
PyconCZ 2019 Serverless Slack bot

Jun 16, 2019

1	Project overview	1
2	Basic setup	3
3	Slack setup	5
3.1	Slack app	5
4	Serverless.js	7
4.1	serverless.yml	7
5	Slack messages	11
6	Sending questions	13
7	Flask app - processing Slack requests	17
7.1	Flask app	17
8	Meeting Report	29
8.1	Sending one report	29
8.2	All reports	30
8.3	Send report	30
9	Final deployment	31
9.1	Enabling Slack Interactive Components	31
9.2	Local invocation	32
10	AWS CloudWatch	35
10.1	CloudWatch Logs	35
10.2	CloudWatch Events	35
11	AWS DynamoDB	37
12	IAM	39
13	AWS Lambda	43
13.1	Function as a service (FAAS)	43
13.2	Handler	43
13.3	Technical details	44

14 Amazon Web Services	45
15 Underlying AWS services	47
16 How to set environment variables	49
16.1 Mac/Linux (Bash)	49
16.2 Windows	49
17 Iam Role	51

CHAPTER 1

Project overview

We are going to build a serverless Slack application to simulate daily standup meeting.

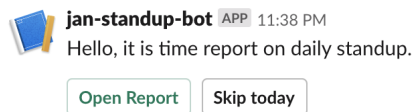
How is it going to work?

1. Application will be installed to our team's `SLACK_CHANNEL`
2. At time event (eg 8:30) (Time to ask questions) - application will send Direct Messages to the users and collect the responses.
3. At time event (eg. 10:00) (Time to send report) - application collects data from the database and send it back to the `SLACK_CHANNEL`

You can also [look at the schema via draw.io](#)

Users will interact with our app via Slack.

1. User receives a menu into direct message (private message)



2. Click on Open Dialog button opens a dialog with questions

 **Daily standup questions.** ×

What did you work on yesterday? (optional)

Writing tutorial for serverless workshop.

What did you work on yesterday?

What is your plan for today? (optional)

Walk through and complete the final bits.

What is your plan for today?


Any impediments? (optional)

no | 2998


Any impediments?


[? Learn more about jan-standup-bot](#) Cancel Submit

3. Application notifies user when dialog is successfully sent

 **jan-standup-bot** APP 11:38 PM
Thank you for your submission.

4. Send a report after a given time

 **jan-standup-bot** APP 12:46 AM
Here are the standup results from 1970-01-01 00:00:00.

 **Jan** APP 12:46 AM

What did you work on yesterday?
Writing tutorial for serverless workshop

What is your plan for today?
Walk through and complete final bits

Any impediments?
no

CHAPTER 2

Basic setup

First we need to prepare our environment.

You will need:

- A terminal program (Powershell, xterm, etc...) and a directory (`sls_workshop`).
- Installed `python3.6` or newer, `nodejs` with `npm`.
- AWS IAM user name (PyconCZ 2019 will receive this before the workshop)

This tutorial will make the best effort to support powershell on windows and any terminal with bash on Linux/Mac. If you are using `git bash` on windows you should be able to follow steps for Linux/Mac

1. Open your terminal
2. Create a project directory: `mkdir sls_workshop`
3. Change your current working directory to one we just created: `cd sls_workshop`
4. Create a virtual environment `python -m venv slsenv`
5. Activate virtual environment
 - Linux/Mac: `. slsenv/bin/activate`
 - Windows: `. .\slsenv\Scripts\activate.ps1`
6. Install following tools with `pip`: `pip install cookiecutter awscli`
7. Use `cookiecutter` to get serverless app template `cookiecutter https://gitlab.com/jans-workshops/pyconcz-2019-slack-bot-template.git`
8. Enter your AWS IAM user name and press enter.

Cookie cutter will create a new directory with our serverless app.

```
sls_app/
├── example-data
│   ├── apigw-block_action.json
│   └── block_action.json
```

(continues on next page)

(continued from previous page)

```
├── cwe-cron.json
├── cwe-questions.json
├── cwe-report.json
├── dialog_submission.json
├── sample_dialog.json
├── verification.json
├── requirements.txt
├── serverless.yml
├── standup_bot
│   ├── __init__.py
│   ├── action_app.py
│   ├── config.py
│   ├── models.py
│   ├── msg_templates.py
│   └── scheduled.py
```

- `serverless.yml` - configuration file for `serverless.js` framework, which describes our deployment
- `example-data/` - directory contains examples of data structures we deal with - good reference once you are comfortable with theory behind it.
- `standup_bot` - python module representing our application
- `standup_bot/action_app.py` - Flask application processing responses from Slack deployed as `slack-responses` AWS Lambda function
- `standup_bot/config.py` - contains some configuration used by both lambda functions
- `standup_bot/models.py` - contains our database models
- `standup_bot/msg_templates.py` - contains slack message [blocks](#) and templates (JSON objects)
- `standup_bot/scheduled.py` - contains logic for scheduled events to send menu or report

Now you are ready to start building the Serverless Slack bot.

Your next step is to set up a [Slack](#)

CHAPTER 3

Slack setup

We will be using Slack web client via your favorite web browser.

1. Join our Slack workspace `slspyconcz2019` or [invite yourself](#)
2. Create a channel `iam_username-team`
3. Make a note with your `CHANNEL_ID` from your channel's url `https://slspyconcz2019.slack.com/messages/<CHANNEL_ID>/`

3.1 Slack app

1. Go to <https://api.slack.com/apps>
2. Click Create New App
3. Fill in `appName` (to keep it simple: `iam_username-bot` and select workspace `slspyconcz2019` workspace)
4. Click Features:Bot Users -> Add a Bot User, keep default values -> click green Add a Bot User
5. Go to OAuth & Permissions -> Install app to Workspace -> Authorize (this generates Bot Token)

Keep the Bot Token handy as you will need it later.

Now basic slack setup is finished, continue to set up the serverless framework

CHAPTER 4

Serverless.js

If you did not install before then install [serverless.js](#) framework:

```
npm install -g serverless
```

You should have finished basic [slack setup](#) and have your AWS credentials.

1. In your terminal go to the `sls_app` directory you created before: `cd sls_app`.
2. Install python requirements plugin: `sls plugin install -n serverless-python-requirements`
3. Install AWS pseudo parameters plugin `sls plugin install -n serverless-pseudo-parameters`
4. Set environment variables in your terminal (in case you do not know how, check [How to set environment variables](#) section:

```
AWS_ACCESS_KEY_ID=<your key ID>
AWS_SECRET_ACCESS_KEY=<your secret key>
AWS_REGION=eu-west-1
AWS_DEFAULT_REGION=eu-west-1
SLACK_TOKEN=<your BOT token>
SLACK_CHANNEL=<"your slack channel ID">
```

4.1 serverless.yml

We are going to briefly describe `serverless.yml` file, please visit [official page with full description](#). Most common feature used in our `serverless.yml` is dynamic variable replacement (reference). Example syntax of variables:

```
ymlKeyXYZ: ${variableSource}
# this is an example of providing a default value as the second parameter
otherYmlKey: ${variableSource, defaultValue}
```

There are many variable sources, please see [the official documentation](#) for more details.

This file describes our environment (cloud provider), *runtime*, *functions*, plugins, other cloud resources and the final package.

Listing 1: serverless.yml

```

1 service: user1
2
3 provider:
4   name: aws
5   runtime: python3.7
6   region: eu-west-1
7   tags:
8     user: ${self:service}
9   environment: # global environment variables available to all functions within the
    ↪ service
10     SLACK_CHANNEL: ${env:SLACK_CHANNEL}
11     SLACK_TOKEN: ${env:SLACK_TOKEN}
12     DYNAMODB_TABLE: ${self:service}
13   iamManagedPolicies:
14     - "arn:aws:iam::#{AWS::AccountId}:policy/${self:service}-serverless-workshop-
    ↪ policy"

```

- **service** - Provides a namespace for our project
- **provider** - Is the environment where are we going to deploy our application (in our case it's AWS). Some attributes can be set at different scopes service, provider, function. Everything we set within provider scope is common for all functions.
- **provider.iamManagedPolicies** - **IAM service role** with permissions our functions require to run.

Note: if you are using this lab on your own account, you will need to define your own *role similar to this one*

Listing 2: serverless.yml

```

16 package:
17   exclude:
18     - node_modules/**
19     - tests/**

```

- **package** - final zip file uploaded to S3 and deployed to AWS Lambda

Listing 3: serverless.yml

```

21 functions:
22   # Lambda function with Flask application to handle Slack communication
23   slack-responses:
24     handler: wsgi_handler.handler
25     events:
26       - http:
27         path: actions
28         method: post
29   # Lambda function triggered by CloudWatch events
30   scheduled-events:
31     handler: standup_bot/scheduled.lambda_handler
32     events:
33       - schedule:
34         description: 'Send questions'
35     # rate: cron(0 8 ? * MON-FRI *) # Mo-Fri 8:00

```

(continues on next page)

(continued from previous page)

```

36     rate: rate(5 minutes) # For testing
37     enabled: false
38     inputTransformer:
39       inputPathsMap:
40         eventTime: "${.time}"
41         source: "${.source}"
42     inputTemplate: '{"source": <source>,"time": <eventTime>,"type": "send_
↪questions"}'
43   - schedule:
44     description: 'Send report.'
45     # rate: cron(0 10 ? * MON-FRI *) # Mo-Fri 10:00
46     rate: rate(5 minutes) # For testing
47     enabled: false
48     inputTransformer:
49       inputPathsMap:
50         eventTime: "${.time}"
51         source: "${.source}"
52     inputTemplate: '{"source": <source>,"time": <eventTime>,"type": "send_
↪report"}'

```

- functions - Properties and settings for AWS Lambda functions.
- functions.<fn-name>.handler - Path to python module containing *lambda_handler function*
- functions.<fn-name>.events - AWS Lambda function triggers

Listing 4: serverless.yml

```

55 plugins:
56   - serverless-python-requirements
57   - serverless-pseudo-parameters
58   - serverless-wsgi

```

- plugins - Serverless.js plugins (some are installed automatically, others must be installed with `sls plugin install -n <plugin-name>`)

Listing 5: serverless.yml

```

61 custom:
62   wsgi:
63     app: standup_bot.action_app.app
64     pythonBin: python3
65   pythonRequirements:
66     slim: true
67     slimPatternsAppendDefaults: true
68     slimPatterns:
69       - "**/*.egg-info*"
70       - "**/*.dist-info*"

```

- custom - Custom variables, which can be referenced as `${self:custom.<variableName>}`

Listing 6: serverless.yml

```

73 resources:
74   Resources:
75     StandupDynamoDbTable:
76       Type: 'AWS::DynamoDB::Table'

```

(continues on next page)

(continued from previous page)

```
77     DeletionPolicy: Retain
78     Properties:
79         AttributeDefinitions:
80             - AttributeName: report_id
81               AttributeType: S
82             - AttributeName: report_user_id
83               AttributeType: S
84         KeySchema:
85             - AttributeName: report_id
86               KeyType: HASH
87             - AttributeName: report_user_id
88               KeyType: RANGE
89         ProvisionedThroughput:
90             ReadCapacityUnits: 1
91             WriteCapacityUnits: 1
92         TableName: ${self:provider.environment.DYNAMODB_TABLE}
```

- resources - definition of additional AWS CloudFormation resources ([serverless.js docs](#), [aws CloudFormation docs](#))

CHAPTER 5

Slack messages

Before we start coding let's find out how Slack is composing the message. You can always refer to [official documentation](#).

We are only going to cover parts we will use through out the guide, please refer to [full message documentation](#) if needed.

Slack message is a simple JSON object.

```
{
  "text": "A message text",
  "blocks": [],
  "mrkdwn": true
}
```

Field	Type	Required?	Description
text	String	Yes	Text of the message usually used inside notification popups. (not required when using blocks)
blocks	Array	No	An array of layout blocks, in the same format as described in the layout block guide

We are going to use a new Slack feature [blocks](#) to compose our messages.

Our main block components will be [Section Block](#) which is similar to the message. And [Action blocks](#) is used for interactive components like buttons and menus. Most important block attributes to understand are: `type`, `block_id`, `fields` and `elements`. Please go to the documentation and try it out using a [block-kit-builder](#).

There are other Slack objects we are going to deal with

CHAPTER 6

Sending questions

Let's write our first part of 1 of ours AWS Lambda functions. Now we are going to work with `standup_bot/scheduled.py` file which represents AWS Lambda function `shecudeld-events` from [schema](#)

```
1 #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  """
4  Standalone lambda function triggered by CWE.
5  """
6  import datetime as dt
7  import json
8  import logging
9
10 from slack import WebClient
11
12 from standup_bot.config import (
13     SLACK_TOKEN,
14     SLACK_CHANNEL,
15     QUESTIONS,
16 )
17 from standup_bot.models import Report
18 from standup_bot.msg_templates import standup_menu_block, report_block
19
20 LOGGER = logging.getLogger(__name__)
21 LOGGER.setLevel(logging.INFO)
22
23 # synchronous slack client
24 SC = WebClient(SLACK_TOKEN)
```

In highlighted lines above, we:

- import the functionality needed later,
- initialize logging
- initialize Slack client - note that we are using synchronous client in order to keep the code beginner friendly. However you it is possible to make the function async using `asyncio` and `asynchronous slack client`.

Let's jump to the very bottom of the file and look at our `lambda_handler`.

We will pay more attention to 28-83 bit later.

```

88 def lambda_handler(lambda_event, lambda_context):
89     """Main lambda handler"""
90     LOGGER.debug(lambda_context)
91     LOGGER.info(lambda_event)
92     LOGGER.info("lambda_event starts:")
93     LOGGER.info(json.dumps(lambda_event))
94
95     # is it our CWE event?
96     if {"type", "time", "source"}.issubset(lambda_event):
97
98         report_id = dt.datetime.strptime(lambda_event["time"], '%Y-%m-%dT%H:%M:%S%Z').
↪strptime("%Y%m%d")
99         LOGGER.info("Report ID: %s", report_id)
100         if lambda_event["type"] == "send_report":
101             send_report(report_id)
102         elif lambda_event["type"] == "send_questions":
103             send_questions(report_id)

```

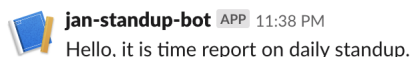
Event source (trigger) of our lambda function will be modified input from a *cloudwatch event*. We are going to deal with two event types `send_questions` and `send_report`.

Code breakdown:

- L90-93 - Logging of input, so we can investigate what is going on via *CloudWatch Logs*.
- L98 - generate `report_id` in format `YYYYMMDD`
- Then trigger entry point function based on event type

We are going to start with Sending a daily menu to the members of our `SLACK_CHANNEL`. Implement functions: `send_menu`, `send_menus` and `send_questions`.

To send a menu to the user via private message, we need to first open the conversation, then we can send a message. Main body of our message will be a `menu_block`, which consists of 2 buttons. Where 1 button opens a dialog with questions and second button allows user to skip todays report.

 **jan-standup-bot** APP 11:38 PM
Hello, it is time report on daily standup.

```

28 def send_menu(user_id, menu_block):
29     """Send menu as private message to the user."""
30
31     response = SC.conversations_open(users=[user_id])
32     post_response = SC.chat_postMessage(
33         channel=response["channel"]["id"], text="Daily menu", blocks=menu_block
34     ),
35     return user_id, post_response

```

On line 33 we are using pre-built layout block from `standup_bot/msg_templates`.

In function `send_menus` we ask slack to get a list of channel members and `send_menu` to each member.

```

38 def send_menus(menu_block):
39     """Send menu to all users from the channel."""

```

(continues on next page)

(continued from previous page)

```

40 members = SC.conversations_members(channel=SLACK_CHANNEL)
41 for user_id in members['members']:
42     yield send_menu(user_id, menu_block)

```

As a last step we define our entry-point function `send_questions`. Where we generate `menu_block` part of the slack message and gather the delivery status responses.

```

45 def send_questions(report_id):
46     """Entry point for daily menu."""
47     menu_block = standup_menu_block(report_id)
48     results = list(send_menus(menu_block))
49     LOGGER.info(results)

```

Our `menu_block` is a function which generates message blocks

Listing 1: `msg_templates.standup_menu_block`

```

34 def standup_menu_block(report_id):
35     """
36     Message block with the menu sent on daily basis.
37
38     Contains Open Dialog and Skip buttons.
39
40     """
41     return [
42         {
43             "type": "section",
44             "text": {
45                 "type": "mrkdwn",
46                 "text": "Hello, it is time report on daily standup.",
47             },
48         },
49         {
50             "type": "actions",
51             "elements": [
52                 {
53                     "type": "button",
54                     "text": {
55                         "type": "plain_text",
56                         "emoji": True,
57                         "text": "Open Report",
58                     },
59                     "style": "primary",
60                     "value": report_id,
61                     "action_id": "standup.action.open_dialog",
62                 },
63                 {
64                     "type": "button",
65                     "text": {"type": "plain_text", "emoji": True, "text": "Skip today"},
66                     "action_id": "standup.action.skip_today",
67                 },
68             ],
69         },
70     ]

```

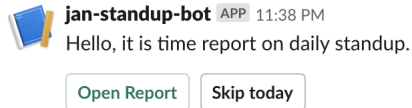
Above we have generated 2 blocks with types `section` and `actions`. We have given a specific `action_id` to

each element in order to recognize which button was clicked by user. Processing of actions is further explained in the next section.

We can now test this part of our code and invoke our function locally with command: `sls invoke local -f scheduled-events --path example-data/cwe-questions.json`.

Make sure you are running this command from within same directory where `serverless.yml` is located. (`sls_app`)

If the invocation was successful, you should receive a private message from your application which looks similar to what you can see in a picture below.



Flask app - processing Slack requests

Now that we can send the menu to the user, we are going to learn how to store the data. Slack communicates with our app via [interactive actions](#) (actions, dialogs, message buttons, or message menus) via HTTP POST to URL we set in slack application settings.

But first let's write some code.

7.1 Flask app

We are going to write our code inside `standup_bot/action_app.py`

It will be a simple application POST processing requests via API Gateway triggering `slack-responses` Lambda function.

Listing 1: `serverless.yml`

```
21 functions:
22   # Lambda function with Flask application to handle Slack communication
23   slack-responses:
24     handler: wsgi_handler.handler
25     events:
26       - http:
27         path: actions
28         method: post
```

To serve the flask app from within AWS Lambda we are going to use `serverless-wsgi` plugin. (Plugin is able to install itself, but you can do it as well: `sls plugin install -n serverless-wsgi`)

Listing 2: `serverless.yml`

```
55 plugins:
56   - serverless-python-requirements
57   - serverless-pseudo-parameters
58   - serverless-wsgi
```

Our application will deal with multiple structures which you can look at inside the folder `example-data`

1. Requests from slack are sent to API Gateway, which triggers our function and passes event similar to what you can see inside `example-data/apigw-block_action.json`
2. Then `serverless-wsgi` and `Flask` transforms this event into `Flask Request`
3. We parse the request body and determine the request type: `block_actions` or `dialog_submission`

Let's break down the `standup_bot/action_app.py` file.

Listing 3: `action_app.py`

```

1  """
2  Main flask app file used to receive incoming http requests from Slack.
3  """
4  import logging
5  from urllib import parse
6
7  from flask import Flask, request, json, make_response
8  from slack import WebClient
9
10 from standup_bot.config import QUESTIONS, SLACK_TOKEN
11 from standup_bot.models import Report
12 from standup_bot.msg_templates import dialog_questions

```

- We are going to need `urllib.parse` to help us with url decoding.
- Then we import `Flask` items we are going to need
- And `Slack` web client from `python-slackclient`
- Next set of imports comes from our cookiecutter template
 - `QUESTIONS` - is a dict of questions we are going to ask. `{ "question1" : "Question text?" }`
 - `SLACK_TOKEN` - so we can respond back to the user
 - `dialog_questions` - is a `Slack dialog` containing `QUESTIONS` We simply iteratively build elements and so our result dialog looks similar to this

Listing 4: `example-data/sample_dialog.json`

```

1  {
2      "callback_id": "standup.action.answers",
3      "elements": [
4          {
5              "hint": "What did you work on yesterday?",
6              "label": "What did you work on yesterday?",
7              "name": "question0",
8              "optional": true,
9              "placeholder": "What did you work on yesterday?",
10             "type": "textarea"
11         },
12         {
13             "hint": "What is your plan for today?",
14             "label": "What is your plan for today?",
15             "name": "question1",
16             "optional": true,
17             "placeholder": "What is your plan for today?",

```

(continues on next page)

(continued from previous page)

```

18         "type": "textarea"
19     },
20     {
21         "hint": "Any impediments?",
22         "label": "Any impediments?",
23         "name": "question2",
24         "optional": true,
25         "placeholder": "Any impediments?",
26         "type": "textarea"
27     }
28 ],
29 "state": "{\"container\": {\"channel_id\": \"DFK2PDRPT\", \"is_
↪ephemeral\": false, \"message_ts\": \"1558433489.000600\", \"type\
↪\": \"message\"}, \"report_id\": \"19700101\"}\",
30 "title": "Daily standup questions."
31 }

```

- **Report** - is our **DynamoDB** database model created with **pynamodb** package. It is possible to access *DynamoDB* directly via **boto3** however **pynamodb** friendlier API.

Listing 5: models.py

```

1  """Dynamo db models."""
2
3  from pynamodb.attributes import MapAttribute, UnicodeAttribute
4  from pynamodb.models import Model
5
6  from standup_bot.config import TABLE_NAME, AWS_REGION
7
8
9  class Report(Model):
10     """
11     Standup report model.
12     """
13
14     class Meta:
15         table_name = TABLE_NAME
16         region = AWS_REGION
17
18         report_id = UnicodeAttribute(hash_key=True)
19         report_user_id = UnicodeAttribute(range_key=True)
20         user_id = UnicodeAttribute()
21         user_name = UnicodeAttribute()
22         display_name = UnicodeAttribute()
23         icon_url = UnicodeAttribute()
24         answers = MapAttribute()

```

In our next step, we initialize our **SlackClient** (SC) and **Flask** app. When a Slack user use an interactive (click button, submit dialog) we configure our Slack application to send a HTTP POST request to our AWS API Gateway URL `<APIGWID>.execute-api.<AWS-region>.amazonaws.com/actions`. And define a route endpoint / actions.

Listing 6: action_app.py

```

14 # synchronous slack client
15 SC = WebClient(SLACK_TOKEN)

```

(continues on next page)

(continued from previous page)

```

16
17 app = Flask(__name__)
18 app.config["SECRET_KEY"] = "you-will-never-guess"
19 app.logger.setLevel(logging.INFO)
20
21
22 @app.route("/actions", methods=["POST"])
23 def actions():
24     """
25     Endpoint /actions to process actions and dialogs.
26
27     This method receives a request from API gateway.
28     Example request: example-data/apigw-block_action.json
29
30     We need to decode body and decide if requests is one of:
31     - block_actions -> will trigger process_block_actions()
32     - dialog_submission -> will trigger process_dialogs()
33
34     Returns
35     -----
36     flask.Response
37
38     """
39

```

Inside function actions we are going to perform following operations:

1. Parse and unquote request body received from AWS API Gateway.
 - Received request is a URL encoded string which contains a prefix payload.
See line 119 inside a file example-data/apigw-block_action.json.
"body": "payload=%7B%22type%22%3A%22block_actions%22%2C%22team%...
 - To find out how does the parsed action body looks like check file: sls_app/example-data/block_action.json
2. Determine Slack action type block_actions or dialog_submission
3. Process the action
4. Respond back to Slack (user)

Listing 7: action_app.py

```

22 @app.route("/actions", methods=["POST"])
23 def actions():
24     """
25     Endpoint /actions to process actions and dialogs.
26
27     This method receives a request from API gateway.
28     Example request: example-data/apigw-block_action.json
29
30     We need to decode body and decide if requests is one of:
31     - block_actions -> will trigger process_block_actions()
32     - dialog_submission -> will trigger process_dialogs()
33
34     Returns

```

(continues on next page)

(continued from previous page)

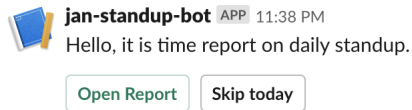
```

35  -----
36  flask.Response
37
38  """
39
40  prefix = "payload="
41  data = request.get_data(as_text=True)[len(prefix):]
42  app.logger.info("req_data %s", data)
43  # app.logger.info("dd %s", data[len(prefix):])
44
45  # action body
46  slack_req_body = json.loads(parse.unquote_plus(data))
47  app.logger.info("Action body: %s", slack_req_body)
48
49  slack_req_type = slack_req_body.get("type")
50
51  action = {"block_actions": process_block_actions, "dialog_submission": process_
↪ dialogs}
52
53  response = action[slack_req_type](slack_req_body)
54  app.logger.info("Response to Action: %s : %s", response, response.get_data())
55  return response

```

Now we need to write appropriate functions to process our action types.

Our first function will be `process_block_actions` which triggered when user clicks on a button from the menu. Button triggers a [Slack block action](#).



We will need to create simple logic inside our function `process_block_actions`, to process each action type correctly. So far we will deal with 2 types:

- `standup.action.open_dialog` - When clicked on Open Dialog button
- `standup.action.skip_today` - The idea is to signal that user decided to skip today's meeting. Implementation of this is left for you as a challenge.

Following example is a JSON (dict) data structure we get after successful parsing of `slack_request` in previous method `actions()`.

Message itself contains a lot of data, however we will focus on highlighted parts.

- L15 `container` - contains data we are going to need to uniquely identify the Slack dialog we create later
- L21 `trigger_id` - is required to trigger the correct dialog
- L26 `message` - contains original message with the menu
- L72 `actions` - contains result value of user's action (click Open Dialog button)

Listing 8: example-data/block_action.json

```

1  {
2  "type": "block_actions",
3  "team": {

```

(continues on next page)

(continued from previous page)

```

4     "id": "TFE4ZTB3L",
5     "domain": "jendaworkspace"
6 },
7 "user": {
8     "id": "UFE4ZTC8J",
9     "username": "loglopl",
10    "name": "loglopl",
11    "team_id": "TFE4ZTB3L"
12 },
13 "api_app_id": "AFM36S3CN",
14 "token": "uJLNkNPcUwaEzPceiCEfb9wC",
15 "container": {
16     "type": "message",
17     "message_ts": "1555663294.000200",
18     "channel_id": "DFK2PDRPT",
19     "is_ephemeral": false
20 },
21 "trigger_id": "601888012770.524169929122.23393cd981d2028028794396a1e104bf",
22 "channel": {
23     "id": "DFK2PDRPT",
24     "name": "directmessage"
25 },
26 "message": {
27     "type": "message",
28     "subtype": "bot_message",
29     "text": "Daily menu",
30     "ts": "1555663294.000200",
31     "username": "jan-standup-bot",
32     "bot_id": "BFLN2CMST",
33     "blocks": [
34         {
35             "type": "section",
36             "block_id": "dgz+G",
37             "text": {
38                 "type": "mrkdwn",
39                 "text": "Hello, it is time report on daily standup.",
40                 "verbatim": false
41             }
42         },
43         {
44             "type": "actions",
45             "block_id": "act",
46             "elements": [
47                 {
48                     "type": "button",
49                     "action_id": "standup.action.open_dialog",
50                     "text": {
51                         "type": "plain_text",
52                         "text": "Open Report",
53                         "emoji": true
54                     },
55                     "style": "primary",
56                     "value": "19700101"
57                 },
58                 {
59                     "type": "button",
60                     "action_id": "standup.action.skip_today",

```

(continues on next page)

(continued from previous page)

```

61         "text": {
62             "type": "plain_text",
63             "text": "Skip today",
64             "emoji": true
65         }
66     }
67 ]
68 }
69 ]
70 },
71 "response_url": "https://hooks.slack.com/actions/TFE4ZTB3L/606986906385/
↪Fc2QQKdICRgBSKiWXVBAS5Qp",
72 "actions": [
73     {
74         "action_id": "standup.action.open_dialog",
75         "block_id": "act",
76         "text": {
77             "type": "plain_text",
78             "text": "Open Report",
79             "emoji": true
80         },
81         "value": "19700101",
82         "type": "button",
83         "style": "primary",
84         "action_ts": "1555663307.531487"
85     }
86 ]
87 }

```

With the information above we can proceed with implementation of `process_block_actions`.

The logic is following:

1. Take the 1st action value from actions array
2. Create a `state_data` to match the dialog with a user.
3. Determine the action type
4. Fill dialog object with questions and other data
5. Send a request back to Slack to open a dialog
6. And respond 200 and empty body if successful

```

58 def process_block_actions(slack_request: dict):
59     """
60     Slack Action processor.
61
62     Here we are going to process decoded slack request "block actions"
63     https://api.slack.com/reference/messaging/blocks#actions
64
65     Example request: example-data/block_action.json
66
67     We will present user with 2 buttons.
68     1. Open dialog - which contains standup questions
69     2. Skip today - to let user pass the meeting
70
71     Returns

```

(continues on next page)

(continued from previous page)

```

72  -----
73  flask.Response
74      Empty response 200 signifies success.
75
76  """
77  action = slack_request["actions"][0]
78  state_data = {"container": slack_request["container"], "report_id": action["value
↪"]}
79  if action["action_id"] == "standup.action.open_dialog":
80      questions = dialog_questions(json.dumps(state_data), QUESTIONS)
81
82      app.logger.info(questions)
83
84      slack_response = SC.dialog_open(
85          dialog=questions, trigger_id=slack_request["trigger_id"]
86      )
87      app.logger.info("Dialog Open: %s", slack_response)
88      return make_response()
89
90  if action["action_id"] == "standup.action.skip_today":
91      # you can try to implement this yourself
92      pass
93
94  return make_response("Unable to process action", 400)

```

If impatient and would like to try out your partially implemented app, go ahead to *second slack setup and deployment* then come back!

Okay, we have successfully showed dialog to the user, now it's time to collect the data.

When user submits the dialog Slack sends a HTTP POST request to our endpoint `/actions`. But this time the `slack_req_type` type is `dialog_submission`.

Dialog submission message contains data about the user and values for the answers.

Note that questions are only represented with IDs `question1,...`

The important part is a `callback_id` field which helps us to identify dialog type. We then use the data previously stored in `state` to identify the user.

Listing 9: example-data/block_action.json

```

1  {
2      "type": "dialog_submission",
3      "token": "uJLNkNPcUwaEzPceiCEfb9wC",
4      "action_ts": "1558430192.474181",
5      "team": {
6          "id": "TFE4ZTB3L",
7          "domain": "jendaworkspace"
8      },
9      "user": {
10         "id": "UFE4ZTC8J",
11         "name": "loglop1"
12     },
13     "channel": {
14         "id": "DFK2PDRPT",
15         "name": "directmessage"
16     },

```

(continues on next page)

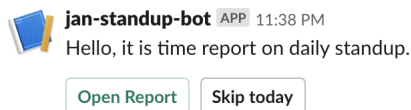
(continued from previous page)

```

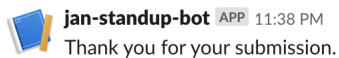
17  "submission": {
18    "question0": "2",
19    "question1": "2",
20    "question2": "2"
21  },
22  "callback_id": "standup.action.answers",
23  "response_url": "https://hooks.slack.com/app/TFE4ZTB3L/641451583301/
  ↪S10UQ9nERHjT2EbfXZNFgS7F",
24  "state": "{\"container\": {\"channel_id\": \"DFK2PDRPT\", \"is_ephemeral\": false, \
  ↪\"message_ts\": \"1558430180.000400\", \"type\": \"message\"}, \"report_id\": \
  ↪\"19700101\"}"
25 }

```

Slack does not have implemented automated updates(answers) to actions. This means when user submits the dialog, the menu stays unchanged.



But we would like to inform the user about successful submission therefore we will do the trick and update the previous message using the timestamp to following:



For this purpose we are going to implement a function `process_dialogs` as follows:

1. Examine `callback_id` to identify dialog type
2. Parse the `state_data` from string to dict
3. Get detailed information about the user from Slack
4. Create a `report` object represented by our database model `Report`
 - To uniquely identify the report for simple a simple query, our main identifier will be `report_id`. It's a simple execution timestamp, which comes from a scheduled event, converted into YYYYMMDD string with suffix of a slack user ID UFE4ZTC8J (assuming our meeting is once a day).

Example

```
time: 2016-12-30T18:44:49Z
user_id: UFE4ZTC8J
```

becomes:

```
20161230_UFE4ZTC8J
```

This way we can later query the database for a given day to get all reports.

5. Save data in the database
6. Inform user via updating a chat message.
7. Send empty response 200

Listing 10: action_app.py

```

97 def process_dialogs(slack_dialog: dict):
98     """
99     Process Slack dialogs.
100
101     Here we are going to collect data from dialog
102
103     example dialog submission: example-data/dialog_submission.json
104
105     Returns
106     -----
107     flask.Response
108         Successful dialog submission requires empty response 200.
109
110     """
111
112     if slack_dialog["callback_id"] == "standup.action.answers":
113         # add one field to answers
114         state_data = json.loads(slack_dialog["state"])
115         # prepare DB record
116
117         # get more user data
118         user_info = SC.users_info(
119             user=slack_dialog["user"]["id"]
120         )
121
122         app.logger.info("UserInfo: %s", user_info)
123
124         display_name = (
125             user_info["user"]["profile"]["display_name"]
126             or user_info["user"]["profile"]["real_name"]
127             or slack_dialog["user"]["name"]
128         )
129
130         user_report = Report(
131             state_data["report_id"],
132             f'{state_data["report_id"]}_{slack_dialog["user"]["id"]}',
133             user_name=slack_dialog["user"]["name"],
134             user_id=slack_dialog["user"]["id"],
135             answers=slack_dialog["submission"],
136             display_name=display_name,
137             icon_url=user_info["user"]["profile"]["image_48"],
138         )
139         # Write to database.
140         user_report.save()
141         # Respond to user
142         SC.chat_update(
143             channel=state_data["container"]["channel_id"],
144             ts=state_data["container"]["message_ts"],
145             text="Thank you for your submission.",
146             blocks=[],
147             as_user=True, # reason specified in slack docs
148         )
149
150         app.logger.info("Adding new answer: %s", user_report._get_json())
151         return make_response()

```

(continues on next page)

(continued from previous page)

```
152  
153     return make_response("Unable to process dialog", 400)
```

This concludes 2/3 of our application, you can now try to deploy and play around. To deploy check out *the second slack setup and deployment* page!

If you are not deploying or you have finished the investigation, please proceed to the last part *sending the report from all users*

CHAPTER 8

Meeting Report

Last step of our application is to report all collected data back to the `SLACK_CHANNEL`

This is going to be again the scheduled event but this time called `send_report`. We will schedule this event after `send_questions` and give users enough time to respond (30m ~ 2h).

Implementation will take place again inside `standup_bot/scheduled.py` file.

Where we are going to write 3 functions in a style that is easy to convert into asynchronous code.

- `one_report` - Send report of 1 user to the `SLACK_CHANNEL`
- `all_reports` - Iterate through all reports collected that day and send them to the `SLACK_CHANNEL`
- `send_report` - Our entry point function, which starts the whole orchestra.

8.1 Sending one report

In this case we assume we already have our `report` object available. And all we need to do is to send a request to the Slack and return response.

```
53 def one_report(report):
54     """Show report of one user."""
55     user_id = report.user_id
56
57     response = SC.chat_postMessage(
58         channel=SLACK_CHANNEL,
59         username=report.display_name,
60         icon_url=report.icon_url,
61         text=f"*{report.display_name}* posted and update for stand up meeting.",
62         blocks=report_block(QUESTIONS, report.answers)
63     )
64
65     return user_id, response
```

8.2 All reports

In method `all_reports` we are going to query for all reports from a given day and apply `one_report` on each.

```
68 def all_reports(report_id):
69     """Show reports of all users."""
70     reports = Report.query(report_id, Report.report_user_id.startswith(report_id))
71
72     SC.chat_postMessage(
73         channel=SLACK_CHANNEL,
74         text=f"Here are the standup results from {(dt.datetime.strptime(report_id, '%Y
75     ↪ %m%d')}*.",
76     )
77     for report in reports:
78         yield one_report(report)
```

Using a generator is very similar to asynchronous programming in Python. So if you are interested into async python but not yet familiar with [generators](#), I strongly suggest to start there.

8.3 Send report

Now we have everything ready and all we need to do is to call function `all_reports`.

```
80 def send_report(report_id):
81     """Entry point for sending reports."""
82     results = list(all_reports(report_id))
83     LOGGER.info("Sent report log: %s", results)
```

Now we can let `lambda_handler` decide what is the appropriate entry point (`send_report` or `send_questions`)

Note that explanation of sending report and question took a bottom up approach. Where we first implemented a single (atomic) operation and in next steps we simply run this in a loop. You can use this approach in asynchronous version of this application and run `all_reports` at the same time, instead iterating through each.

Final deployment

To deploy our app we need to go through few steps:

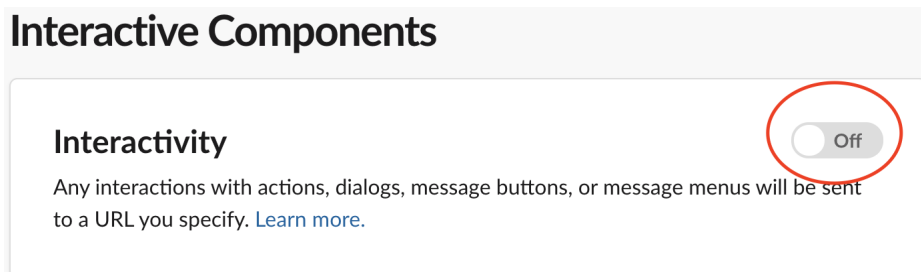
1. `cd sls_app` (you should be already there)
2. `sls deploy` - this will deploy our app to AWS and output API Gateway endpoint

```
endpoints:  
  POST - https://jvb1c6tyd6.execute-api.eu-west-1.amazonaws.com/dev/actions
```

3. Enable Slack Interactive Components

9.1 Enabling Slack Interactive Components

1. Navigate to <https://api.slack.com/apps>
2. Click on your application
3. Features -> **Interactive Components**
4. Set interactive components to On



5. Fill in the endpoint URL

Interactivity

On 

Any interactions with actions, dialogs, message buttons, or message menus will be sent to a URL you specify. [Learn more.](#)

Request URL

`https://<api-gw-ID>.execute-api.<region>.amazonaws.com/dev/actions`

We'll send an HTTP POST request with information to this URL when users interact with a component (like a button or dialog).

6. Hit the **Save Changes** button

9.2 Local invocation

Because our Cloud Watch events are disabled to prevent unwanted executions before the production/test stage.

```

30 scheduled-events:
31   handler: standup_bot/scheduled.lambda_handler
32   events:
33     - schedule:
34       description: 'Send questions'
35       # rate: cron(0 8 ? * MON-FRI *) # Mo-Fri 8:00
36       rate: rate(5 minutes) # For testing
37       enabled: false
38       inputTransformer:
39         inputPathsMap:
40           eventTime: "$.time"
41           source: "$.source"
42       inputTemplate: '{"source": <source>,"time": <eventTime>,"type": "send_
31 ↪ questions"}'
43     - schedule:
44       description: 'Send report.'
45       # rate: cron(0 10 ? * MON-FRI *) # Mo-Fri 10:00
46       rate: rate(5 minutes) # For testing
47       enabled: false
48       inputTransformer:
49         inputPathsMap:
50           eventTime: "$.time"
51           source: "$.source"
52       inputTemplate: '{"source": <source>,"time": <eventTime>,"type": "send_
31 ↪ report"}'

```

You may have noticed that `rate` is set to `rate` which is CloudWatch scheduled event expression and this is also not correct for production, however it's a good way to observe behaviour triggered by AWS.

For now we invoke our scheduled functions locally and manually via serverless framework commands.

To `send_questions` run

```
sls invoke local -f scheduled-events --path example-data/cwe-questions.json
```

Then take some time to collect the data and once you are ready. Trigger `send_report` using the command

```
sls invoke local -f scheduled-events --path example-data/cwe-report.json
```


CHAPTER 10

AWS CloudWatch

10.1 CloudWatch Logs

10.2 CloudWatch Events

CHAPTER 11

AWS DynamoDB

DynamoDB is a no relational database in AWS. This will only cover the part required for the workshop but feel free to refer to the [official documentation](#).

In this workshop, we need a way to store our state, this mean we need the questions and answers of our users in a database so we can publish the results to the chat when the time comes.

To do so, at `serverless.yml` we create a table with two items:

Listing 1: serverless.yml

```
75 StandupDynamoDbTable:
76   Type: 'AWS::DynamoDB::Table'
77   DeletionPolicy: Retain
78   Properties:
79     AttributeDefinitions:
80       - AttributeName: report_id
81         AttributeType: S
82       - AttributeName: report_user_id
83         AttributeType: S
84     KeySchema:
85       - AttributeName: report_id
86         KeyType: HASH
87       - AttributeName: report_user_id
88         KeyType: RANGE
89     ProvisionedThroughput:
90       ReadCapacityUnits: 1
91       WriteCapacityUnits: 1
92     TableName: ${self:provider.environment.DYNAMODB_TABLE}
```

- `report_id`: incremental ID of every report
- `report_user_id`: the user ID that submitted the dialog

After submitting the questions, this table will be updated with:

- `answers`: dictionary with every question number as a key and the answer as value

When the standup time comes, `scheduled.py` will trigger `all_reports` functions.

CHAPTER 12

IAM

IAM provides identity management in AWS. It can create users, roles and policies. For more information please refer to the [official documentation](#).

For this workshop, every user will have an IAM user with access to the console and API credentials so it can bootstrap the serverless deployment.

Every user will only have access to its own resources. In order to do so, a separate Cloudformation stack creates all users according to its policy.

Here there are the policies that the user will have. Please note some AWS services won't fully provide fine grained permission access level as of this time of writing.

```
IAMUserPolicy{{ loop.index }}:
Type: AWS::IAM::ManagedPolicy
Properties:
  ManagedPolicyName: user{{ loop.index }}-serverless-workshop-policy
  PolicyDocument:
    Version: '2012-10-17'
    Statement:
      - Action:
          - ec2:CreateNetworkInterface
          - ec2:DeleteNetworkInterface
          - ec2:DescribeNetworkInterfaces
          - dynamodb:List*
          - dynamodb:DescribeReservedCapacity
          - dynamodb:DescribeLimits
          - dynamodb:DescribeTimeToLive
          - cloudformation:DescribeStackEvents
          - cloudformation:ValidateTemplate
          - cloudformation:ListStacks
          - s3:GetEncryptionConfiguration
          - s3:PutEncryptionConfigurations
          - s3:PutBucketAcl
        Effect: Allow
        Resource: "*"

```

(continues on next page)

(continued from previous page)

```

- Action:
  - s3:*
  Effect: Allow
  Resource: "arn:aws:s3:::user{{ loop.index }}-dev*"

- Action:
  - cloudformation:DescribeStacks
  - cloudformation:CreateStack
  - cloudformation:DescribeStackResource
  - cloudformation>DeleteStack
  - cloudformation:UpdateStack
  - cloudformation:ListStackResources
  Effect: Allow
  Resource: !Sub "arn:aws:cloudformation:${AWS::Region}:${AWS::AccountId}:stack/
↪user{{ loop.index }}-dev/*"

- Action:
  - iam:ChangePassword
  Effect: Allow
  Resource: !Sub "arn:aws:iam::${AWS::AccountId}:user/user{{ loop.index }}"

- Action:
  - iam:GetRole
  - iam:CreateRole
  - iam>DeleteRole
  - iam:PassRole
  - iam:PutRolePolicy
  - iam>DeleteRolePolicy
  - iam:AttachRolePolicy
  - iam:DetachRolePolicy
  Effect: Allow
  Resource: !Sub "arn:aws:iam::${AWS::AccountId}:role/user{{ loop.index }}-dev-${
↪AWS::Region}-lambdaRole"

- Effect: Allow
  Action:
    - dynamodb:Query
    - dynamodb:Scan
    - dynamodb:Get*
    - dynamodb:PutItem
    - dynamodb:UpdateItem
    - dynamodb>DeleteTable
    - dynamodb:CreateTable
    - dynamodb>DeleteItem
    - dynamodb:DescribeTable
    - dynamodb:TagResource
  Resource: !Sub "arn:aws:dynamodb:${AWS::Region}:${AWS::AccountId}:table/user{
↪{{ loop.index }}"

- Effect: Allow
  Action:
    - lambda:GetFunction
    - lambda:GetFunctionConfiguration
    - lambda:CreateFunction
    - lambda>DeleteFunction
    - lambda:AddPermission

```

(continues on next page)

(continued from previous page)

```

    - lambda:ListVersionsByFunction
    - lambda:AddPermission
    - lambda:RemovePermission
    - lambda:PublishVersion
    - lambda:UpdateFunctionCode
    - lambda:UpdateFunctionConfiguration
    Resource: !Sub "arn:aws:lambda:${AWS::Region}:${AWS::AccountId}:function:user{
↪ { loop.index }}*"

- Effect: Allow
  Action:
    - events:PutRule
    - events>DeleteRule
    - events:DescribeRule
    - events:RemoveTargets
    - events:PutTargets
    Resource: !Sub "arn:aws:events:${AWS::Region}:${AWS::AccountId}:rule/user{{ ↪
↪ loop.index }}-dev*"

- Effect: Allow
  Action:
    - apigateway:*
  Resource: "*"

- Effect: Allow
  Action:
    - logs:List*
    - logs:Describe*
    - logs:CreateLogGroup
    - logs>DeleteLogGroup
  Resource: "*"

- Effect: Allow
  Action:
    - logs:*
  Resource: "*"
  Condition:
    StringLike:
      "logs:ResourceTag/user": "user{{ loop.index }}"

```


CHAPTER 13

AWS Lambda

AWS Lambda is a function as a service in AWS Cloud, we will cover only necessary parts but you can always refer to the [official documentation](#).

We are going to use these terms throughout the guide: `Function`, `Runtime`, `Event source` and `Log streams`. Please make yourself familiar with [basic concepts of AWS Lambda](#).

13.1 Function as a service (FAAS)

With FAAS you only need to provide your code and select the runtime (interpreter). We are going to use `python3.7` runtime.

AWS Lambda can either be a simple python file or entire module. We need to specify a `handler` to tell lambda where to execute our code.

More details about [python in AWS Lambda](#)

13.2 Handler

Handler is usually a function `lambda_handler` which accepts 2 arguments.

Name of handler function can be anything, you just need to point the `handler` to the function, which accepts 2 arguments.

```
def lambda_handler(event, context):  
    pass
```

`event` - Contains data from `Event Source` usually a Python dict type. It can also be list, str, int, float, or `NoneType` type. `context` - Contains runtime information to your handler. This parameter is of the `LambdaContext` type.

If needed visit [more detailed handler documentation](#)

13.3 Technical details

AWS Lambda function is a amazon linux container running python interpreter.

First invocation of your lambda function starts the whole container, which loads all your code into memory and executes `lambda_handler`. Then container stays idle for some time until amazon reclaim it's resources. During the idle time, your code stays loaded and only `lambda_handler` is executed. This behaviour cold and hot start of AWS Lambda function.

Cold - invocation starts new container and executes `lambda_handler` Hot - invocation executes `lambda_handler` only Check this [article](#) for more information about [lambda container lifetime](#).

You can use this feature to preserve open sessions by declaring them outside `lambda_handler`.

One example for thousands words.

Listing 1: `session_not_reused.py`

```
import boto3

def lambda_handler(event, context):
    session = boto3.Session()
    client = session.client('iam')
    client.get_user(UserName=event['iam_username'])
```

In this example the `session` object is created every time our lambda is called, which can lead to api throttling or different side effects.

We can fix this by declaring our session outside `lambda_handler` therefore it will be created only when lambda function is invoked from a cold state.

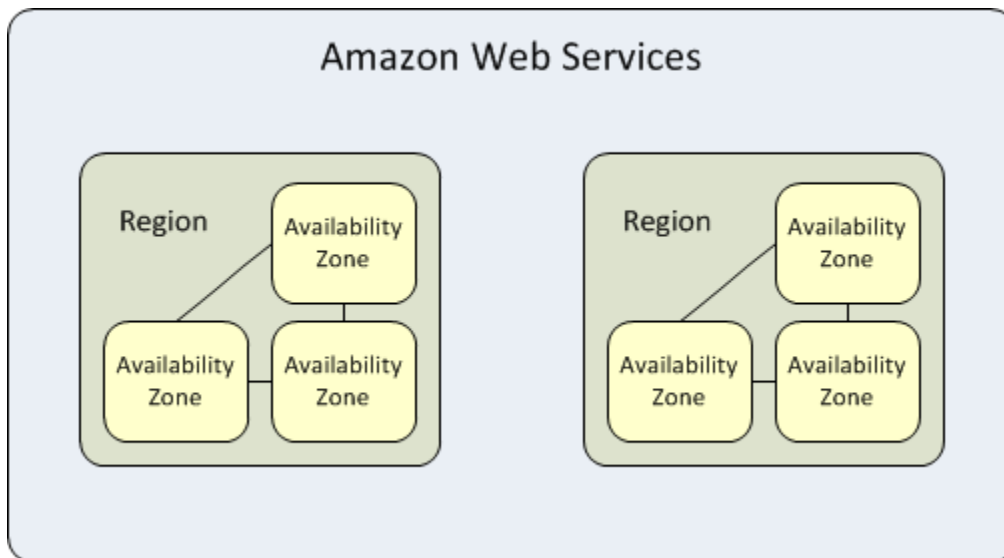
Listing 2: `session_reused.py`

```
import boto3

session = boto3.Session()
client = session.client('iam')

def lambda_handler(event, context):
    client.get_user(UserName=event['iam_username'])
```


These locations are composed of Regions and Availability Zones. Each Region is a separate geographic area. Each Region has multiple, isolated locations known as Availability Zones.



Underlying AWS services

- *IAM* - AWS Identity and Access Management (IAM), With IAM, you can centrally manage users, security credentials such as access keys, and permissions that control which AWS resources users and applications can access.
- **S3 (Simple Storage Service) - is storage for the internet.** Designed for durability of 99.999999999% of objects across multiple Availability Zones
- *Lambda* - Function as a service
- *Cloud Watch* - Amazon CloudWatch provides a reliable, scalable, and flexible monitoring solution
- *API Gateway* - Amazon API Gateway enables you to create and deploy your own REST and WebSocket APIs at any scale.
- *DynamoDB* - Amazon DynamoDB is a fully managed NoSQL database
- *CloudFormation* - AWS Infrastructure as a Code

How to set environment variables

16.1 Mac/Linux (Bash)

```
export AWS_ACCESS_KEY_ID=<your key ID>
export AWS_SECRET_ACCESS_KEY=<your secret key>
export AWS_REGION=us-east-1
export AWS_DEFAULT_REGION=us-east-1
export SLACK_TOKEN=<your BOT token>
export SLACK_CHANNEL=<your slack channel ID>
```

16.2 Windows

Powershell

```
$env:AWS_ACCESS_KEY_ID = "<your key ID>"
$env:AWS_SECRET_ACCESS_KEY = "<your secret key>"
$env:AWS_REGION = "us-east-1"
$env:AWS_DEFAULT_REGION = "us-east-1"
$env:SLACK_TOKEN = "<your BOT token>"
$env:SLACK_CHANNEL = "<your slack channel ID>"
```

Git-bash

```
export AWS_ACCESS_KEY_ID=<your key ID>
export AWS_SECRET_ACCESS_KEY=<your secret key>
export AWS_REGION=us-east-1
export AWS_DEFAULT_REGION=us-east-1
export SLACK_TOKEN=<your BOT token>
export SLACK_CHANNEL=<your slack channel ID>
```

CMD (not recommended)

```
setx AWS_ACCESS_KEY_ID "<your key ID>"
setx AWS_SECRET_ACCESS_KEY "<your secret key>"
setx AWS_REGION "us-east-1"
setx AWS_DEFAULT_REGION "us-east-1"
setx SLACK_TOKEN "<your BOT token>"
setx SLACK_CHANNEL "<your slack channel ID>"
```

CHAPTER 17

Iam Role

Workshop apps hosted inside our environment are protected via managed policy. If you are going through this in your own environment, please use following service role for lambdas.

```
1 provider:
2   # Lambda function's IAM Role
3   iamRoleStatements:
4     - Effect: Allow
5       Action:
6         # Allow lambda to create network interface in vpc
7         - ec2:CreateNetworkInterface
8         - ec2:DeleteNetworkInterface
9         - ec2:DescribeNetworkInterfaces
10        # Allow lambda to write logs
11        - logs:CreateLogGroup
12        - logs:CreateLogStream
13        - logs:PutLogEvents
14      Resource:
15        - "*"
16    - Effect: Allow
17      Sid: AllowDynamoDBAccess
18      Action:
19        - dynamodb:Query
20        - dynamodb:Scan
21        - dynamodb:GetItem
22        - dynamodb:PutItem
23        - dynamodb:UpdateItem
24        - dynamodb>DeleteItem
25        - dynamodb:DescribeTable
26      Resource: "arn:aws:dynamodb:${opt:region, self:provider.region}:*:table/$
    ↪{self:provider.environment.DYNAMODB_TABLE}"
```

Remove the section `provider.iamManagedPolicies` from our *original serverless.yaml* and insert `iamRoleStatements` section from above.