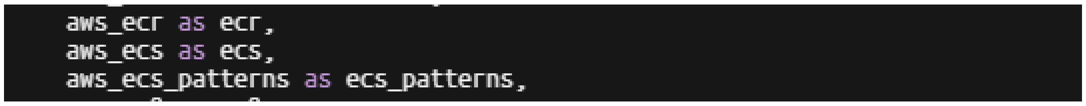
Now, we need to tag our existing image created in last section to match the name of our repository. Then we can push our image to our ECR repository. Execute the following commands:

```
docker tag api_test:latest \
315997497220.dkr.ecr.us-east 1.amazonaws.com/sikandarsprint5:latest

docker push \
315997497220.dkr.ecr.us-east-1.amazonaws.com/sikandarsprint5:latest

docker pull \
315997497220.dkr.ecr.us-east-1.amazonaws.com/sikandarsprint5:latest
```

Your image should now be published in the ECR.

Amazon ECS

You can easily create ECS clusters for either EC2 or Fargate Services via CDK. You will need the following new imports:

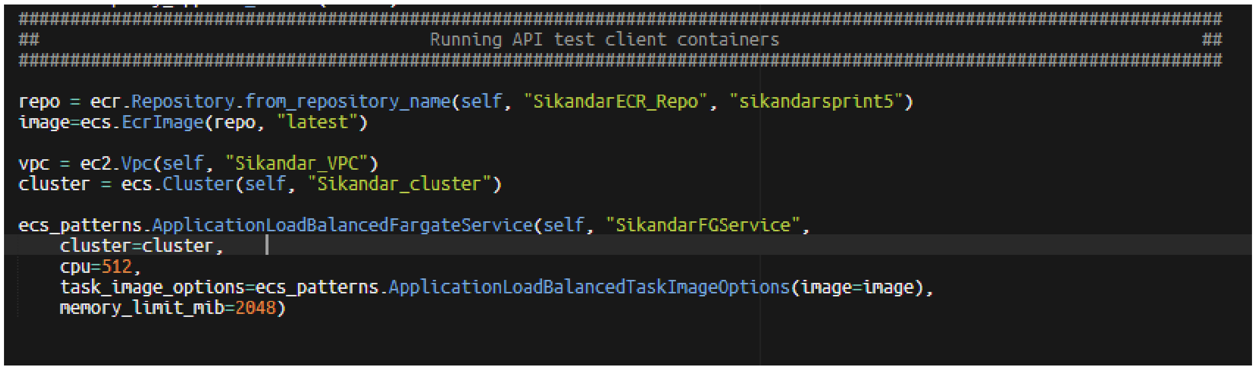


```
    aws_ecr as ecr,
    aws_ecs as ecs,
    aws_ecs_patterns as ecs_patterns,
```

Make sure to add the same in requirements.txt and run pip install again to download the new dependencies.

Now, we just need to create a few resources to instantiate our container on an ECS cluster. We use Fargate Service because of its ease of use.

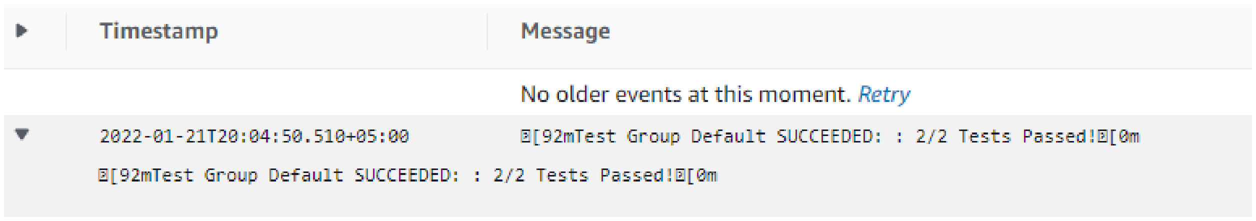
UPDATE: <u>It is not recommended to use an ECS service to deploy our test because it is used as a continuously deployed app. It means it will keep reloading our test, when we only need to run it once. Use Fargate or EC2 Scheduled Task instead.</u>



```
######################################################################################################################
##                                        Running API test client containers                                       ##
######################################################################################################################

repo = ecr.Repository.from_repository_name(self, "SikandarECR_Repo", "sikandarsprint5")
image=ecs.EcrImage(repo, "latest")

vpc = ec2.Vpc(self, "Sikandar_VPC")
cluster = ecs.Cluster(self, "Sikandar_cluster")

ecs_patterns.ApplicationLoadBalancedFargateService(self, "SikandarFGService",
    cluster=cluster,
    cpu=512,
    task_image_options=ecs_patterns.ApplicationLoadBalancedTaskImageOptions(image=image),
    memory_limit_mib=2048)
```

We can also pass environment variables from our stack to our container. One example could be passing the API URL endpoint via the API.url attribute. It can be accessed by the container so that URL does not need to be hardcoded in the Dockerfile.

We can see the Cloudwatch logs of tasks run in ECS. Here is one showing what the correct output should be:



```
    Timestamp                          Message
                                       No older events at this moment. Retry
    2022-01-21T20:04:50.510+05:00      ▯[92mTest Group Default SUCCEEDED: : 2/2 Tests Passed!▯[0m
    ▯[92mTest Group Default SUCCEEDED: : 2/2 Tests Passed!▯[0m
```

Troubleshooting

AWS Related Issues:

1. **<u>VPC quota reached:</u>**
   By default, AWS accounts have a VPC instance limit of 5 per region. To get another VPC, I had to deploy in another region (us-east-1)

2. **S3 Bucket Limit reached**
   By default, AWS accounts have an S3 bucket limit of 100 buckets. As I had to bootstrap in another region, the staging bucket could not be created with the limit reached. So, I deleted my staging buckets in us-east-2 to free up some S3 bucket quota.

3. **No space left on device in Cloud9**
   You can increase space on your Cloud9 IDE by going the EC2 Instance > Storage > Increase volume to 30 GB. Close the IDE, wait 5 minutes, restart the IDE and increase filesystem size to match partition size by following this link: recognize-expanded-volume-linux

PyRestTest:

1. **Pycurl is difficult to install on Windows:**

Recommend installing PyRestTest on Linux to easily install its dependencies.

Docker:

1. **pyresttest: error: no such option: -c**
   Upon running "docker run", this error may appear. The reason is that the default entrypoint for the pyresttest image in the Dockerfile is /bash/sh/. Changing entrypoint to "pyresttest" resolves this issue.
2. **Docker push results in timeout:**
   This can be for any number of reasons like incorrect configuration of AWS ECR login or docker login. In my case, it was because the tagged image name and the ECR repository name did not match. It will not throw an error at runtime but it is required that the image name and repository name are same.