Welcome everyone. Today, we're going to do a quick deep dive into server less workflows.
These slides and other resources will all be available via a link at the end, so don't worry about trying to screen cap everything.

Scott Blake

Flywire

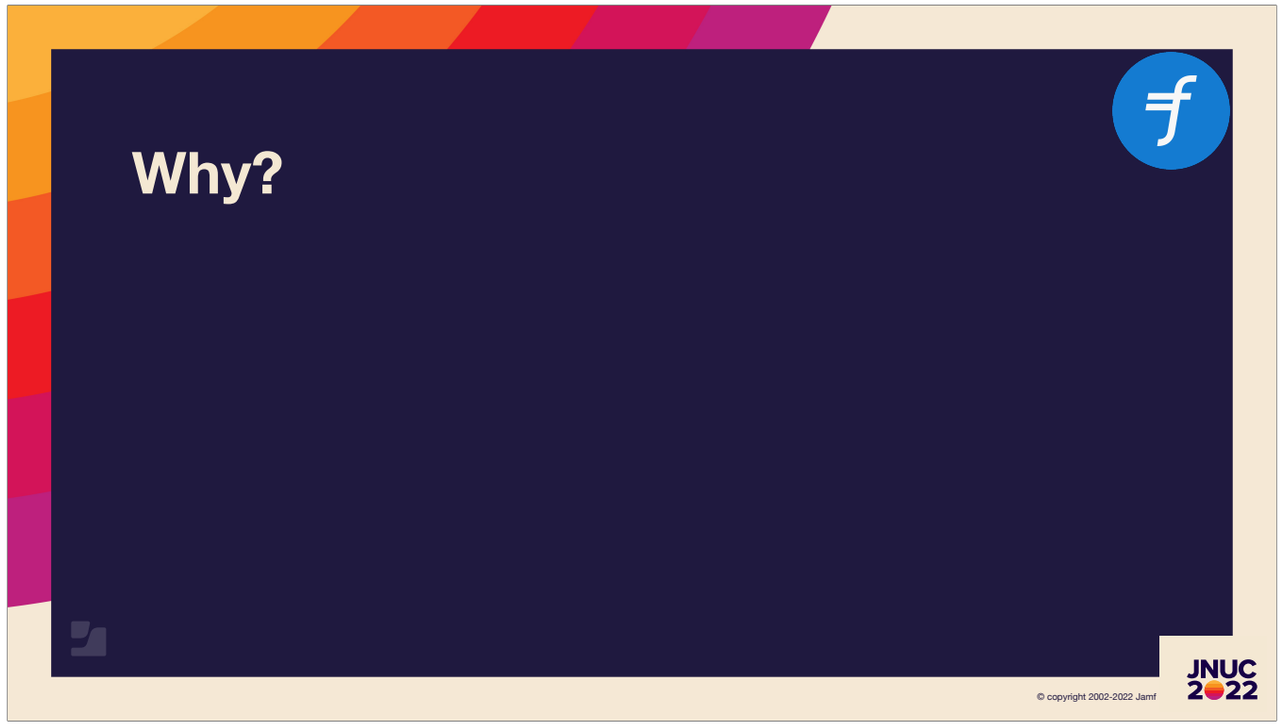
I'm Scott Blake, a Client Platform Engineer at Flywire. I've been here about a year. Before that, I worked in Higher Ed for 20 years. I've been managing Apple devices since the days of Leopard and Snow Leopard. If you're looking for me on the internet, I am @MScottBlake on most sites.

Take notes if you want... or don't.
I uploaded a copy of these slides and the scripts to my Github repo if you want to follow along.
This slide will appear again at the end, so don't worry if you didn't get it in time.

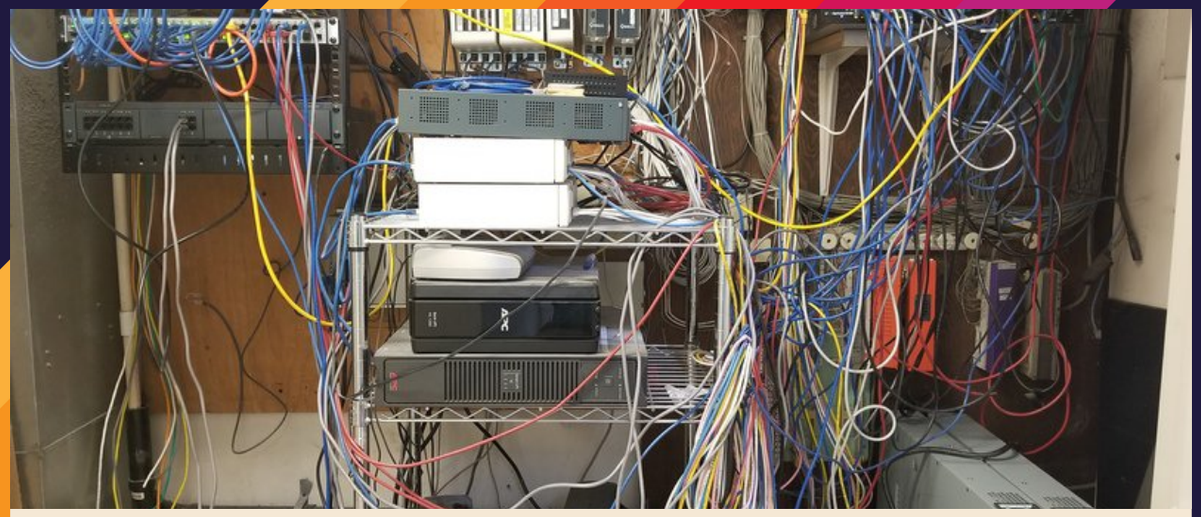
Agenda

Why?
Local Environment
Common Services
Examples

JNUC 2022



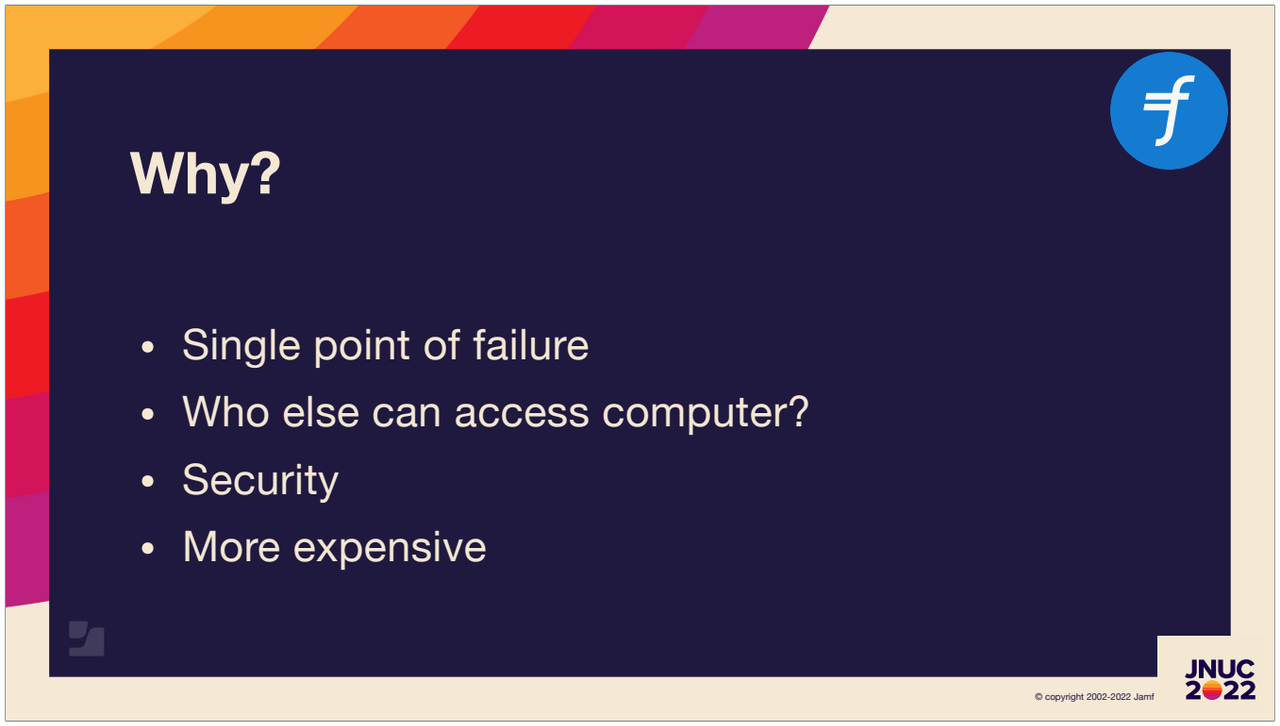
Let's start off with a simple question. Why do we care about server less?



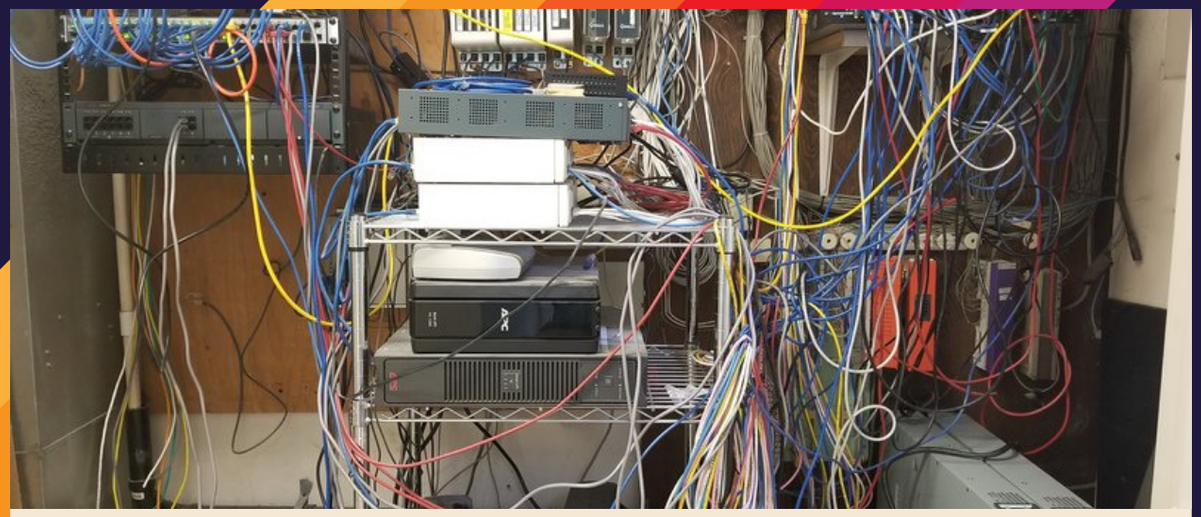
Closet

JNUC 2022

If you have a computer that runs automated scripts from a network closet or an office corner, type a ONE in chat.
If you have scripts on your local computer that you run occasionally, type a TWO in chat.



Back to the question...
Most people don't have backups of scripts run in environments like this.
In my experience, it's common that only 1 person fully knows what it does.
They are often auto login and not patched like other computers.
Hardware is expensive to buy and maintain.

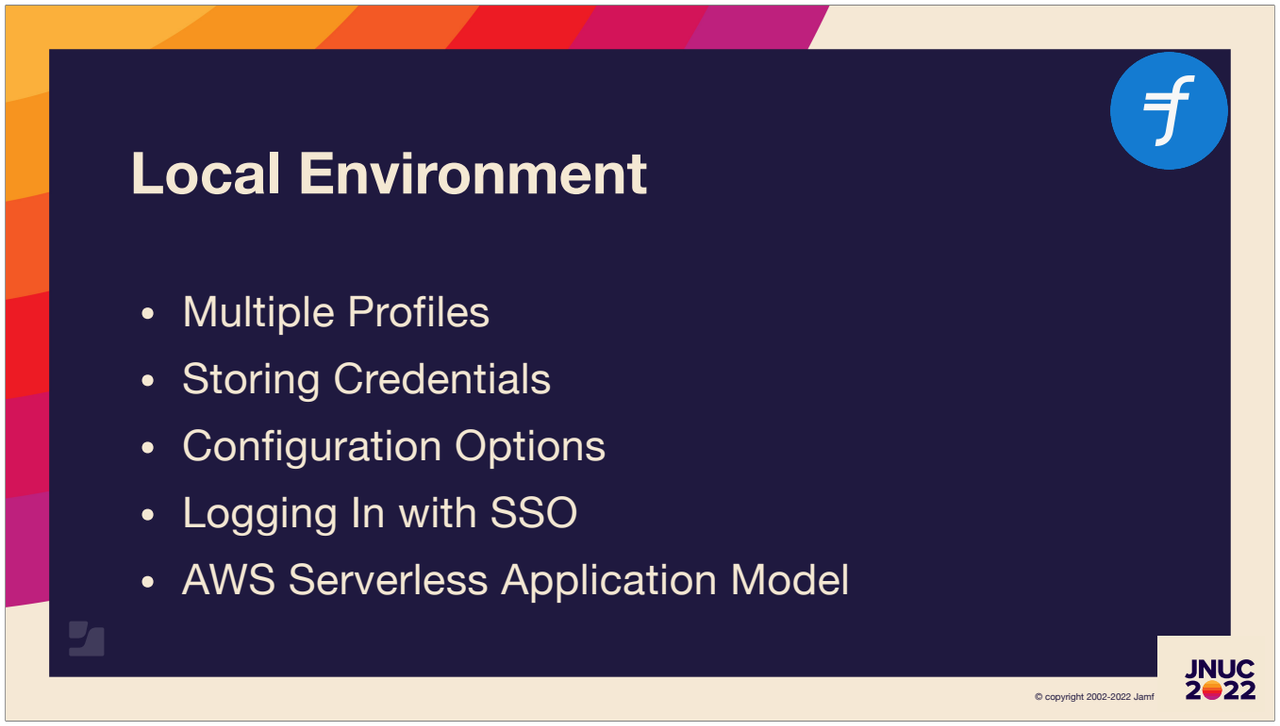


Closet

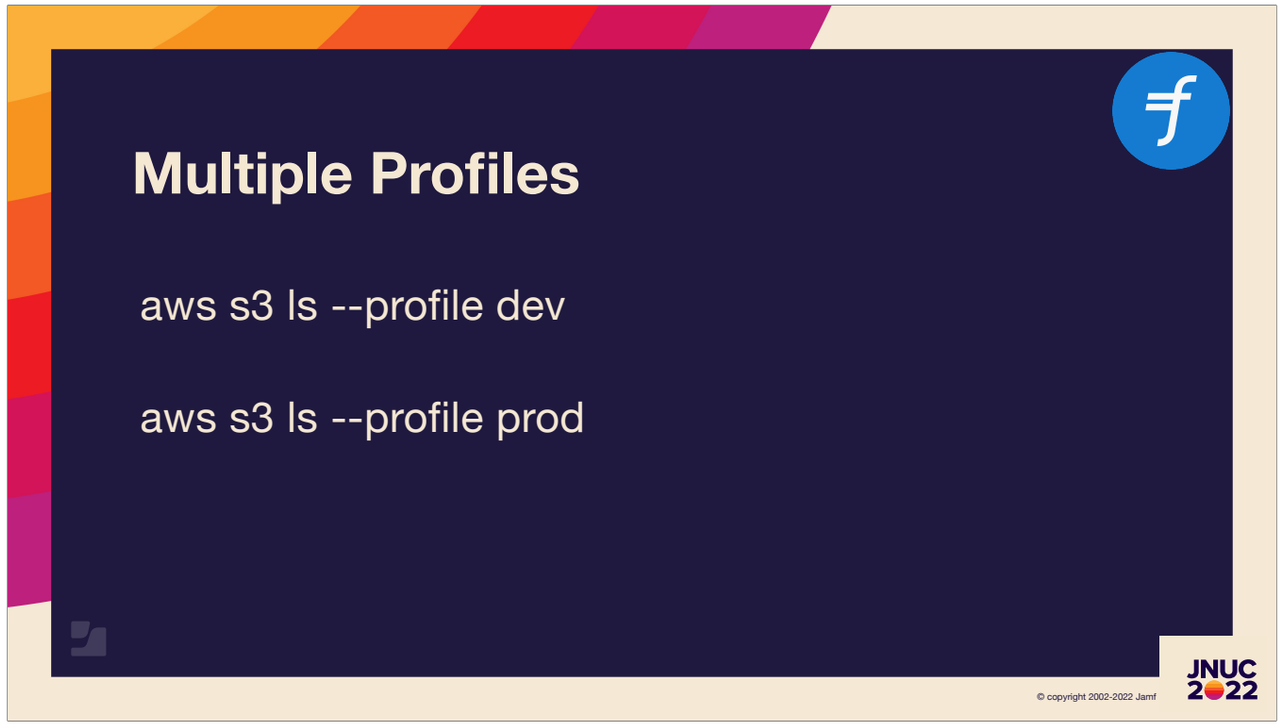
JNUC 2022

In essence, the goal of this talk is to take you from this.

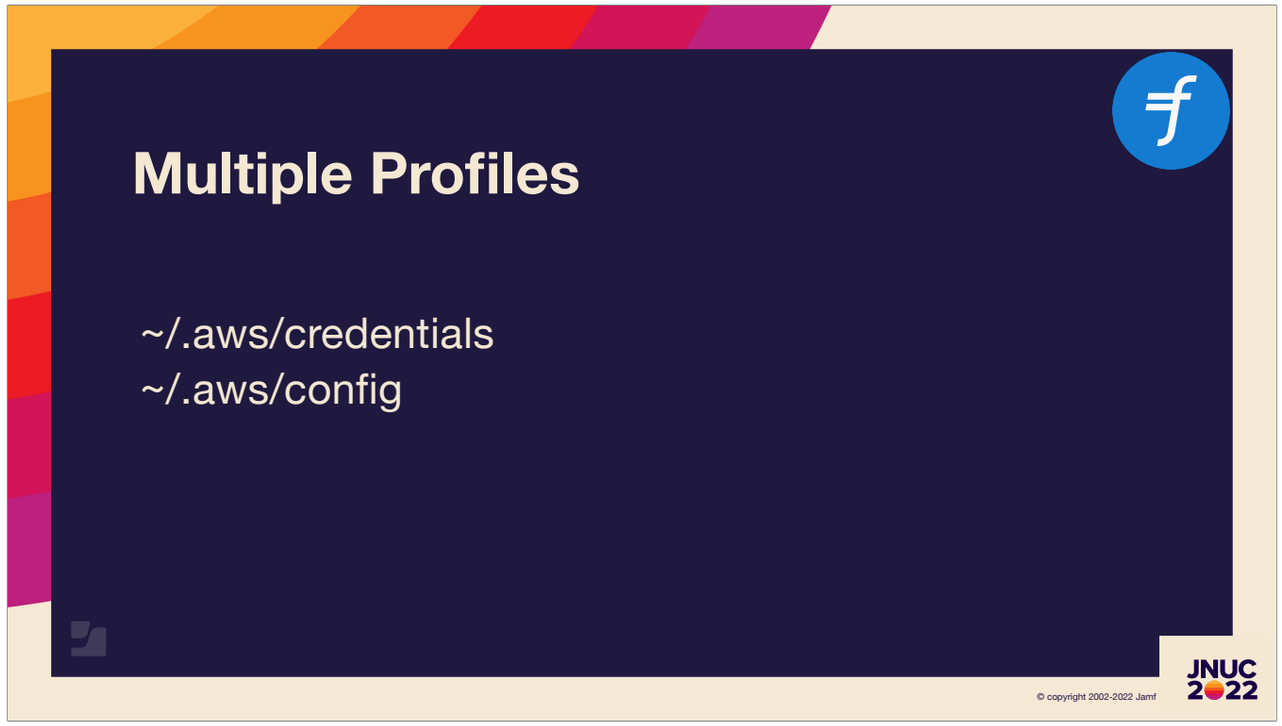
To this.
Repeatable code, stored in a central location, and only costing money when it runs.
There are many ways to achieve this. I'm going to walk through an example with AWS and the Jamf Pro API, but these concepts are certainly not limited to either. Azure, Google, and others all have similar services, and you can apply these concepts to any vendor API.



To get there, we need to talk about a baseline local environment.

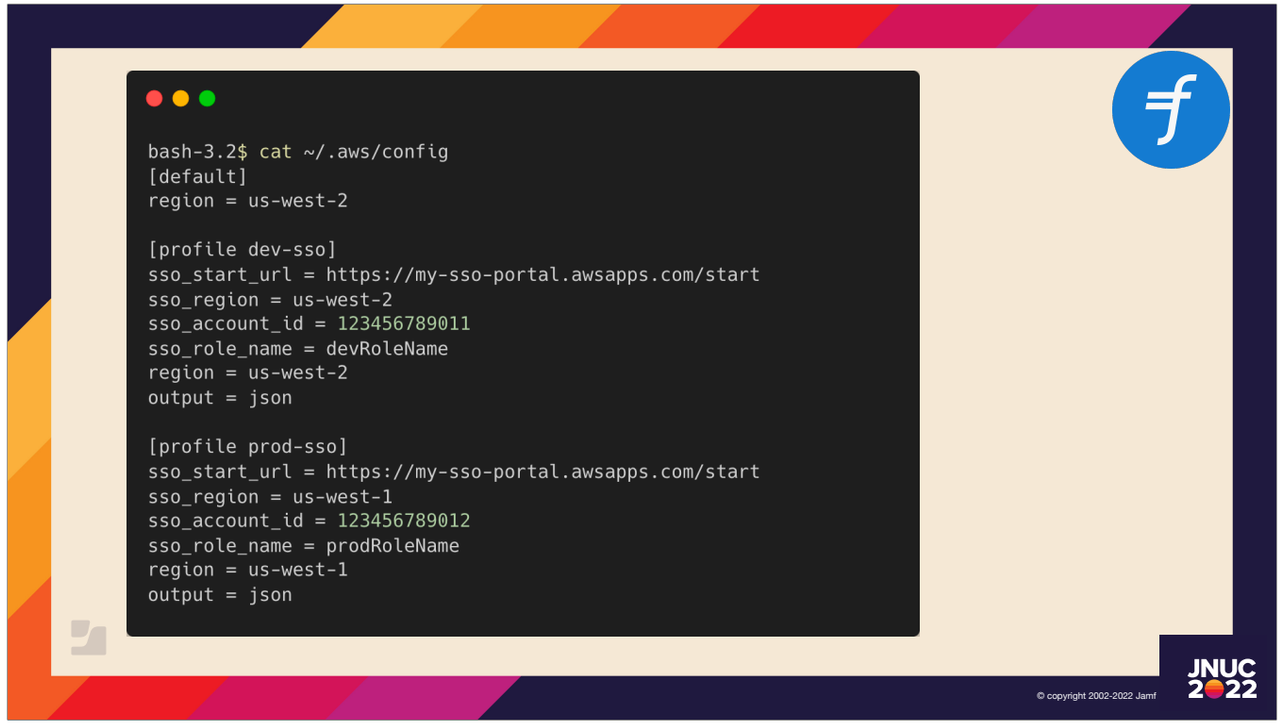


The AWS CLI is so powerful that I could probably spend an entire session talking about it. One of the most useful parameters it accepts is the --profile flag. This allows you to easily declare an entirely different environment or even different account to work with.

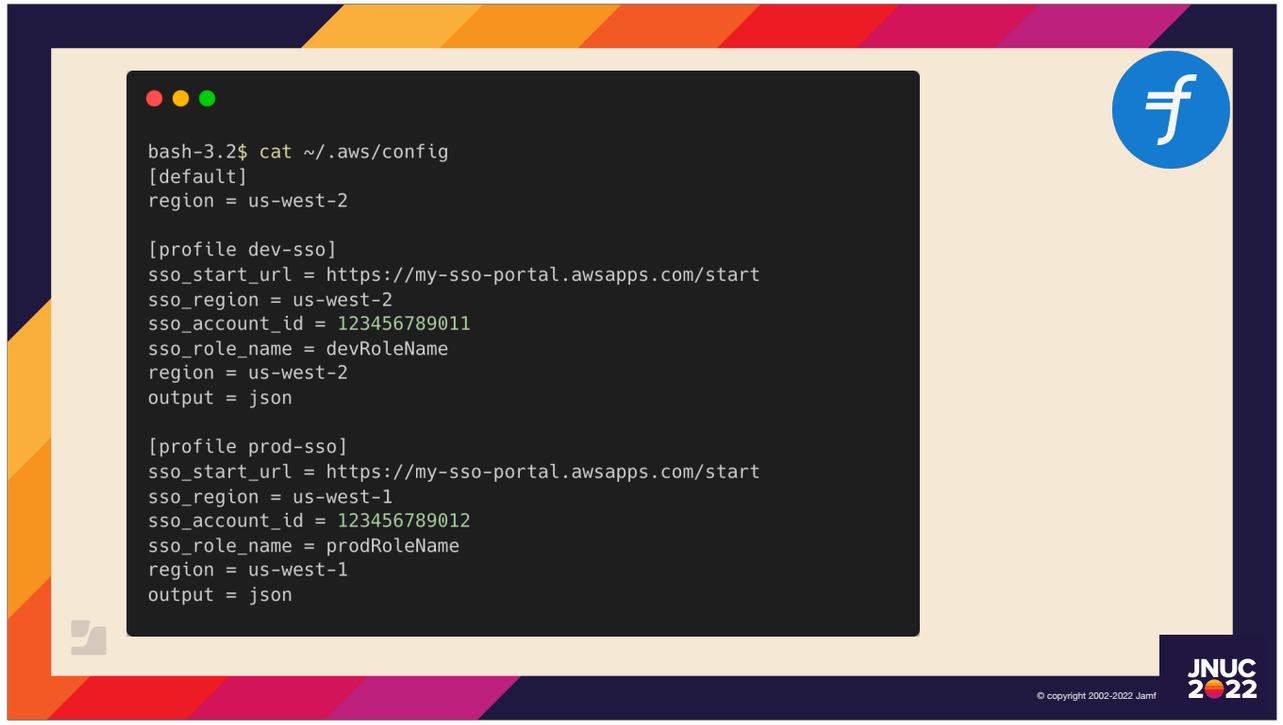


To configure your different profiles, you will be modifying these two files. You will definitely have a config file, but depending on your environment, you may not need a credentials file.
Each environment is different and I suggest you read the AWS documentation to get a better idea of what's needed. I'll show some examples, but they are nowhere near the full capabilities.

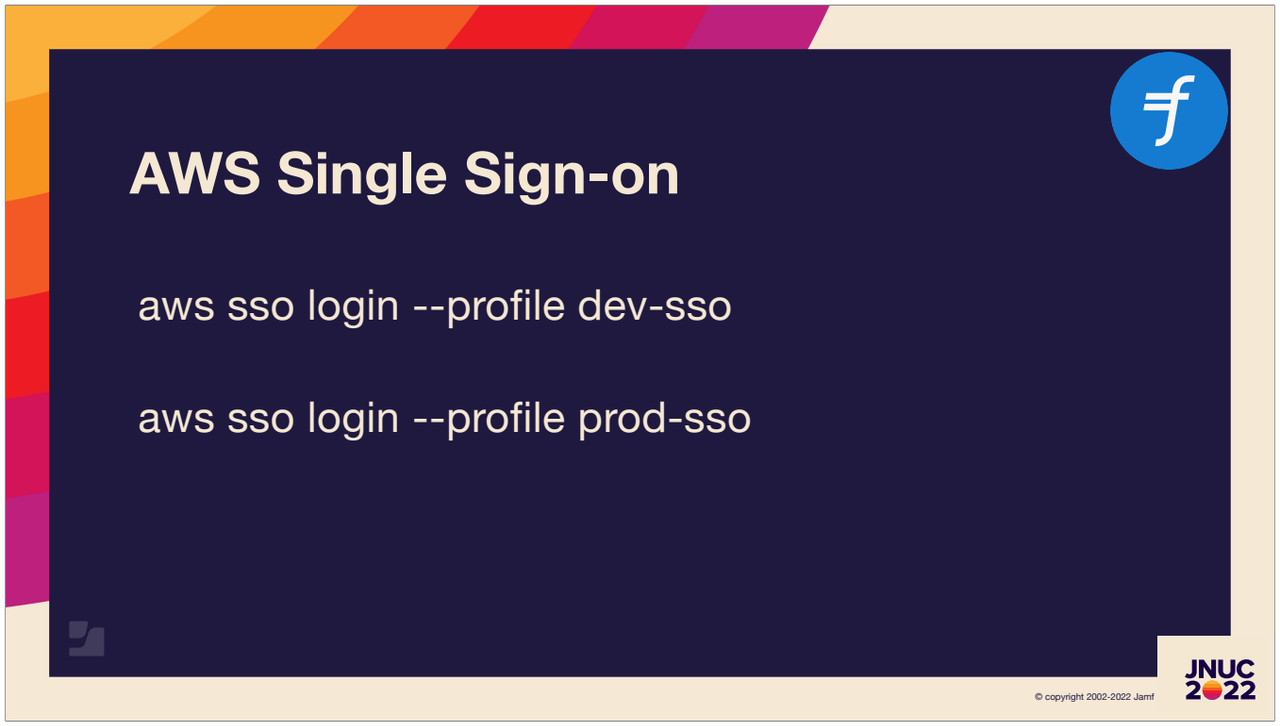
The most basic AWS credentials are an Access Key ID and a Secret Access Key. These fields can go into a credentials file as shown. Notice that in my example, the default is the same as dev. This isn't necessary, but it means I don't have to type --profile on every dev iteration.



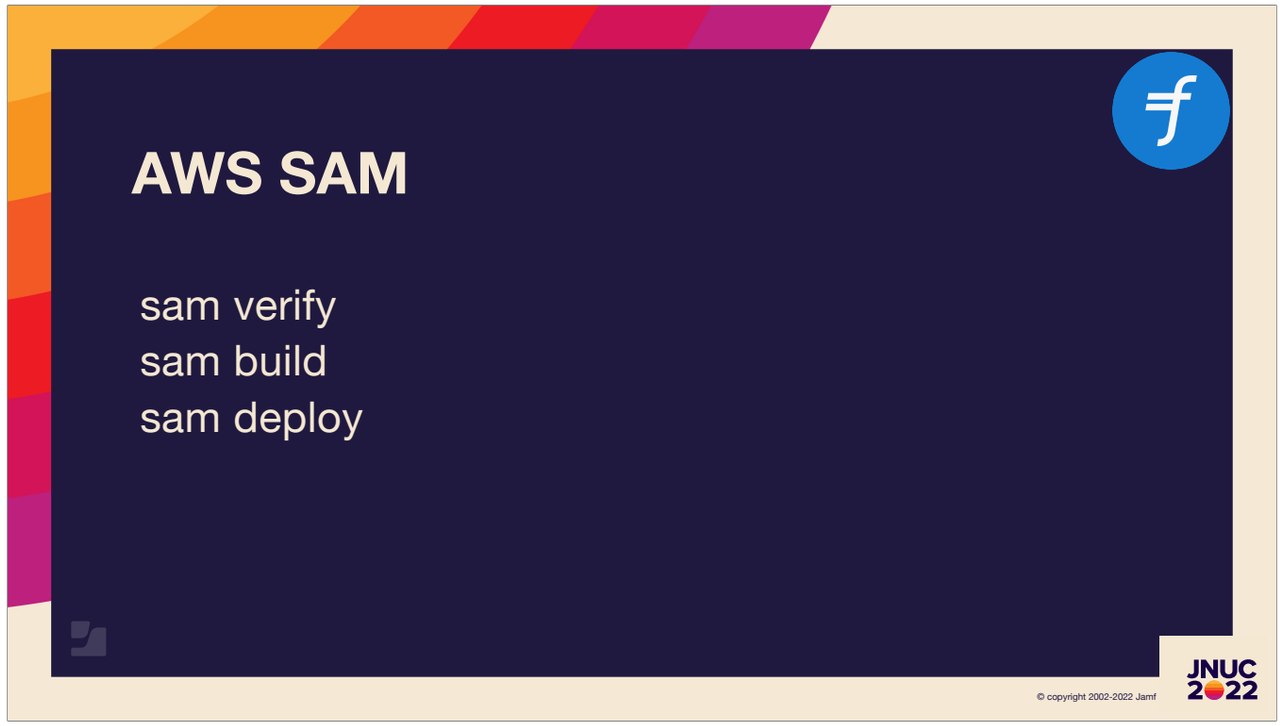
The config file is way more powerful. The data stored in this file serves as a default. You can override them in your commands, but if you don't include them, it looks here. Notice that the headers contain the word *profile*. The credentials file doesn't use that keyword.



If your organization uses SSO, those settings go into config. This example shows that you can not only define different regions, but you can also define entirely different accounts and roles.
I used different profile names, but you can and will use the same names in credentials and config files.

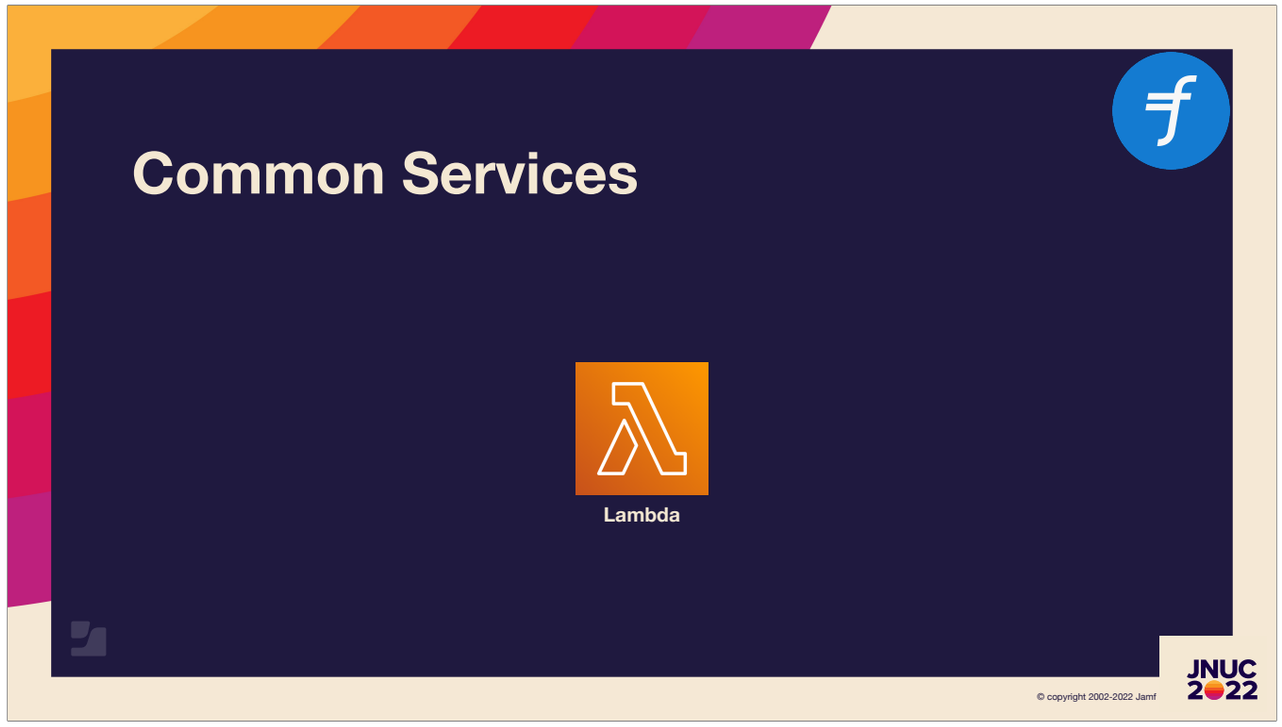


Signing into SSO is not difficult once it's configured. If you were to run both of these commands, signing into prod will create a second set of temporary credentials and does not override dev. You end up being logged into both at the same time.

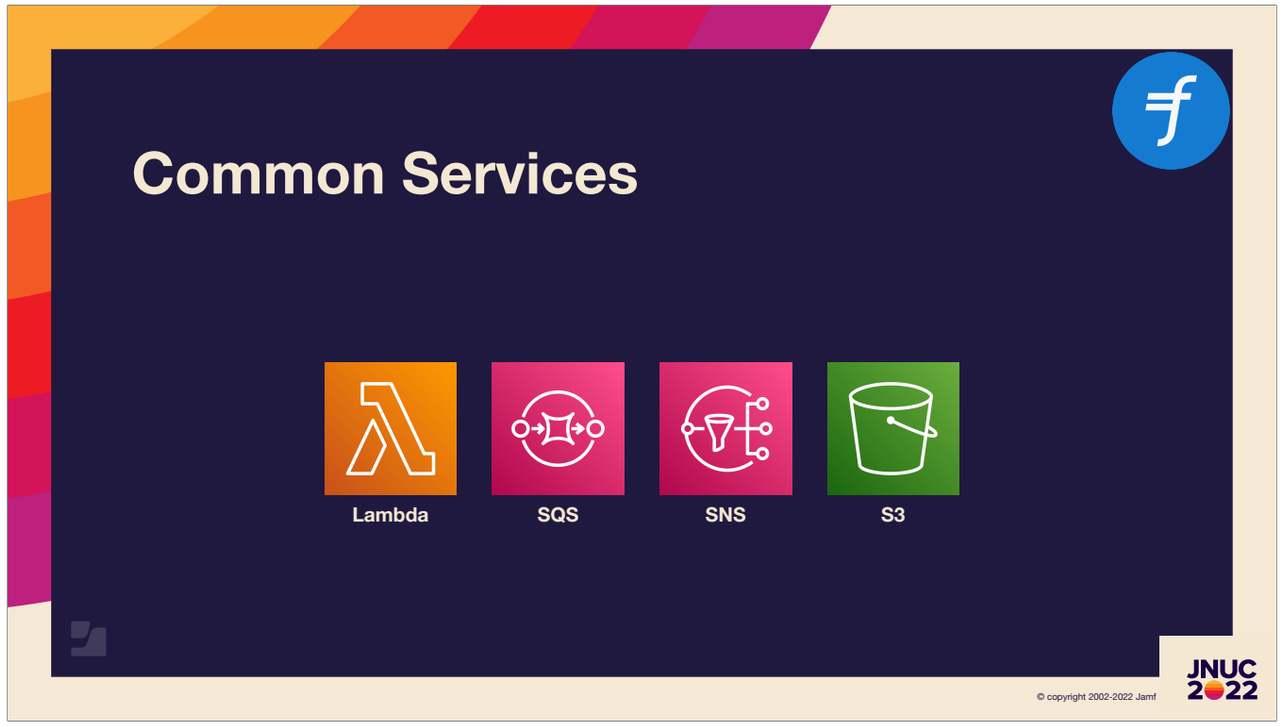


The SAM CLI has quite a few commands, but the 3 most common are verify, build, and deploy. The verify command makes sure that your template is valid. The build command takes your code and puts it into a deployable format. This step must be done before deploy if any changes are made.

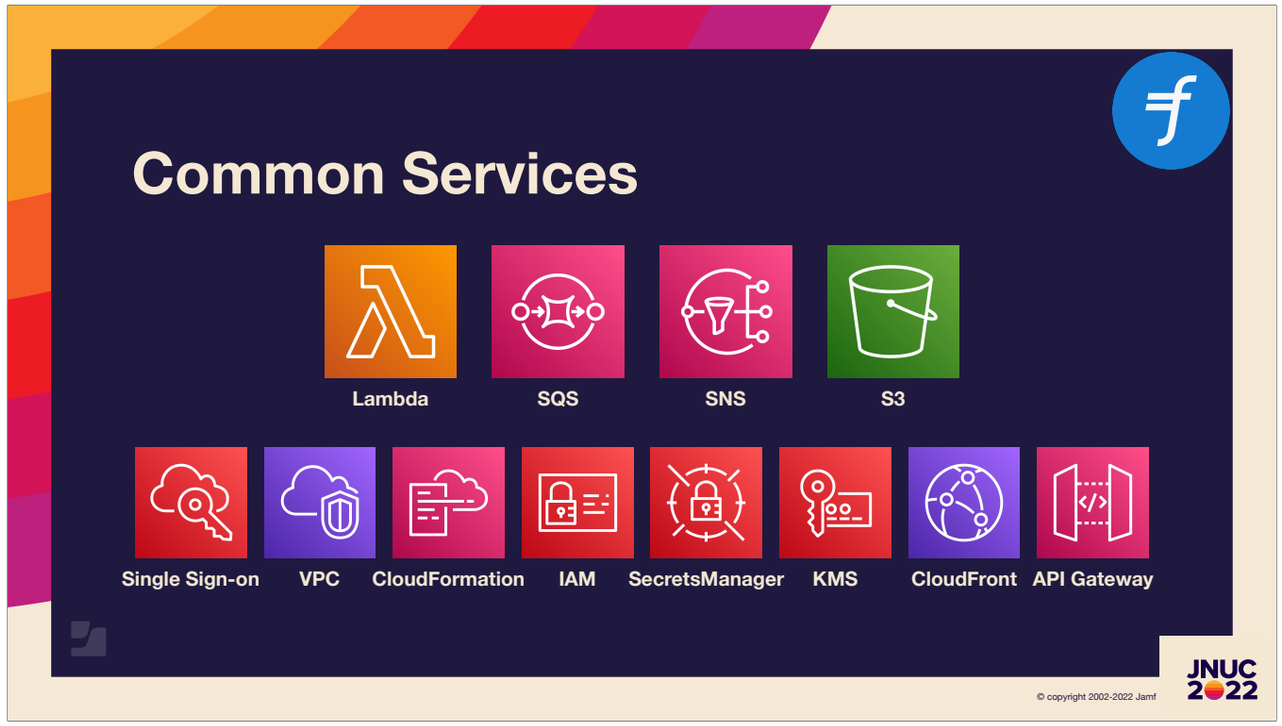
The deploy command is the one you use to push your changes to AWS. The first time you deploy to each profile, you need to use the --guided parameter. It will prompt you for more information and it will save your answers to a local toml file. Subsequent deploy runs will pull values from that file so you don't need it to be guided.



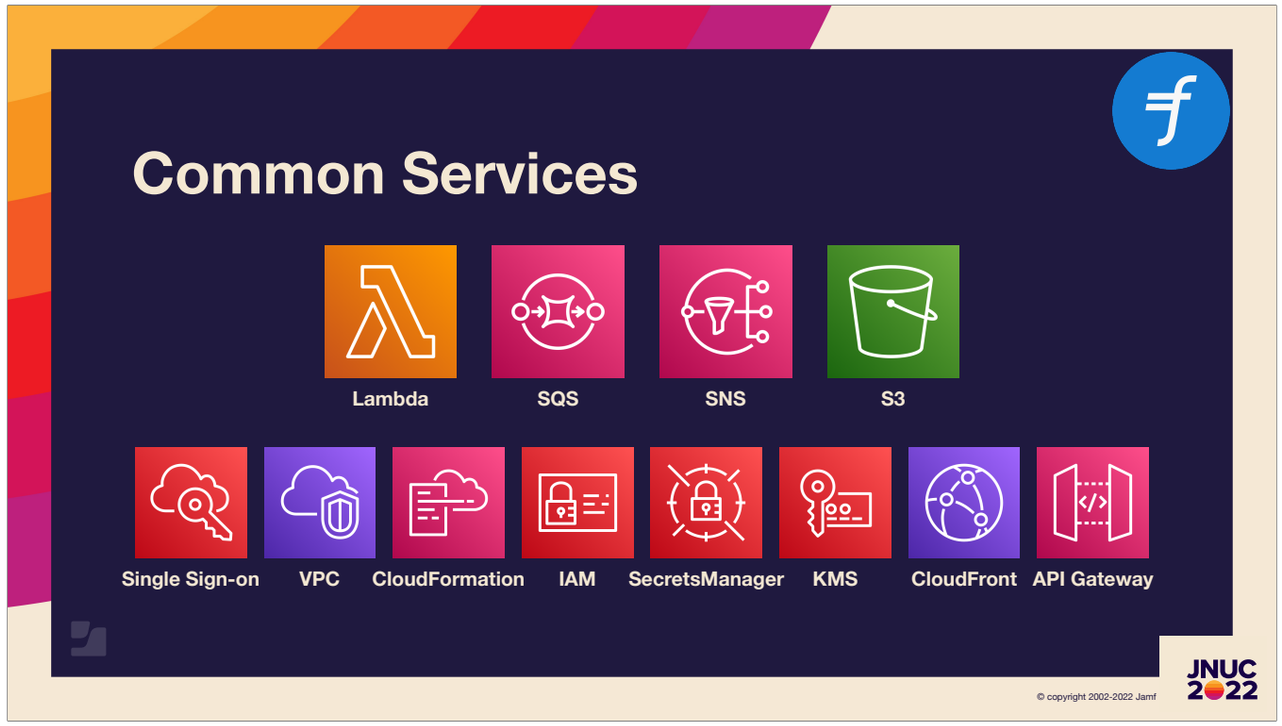
For the purposes of this talk, we're mainly talking about Lambdas. This service allows you to run code in the cloud without having to worry about the server. You are only charged during runtime. Compare that to a computer in the corner where you are paying for electricity and hardware refreshes.



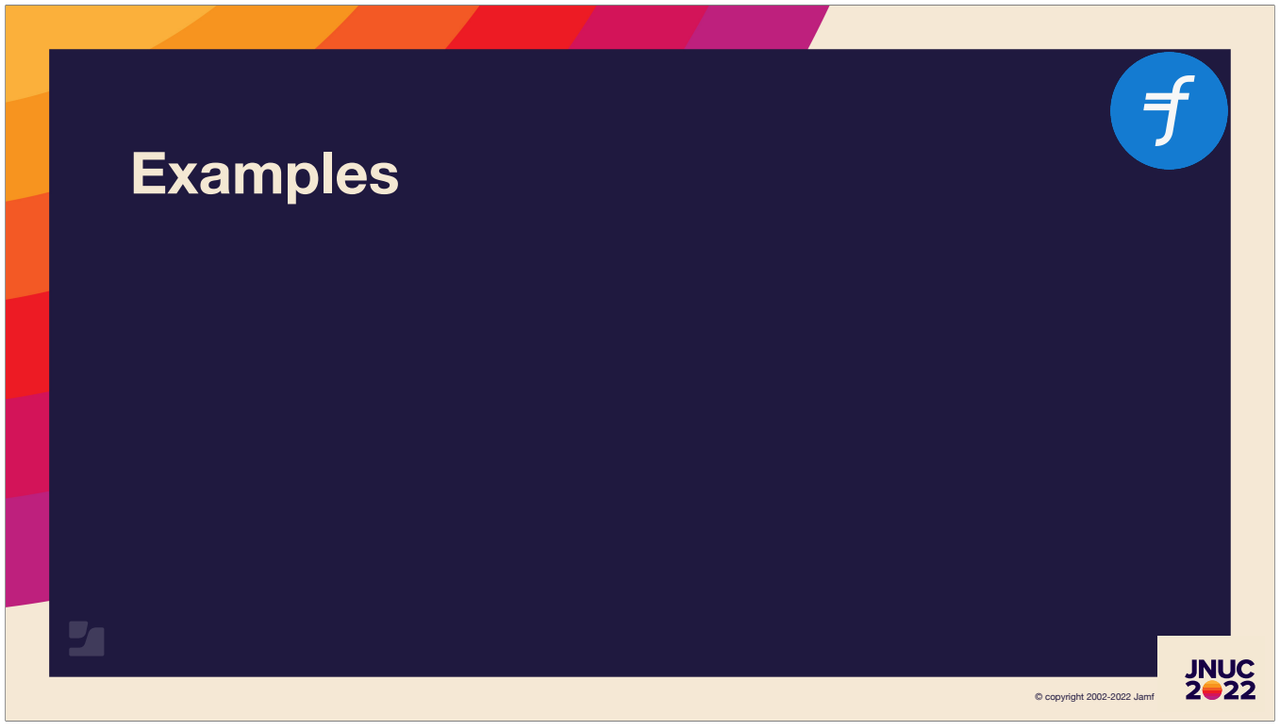
S3 gives us a place to store the code. SQS and SNS are services we can use to communicate between Lambdas or other services. SQS is a queue, and ensures the messages are handled. SNS is a notification, or put another way, messaging via UDP. There is no guarantee that it made it. Each is useful in different scenarios.



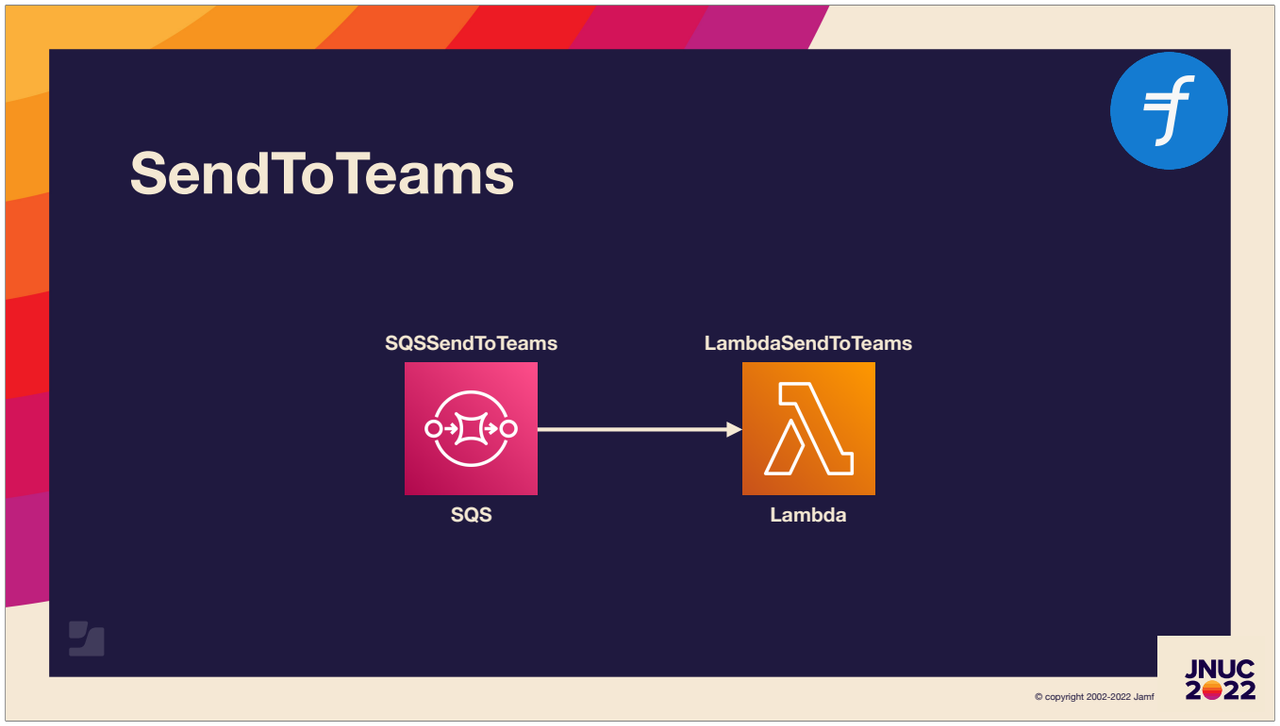
Additionally, here are some of the other services we'll touch on today. Some of these are configured automatically. They are useful to know, but may not be crucial to know in-depth.



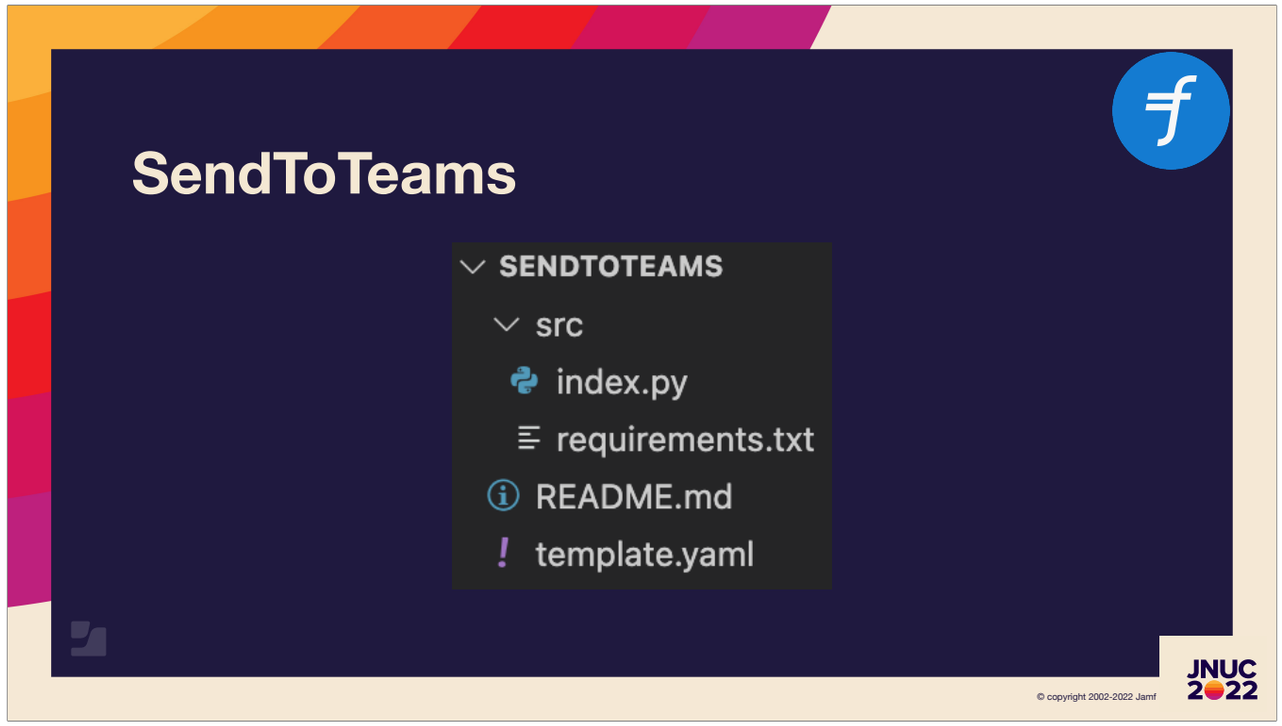
For instance, VPC (or Virtual Private Cloud) is like a VLAN and lets you isolate your infrastructure. You can use IAM (or Identity and Access Management) to grant access to SecretsManager which uses KMS (or Key Management Service) to secure its secrets.



Let's head into a few examples.

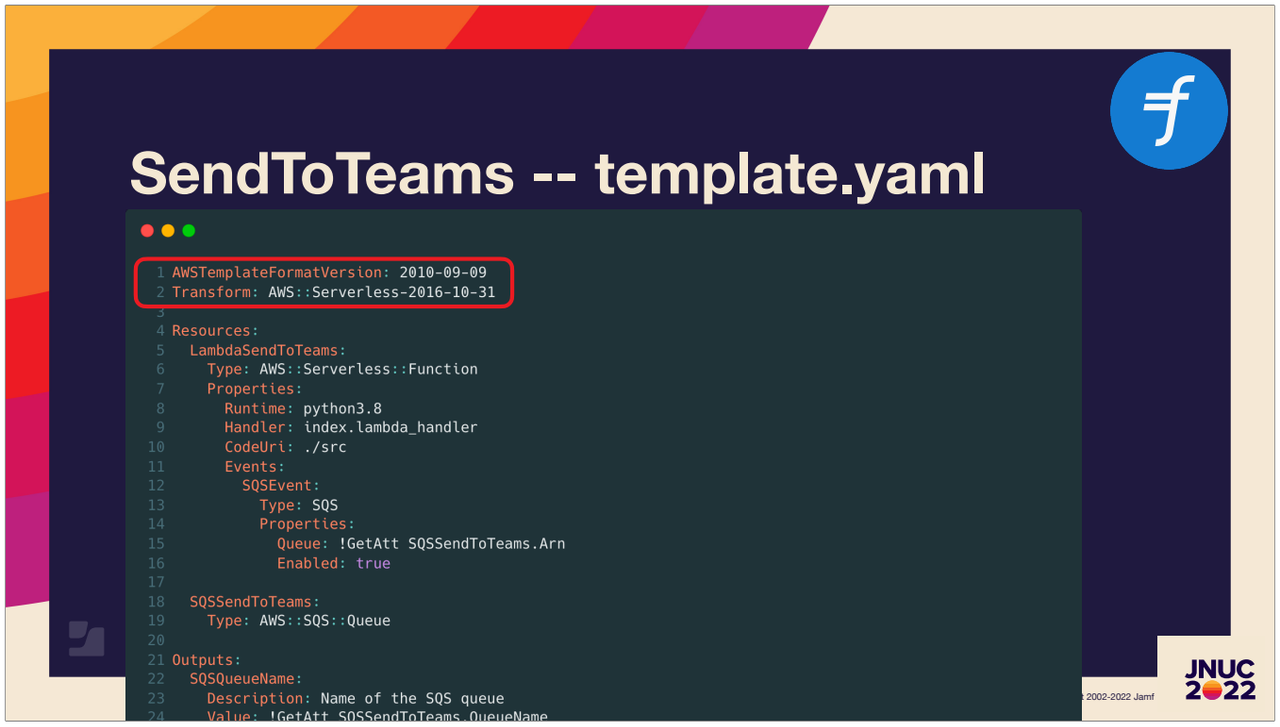


The first example is a simple Lambda that sends a message to a Microsoft Teams channel. It triggers automatically whenever a message appears in a specific SQS queue.

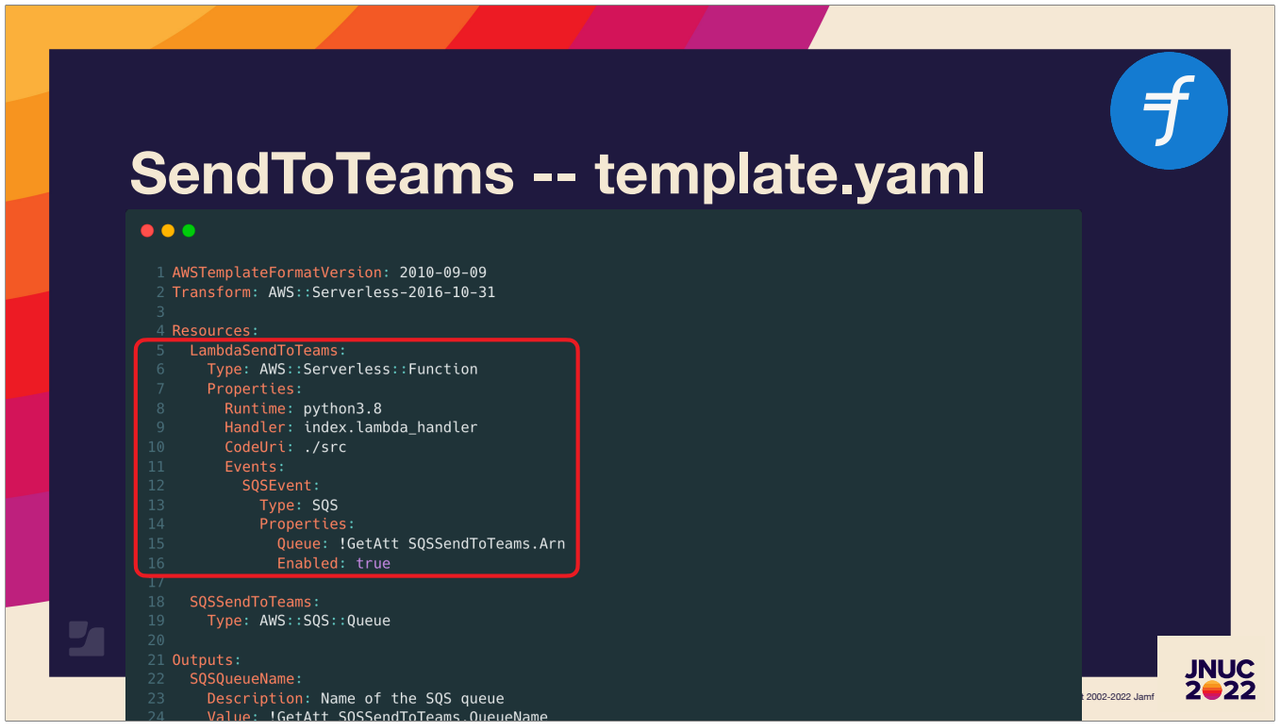


Here's the folder structure of this project. There is a template file in the root of the directory. This file is written in the AWS Serverless Application Model syntax. There is also a folder containing the lambda's code: a python script and a corresponding requirements text file. That's all there is to make this work. Of course, any good project needs a good readme as well.

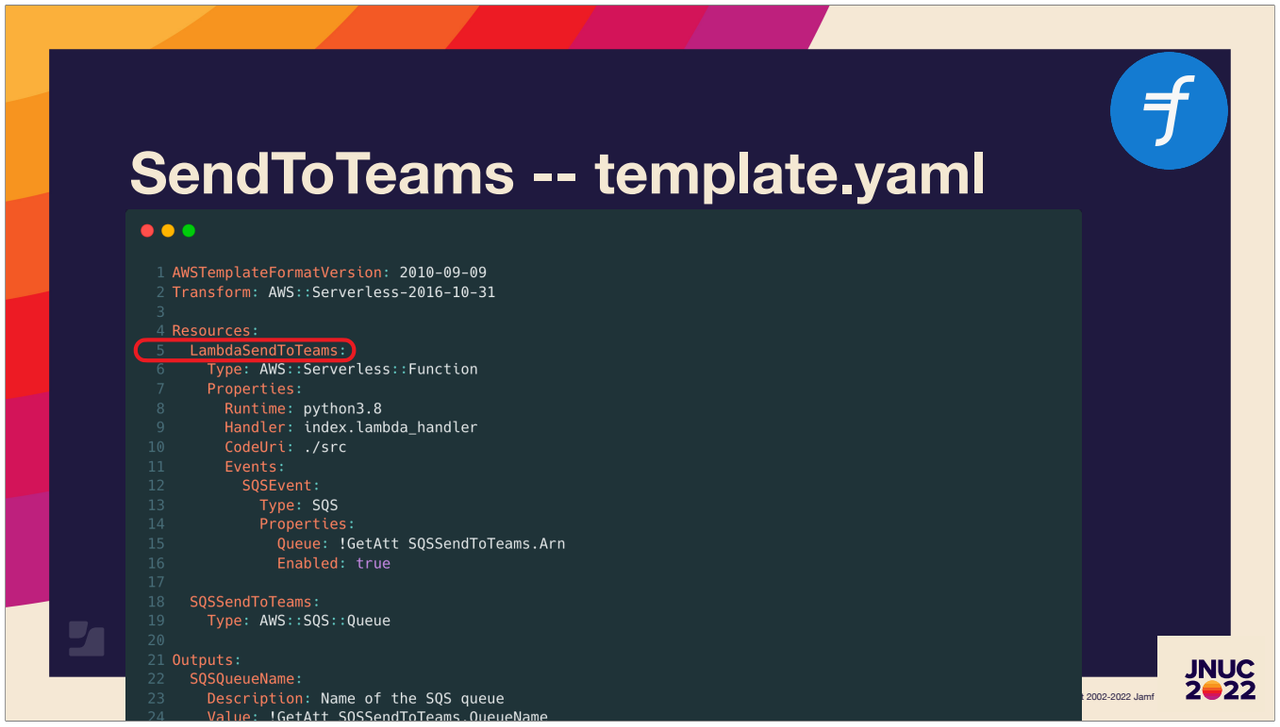
Here's the template yaml file.



The AWSTemplateFormatVersion is optional. It identifies the capabilities of the template.
The Transform key defines a macro used to transform the template into a compliant CloudFormation template. Within the Resources section...

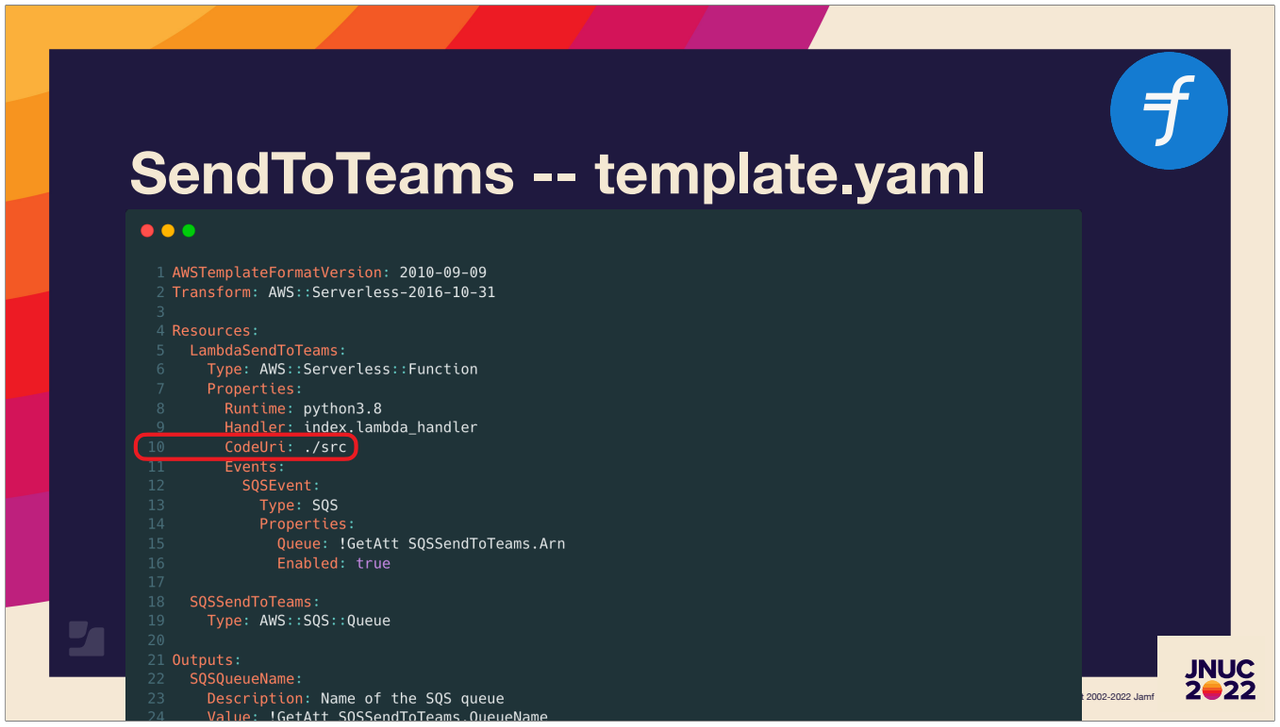


We define the AWS resources used in this stack. Here we have a Lambda being defined.

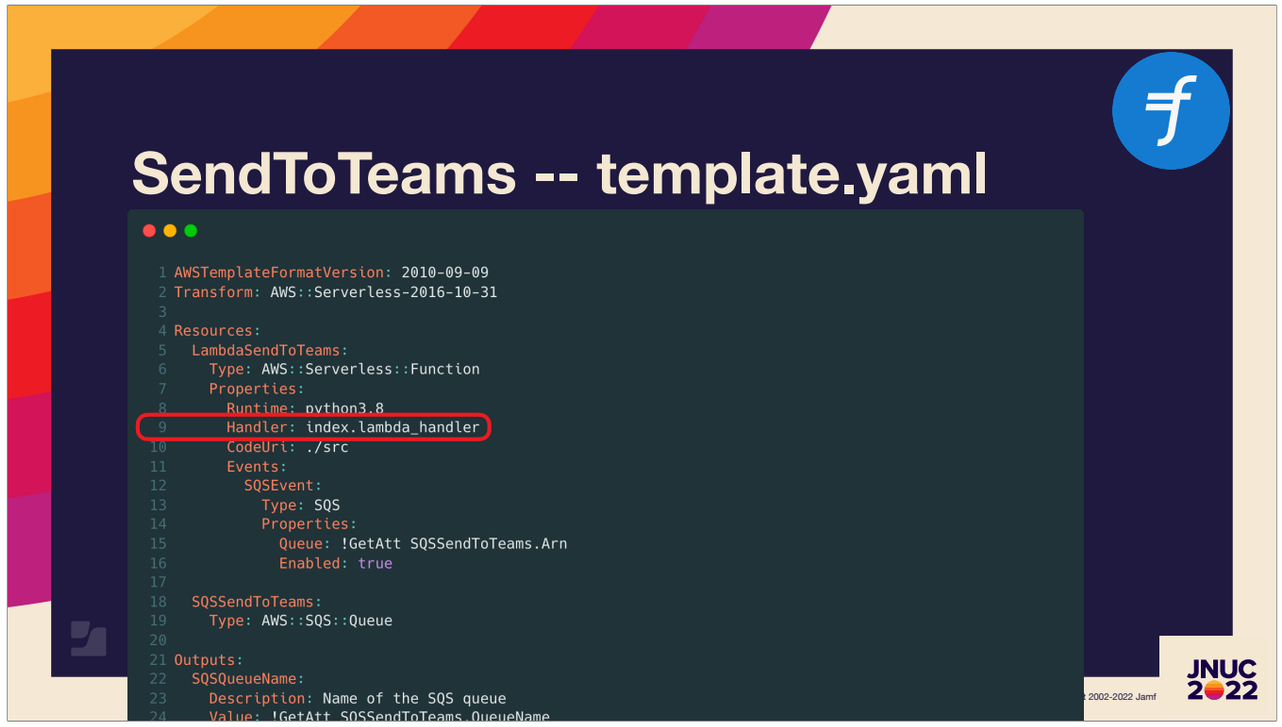


This is the name we are giving the resource.

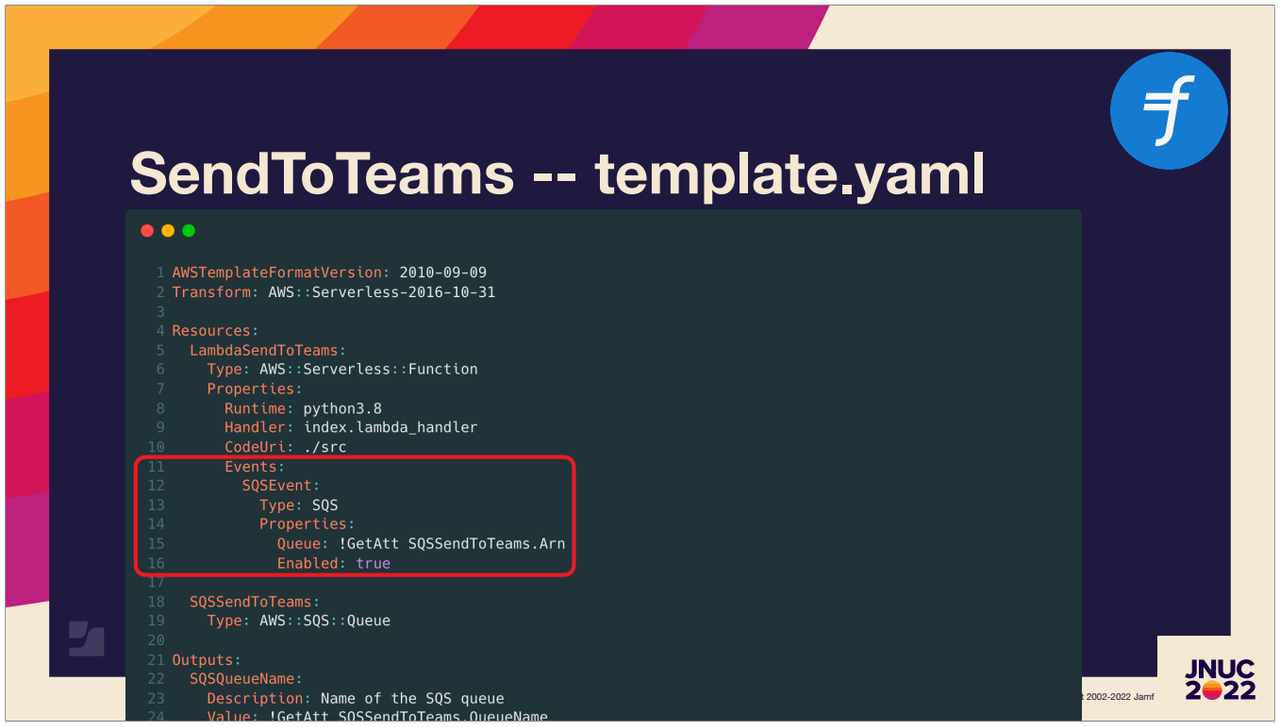
Here we are defining the environment to use. In this case, we are running python 3.8 code.



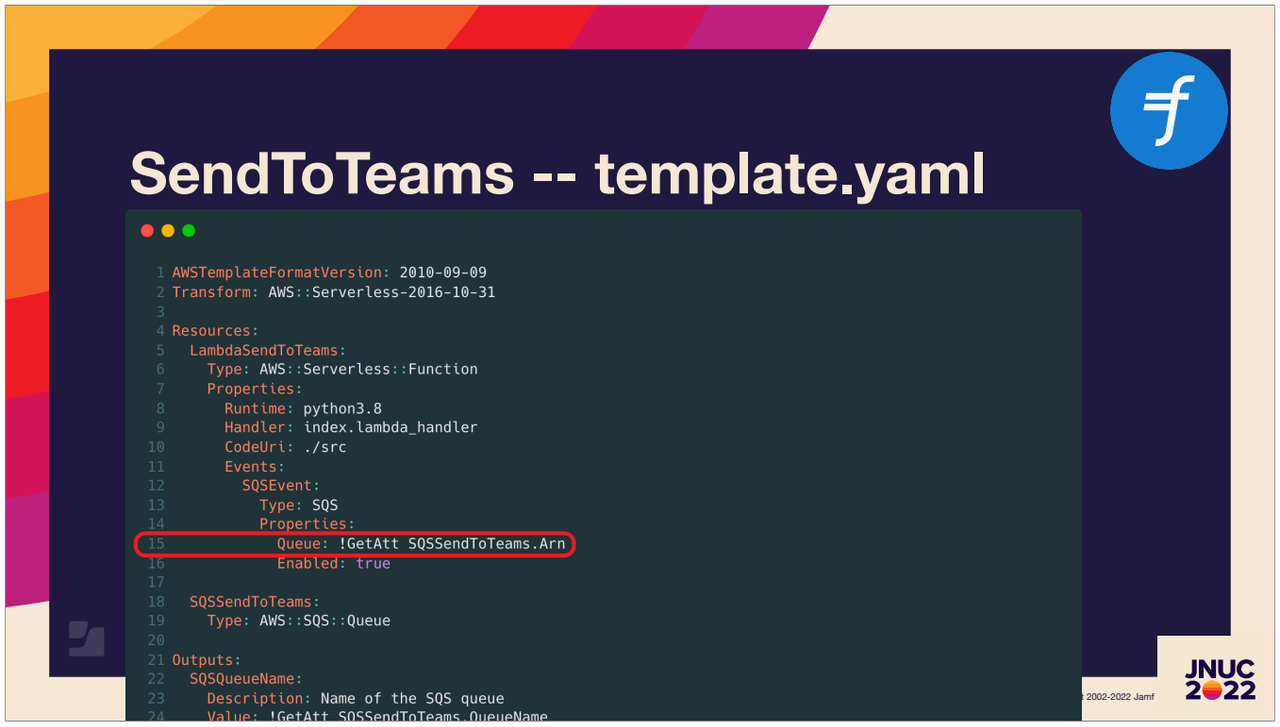
When it runs, it will look in the src directory.



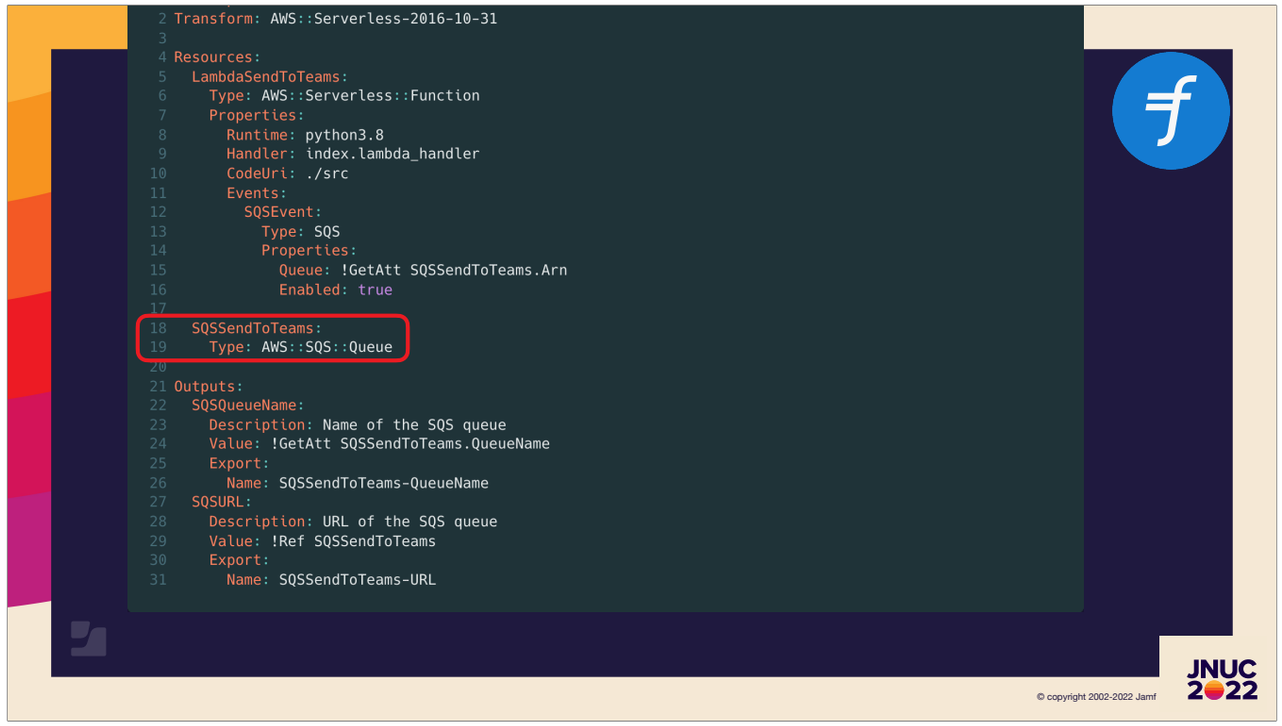
If it finds that directory, it will call the lambda_handler function within the index file.



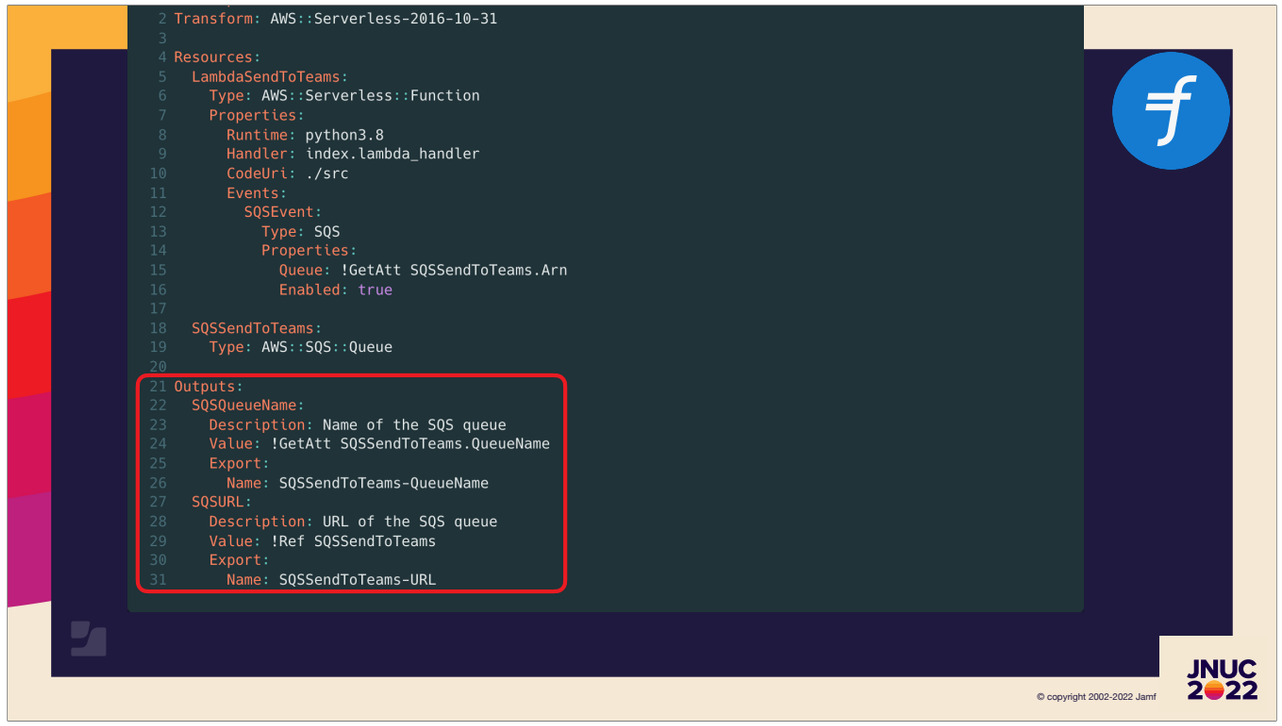
The code is triggered on this event. We give it an arbitrary name because we can have multiple events.



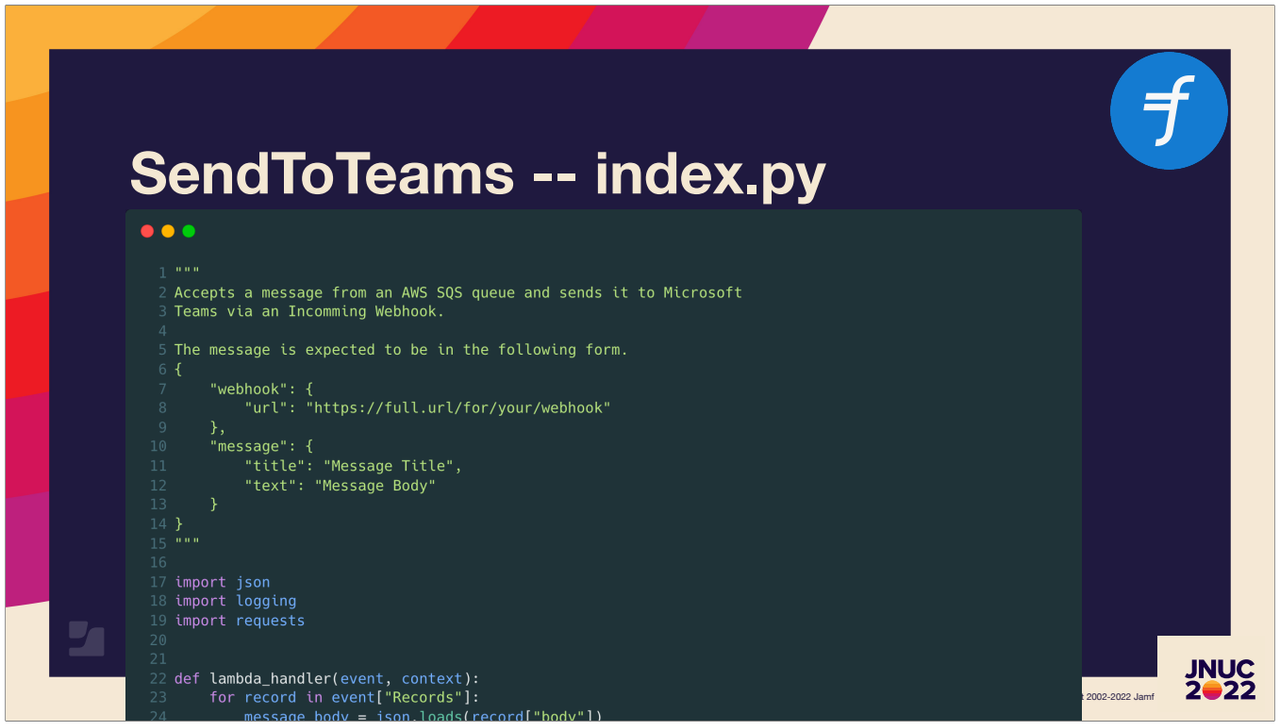
This line designates the specific SQS queue to use. Here we are using a SAM Template shortcut to get the Amazon Resource Name of the SQSSendToTeams queue. This attribute is globally unique in AWS.



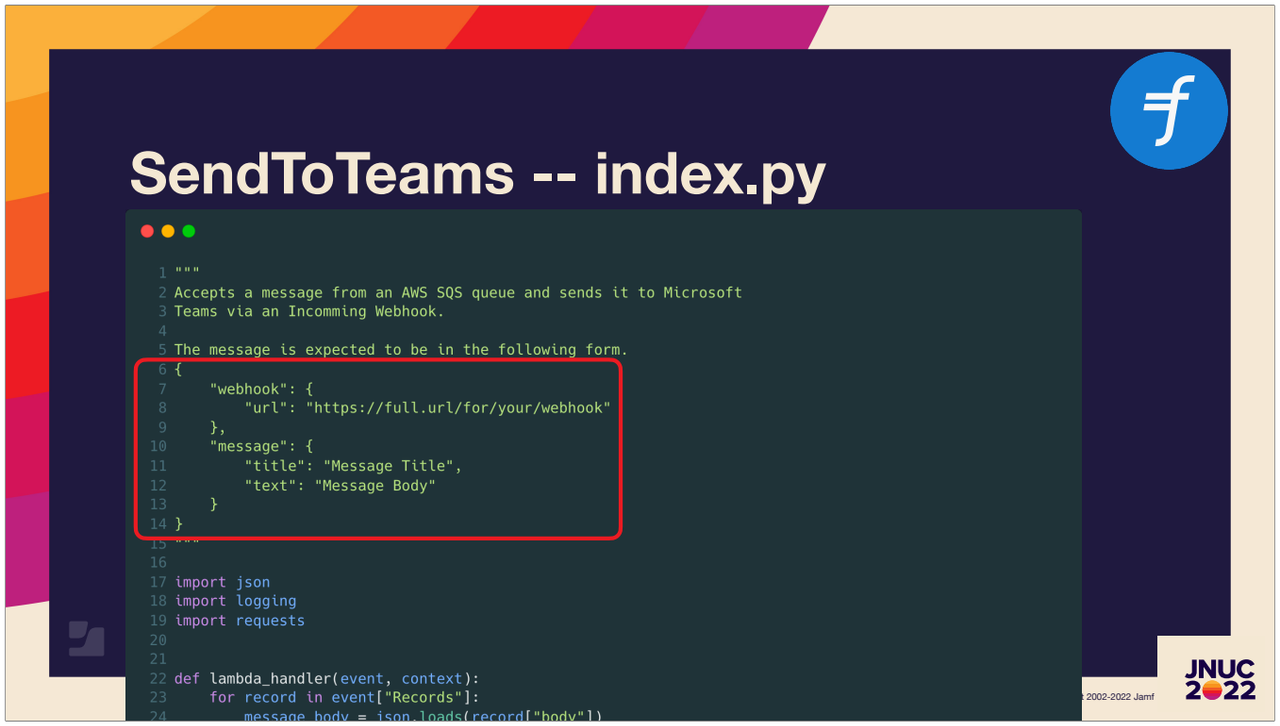
Next, we configure an SQS queue and name it. You'll notice this is the same name referenced in the Lambda. There are tons of other configuration options for both of these resources. These are just the requirements to make this workflow work.



Finally, we get into a section that is not covered in many tutorials. Let's say you have other code somewhere and you want to reference this SQS queue. The outputs section exposes details to resources outside of this stack. In this case, we are defining a Name and URL that we can target to add messages for processing by the Lambda.



Here's the template yaml file.

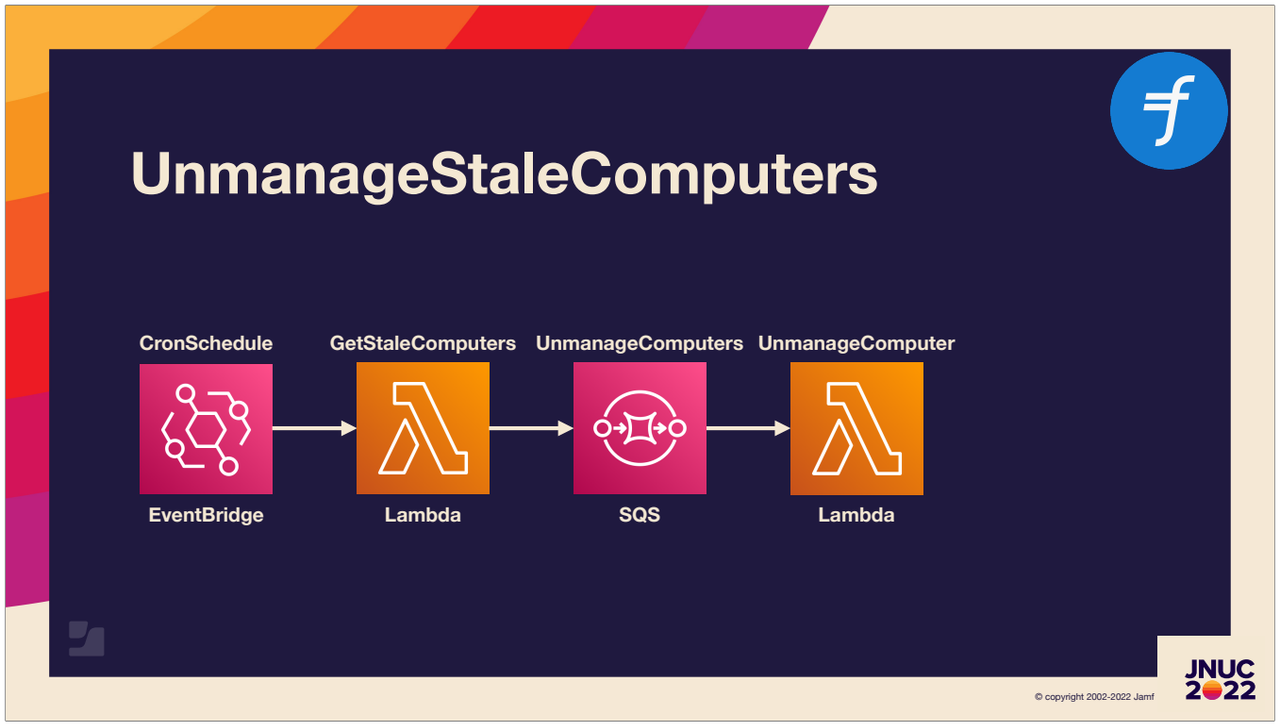


Before we get into the script itself, I wanted to call out that I like to always document the expected format within the comments. I find it easier to comprehend what a script is doing when I know what the input looks like.

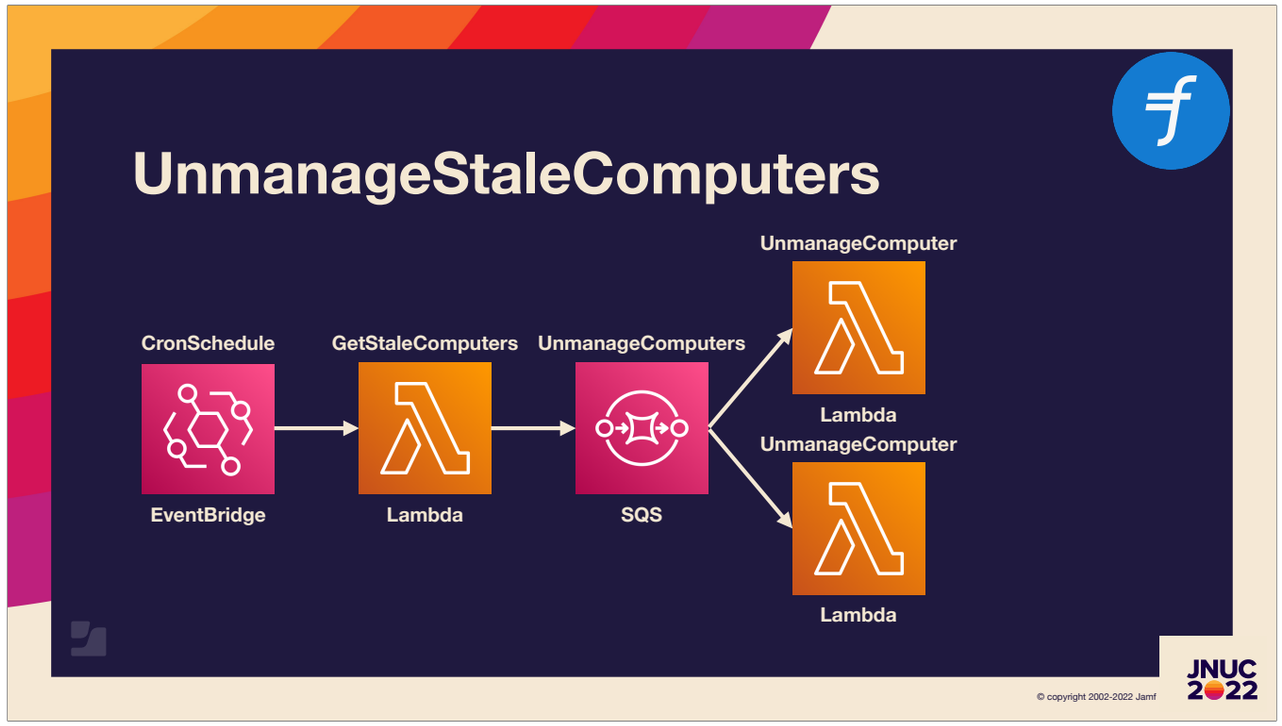
```
    },
    "message": {
        "title": "Message Title",
        "text": "Message Body"
    }
}
"""

import json
import logging
import requests


def lambda_handler(event, context):
    for record in event["Records"]:
        message_body = json.loads(record["body"])

        url = message_body["webhook"]["url"]
        msg = json.dumps(message_body["message"])

        try:
            requests.post(url, data=msg)
        except requests.exceptions.RequestException as e:
            LOGGER.error(e)
            return None


logging.basicConfig(format='%(levelname)s: %(asctime)s: %(message)s')
LOGGER = logging.getLogger(__name__)
LOGGER.setLevel(logging.INFO)

```


There's not much to the script. It takes a group of queued messages as input, loops over each message and sends it to the designated web hook url. Each Teams channel uses a different URL, so it is part of the message. This makes the script more generic and means that we don't have to duplicate it for each channel we want to define.

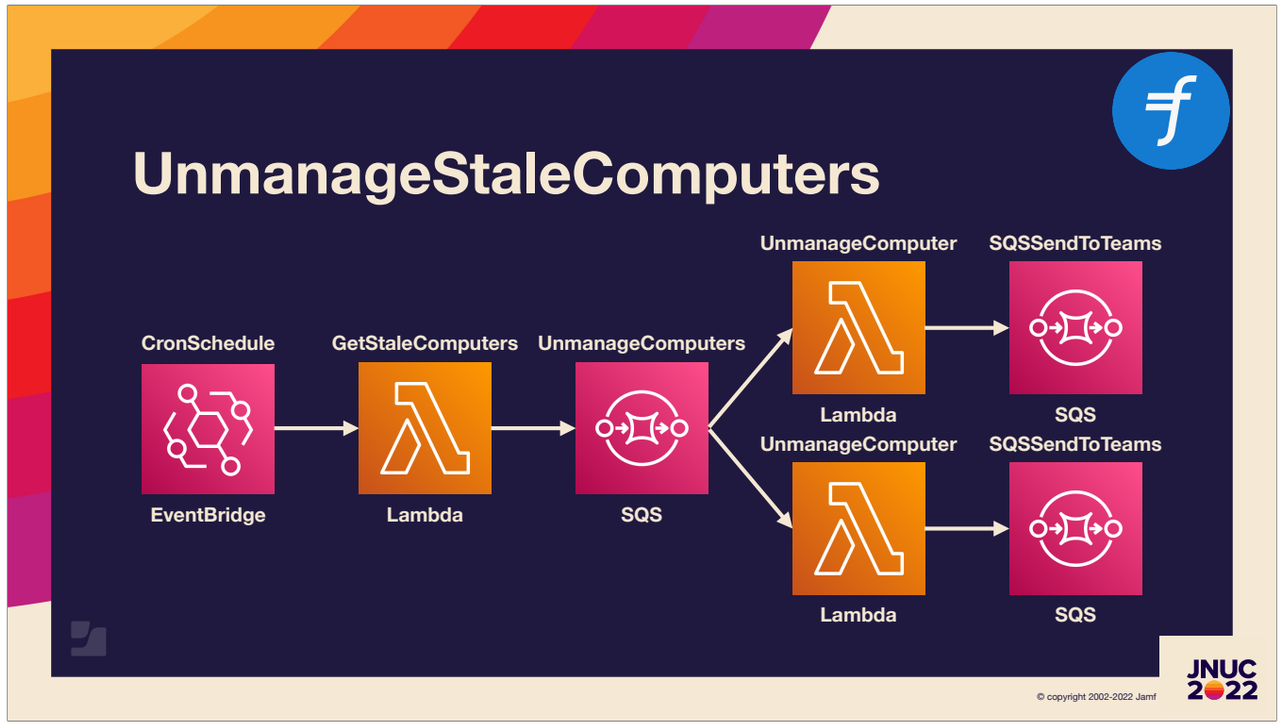
Note that there is a function named lambda_handler in here. This matches the template definition and will therefore be the function called in the script. We can add more functions without worrying about the starting point.



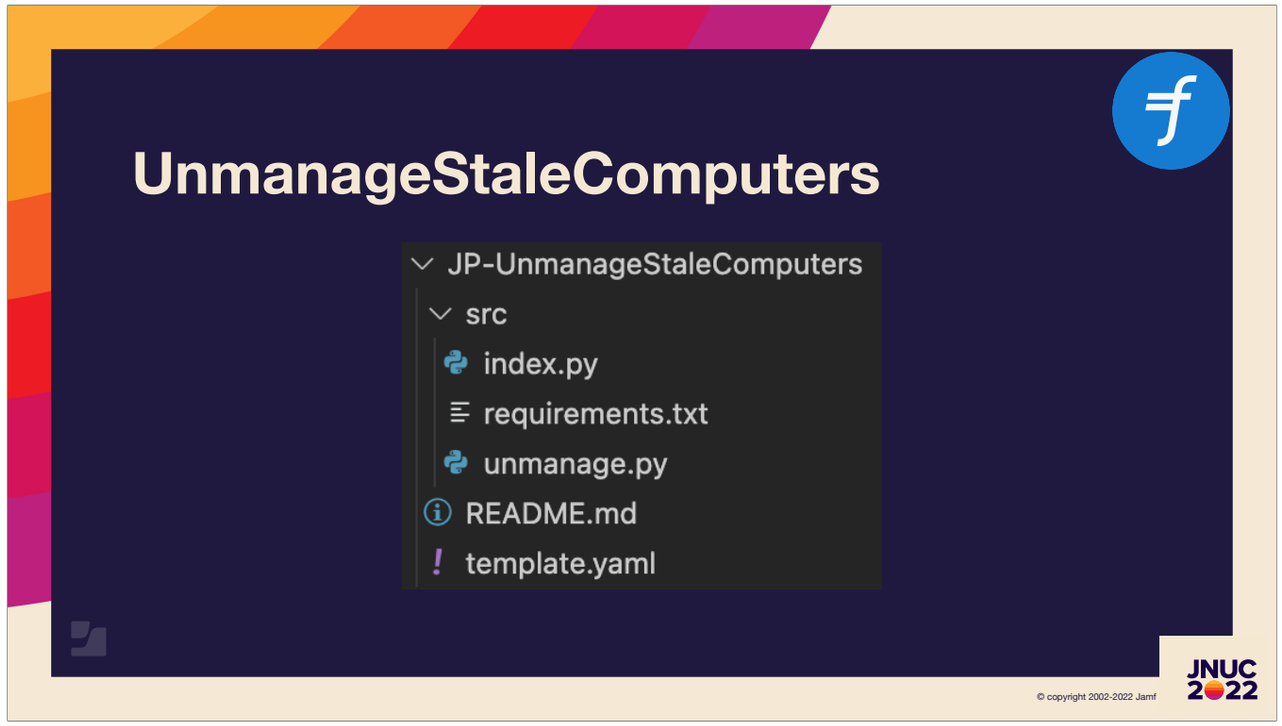
Our next example is a little more complex. It is triggered on a schedule and gets a list of computers from a Jamf Pro Advanced Search. It takes that list of devices and puts each of them into a queue which triggers the other Lambda to unmanage the computer. You can think of this as a one-to-many relationship.



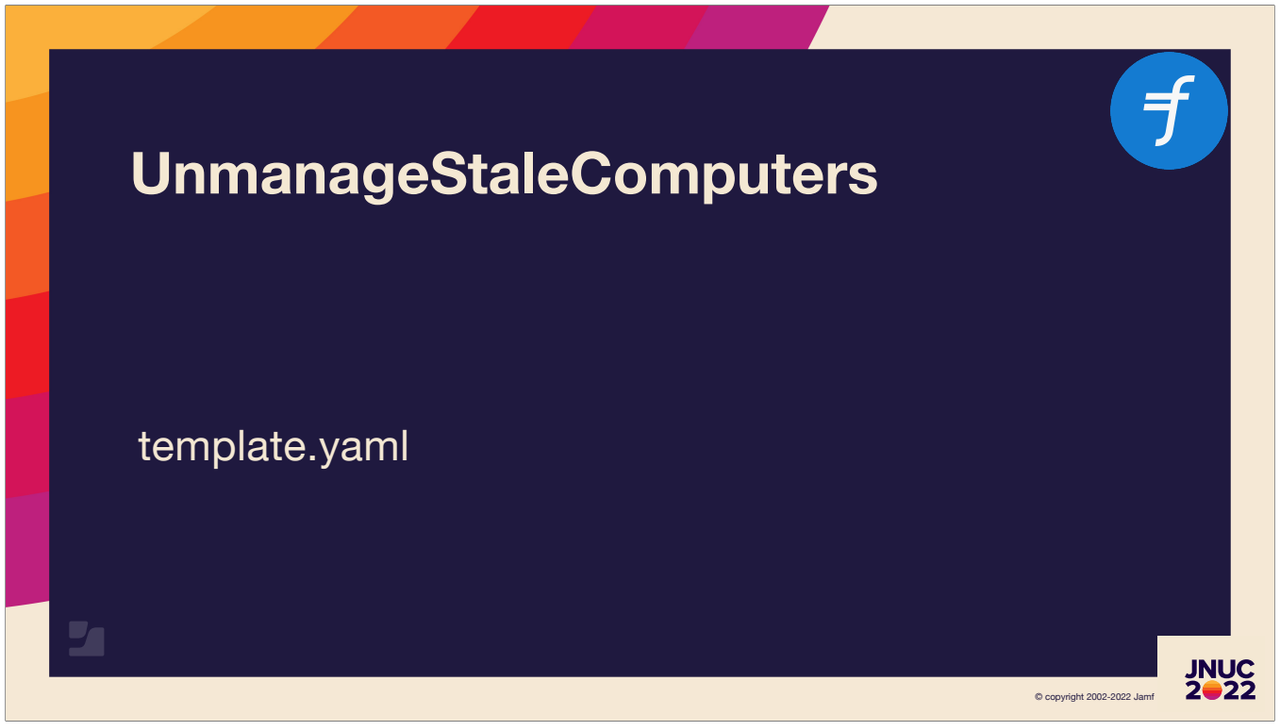
You can think of this as a one-to-many relationship. In fact, when that Lambda completes, it sends a message to the SendToTeams queue.



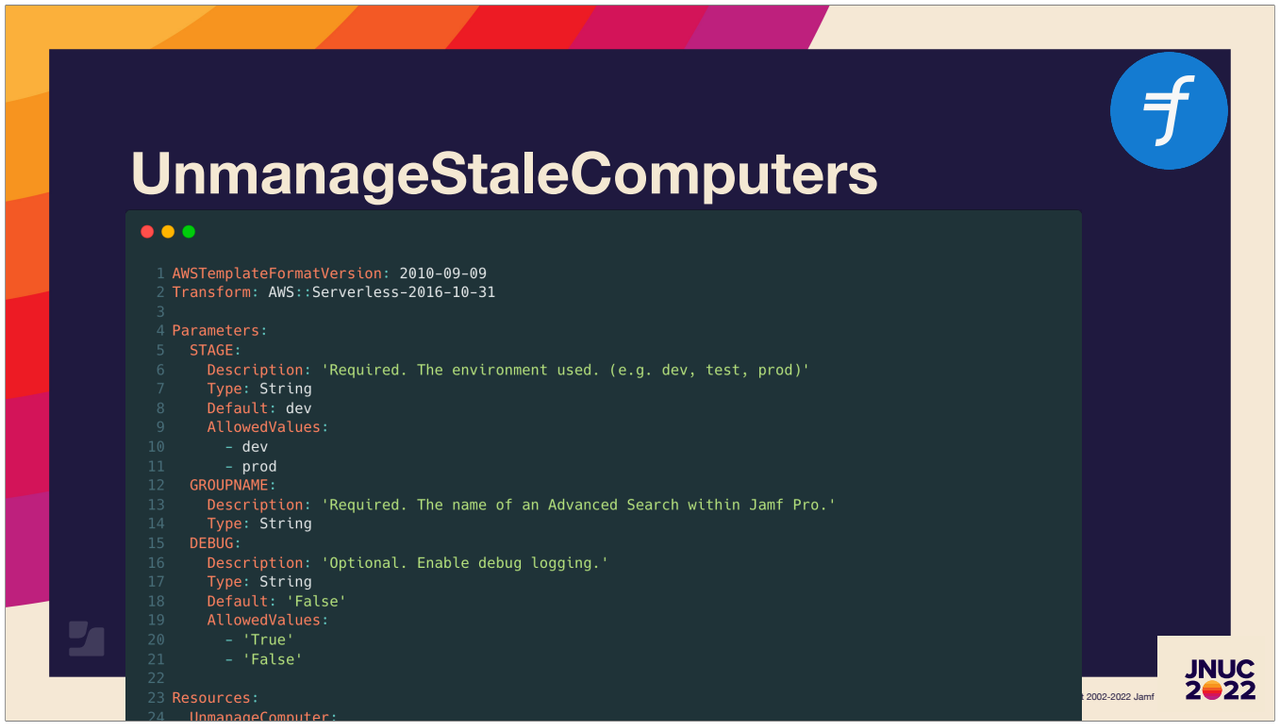
In fact, when that Lambda completes, it sends a message to the SendToTeams queue. As you can see, you can get pretty complex pretty quick. The cool thing is that the structure of this stack isn't much more complex than the first.



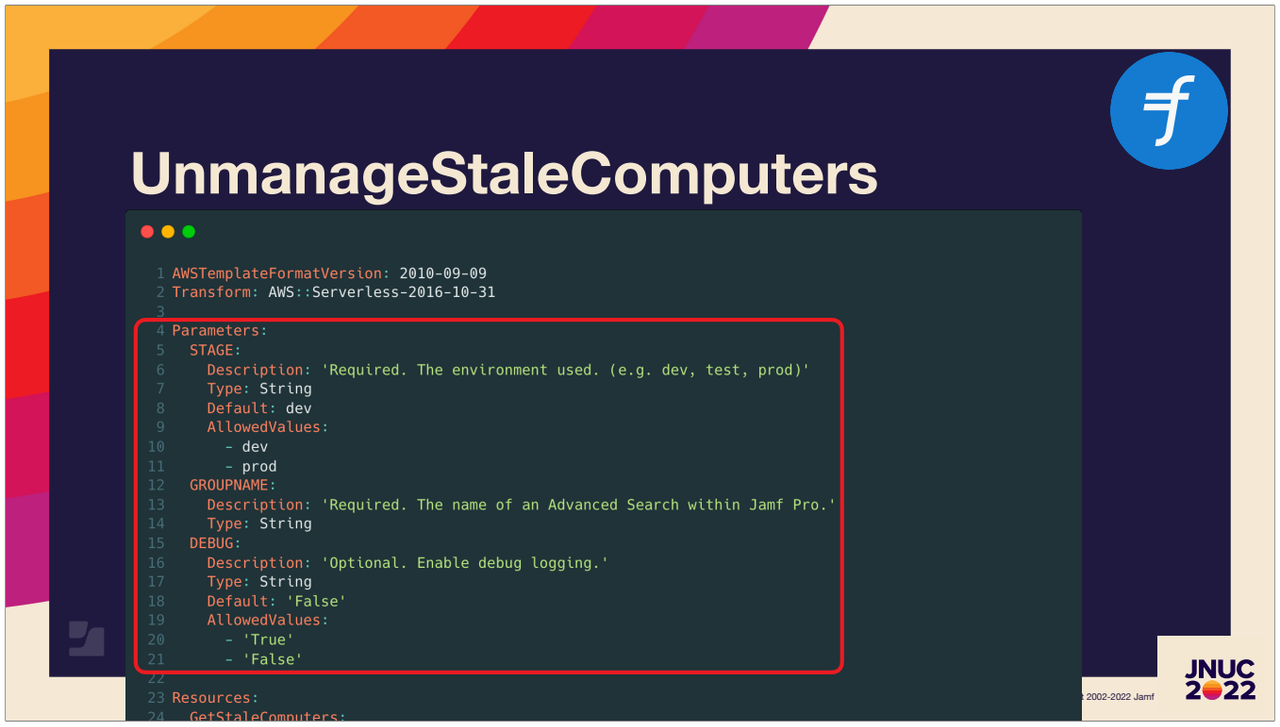
As you can see, we have an extra python file inside the source directory, but other than that, this project looks the same. Let's dig into the template again. I'm going to pick up the pace a bit but I'll still highlight some key differences.



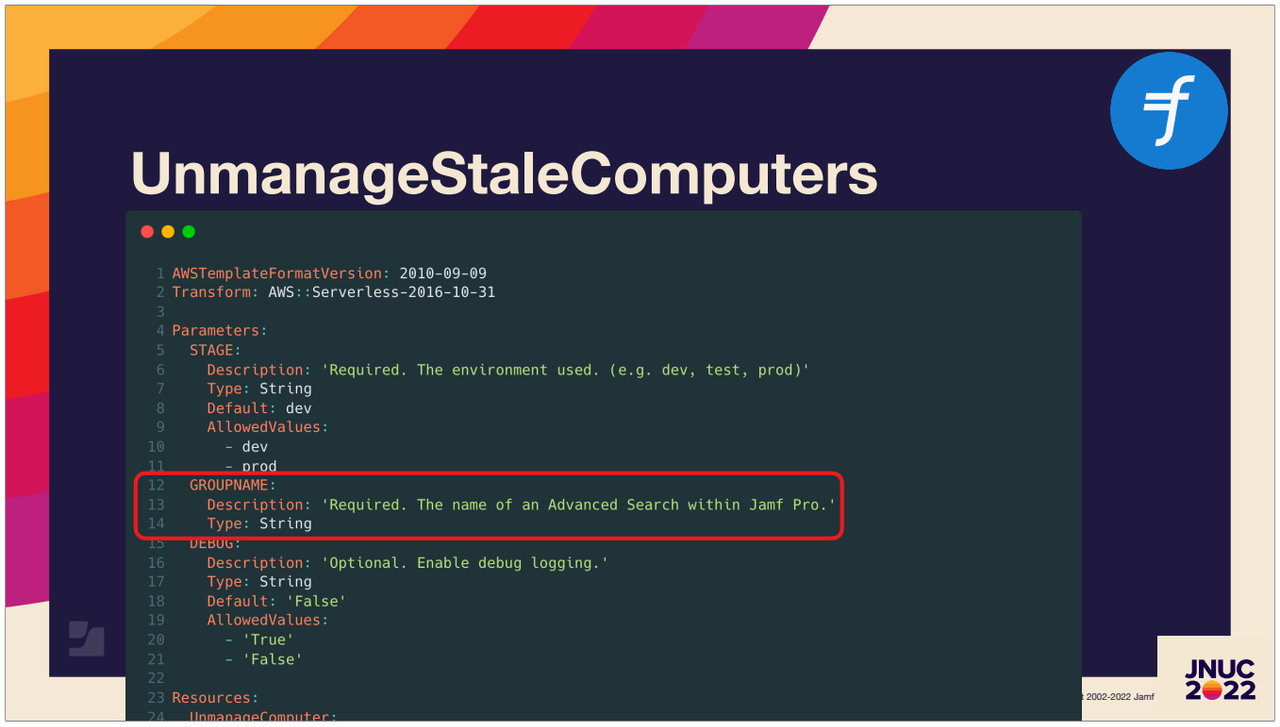
Let's start with the template again.



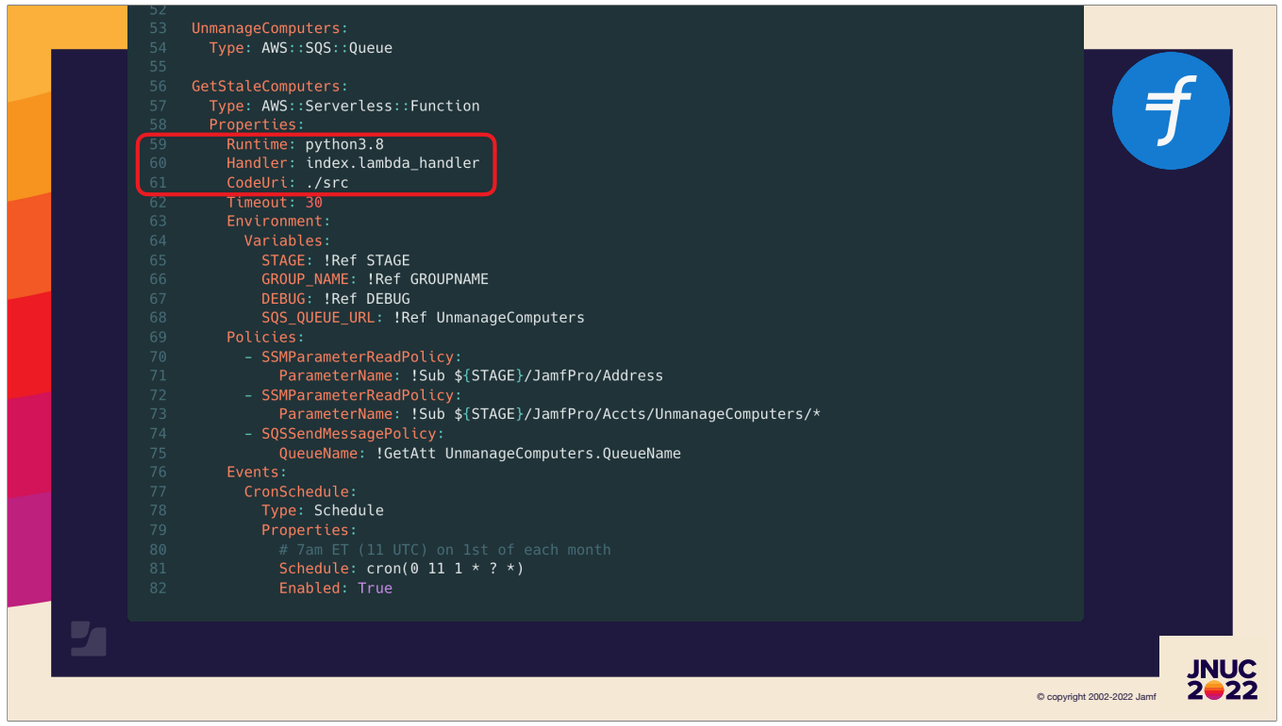
Here's the template file.



Right away, you'll see there is a new *Parameters* section. This is a way for you to add variables into your templates. In this case, GroupName takes the string name of an Advanced Search from Jamf Pro. This lets Jamf Pro do all the heavy lifting with its inventory without adding the overhead of a Smart Group.



Right away, you'll see there is a new *Parameters* section. This is a way for you to add variables into your templates. In this case, GroupName takes the string name of an Advanced Search from Jamf Pro. This lets Jamf Pro do all the heavy lifting with its inventory without adding the overhead of a Smart Group.



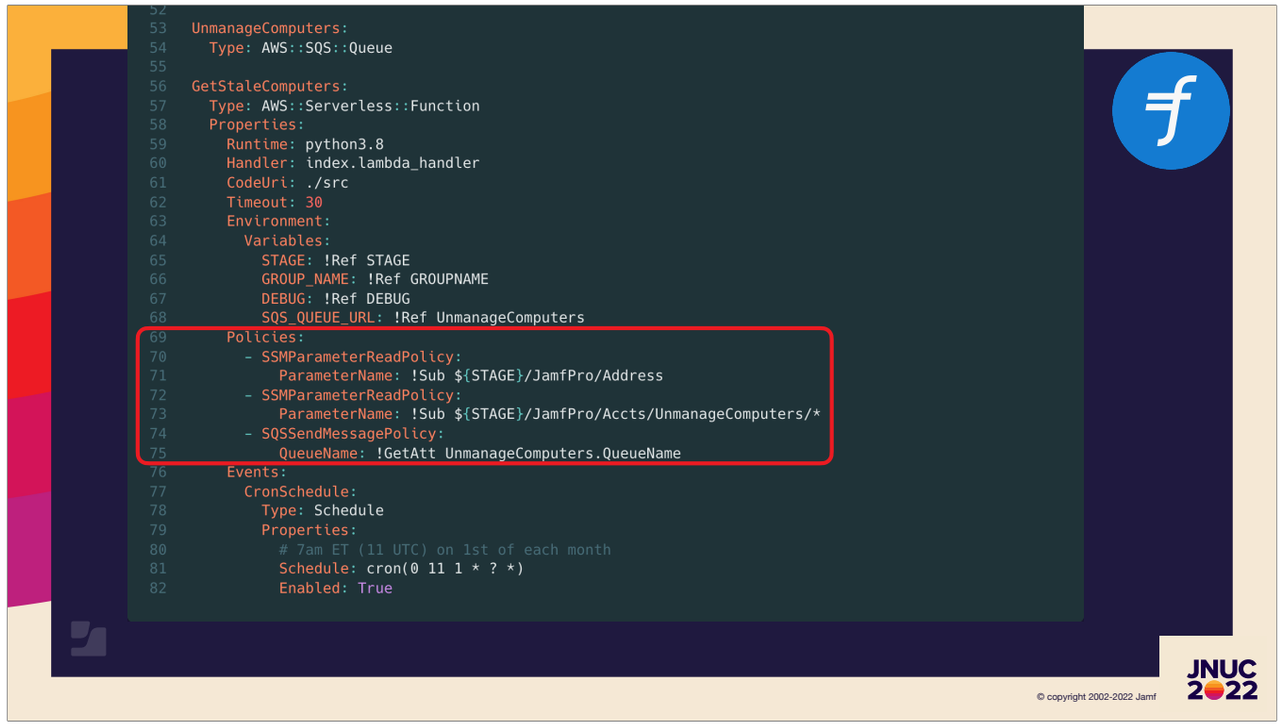
You'll notice that this section is the same as the last example. It will look in the source directory for a file called index, then call the lambda_handler function in a python 3.8 runtime.

```yaml
52
53  UnmanageComputers:
54    Type: AWS::SQS::Queue
55
56  GetStaleComputers:
57    Type: AWS::Serverless::Function
58    Properties:
59      Runtime: python3.8
60      Handler: index.lambda_handler
61      CodeUri: ./src
62      Timeout: 30
63      Environment:
64        Variables:
65          STAGE: !Ref STAGE
66          GROUP_NAME: !Ref GROUPNAME
67          DEBUG: !Ref DEBUG
68          SQS_QUEUE_URL: !Ref UnmanageComputers
69      Policies:
70        - SSMParameterReadPolicy:
71            ParameterName: !Sub ${STAGE}/JamfPro/Address
72        - SSMParameterReadPolicy:
73            ParameterName: !Sub ${STAGE}/JamfPro/Accts/UnmanageComputers/*
74        - SQSSendMessagePolicy:
75            QueueName: !GetAtt UnmanageComputers.QueueName
76      Events:
77        CronSchedule:
78          Type: Schedule
79          Properties:
80            # 7am ET (11 UTC) on 1st of each month
81            Schedule: cron(0 11 1 * ? *)
82            Enabled: True
```

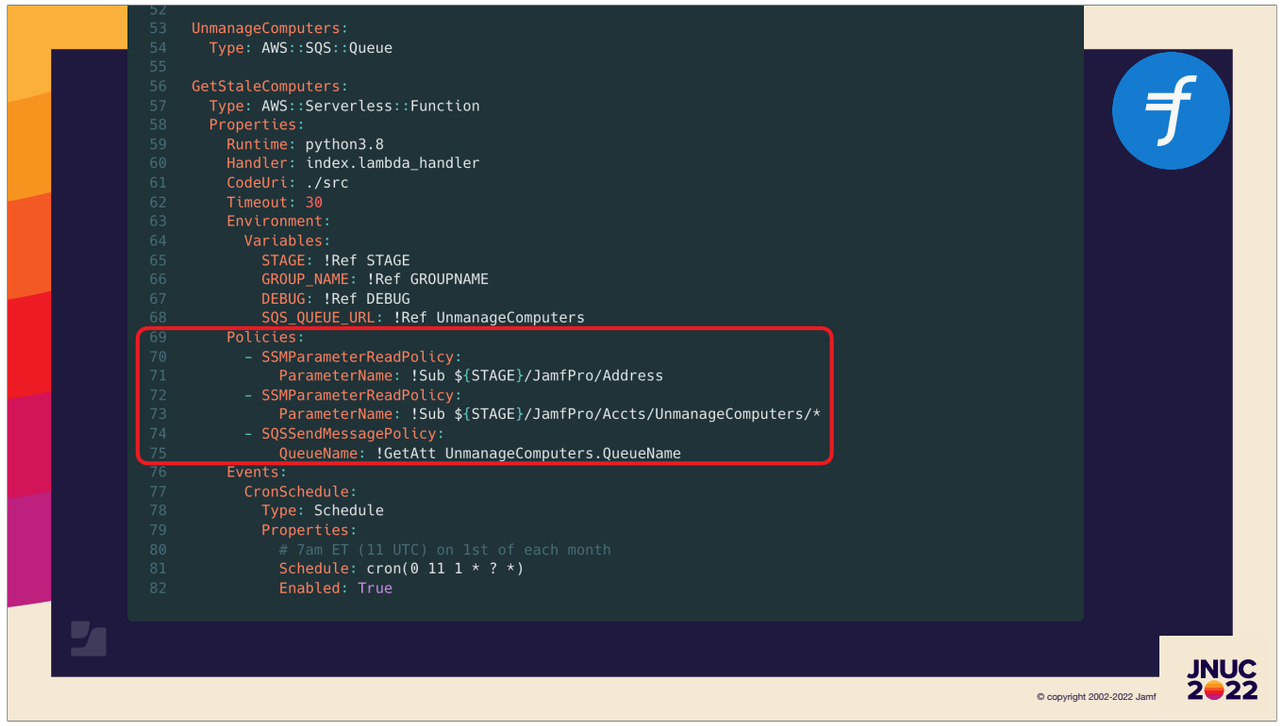

This is how you get the template parameters into a server less resource. The *Ref* function is a shortcut that references an element. Similarly, *Sub* is a function used to place a variable into a string.

```
52
53  UnmanageComputers:
54    Type: AWS::SQS::Queue
55
56  GetStaleComputers:
57    Type: AWS::Serverless::Function
58    Properties:
59      Runtime: python3.8
60      Handler: index.lambda_handler
61      CodeUri: ./src
62      Timeout: 30
63      Environment:
64        Variables:
65          STAGE: !Ref STAGE
66          GROUP_NAME: !Ref GROUPNAME
67          DEBUG: !Ref DEBUG
68          SQS_QUEUE_URL: !Ref UnmanageComputers
69      Policies:
70        - SSMParameterReadPolicy:
71            ParameterName: !Sub ${STAGE}/JamfPro/Address
72        - SSMParameterReadPolicy:
73            ParameterName: !Sub ${STAGE}/JamfPro/Accts/UnmanageComputers/*
74        - SQSSendMessagePolicy:
75            QueueName: !GetAtt UnmanageComputers.QueueName
76      Events:
77        CronSchedule:
78          Type: Schedule
79          Properties:
80            # 7am ET (11 UTC) on 1st of each month
81            Schedule: cron(0 11 1 * ? *)
82            Enabled: True
```


This is how you get the template parameters into a server less resource. The *Ref* function is a shortcut that references an element. Similarly, *Sub* is a function used to place a variable into a string.



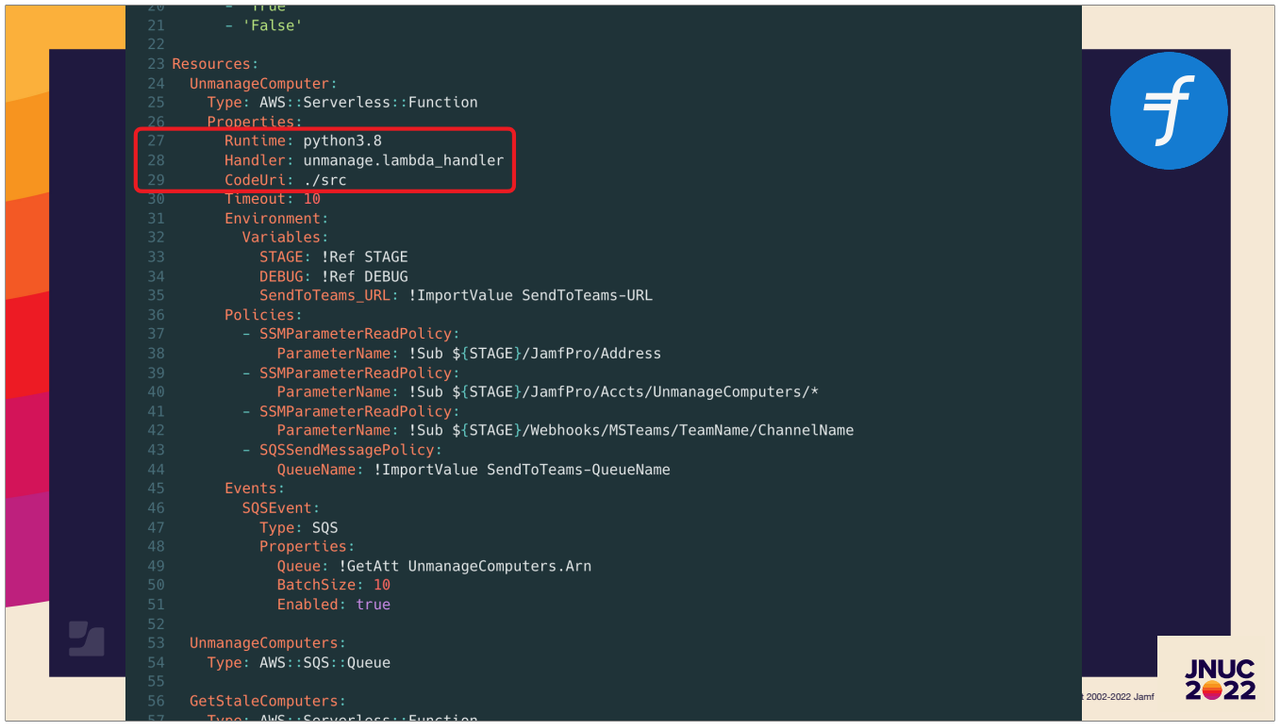
The Policies section configures IAM, or Identity and Access Management, policies. In this example, we are granting the Lambda the ability to read values from the Parameter Store as well as the ability to send a message to the SQS queue.



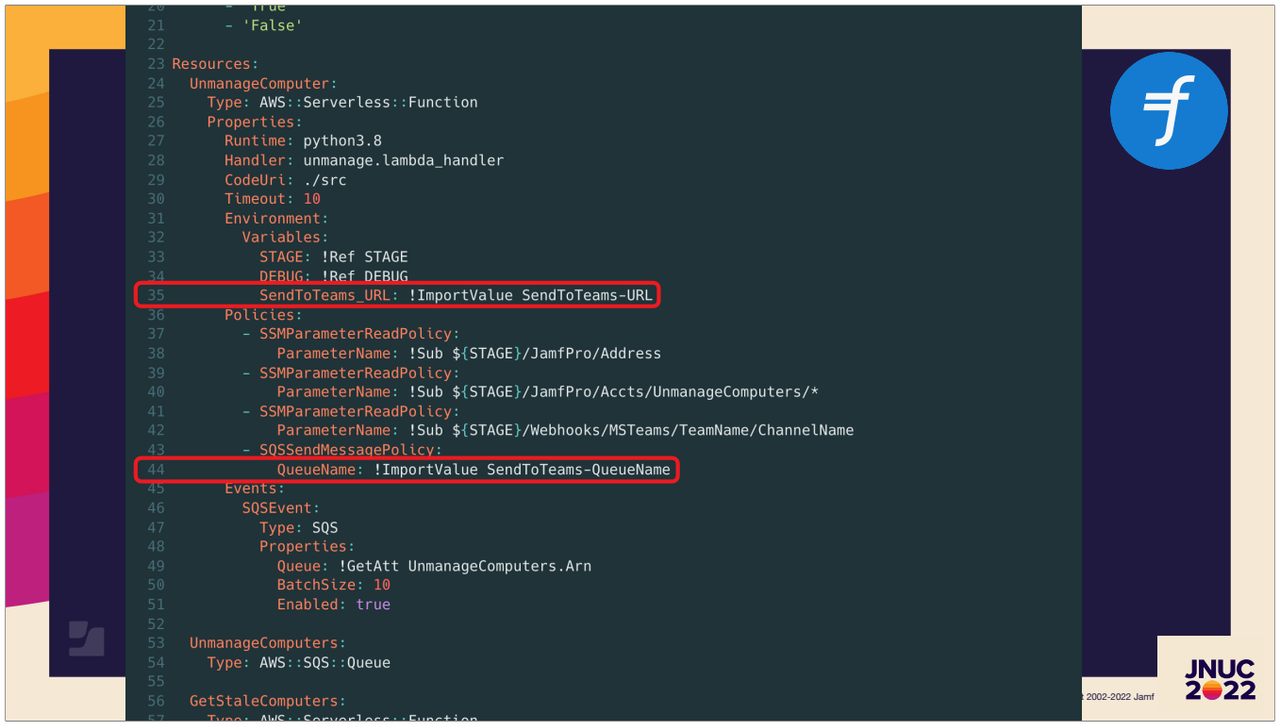
SecretsManager can also be used here instead of Parameter Store. They have very similar features. It just happens that when I wrote this stack, I used the Parameter Store instead. By the time you're seeing this, I may have both examples in the resources.

```
52
53  UnmanageComputers:
54    Type: AWS::SQS::Queue
55
56  GetStaleComputers:
57    Type: AWS::Serverless::Function
58    Properties:
59      Runtime: python3.8
60      Handler: index.lambda_handler
61      CodeUri: ./src
62      Timeout: 30
63      Environment:
64        Variables:
65          STAGE: !Ref STAGE
66          GROUP_NAME: !Ref GROUPNAME
67          DEBUG: !Ref DEBUG
68          SQS_QUEUE_URL: !Ref UnmanageComputers
69      Policies:
70        - SSMParameterReadPolicy:
71            ParameterName: !Sub ${STAGE}/JamfPro/Address
72        - SSMParameterReadPolicy:
73            ParameterName: !Sub ${STAGE}/JamfPro/Accts/UnmanageComputers/*
74        - SQSSendMessagePolicy:
75            QueueName: !GetAtt UnmanageComputers.QueueName
76      Events:
77        CronSchedule:
78          Type: Schedule
79          Properties:
80            # 7am ET (11 UTC) on 1st of each month
81            Schedule: cron(0 11 1 * ? *)
82            Enabled: True
```

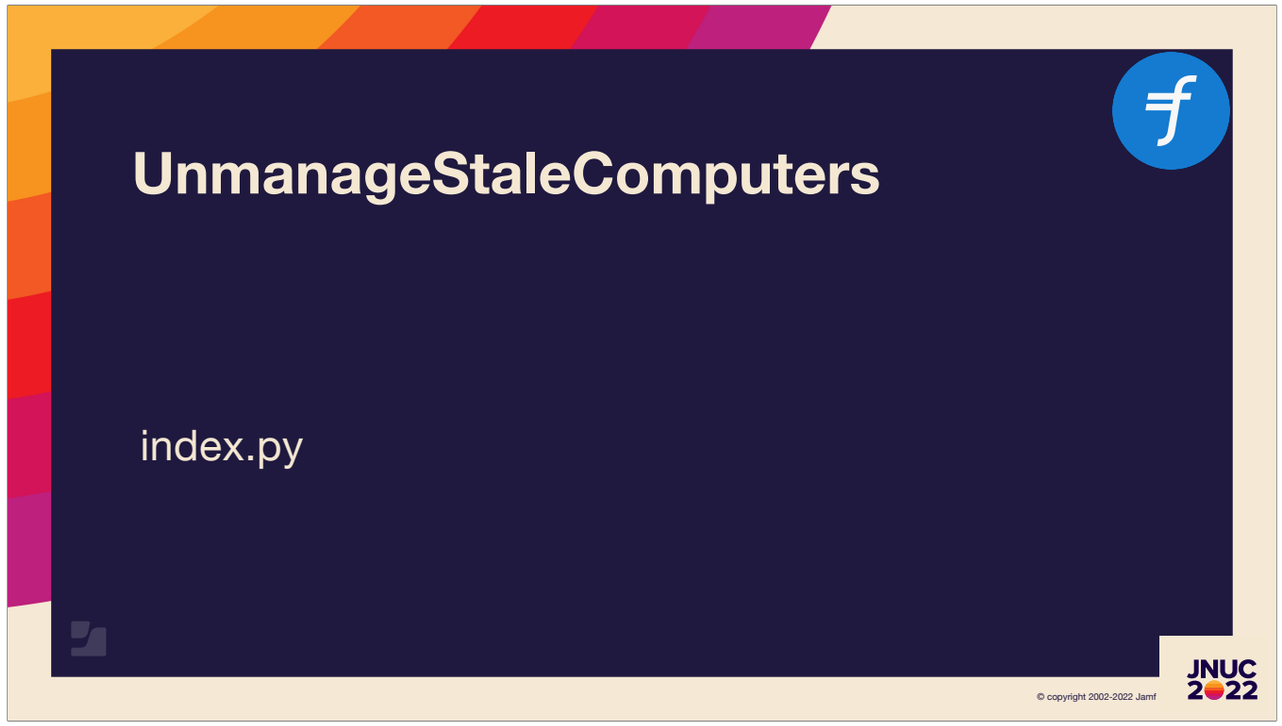

The previous Lambda example used an SQS event to trigger. This one uses a Schedule event. There are two different ways to describe the schedule, *cron* and *rate*. I'll have a link at the end for more information on all of the options.



This Lambda is slightly different. It is calling a function in the unmanaged python file instead of index.



In the first example, we exposed a couple values with the Output property. To access these values, you use the *ImportValue* function. The script needs the URL to send the messages, but we also need the Name to grant the permissions to the Lambda to send messages.



Moving on to index.py.

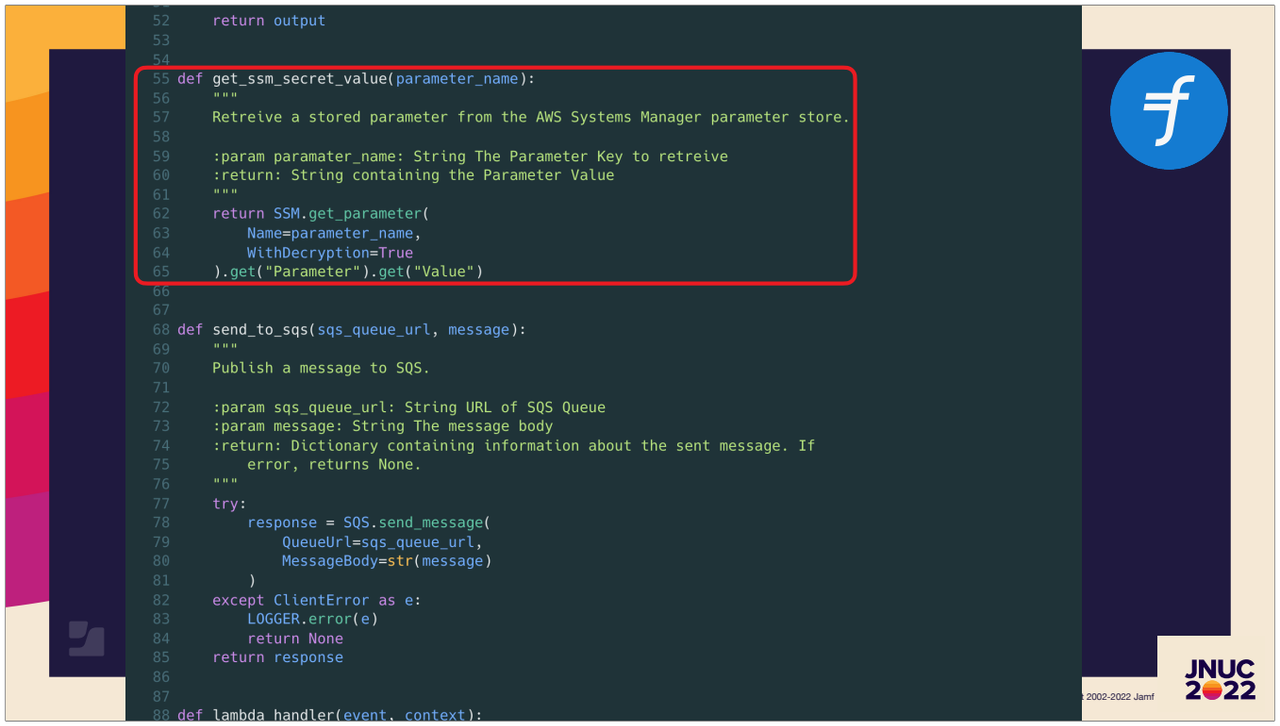
UnmanageStaleComputers

```python
"""
Given a Jamf Pro Advanced Search name, generate a list of computer IDs
and add them to an AWS SQS queue. The Advanced Search should contain
computers that have not checked in within a designated timeframe.
"""

import os
import json
import urllib
import requests
import logging
import boto3
from botocore.exceptions import ClientError
from http.client import HTTPConnection


def api_get_advancedcomputersearch_by_name(name):
    """
    Retrieve saved search data given a group name.

    :param name: String The name of a Jamf Pro Advanced Search
    :return: JSON Object containing information about the request's get
        action. If error, returns None.
    """
```


JNUC 2022

```
        return

    computer_ids = get_computer_ids_from_json(json_result)
    if computer_ids is None:
        LOGGER.error('Failed to retreive Computer IDs from the returned JSON.')
        return
    elif computer_ids == []:
        LOGGER.info('No Computer IDs returned. The search appears to be empty.')
        return

    for computer_id in computer_ids:
        LOGGER.info(f'Sending Computer ID \'{computer_id}\' to the queue to be unmanaged.')
        send_to_sqs(sqs_queue_url, computer_id)


SQS = boto3.client('sqs')
SSM = boto3.client('ssm')
STAGE = os.getenv("STAGE", "dev").lower()
DEBUG = os.getenv("DEBUG", "False").lower()

logging.basicConfig(format='%(levelname)s: %(asctime)s: %(message)s')
LOGGER = logging.getLogger(__name__)
if DEBUG == 'true':
    LOGGER.setLevel(logging.DEBUG)
    HTTPConnection.debuglevel = 1
else:
    LOGGER.setLevel(logging.INFO)

API_URL = get_ssm_secret_value(f'/{STAGE}/JamfPro/Address')
API_USER = get_ssm_secret_value(f'/{STAGE}/JamfPro/Accts/UnmanageComputers/Username')
API_PASS = get_ssm_secret_value(f'/{STAGE}/JamfPro/Accts/UnmanageComputers/Password')
```


One of the first things this script does is pull secrets from the Parameter Store. Like I said earlier, this can and probably should be done with SecretsManager instead. The values you see here are the stored names of the parameters in AWS.

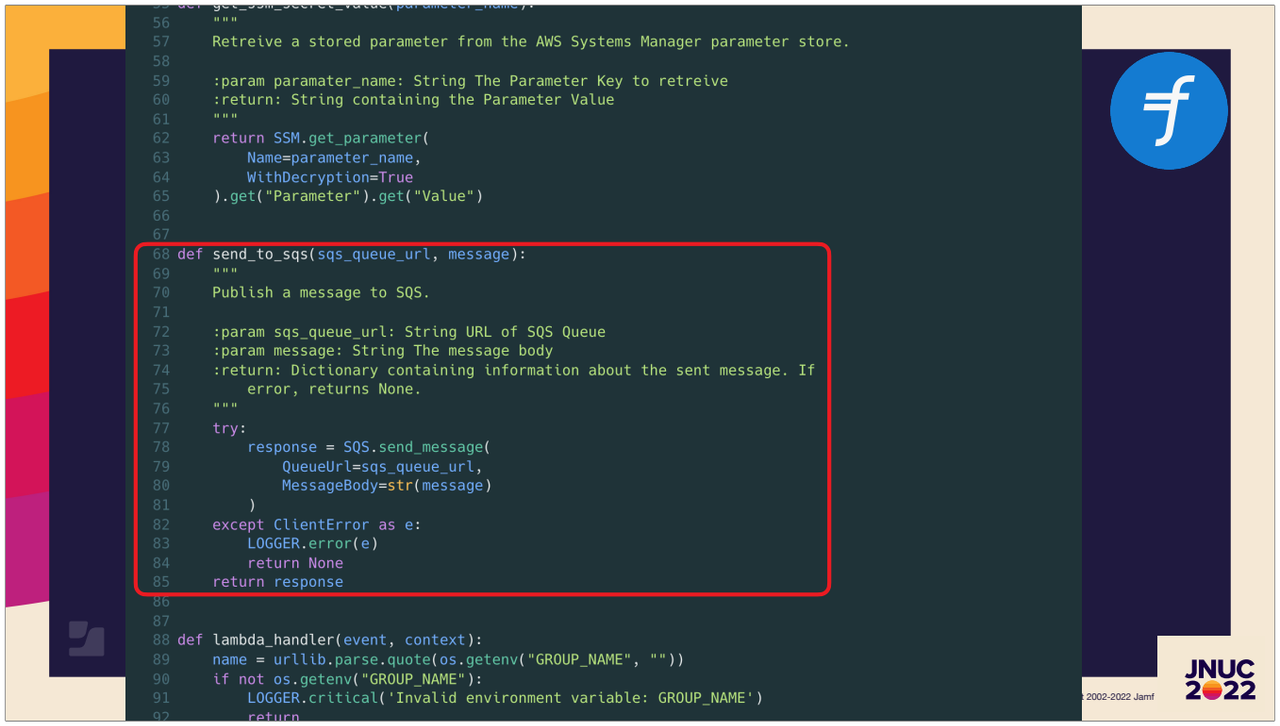


AWS provides a python module called *boto3* that makes interacting with service quite easy. Here we have a helper function that pulls a value given a parameter name. If we got our policies correct in the template, these integrations should just work.

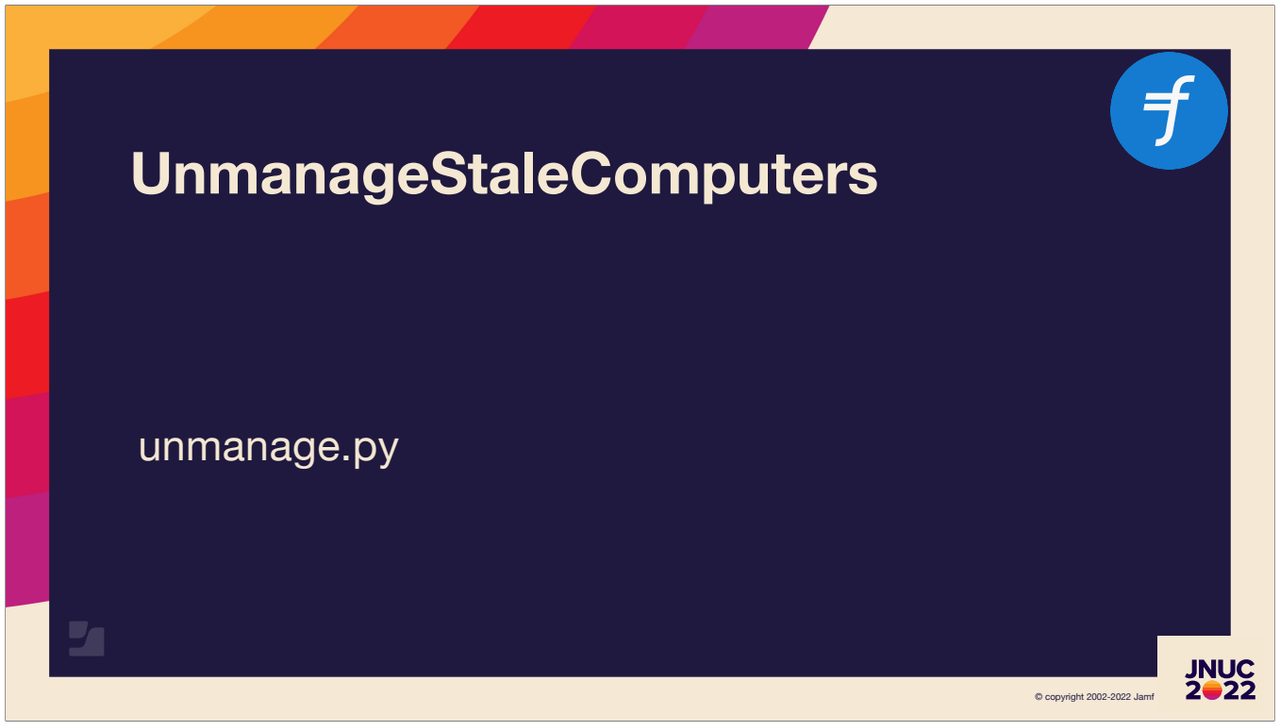
After doing some error checking, it queries the Jamf Pro Classic API and then parses those results. It then loops over each ID and sends a message to the defined SQS queue.

After doing some error checking, it queries the Jamf Pro Classic API and then parses those results. It then loops over each ID and sends a message to the defined SQS queue. Let's take a look at those functions as well.

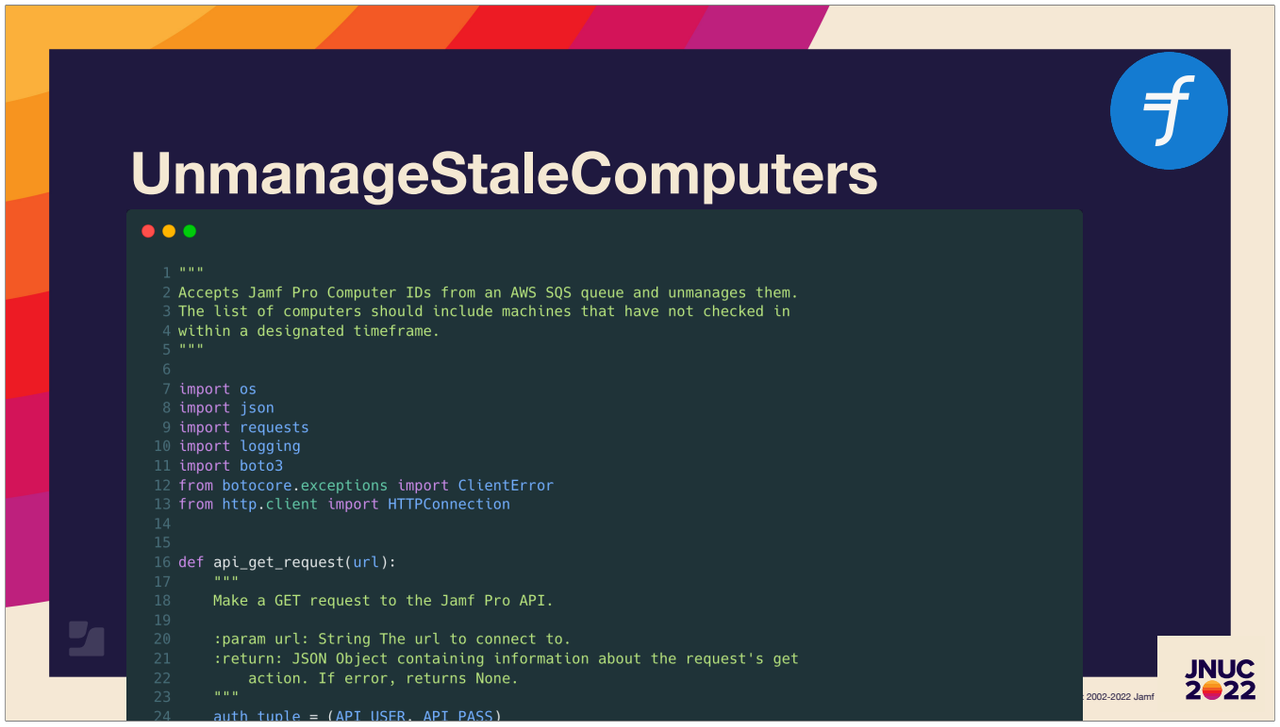
The Jamf Pro Classic API integration relies on the values from the secrets to make the connection, but this way keeps those values secure. Otherwise, this function is pretty standard stuff when you are working with a vendor API. Set some headers, define your URL, and invoke the requests module.



Similarly, when we have a Computer ID that needs to be unmanaged, we just send it as a message to SQS and that script takes over.



Now, let's check out unmanage.py.

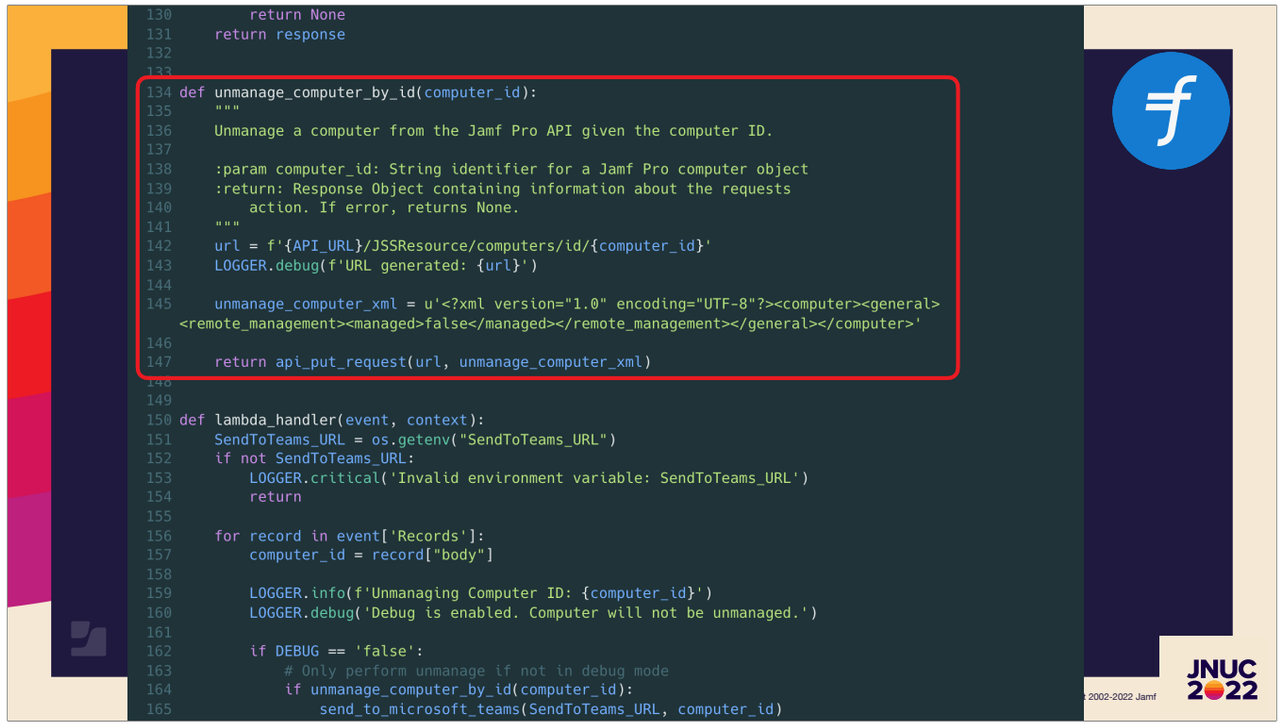


This file is mostly the same stuff, but there a couple helper functions to facilitate API GET and PUT requests. This is mostly done because of the error handling and for consistent returns. It means that most of the rest of the script is formatting data and calling those helper functions.

```
    :return: Dictionary containing each cookie from the PUT request. If error, returns None.
    """
    auth_tuple = (API_USER, API_PASS)

    try:
        r = requests.put(url, auth=auth_tuple, data=data)
        LOGGER.debug(str(r.content))
    except requests.exceptions.RequestException as e:
        LOGGER.error(e)
        return None

    return r.cookies


def get_computer_name_by_id(computer_id):
    """
    Get a computer record from a Jamf Pro API given the computer ID.

    :param computer_id: String identifier for a Jamf Pro computer object
    :return: Response Object containing information about the requests get
        action. If error, returns None.
    """
    url = f'{API_URL}/JSSResource/computers/id/{computer_id}/subset/General'
    LOGGER.debug(f'URL generated: {url}')

    computer_record = api_get_request(url)

    return computer_record["computer"]["general"]["name"]


def get_ssm_secret_value(parameter_name):
    """
    Retreive a stored parameter from the AWS Systems Manager parameter store.

    :param paramater_name: String The Parameter Key to retreive
    :return: String containing the Parameter Value
    """
```


JNUC 2022

This file is mostly the same stuff, but there a couple helper functions to facilitate API GET and PUT requests. This is mostly done because of the error handling and for consistent returns. It means that most of the rest of the script is formatting data and calling those helper functions.

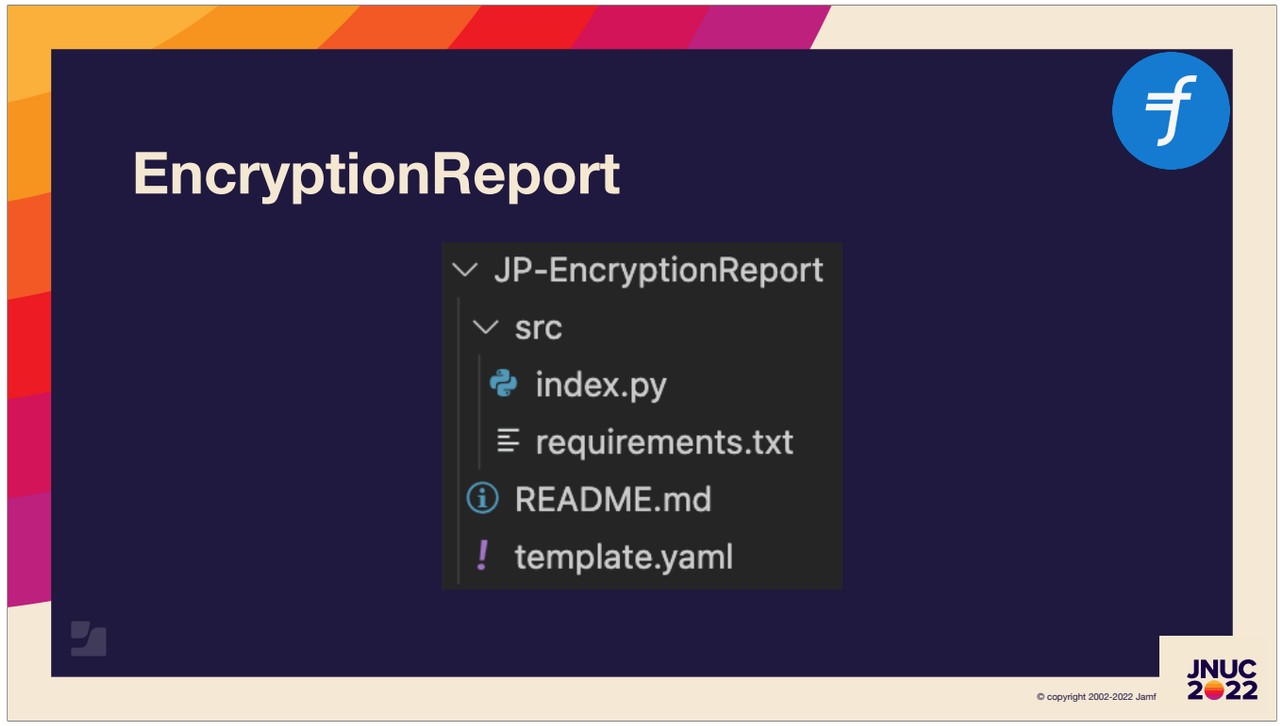


The unmanage_computer_by_id function defines the XML necessary to unmanaged a computer in Jamf Pro. It takes a Computer ID as input and sends that XML to the API. If it fails, it doesn't send the message to Teams.

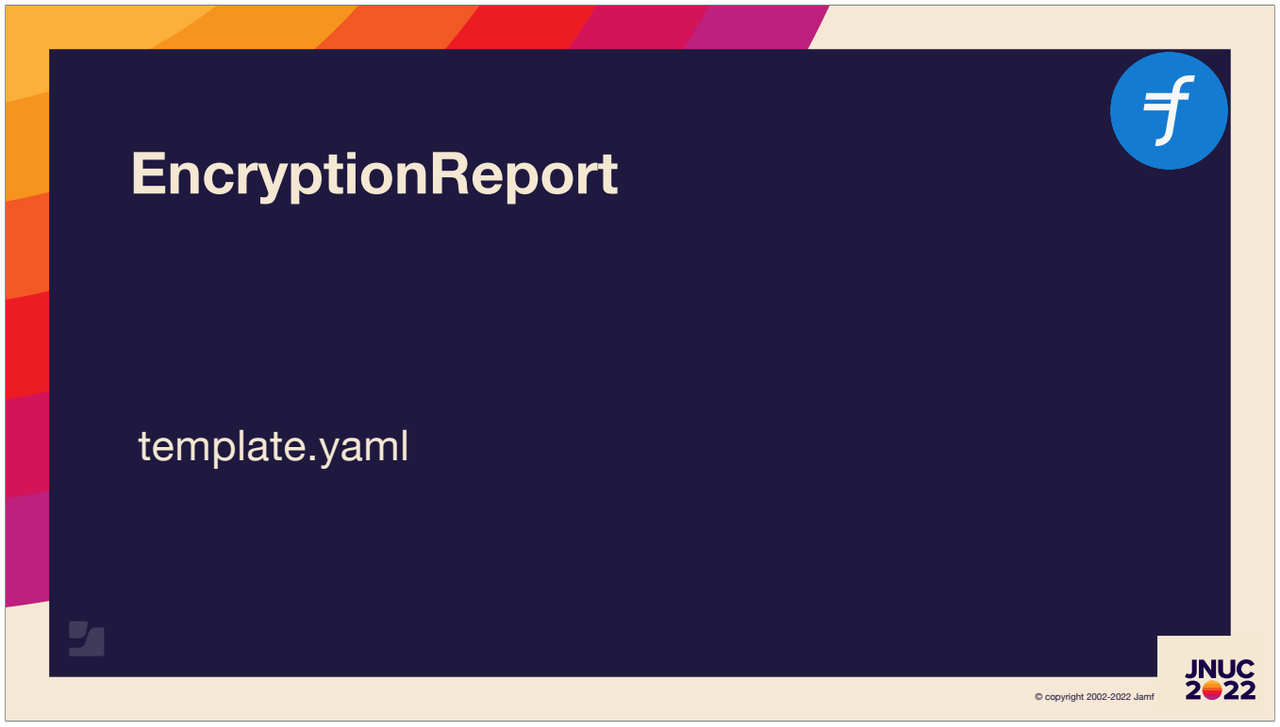
The unmanage_computer_by_id function defines the XML necessary to unmanaged a computer in Jamf Pro. It takes a Computer ID as input and sends that XML to the API. If it fails, it doesn't send the message to Teams.

Have you ever wanted to run a report in Jamf Pro, but it wasn't quite able to do what you needed? I had a situation where I needed to share a report with my Security team that was split by Site. In Jamf Pro, site isn't something you can add to the scope, so I had to script a solution.

I could have just run the script on my local computer, but then no one else would ever know it existed, so I made a serverless stack to run it for me. This one is a little different from the others in that it contains an SNS queue.



This should start looking pretty familiar by now. There are certainly times where you'll need more files, but most MacAdmin tasks tend to be small, so this is what most of my stacks tend to look like.

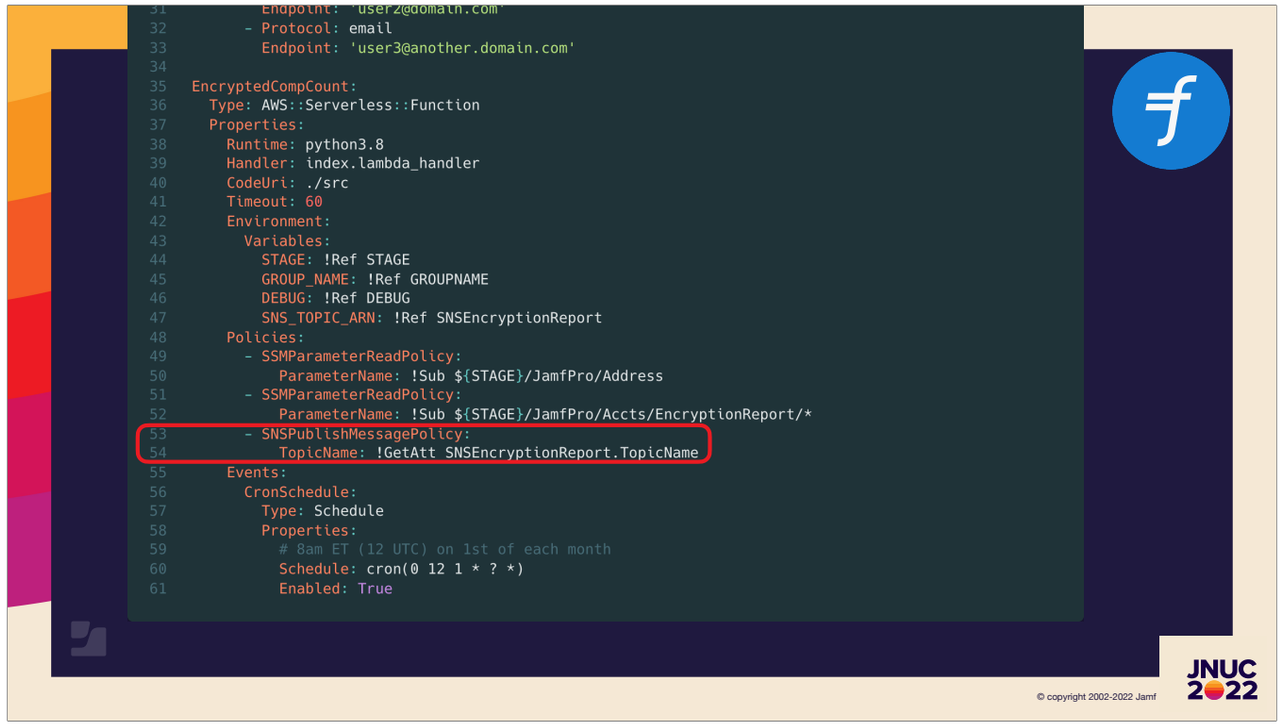


Let's start with the template again.

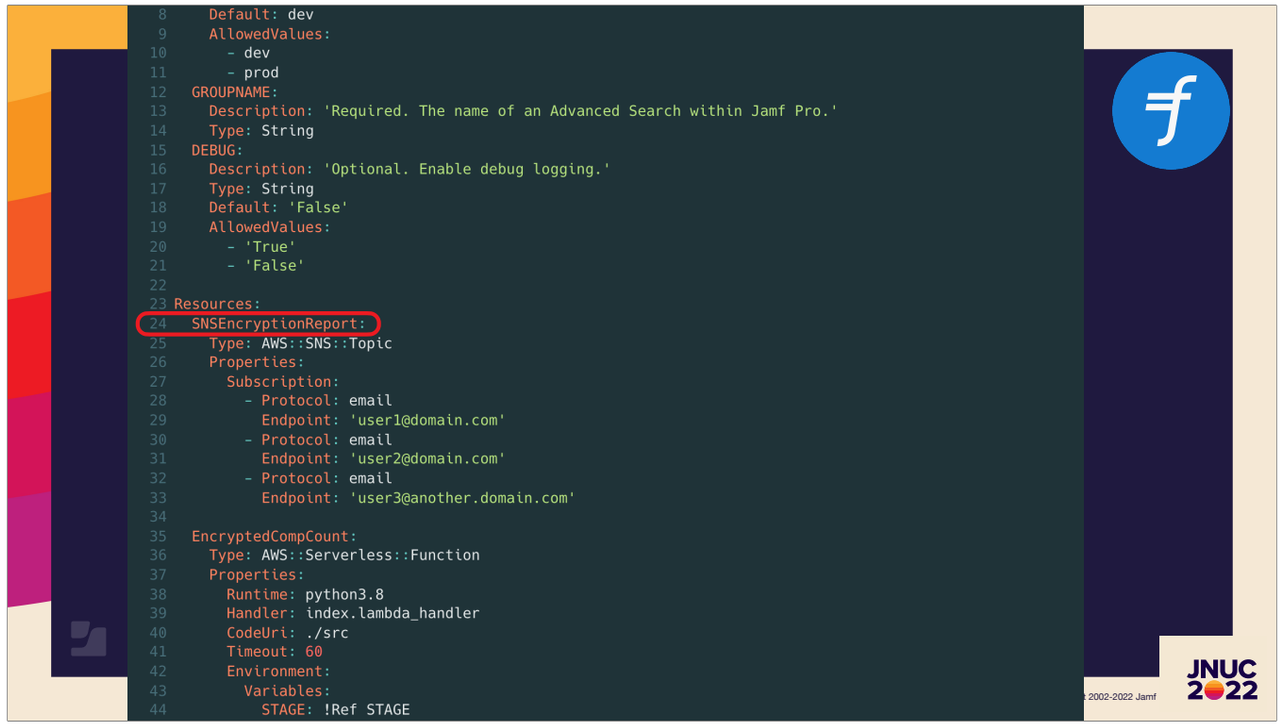
Here's the template file.

```
        Endpoint: 'user2@domain.com'
      - Protocol: email
        Endpoint: 'user3@another.domain.com'

EncryptedCompCount:
  Type: AWS::Serverless::Function
  Properties:
    Runtime: python3.8
    Handler: index.lambda_handler
    CodeUri: ./src
    Timeout: 60
    Environment:
      Variables:
        STAGE: !Ref STAGE
        GROUP_NAME: !Ref GROUPNAME
        DEBUG: !Ref DEBUG
        SNS_TOPIC_ARN: !Ref SNSEncryptionReport
    Policies:
      - SSMParameterReadPolicy:
          ParameterName: !Sub ${STAGE}/JamfPro/Address
      - SSMParameterReadPolicy:
          ParameterName: !Sub ${STAGE}/JamfPro/Accts/EncryptionReport/*
      - SNSPublishMessagePolicy:
          TopicName: !GetAtt SNSEncryptionReport.TopicName
    Events:
      CronSchedule:
        Type: Schedule
        Properties:
          # 8am ET (12 UTC) on 1st of each month
          Schedule: cron(0 12 1 * ? *)
          Enabled: True
```

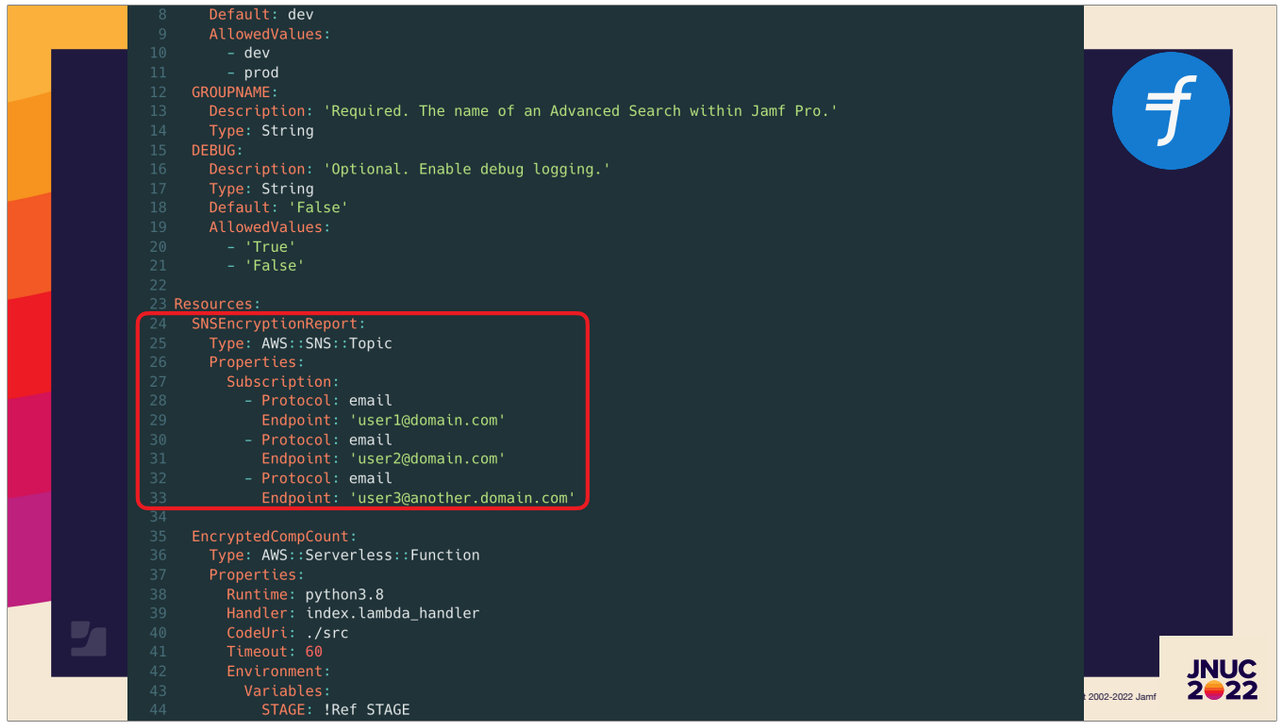

You'll notice that it's pretty short. Most everything in here has been covered in previous examples, except the SNS portions.



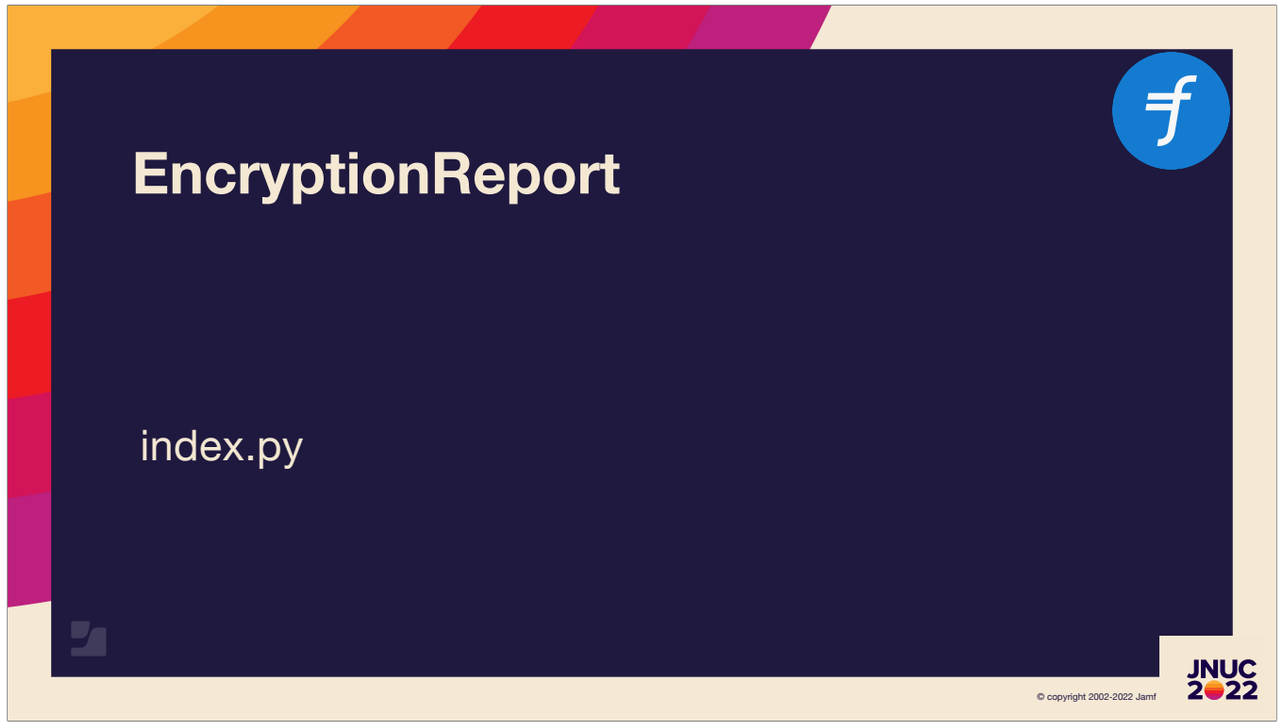
This little section within Policies makes sure that the Lambda can send messages to the SNS service. I sometimes find it hard to keep track of which attributes are needed for different policies. Unfortunately, I don't have any tips on remembering them either.



Just make sure it matches this.



You can create an empty SNS Topic like you do for SQS, but those messages won't go anywhere unless you also add subscriptions. They could be emails as you see here, http addresses, an SQS queue, a Lambda, and more. You can mix and match according to your needs.



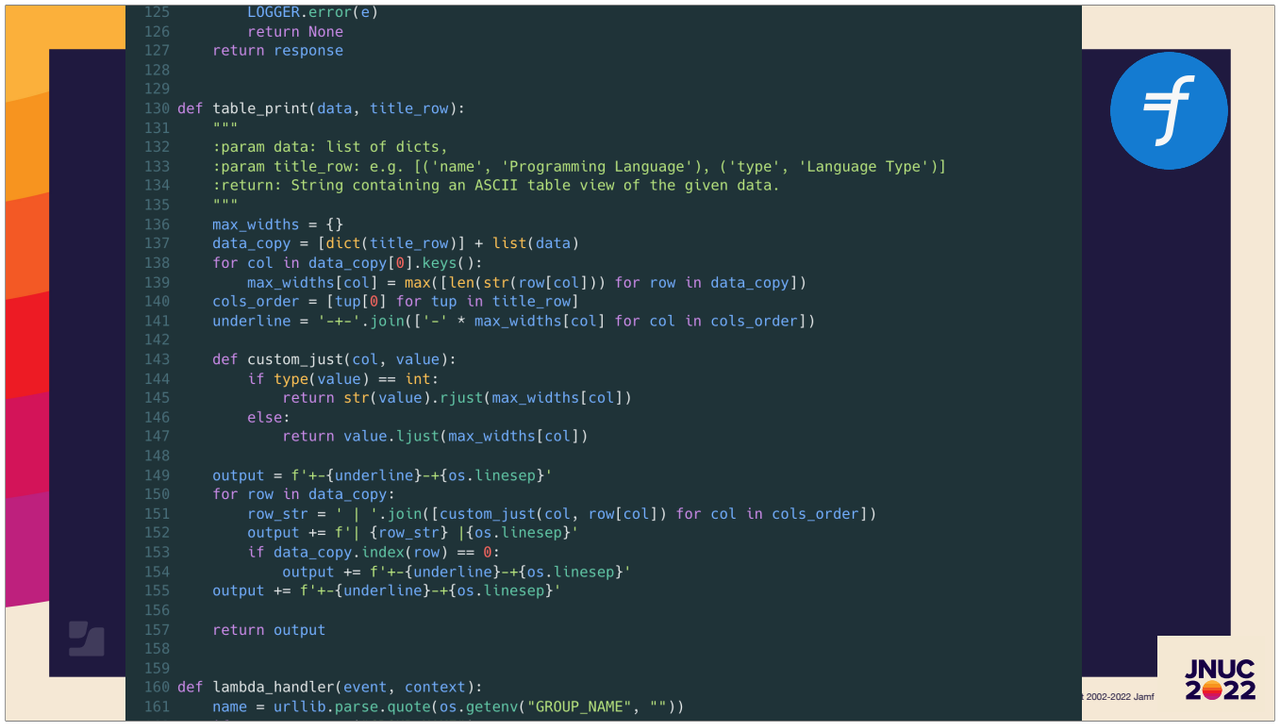
Moving on to index.py.

This script is pretty cool. Like I said earlier, Jamf Pro couldn't do what I wanted and split it by sites, so I created an Advanced Search that had all of the data I needed except the site.

So this script gets all of the sites from the Jamf Pro Classic API. It does it in XML, because sometimes you need to use XML and I wanted to show some sample code doing just that.

```python

def get_encrypted_count_by_site_id(site_id):
    """
    Gets the number of encrypted devices within a given site.

    :param site_id: Integer The ID of a Jamf Pro site.
    :return: Integer containing the number of computers that are encrypted. If error, returns None.
    """
    change_site_xml = f'<advanced_computer_search><site><id>{site_id}</id></site>
</advanced_computer_search>'

    auth_tuple = (API_USER, API_PASS)
    name = urllib.parse.quote(os.getenv("GROUP_NAME", ""))

    url = f'{API_URL}/JSSResource/advancedcomputersearches/name/{name}'
    LOGGER.debug(f'URL generated: {url}')

    r = api_put_request(url, change_site_xml)
    cookies = dict(APBALANCEID=r.get('APBALANCEID'))

    try:
        xml = requests.get(url, auth=auth_tuple, cookies=cookies).content
    except requests.exceptions.RequestException as e:
        LOGGER.error(e)
        return None

    try:
        encrypted_count = ElementTree.fromstring(xml).findtext('computers/size')
    except ElementTree.ParseError as e:
        LOGGER.debug('Value of the \'xml\' parameter: '+str(xml))
        LOGGER.error(e)
        return None

    return encrypted_count


```


It then loops through each site and calls this function. It changes the Site ID on a specific Advanced Search, gets the number of computers returned, and then moves onto the next site.



Afterwards, it compiles all of those values into an ASCII table and sends the data to SNS. Afterwards, it sets the Site ID back to the global context so Site Admins can't mess with it.

```
    titles = [('count', 'Count'),
              ('site', 'Jamf Pro Site Name')]

    table = table_print(data, titles)

    print(table)
    send_to_sns(sns_topic_arl, 'Jamf Pro Encryption Report', table)

    # Set the site back to -1 to set it back to 'Full JSS"
    url = f'{API_URL}/JSSResource/advancedcomputersearches/name/{name}'
    change_site_xml = '<advanced_computer_search><site><id>-1</id></site>
</advanced_computer_search>'
    api_put_request(url, change_site_xml)


SNS = boto3.client('sns')
SSM = boto3.client('ssm')
STAGE = os.getenv("STAGE", "dev").lower()
DEBUG = os.getenv("DEBUG", "False").lower()

logging.basicConfig(format='%(levelname)s: %(asctime)s: %(message)s')
LOGGER = logging.getLogger(__name__)
if DEBUG == 'true':
    LOGGER.setLevel(logging.DEBUG)
else:
    LOGGER.setLevel(logging.INFO)

API_URL = get_ssm_secret_value(f'/{STAGE}/JamfPro/Address')
API_USER = get_ssm_secret_value(f'/{STAGE}/JamfPro/Accts/EncryptionReport/Username')
API_PASS = get_ssm_secret_value(f'/{STAGE}/JamfPro/Accts/EncryptionReport/Password')

```


Afterwards, it compiles all of those values into an ASCII table and sends the data to SNS. Afterwards, it sets the Site ID back to the global context so Site Admins can't mess with it.

I could have gotten more fancy with this one, but it got the job done. I could have also sent this data to Teams, or I could have used Amazon Simple Email Service (or SES) to send HTML messages. Sometimes, simple is better and I wanted to highlight that it's not that hard to get something like that into the cloud.

AWS CLI Resources

- AWS Command Line Interface
- AWS CLI Documentation

JNUC 2022

AWS SAM Resources

- AWS SAM
- AWS SAM Developer Guide
- AWS SAM CLI configuration file
- AWS SAM CLI command reference
- SAM resource and property reference

JNUC 2022

Boto3 Resources

- AWS SDK for Python (Boto3)
- Boto3 Documentation
- Boto3 API Reference
- Code Examples

JNUC 2022

Other Resources

- Lambda runtimes
- Office 365 connector card
- AWS Schedule Rules

JNUC 2022

Here's the link to the resources I promised earlier. Hopefully, you found at least some of this information helpful. I wish I had more time to really dive in to some of this. If there's anything in particular that you'd like to see me talk about in more detail next year, please let me know.

Thank you for listening, and thank you to Jamf for putting together this great conference.