



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

JNUC  
2022

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.



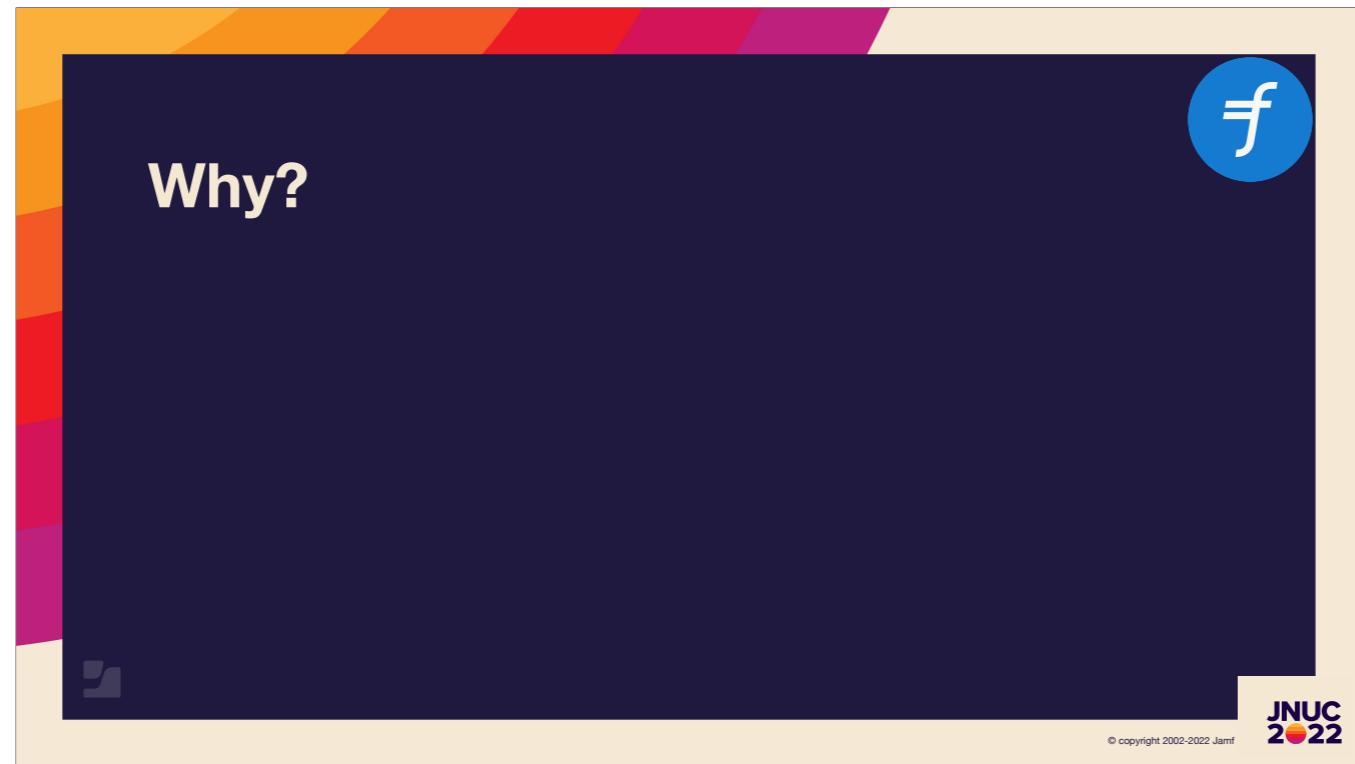
# Agenda

Why?  
Local Environment  
Common Services  
Examples

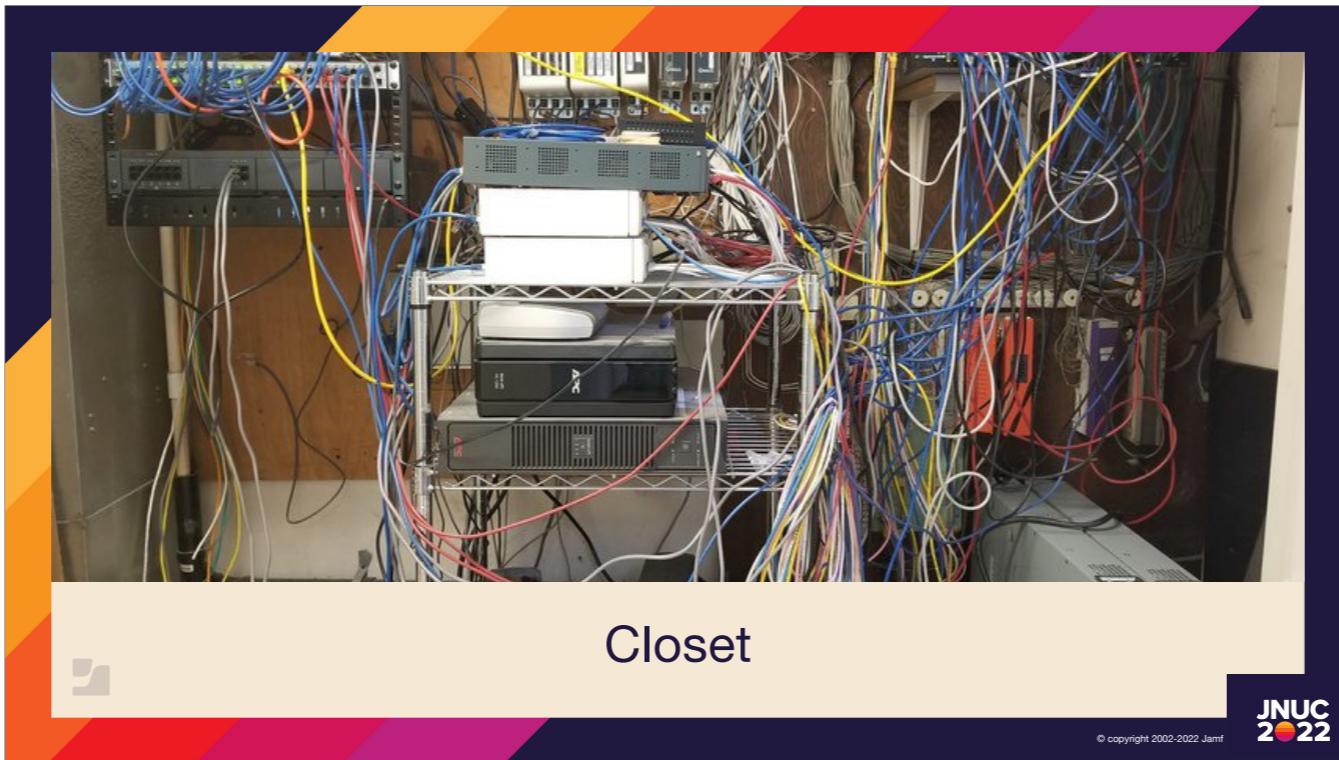


JNUC  
2022

© copyright 2002-2022 Jamf



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



Closet

© copyright 2002-2022 Jamf

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.

# Why?

- Single point of failure
- Who else can access computer?
- Security
- More expensive

© copyright 2002-2022 Jamf

JNUC  
2022

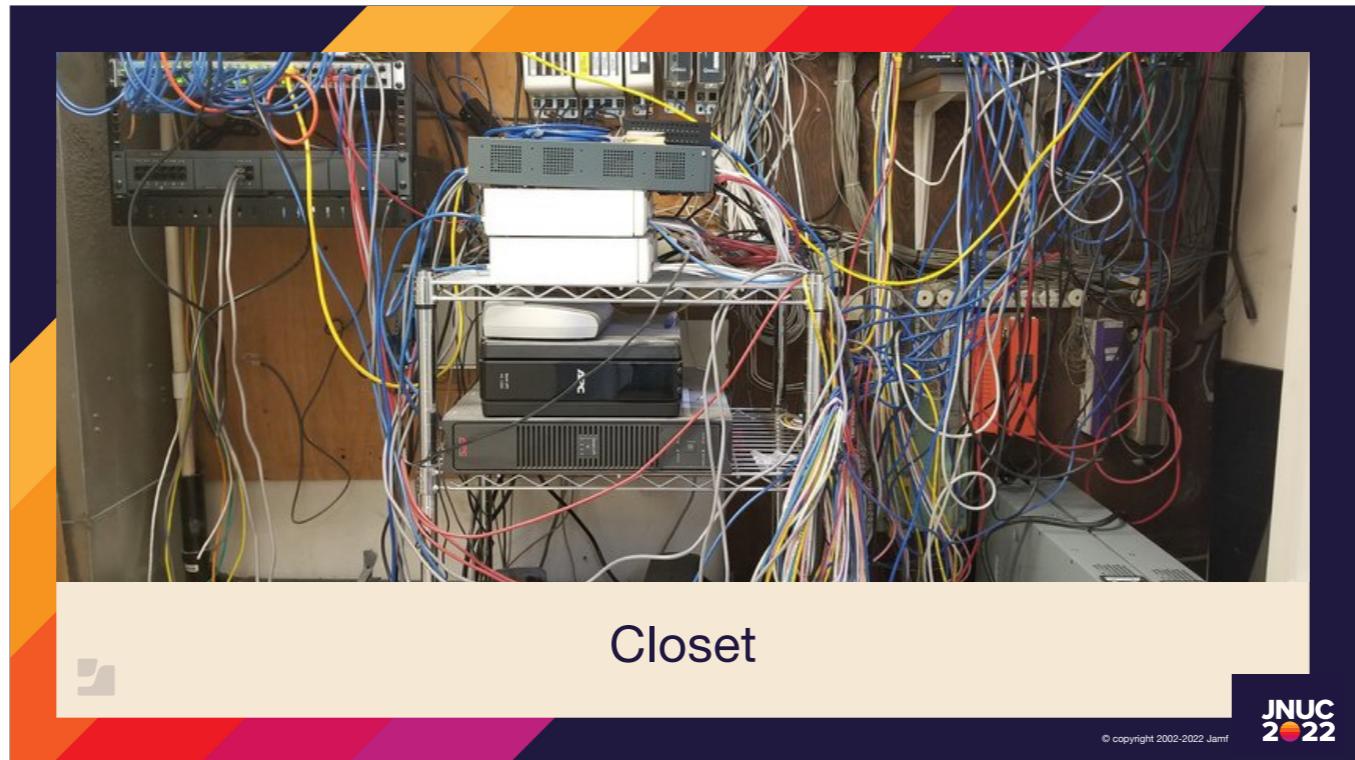
Back to the question...

Most people don't have backups of scripts run in environments like this.

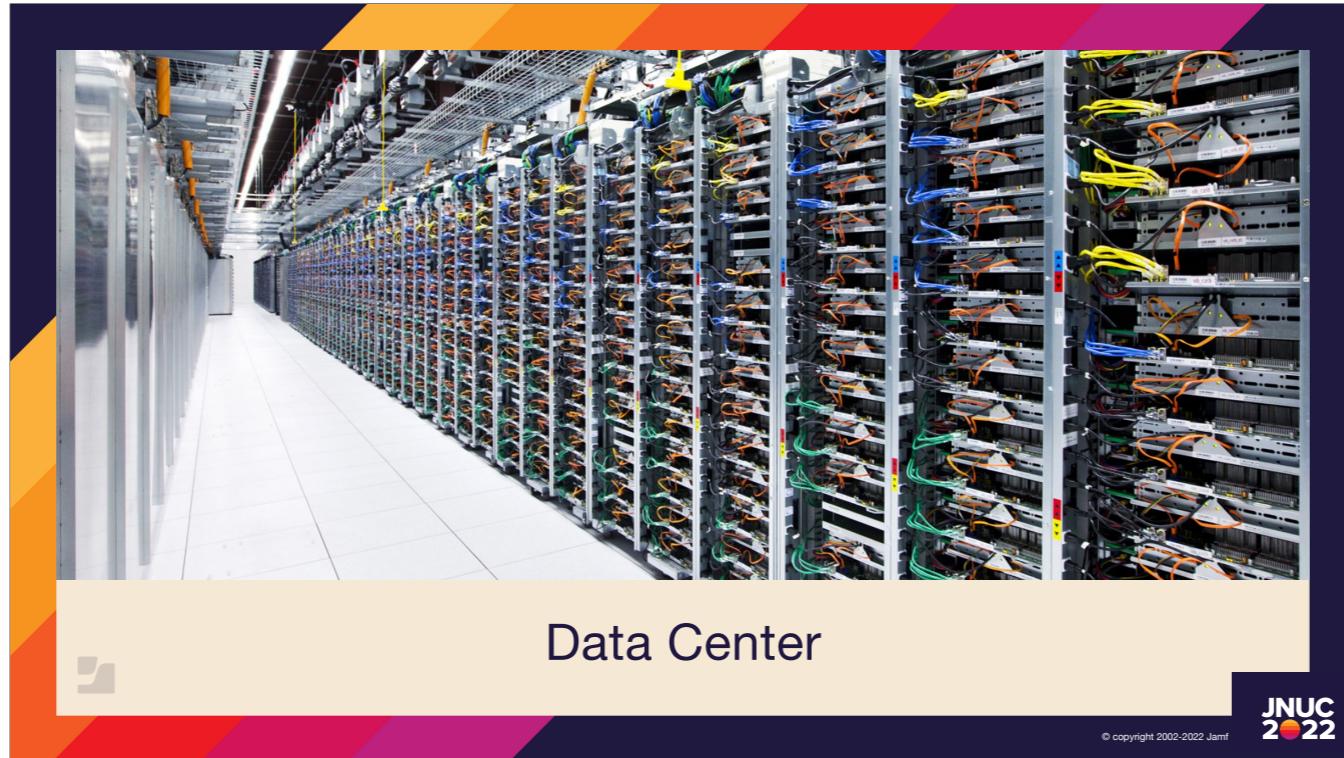
In my experience, it's common that only 1 person fully knows what it does.

They are often auto login and not patched like other computers.

Hardware is expensive to buy and maintain.



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



Data Center



JNUC  
2022

© copyright 2002-2022 Jamf

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.



## Local Environment

- Multiple Profiles
- Storing Credentials
- Configuration Options
- Logging In with SSO
- AWS Serverless Application Model

JNUC  
2022

© copyright 2002-2022 Jamf

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



## Multiple Profiles

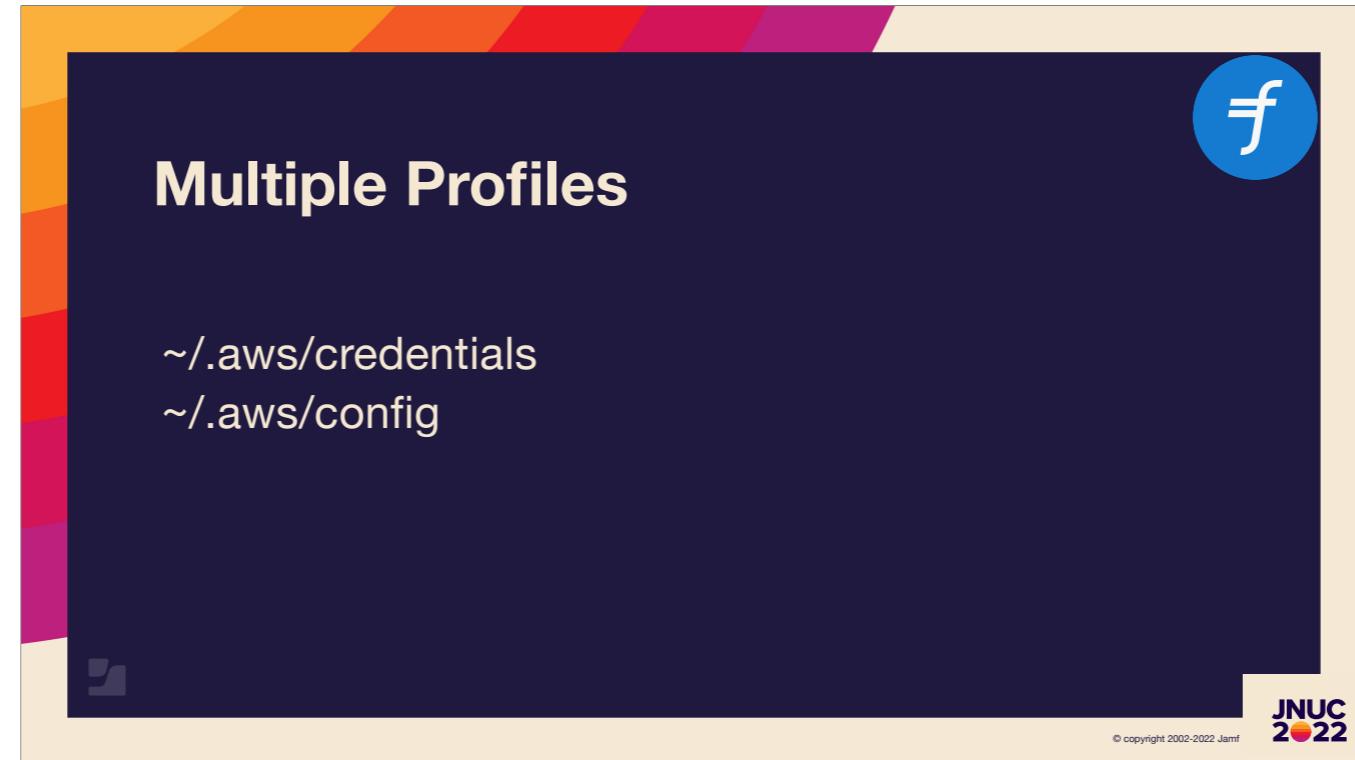
```
aws s3 ls --profile dev
```

```
aws s3 ls --profile prod
```

© copyright 2002-2022 Jamf

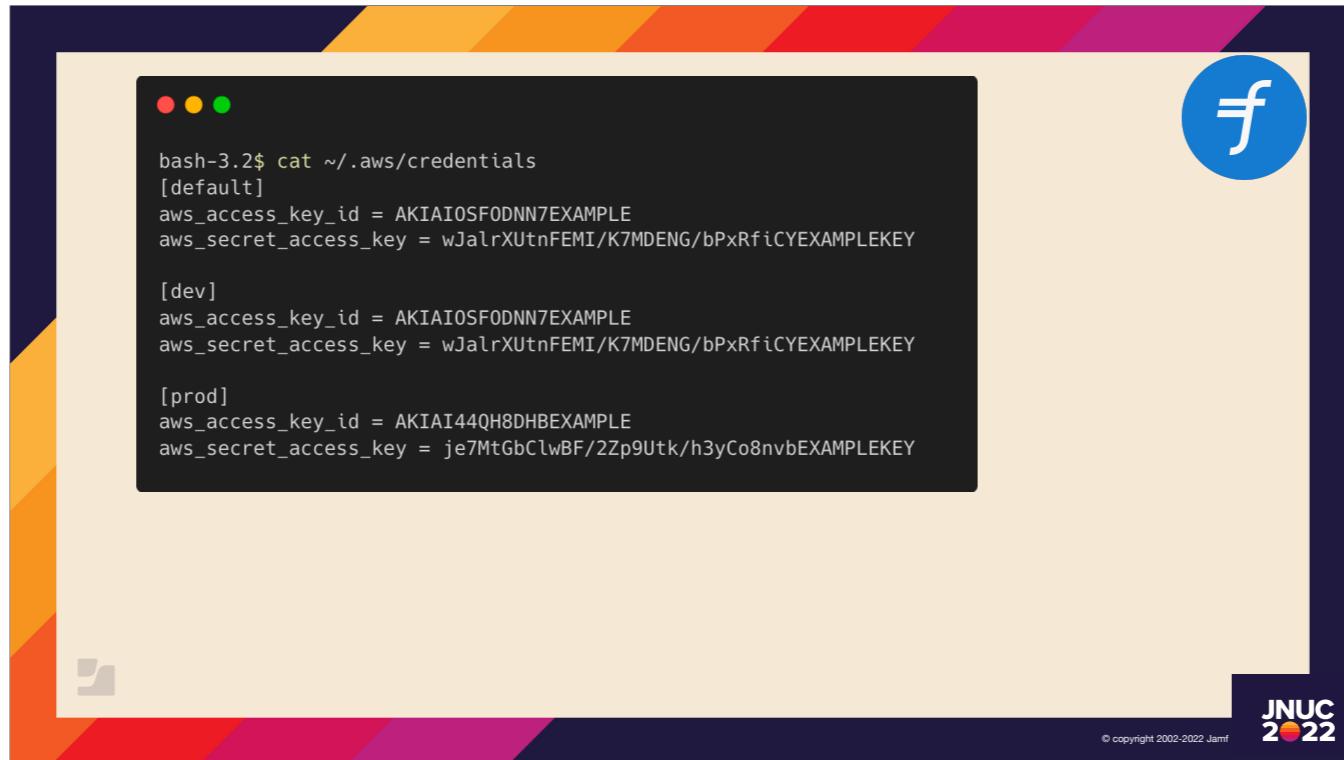
JNUC  
2022

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.

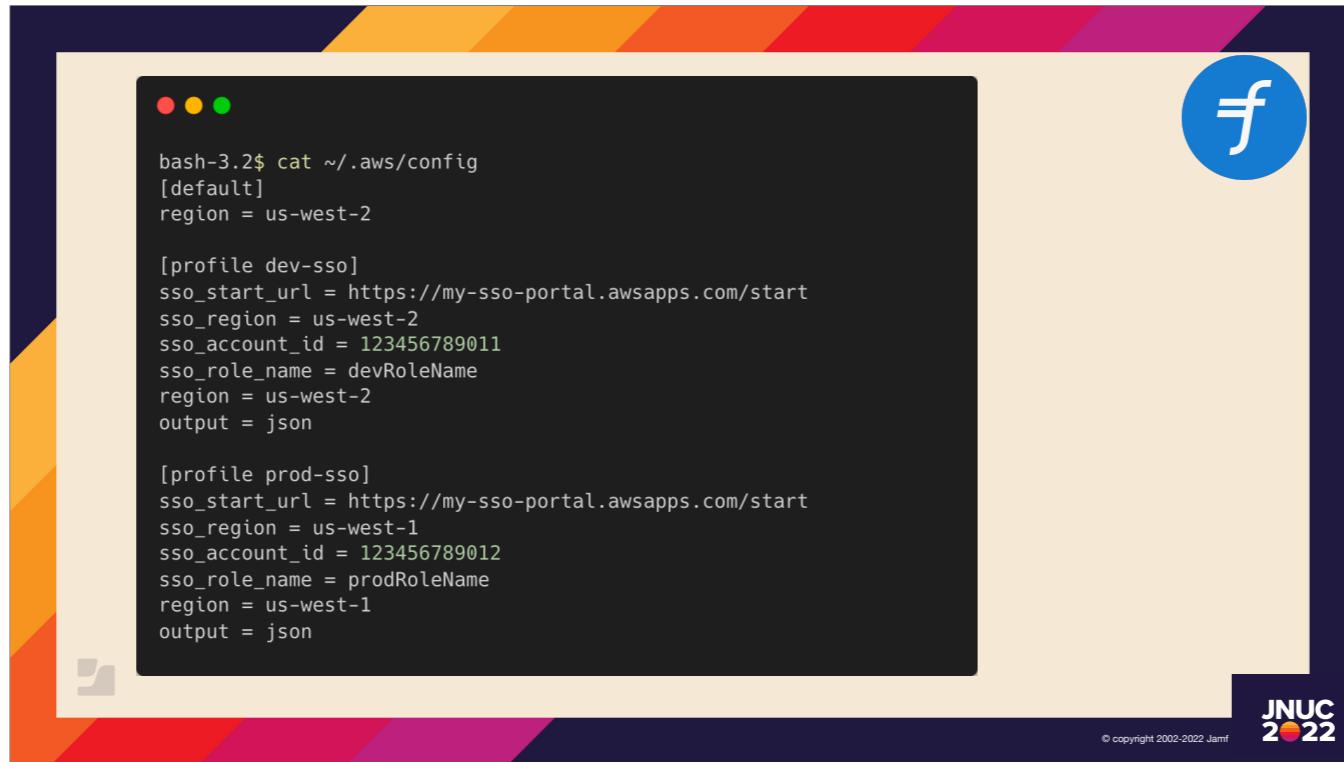


To configure your different profiles, you will be modifying these two files. You will definitely have a config file, but depending on your environment, you may not need a credentials file.

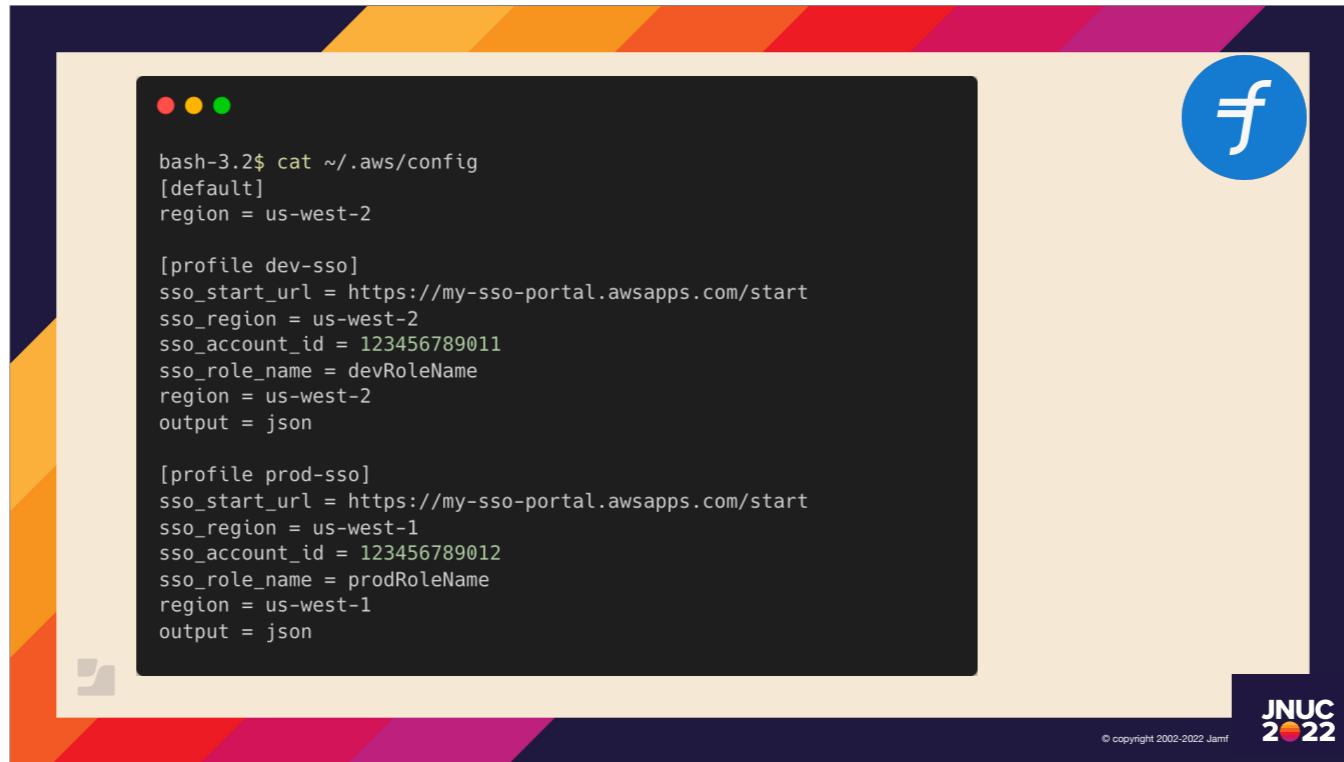
Each environment is different and I suggest you read the AWS documentation to get a better idea of what's needed. I'll show some examples, but they are nowhere near the full capabilities.



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.

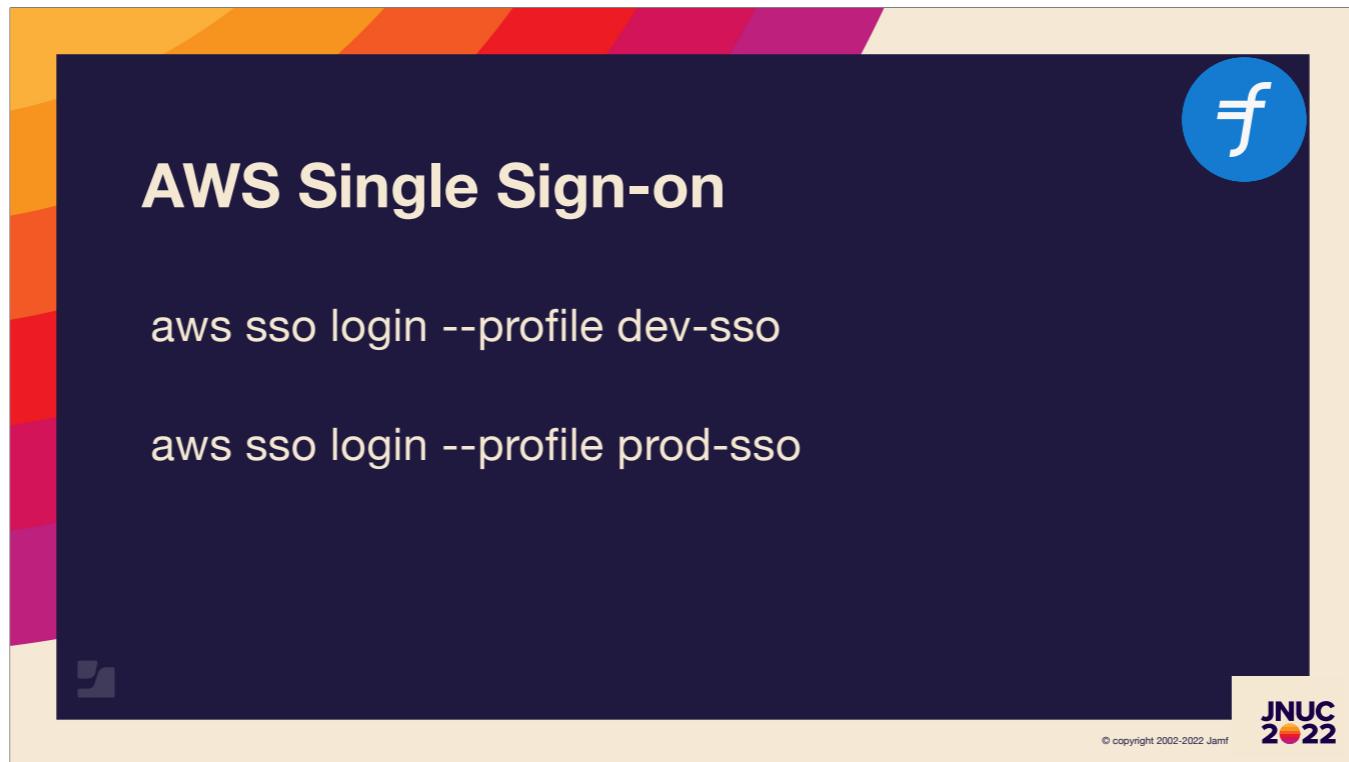


The config file is way more powerful. The data stored in this file serves as a default. You can override them in your commands, but if you don't include them, it looks here. Notice that the headers contain the word *profile*. The credentials file doesn't use that *keyword*.

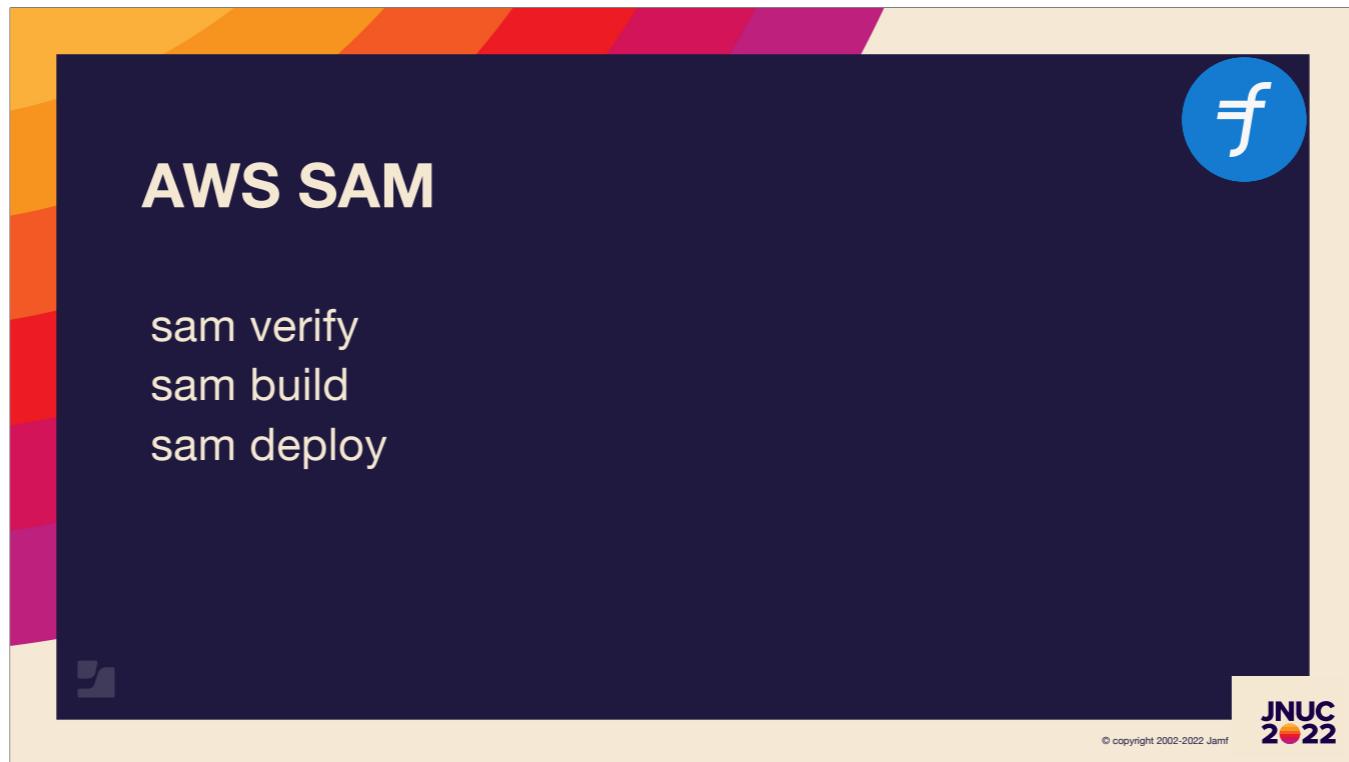


If your organization uses SSO, those settings go into config. This example shows that you can not only define different regions, but you can also define entirely different accounts and roles.

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.



## AWS SAM

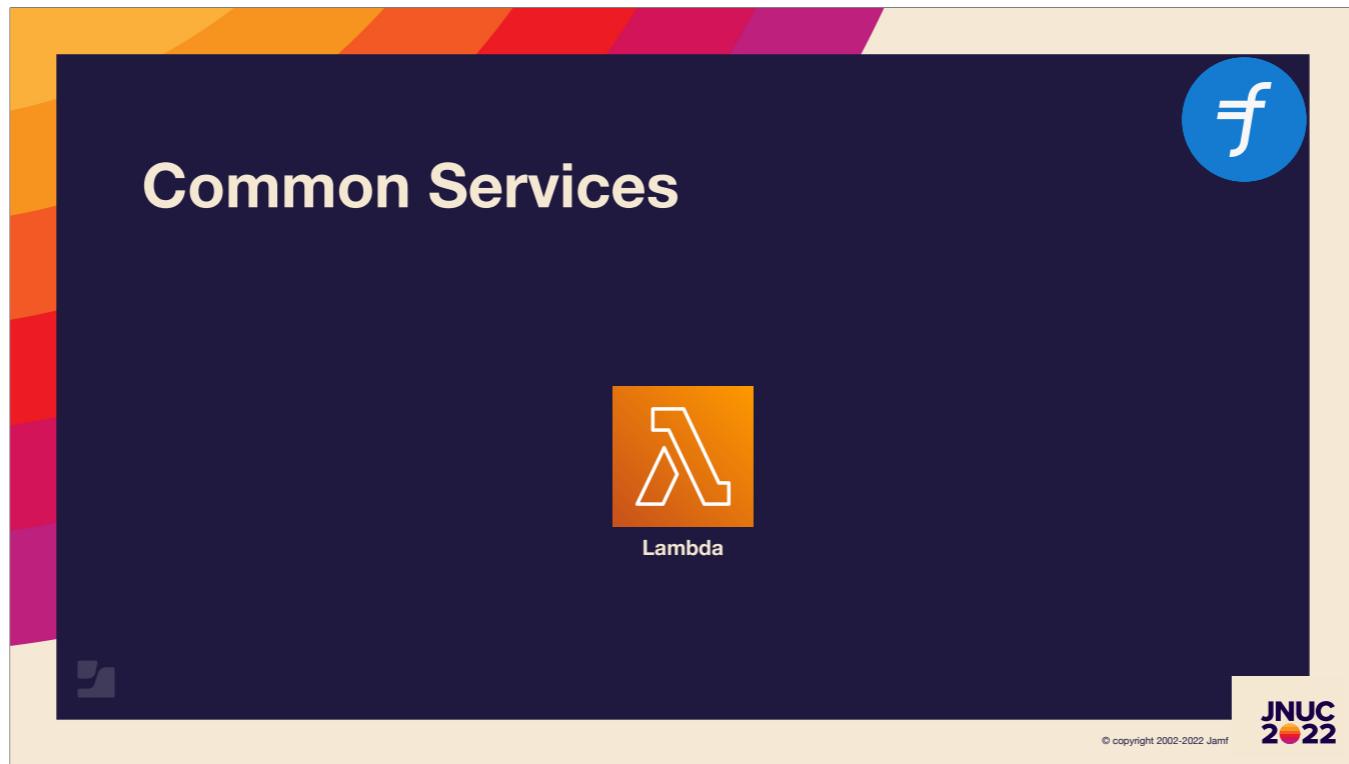
```
sam deploy --config-env dev --guided  
sam deploy --config-env prod --guided
```

```
sam deploy --config-env dev  
sam deploy --config-env prod
```

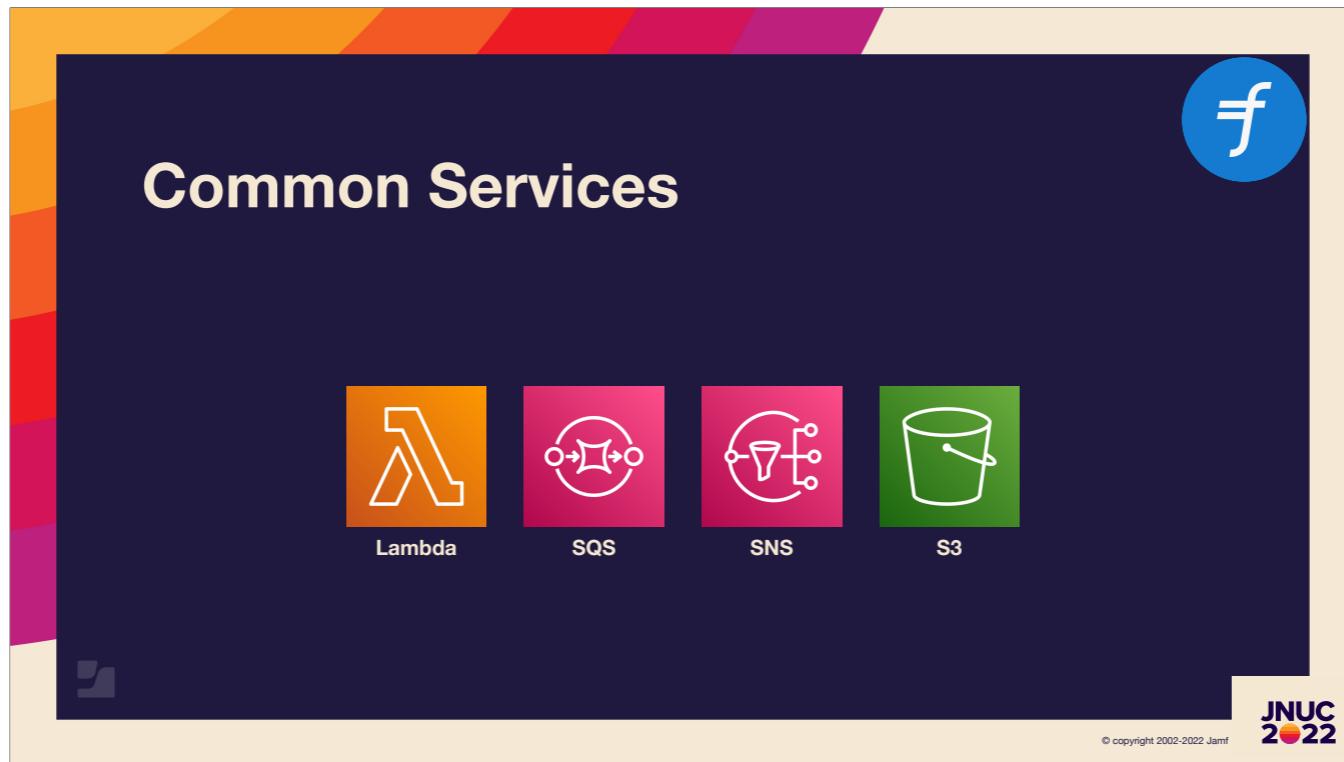
JNUC  
2022

© copyright 2002-2022 Jamf

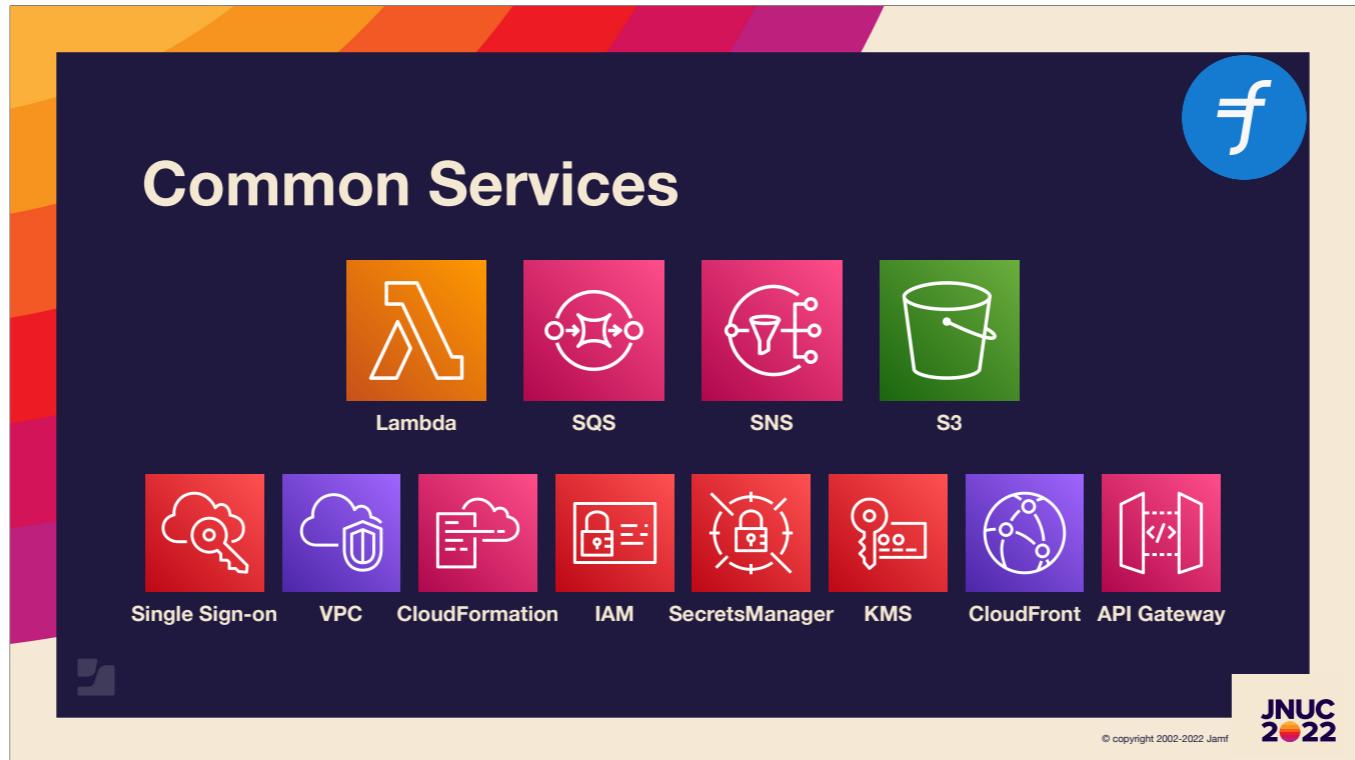
The deploy command is the one you use to push your changes to AWS. The first time you deploy to each profile, you need to use the --guided parameter. It will prompt you for more information and it will save your answers to a local toml file. Subsequent deploy runs will pull values from that file so you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



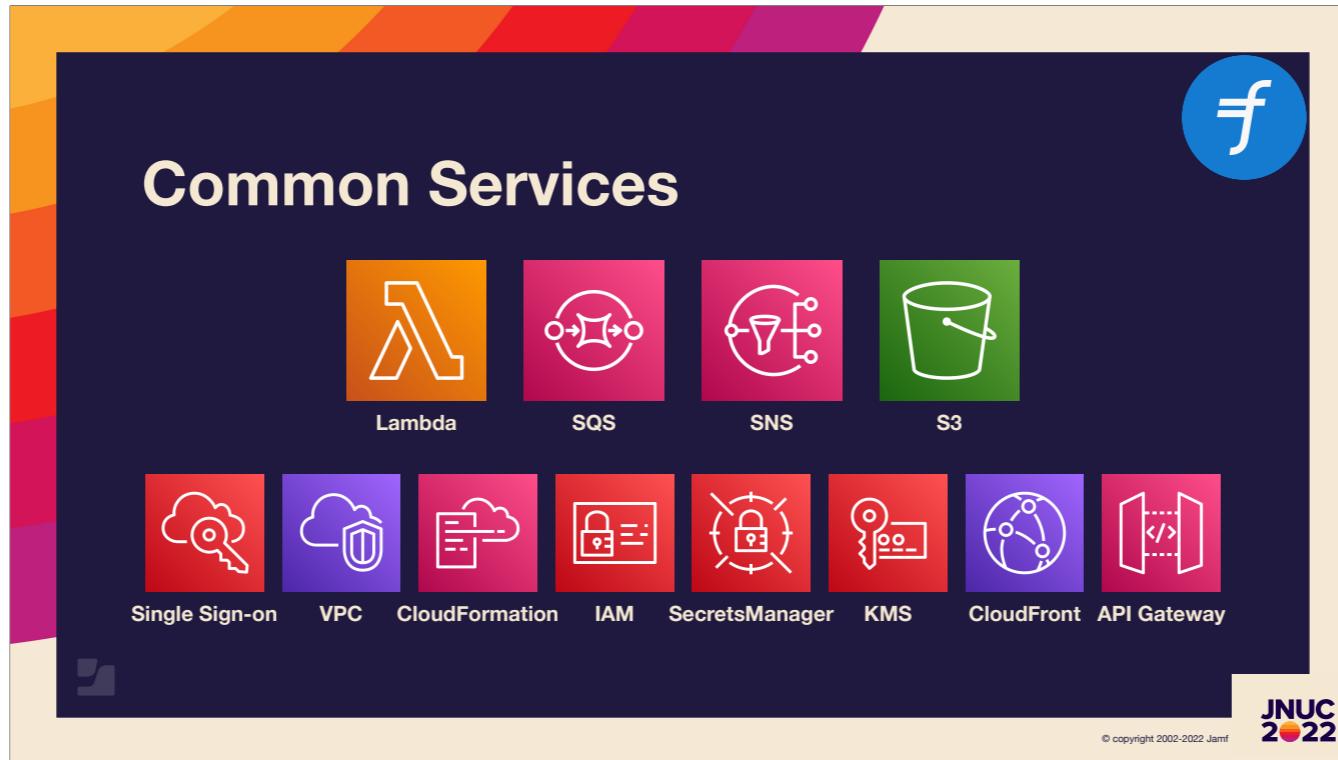
For the purposes of this talk, we're mainly talking about Lambdas. This service allows you to run code in the cloud without having to worry about the server. You are only charged during runtime. Compare that to a computer in the corner where you are paying for electricity and hardware refreshes.



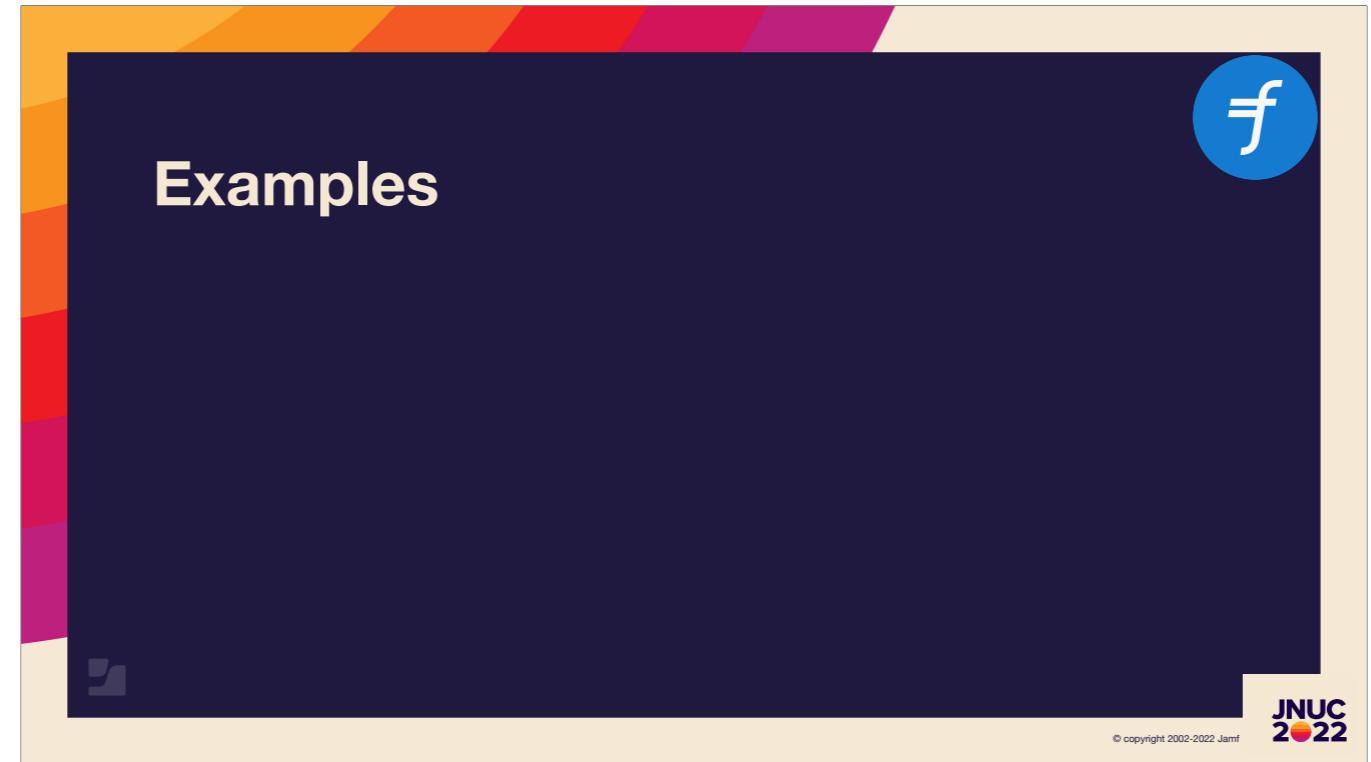
S3 gives us a place to store the code. SQS and SNS are services we can use to communicate between Lambdas or other services. SQS is a queue, and ensures the messages are handled. SNS is a notification, or put another way, messaging via UDP. There is no guarantee that it made it. Each is useful in different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



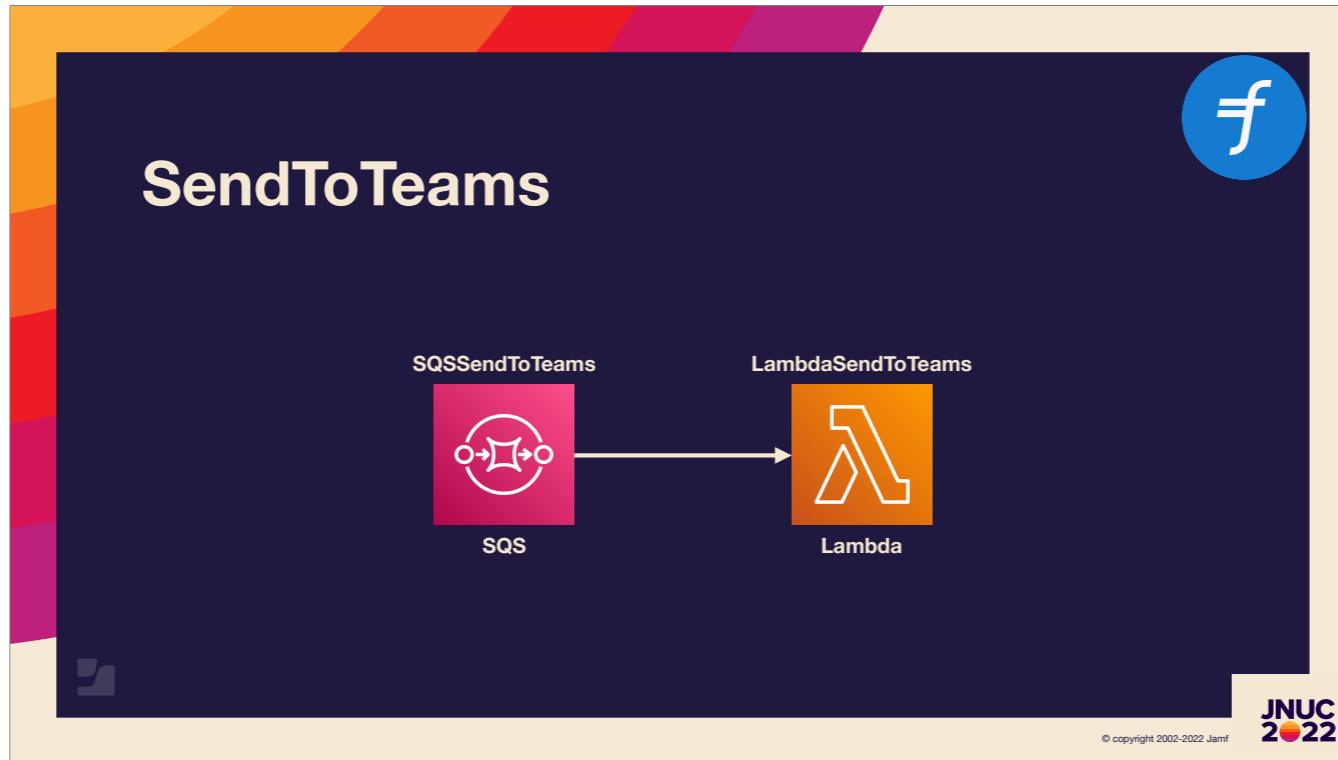
Additionally, here are some of the other services we'll touch on today. Some of these are configured automatically. They are useful to know, but may not be crucial to know **in-depth**.



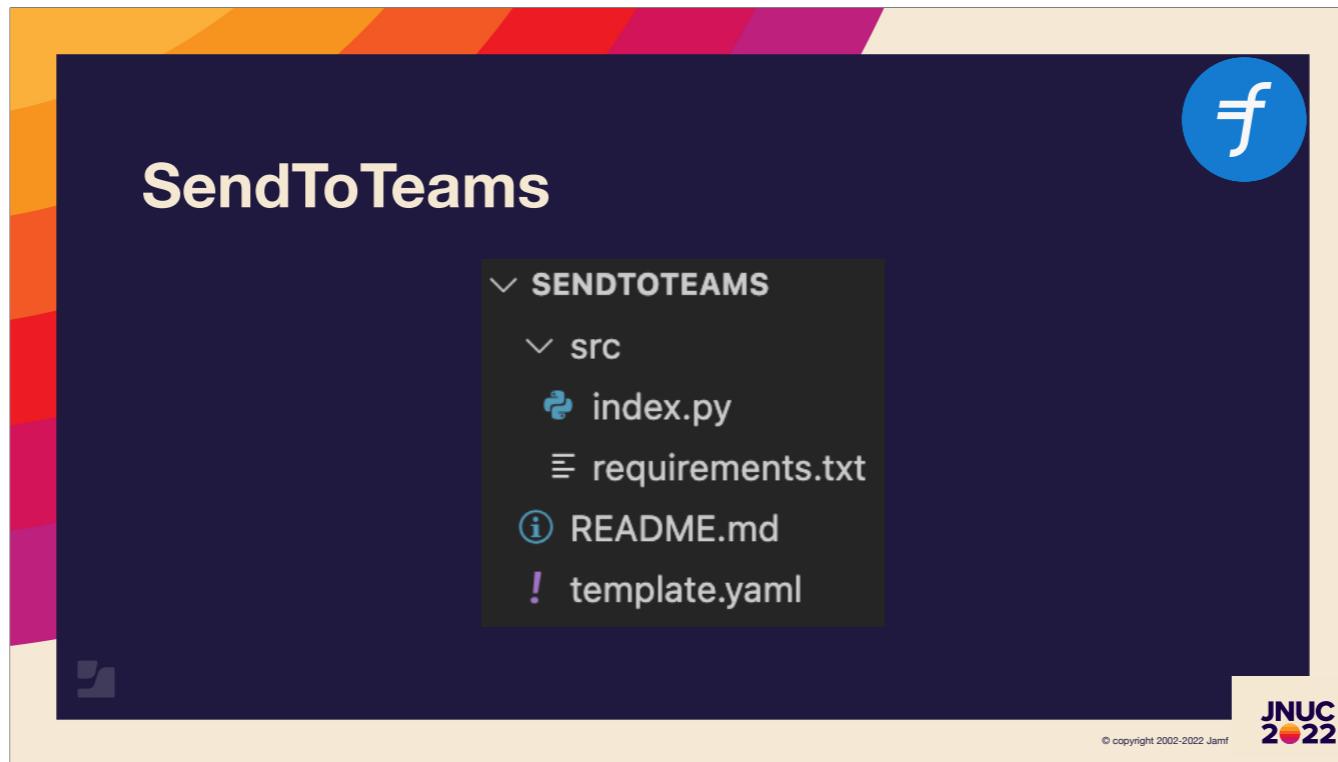
For instance, VPC (or Virtual Private Cloud) is like a VLAN and lets you isolate your infrastructure. You can use IAM (or Identity and Access Management) to grant access to SecretsManager which uses KMS (or Key Management Service) to secure its secrets.



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.

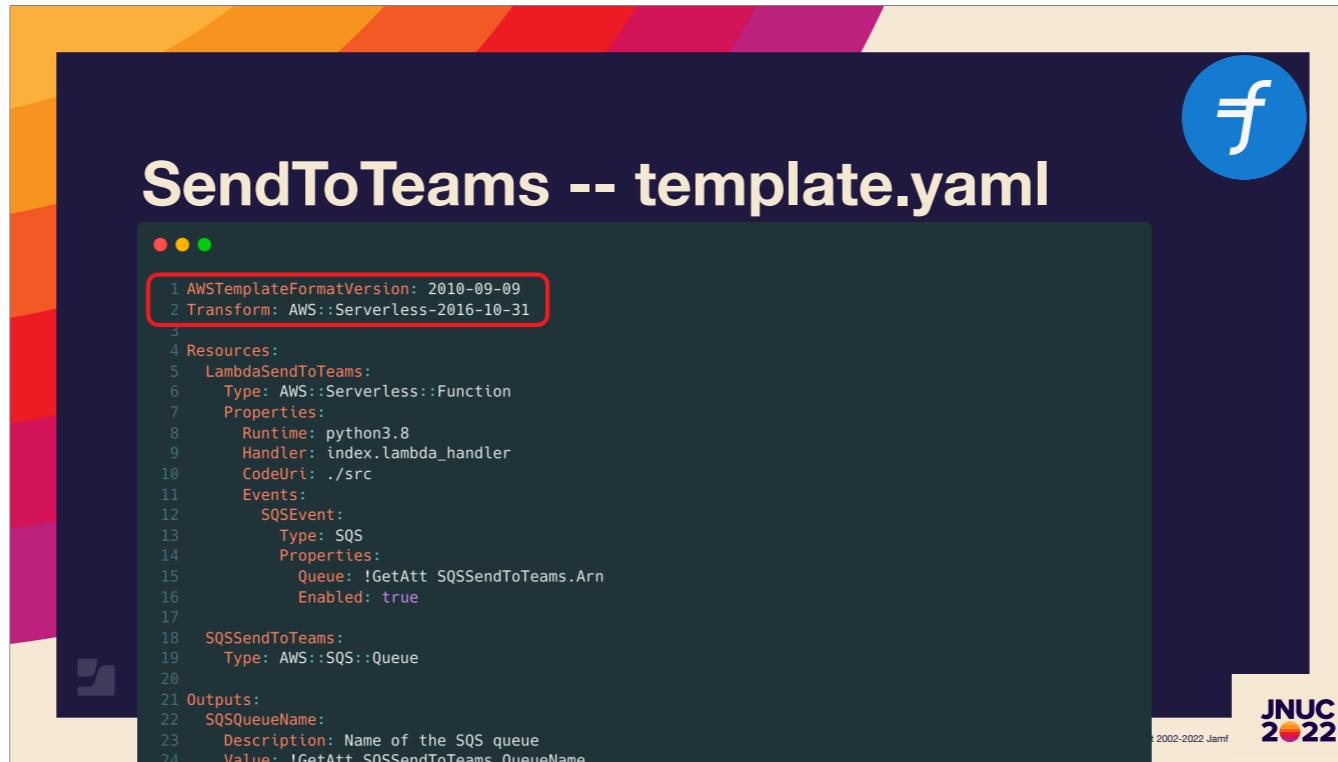


Here's the folder structure of this project. There is a template file in the root of the directory. This file is written in the AWS Serverless Application Model syntax. There is also a folder containing the lambda's code: a python script and a corresponding requirements text file. That's all there is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

## SendToTeams -- template.yaml

```
1 AWSTemplateFormatVersion: 2010-09-09
2 Transform: AWS::Serverless-2016-10-31
3
4 Resources:
5   LambdaSendToTeams:
6     Type: AWS::Serverless::Function
7     Properties:
8       Runtime: python3.8
9       Handler: index.lambda_handler
10      CodeUri: ./src
11      Events:
12        SQSEvent:
13          Type: SQS
14          Properties:
15            Queue: !GetAtt SQSSendToTeams.Arn
16            Enabled: true
17
18   SQSSendToTeams:
19     Type: AWS::SQS::Queue
20
21 Outputs:
22   SQSQueueName:
23     Description: Name of the SQS queue
24     Value: !GetAtt SQSSendToTeams.QueueName
```

Here's the template yaml [file](#).



## SendToTeams -- template.yaml

```
1 AWSTemplateFormatVersion: 2010-09-09
2 Transform: AWS::Serverless-2016-10-31
3
4 Resources:
5   LambdaSendToTeams:
6     Type: AWS::Serverless::Function
7     Properties:
8       Runtime: python3.8
9       Handler: index.lambda_handler
10      CodeUri: ./src
11      Events:
12        SQSEvent:
13          Type: SQS
14          Properties:
15            Queue: !GetAtt SQSSendToTeams.Arn
16            Enabled: true
17
18   SQSSendToTeams:
19     Type: AWS::SQS::Queue
20
21 Outputs:
22   SQSQueueName:
23     Description: Name of the SQS queue
24     Value: !GetAtt SQSSendToTeams.QueueName
```

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](#)...

## SendToTeams -- template.yaml

```
1 AWSTemplateFormatVersion: 2010-09-09
2 Transform: AWS::Serverless-2016-10-31
3
4 Resources:
5   LambdaSendToTeams:
6     Type: AWS::Serverless::Function
7     Properties:
8       Runtime: python3.8
9       Handler: index.lambda_handler
10      CodeUri: ./src
11      Events:
12        SQSEvent:
13          Type: SQS
14          Properties:
15            Queue: !GetAtt SQSSendToTeams.Arn
16            Enabled: true
17
18   SQSSendToTeams:
19     Type: AWS::SQS::Queue
20
21 Outputs:
22   SQSQueueName:
23     Description: Name of the SQS queue
24     Value: !GetAtt SQSSendToTeams.QueueName
```

JNUC  
2022

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

## SendToTeams -- template.yaml

```
1 AWSTemplateFormatVersion: 2010-09-09
2 Transform: AWS::Serverless-2016-10-31
3
4 Resources:
5   LambdaSendToTeams:
6     Type: AWS::Serverless::Function
7     Properties:
8       Runtime: python3.8
9       Handler: index.lambda_handler
10      CodeUri: ./src
11      Events:
12        SQSEvent:
13          Type: SQS
14          Properties:
15            Queue: !GetAtt SQSSendToTeams.Arn
16            Enabled: true
17
18   SQSSendToTeams:
19     Type: AWS::SQS::Queue
20
21 Outputs:
22   SQSQueueName:
23     Description: Name of the SQS queue
24     Value: !GetAtt SQSSendToTeams.QueueName
```

This is the name we are giving the resource.

The screenshot shows a terminal window with a dark background and a blue header bar. The header bar features a white 'f' icon inside a blue circle on the right side. The title of the terminal window is "SendToTeams -- template.yaml". The terminal content displays a YAML configuration file for an AWS Lambda function. The file defines a resource named "LambdaSendToTeams" which is a Serverless Function. It specifies the runtime as "python3.8" (line 8), which is highlighted with a red rectangle. Other properties include the handler "index.lambda\_handler", code URI "./src", and an event trigger "SQSEvent" from an SQS queue named "SQSSendToTeams". The queue is defined in the next section. The file also includes sections for outputs and a copyright notice at the bottom.

```
1 AWSTemplateFormatVersion: 2010-09-09
2 Transform: AWS::Serverless-2016-10-31
3
4 Resources:
5   LambdaSendToTeams:
6     Type: AWS::Serverless::Function
7     Properties:
8       Runtime: python3.8
9       Handler: index.lambda_handler
10      CodeUri: ./src
11      Events:
12        SQSEvent:
13          Type: SQS
14          Properties:
15            Queue: !GetAtt SQSSendToTeams.Arn
16            Enabled: true
17
18   SQSSendToTeams:
19     Type: AWS::SQS::Queue
20
21 Outputs:
22   SQSQueueName:
23     Description: Name of the SQS queue
24     Value: !GetAtt SQSSendToTeams.QueueName

```

© 2002-2022 Jamf

JNUC  
2022

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

## SendToTeams -- template.yaml

```
1 AWSTemplateFormatVersion: 2010-09-09
2 Transform: AWS::Serverless-2016-10-31
3
4 Resources:
5   LambdaSendToTeams:
6     Type: AWS::Serverless::Function
7     Properties:
8       Runtime: python3.8
9       Handler: index.lambda_handler
10      CodeUri: ./src
11      Events:
12        SQSEvent:
13          Type: SQS
14          Properties:
15            Queue: !GetAtt SQSSendToTeams.Arn
16            Enabled: true
17
18      SQSSendToTeams:
19        Type: AWS::SQS::Queue
20
21 Outputs:
22   SQSQueueName:
23     Description: Name of the SQS queue
24     Value: !GetAtt SQSSendToTeams.QueueName
```

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

## SendToTeams -- template.yaml

```
1 AWSTemplateFormatVersion: 2010-09-09
2 Transform: AWS::Serverless-2016-10-31
3
4 Resources:
5   LambdaSendToTeams:
6     Type: AWS::Serverless::Function
7     Properties:
8       Runtime: python3.8
9     Handler: index.lambda_handler
10    CodeUri: ./src
11    Events:
12      SQSEvent:
13        Type: SQS
14        Properties:
15          Queue: !GetAtt SQSSendToTeams.Arn
16          Enabled: true
17
18   SQSSendToTeams:
19     Type: AWS::SQS::Queue
20
21 Outputs:
22   SQSQueueName:
23     Description: Name of the SQS queue
24     Value: !GetAtt SQSSendToTeams.QueueName
```

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

## SendToTeams -- template.yaml

```
1 AWSTemplateFormatVersion: 2010-09-09
2 Transform: AWS::Serverless-2016-10-31
3
4 Resources:
5   LambdaSendToTeams:
6     Type: AWS::Serverless::Function
7     Properties:
8       Runtime: python3.8
9       Handler: index.lambda_handler
10      CodeUri: ./src
11      Events:
12        SQSEvent:
13          Type: SQS
14          Properties:
15            Queue: !GetAtt SQSSendToTeams.Arn
16            Enabled: true
17
18      SQSSendToTeams:
19        Type: AWS::SQS::Queue
20
21 Outputs:
22   SQSQueueName:
23     Description: Name of the SQS queue
24     Value: !GetAtt SQSSendToTeams.QueueName
```

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

## SendToTeams -- template.yaml

```
1 AWSTemplateFormatVersion: 2010-09-09
2 Transform: AWS::Serverless-2016-10-31
3
4 Resources:
5   LambdaSendToTeams:
6     Type: AWS::Serverless::Function
7     Properties:
8       Runtime: python3.8
9       Handler: index.lambda_handler
10      CodeUri: ./src
11      Events:
12        SQSEvent:
13          Type: SQS
14          Properties:
15            Queue: !GetAtt SQSSendToTeams.Arn
16            Enabled: true
17
18   SQSSendToTeams:
19     Type: AWS::SQS::Queue
20
21 Outputs:
22   SQSQueueName:
23     Description: Name of the SQS queue
24     Value: !GetAtt SQSSendToTeams.QueueName
```

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.

The screenshot shows a portion of an AWS CloudFormation YAML template. The template defines a Lambda function named `LambdaSendToTeams` with Python 3.8 runtime and a specific handler. It also defines an SQS queue named `SQSSendToTeams`. The `SQSSendToTeams` resource is highlighted with a red box.

```
2 Transform: AWS::Serverless-2016-10-31
3
4 Resources:
5   LambdaSendToTeams:
6     Type: AWS::Serverless::Function
7     Properties:
8       Runtime: python3.8
9       Handler: index.lambda_handler
10      CodeUri: ./src
11      Events:
12        SQSEvent:
13          Type: SQS
14          Properties:
15            Queue: !GetAtt SQSSendToTeams.Arn
16            Enabled: true
17
18  SQSSendToTeams:
19    Type: AWS::SQS::Queue
20
21 Outputs:
22   SQSQueueName:
23     Description: Name of the SQS queue
24     Value: !GetAtt SQSSendToTeams.QueueName
25     Export:
26       Name: SQSSendToTeams-QueueName
27   SQSURL:
28     Description: URL of the SQS queue
29     Value: !Ref SQSSendToTeams
30     Export:
31       Name: SQSSendToTeams-URL
```

**JNUC 2022**

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.



```
2 Transform: AWS::Serverless-2016-10-31
3
4 Resources:
5   LambdaSendToTeams:
6     Type: AWS::Serverless::Function
7     Properties:
8       Runtime: python3.8
9       Handler: index.lambda_handler
10      CodeUri: ./src
11      Events:
12        SQSEvent:
13          Type: SQS
14          Properties:
15            Queue: !GetAtt SQSSendToTeams.Arn
16            Enabled: true
17
18   SQSSendToTeams:
19     Type: AWS::SQS::Queue
20
21 Outputs:
22   SQSQueueName:
23     Description: Name of the SQS queue
24     Value: !GetAtt SQSSendToTeams.QueueName
25     Export:
26       Name: SQSSendToTeams-QueueName
27   SQSURL:
28     Description: URL of the SQS queue
29     Value: !Ref SQSSendToTeams
30     Export:
31       Name: SQSSendToTeams-URL
```



© copyright 2002-2022 Jamf

**JNUC**  
**2022**

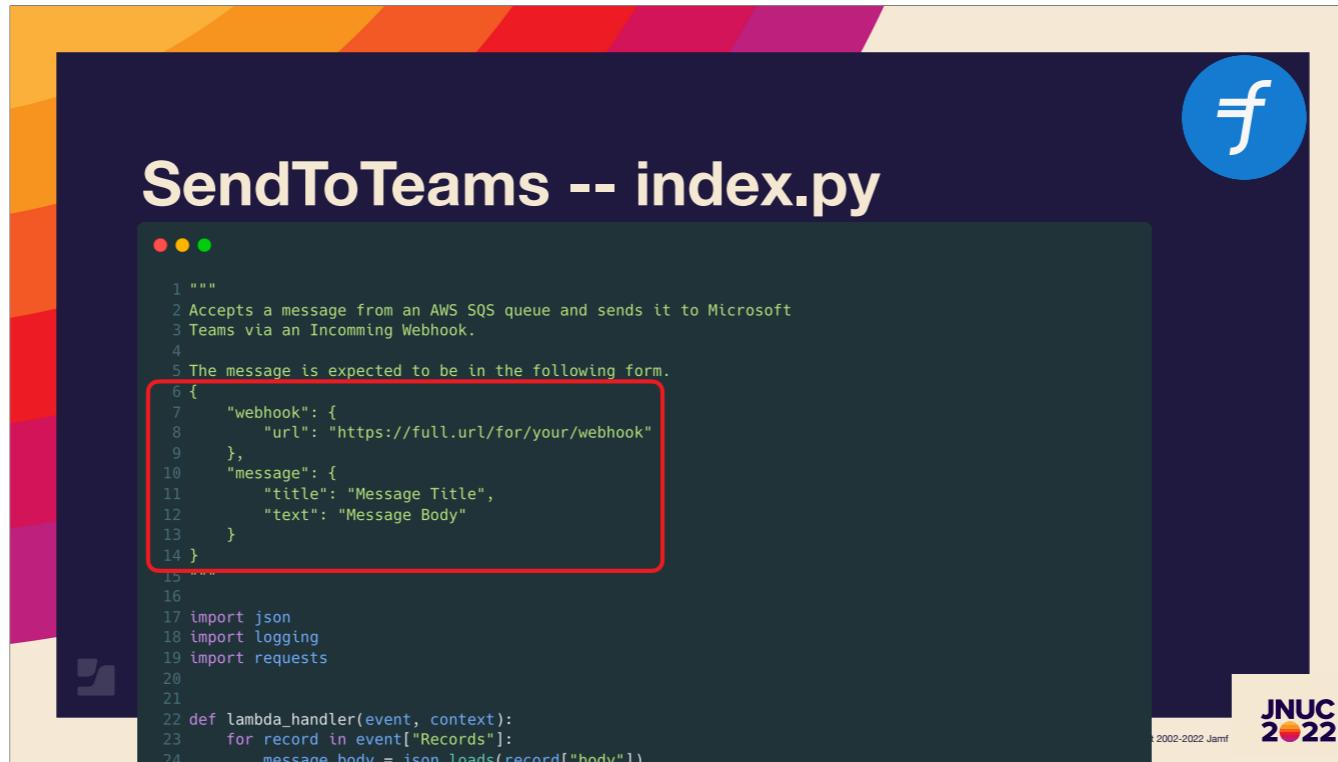
Finally, we get into a section that is not covered in many tutorials. Let's say you have other code somewhere and you want to reference this SQS queue. The outputs section exposes details to resources outside of this stack. In this case, we are defining a Name and URL that we can target to add messages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

## SendToTeams -- index.py

```
1 """
2 Accepts a message from an AWS SQS queue and sends it to Microsoft
3 Teams via an Incoming Webhook.
4
5 The message is expected to be in the following form.
6 {
7     "webhook": {
8         "url": "https://full.url/for/your/webhook"
9     },
10    "message": {
11        "title": "Message Title",
12        "text": "Message Body"
13    }
14 }
15 """
16
17 import json
18 import logging
19 import requests
20
21
22 def lambda_handler(event, context):
23     for record in event["Records"]:
24         message_body = json.loads(record["body"])
```

JNUC  
2022

Here's the template yaml [file](#).



## SendToTeams -- index.py

```
1 """
2 Accepts a message from an AWS SQS queue and sends it to Microsoft
3 Teams via an Incoming Webhook.
4
5 The message is expected to be in the following form.
6 {
7     "webhook": {
8         "url": "https://full.url/for/your/webhook"
9     },
10    "message": {
11        "title": "Message Title",
12        "text": "Message Body"
13    }
14 }
15 """
16
17 import json
18 import logging
19 import requests
20
21
22 def lambda_handler(event, context):
23     for record in event["Records"]:
24         message_body = json.loads(record["body"])
```

The slide features a decorative background with overlapping colored squares (yellow, orange, red, pink, purple) and a blue circular logo with a white 'f' in the top right corner. In the bottom right corner, there is a small logo for 'JNUC 2022' with the text '© 2002-2022 Jamf' below it.

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.

```
9     },
10    "message": {
11      "title": "Message Title",
12      "text": "Message Body"
13    }
14 }
15 """
16
17 import json
18 import logging
19 import requests
20
21
22 def lambda_handler(event, context):
23     for record in event["Records"]:
24         message_body = json.loads(record["body"])
25
26         url = message_body["webhook"]["url"]
27         msg = json.dumps(message_body["message"])
28
29         try:
30             requests.post(url, data=msg)
31         except requests.exceptions.RequestException as e:
32             LOGGER.error(e)
33         return None
34
35
36 logging.basicConfig(format='%(levelname)s: %(asctime)s: %(message)s')
37 LOGGER = logging.getLogger(__name__)
38 LOGGER.setLevel(logging.INFO)
39
```

© copyright 2002-2022 Jamf

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.

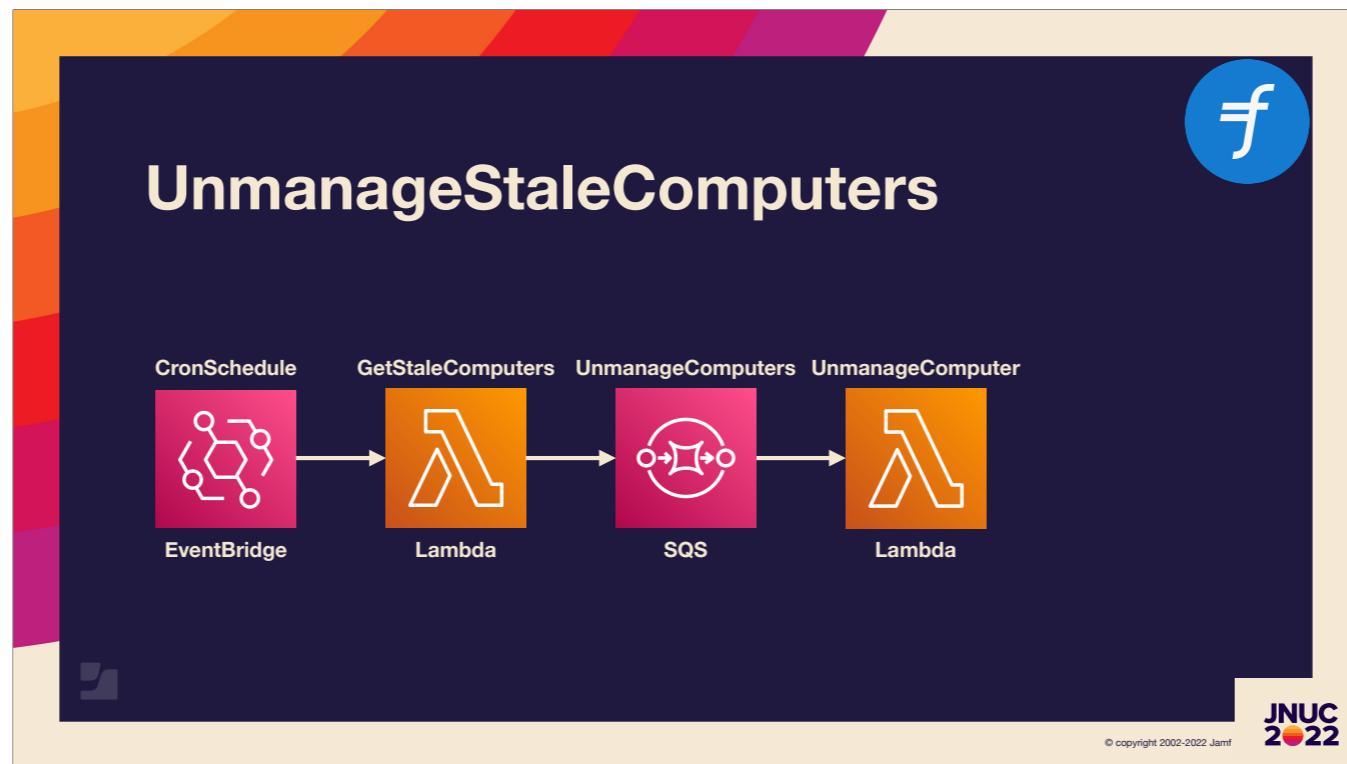


```
9     },
10    "message": {
11      "title": "Message Title",
12      "text": "Message Body"
13    }
14 }
15 """
16
17 import json
18 import logging
19 import requests
20
21
22 def lambda_handler(event, context):
23     for record in event['Records']:
24         message_body = json.loads(record["body"])
25
26         url = message_body["webhook"]["url"]
27         msg = json.dumps(message_body["message"])
28
29         try:
30             requests.post(url, data=msg)
31         except requests.exceptions.RequestException as e:
32             LOGGER.error(e)
33         return None
34
35
36 logging.basicConfig(format='%(levelname)s: %(asctime)s: %(message)s')
37 LOGGER = logging.getLogger(__name__)
38 LOGGER.setLevel(logging.INFO)
39
```

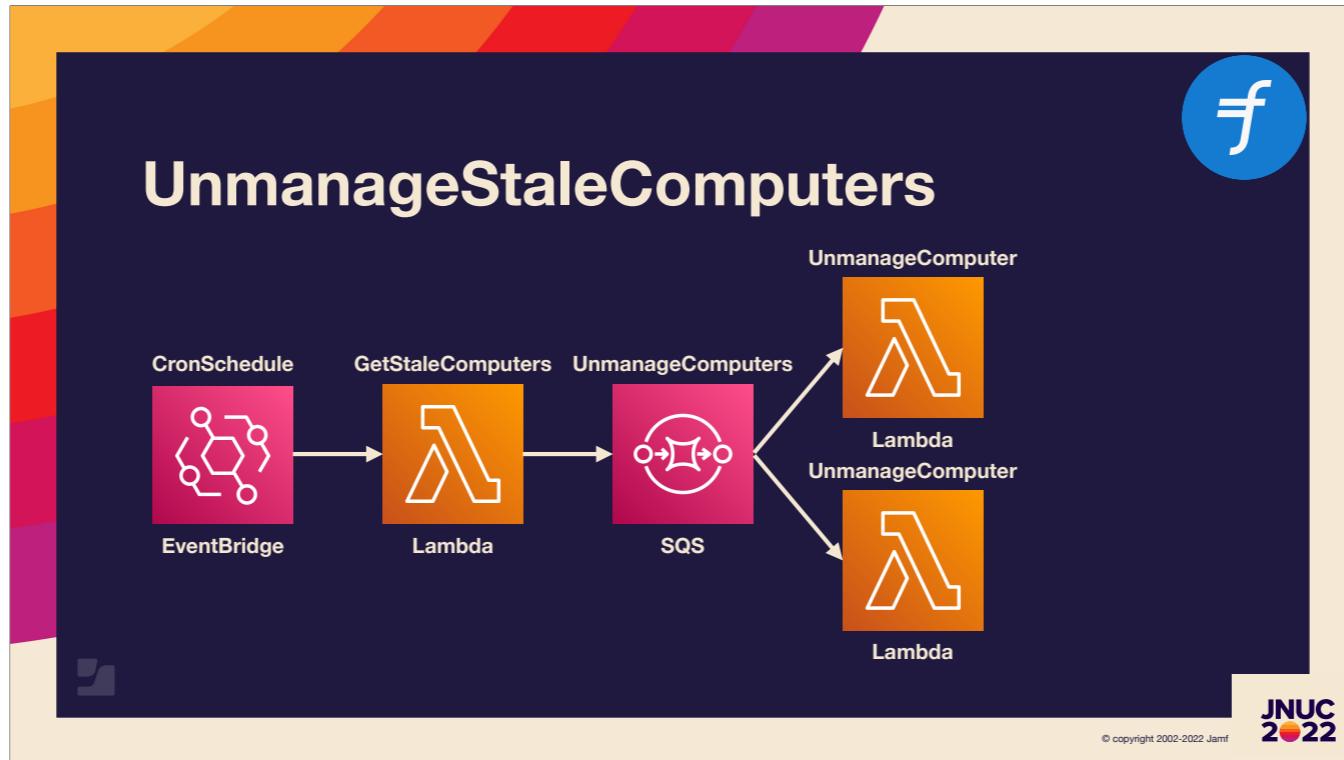


© copyright 2002-2022 Jamf

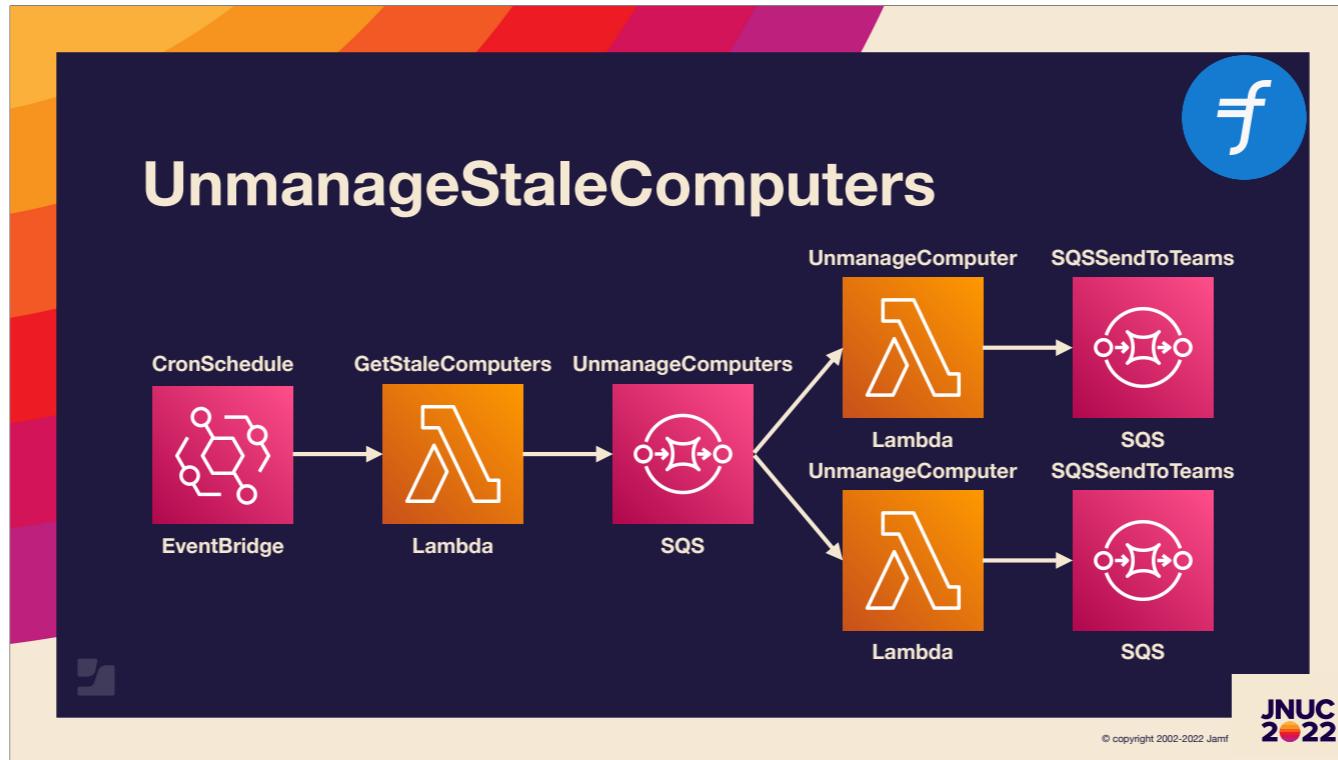
Note that there is a function named `lambda_handler` in here. This matches the template definition and will therefore be the function called in the script. We can add more functions without worrying about the starting point.



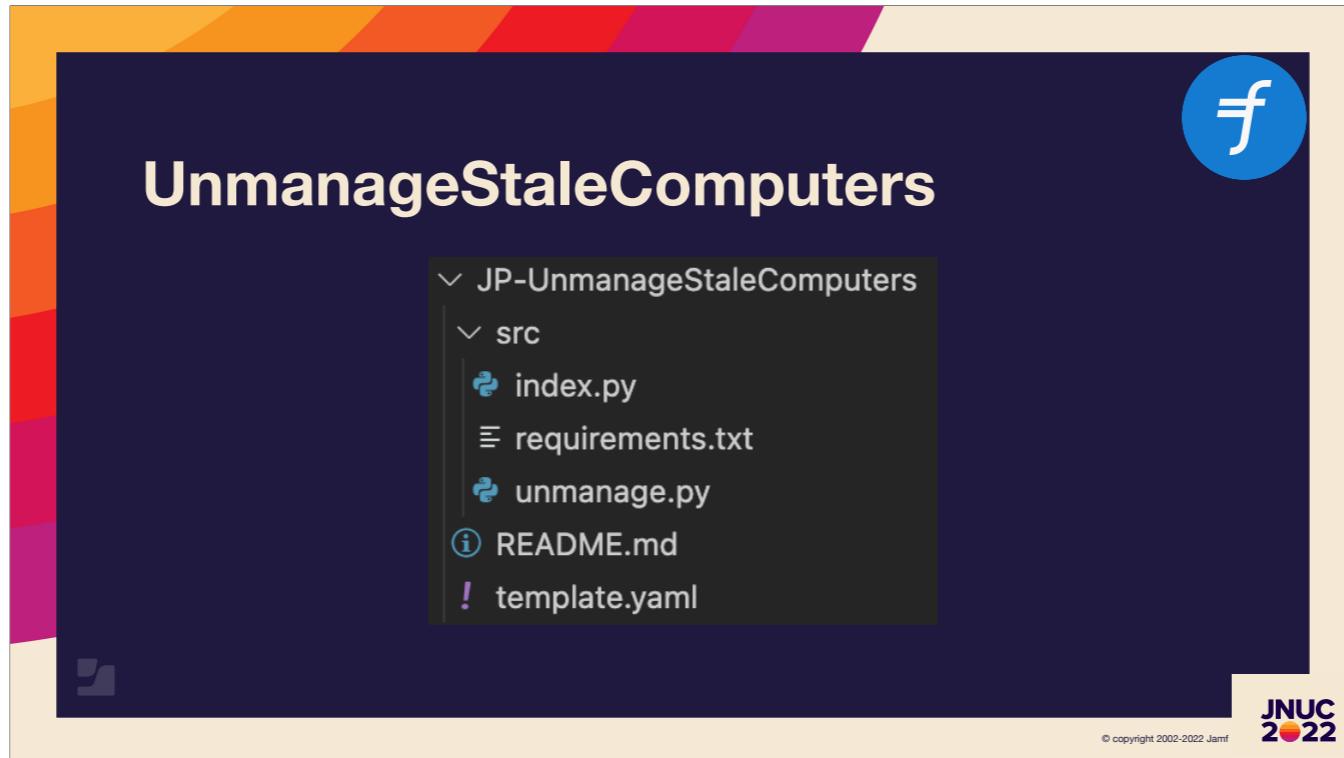
Our next example is a little more complex. It is triggered on a schedule and gets a list of computers from a Jamf Pro Advanced Search. It takes that list of devices and puts each of them into a queue which triggers the other Lambda to unmanage the computer. You can think of this as a one-to-many relationship.



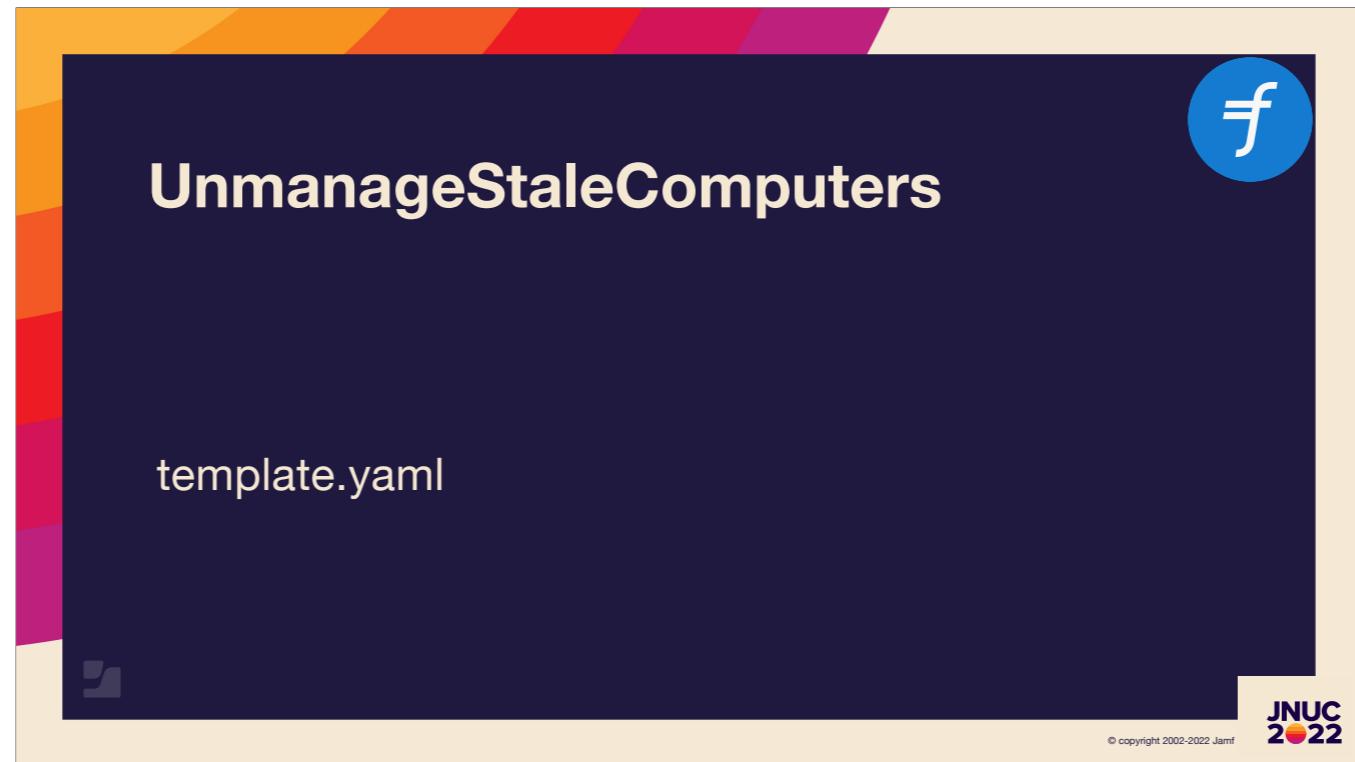
You can think of this as a one-to-many relationship. In fact, when that Lambda **completes**, it sends a message to the SendToTeams queue.



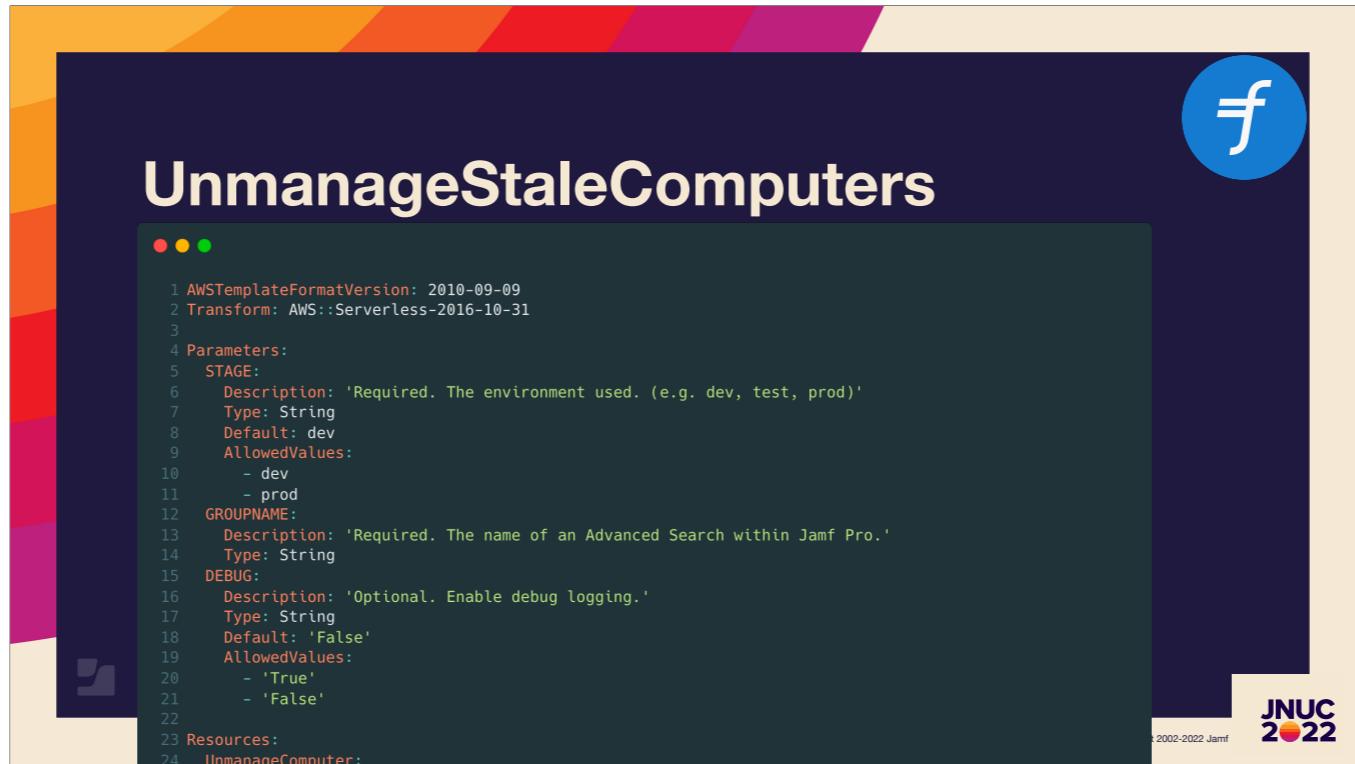
In fact, when that Lambda **completes**, it sends a message to the **SendToTeams** queue. As you can see, you can get pretty complex pretty quick. The cool thing is that the structure of this stack isn't much more complex than the first.



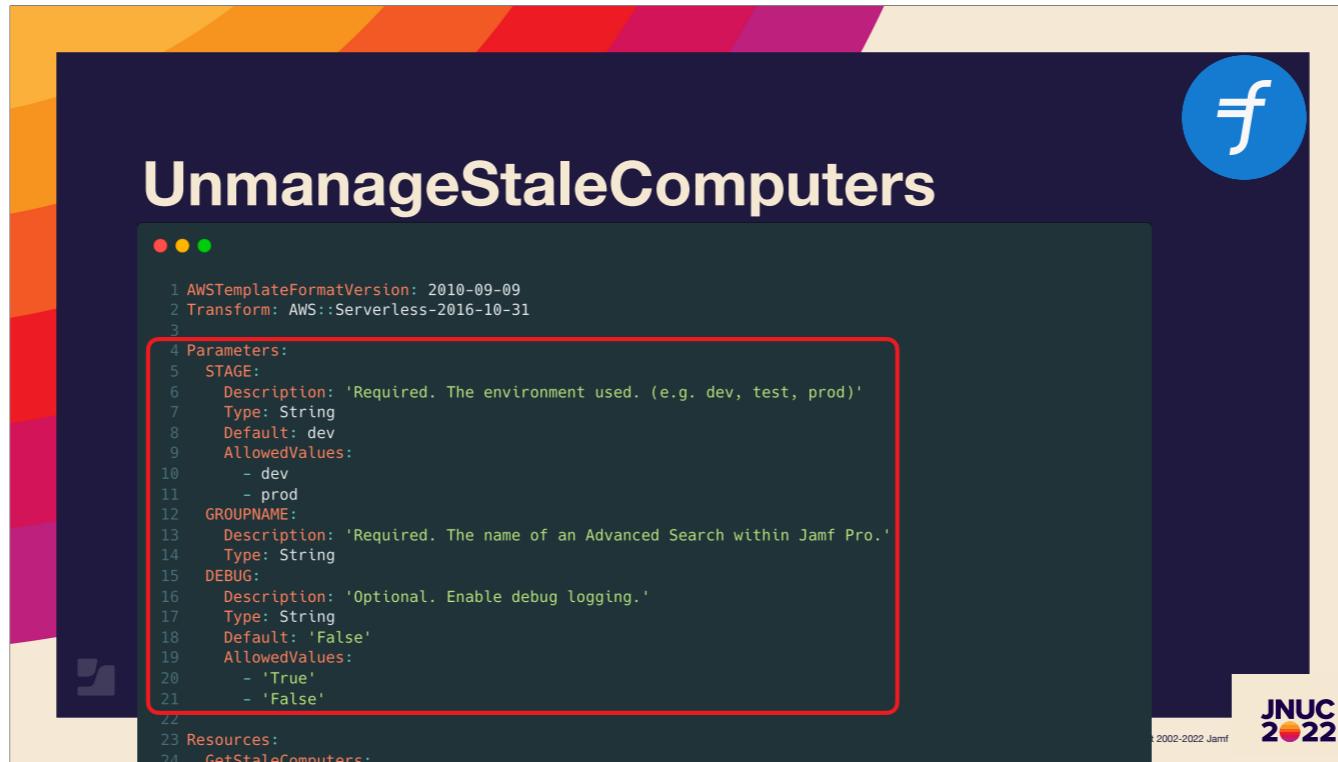
As you can see, we have an extra python file inside the source directory, but other than that, this project looks the same. Let's dig into the template again. I'm going to pick up the pace a bit but I'll still highlight some key differences.



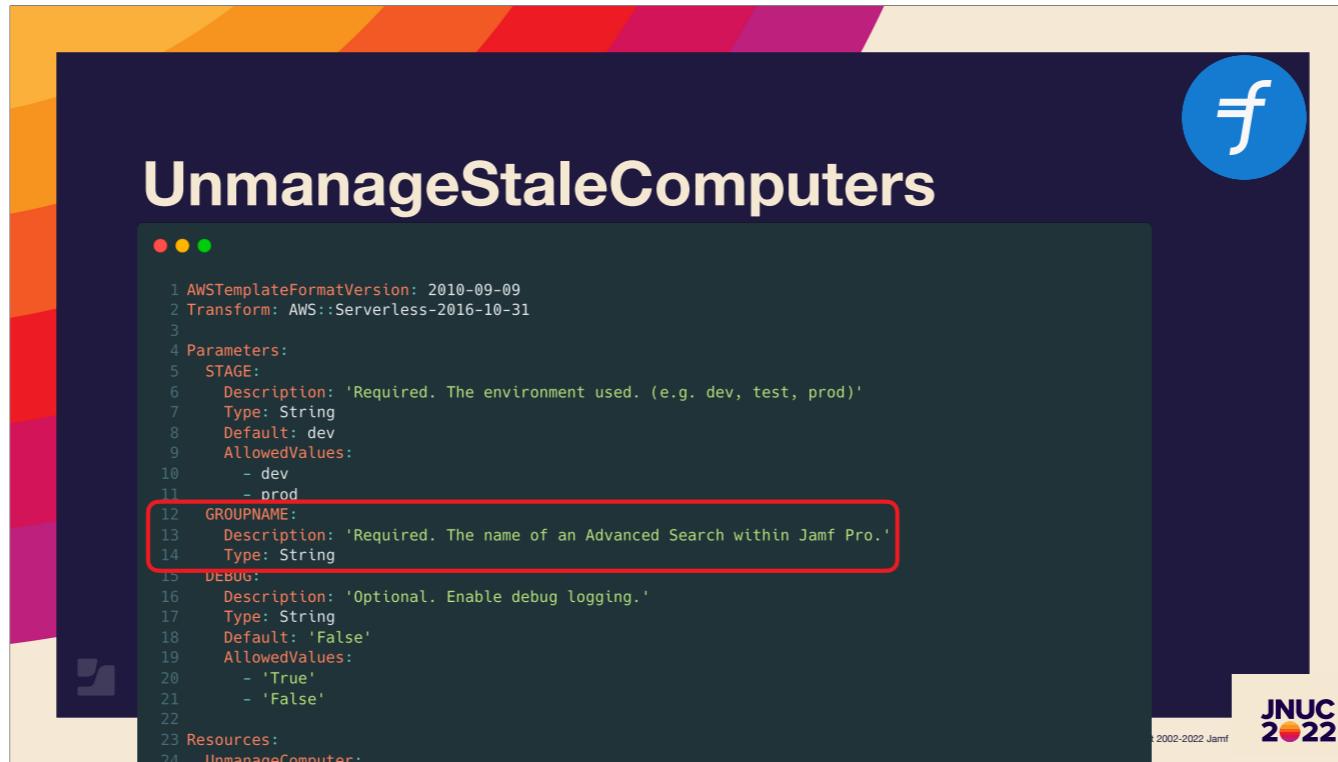
Let's start with the template again.



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.



```
52 UnmanageComputers:
53   Type: AWS::SQS::Queue
54
55 GetStaleComputers:
56   Type: AWS::Serverless::Function
57   Properties:
58     Runtime: python3.8
59     Handler: index.lambda_handler
60     CodeUri: ./src
61
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
70   Policies:
71     - SSMPolicy:
72       ParameterName: !Sub ${STAGE}/JamfPro/Address
73     - SSMPolicy:
74       ParameterName: !Sub ${STAGE}/JamfPro/Accts/UnmanageComputers/*
75     - SQSSendMessagePolicy:
76       QueueName: !GetAtt UnmanageComputers.QueueName
77
78 Events:
79   CronSchedule:
80     Type: Schedule
81     Properties:
82       # 7am ET (11 UTC) on 1st of each month
83       Schedule: cron(0 11 1 * ? *)
84       Enabled: True
```



JNUC  
2022

© copyright 2002-2022 Jamf

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.



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



© copyright 2002-2022 Jamf

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.



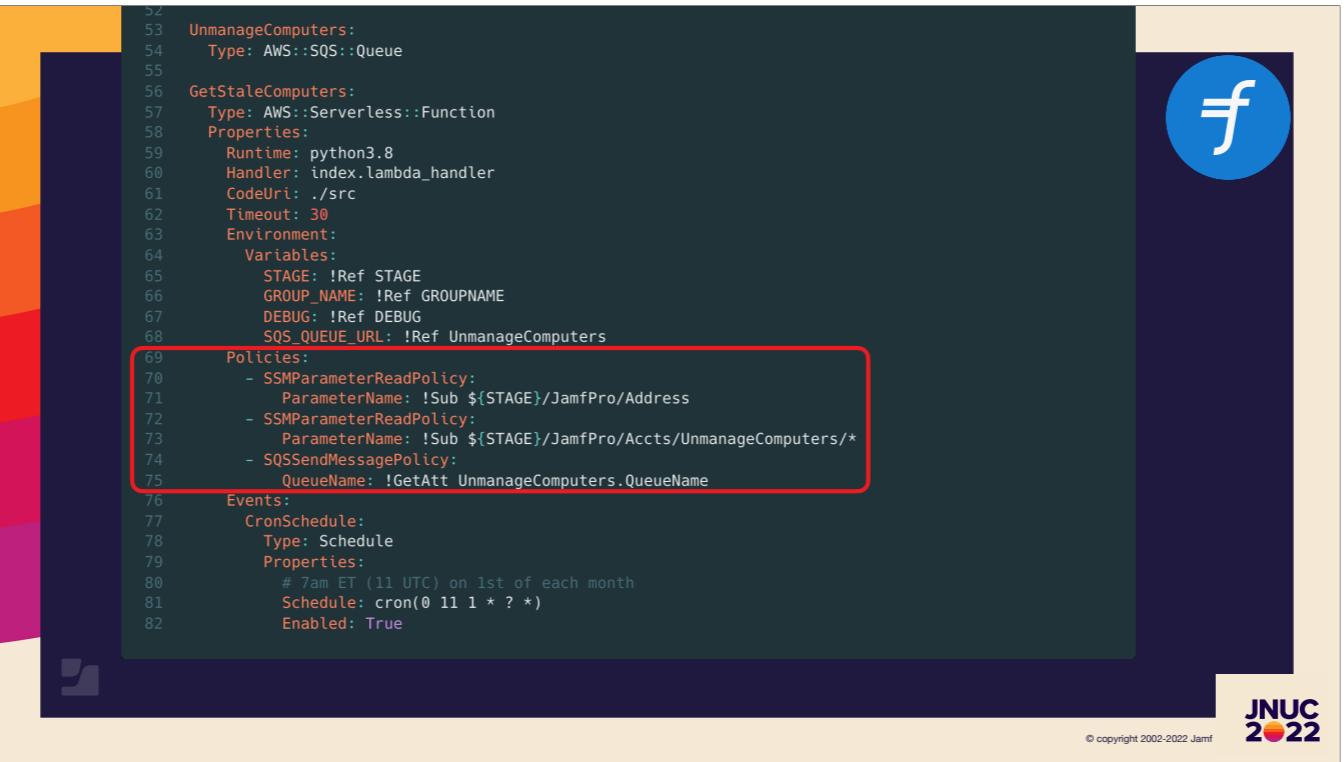
```
52 UnmanageComputers:
53   Type: AWS::SQS::Queue
54
55 GetStaleComputers:
56   Type: AWS::Serverless::Function
57   Properties:
58     Runtime: python3.8
59     Handler: index.lambda_handler
60     CodeUri: ./src
61     Timeout: 30
62     Environment:
63       Variables:
64         STAGE: !Ref STAGE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



© copyright 2002-2022 Jamf

**JNUC  
2022**

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

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.

JNUC  
2022

© copyright 2002-2022 Jamf



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



© copyright 2002-2022 Jamf

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



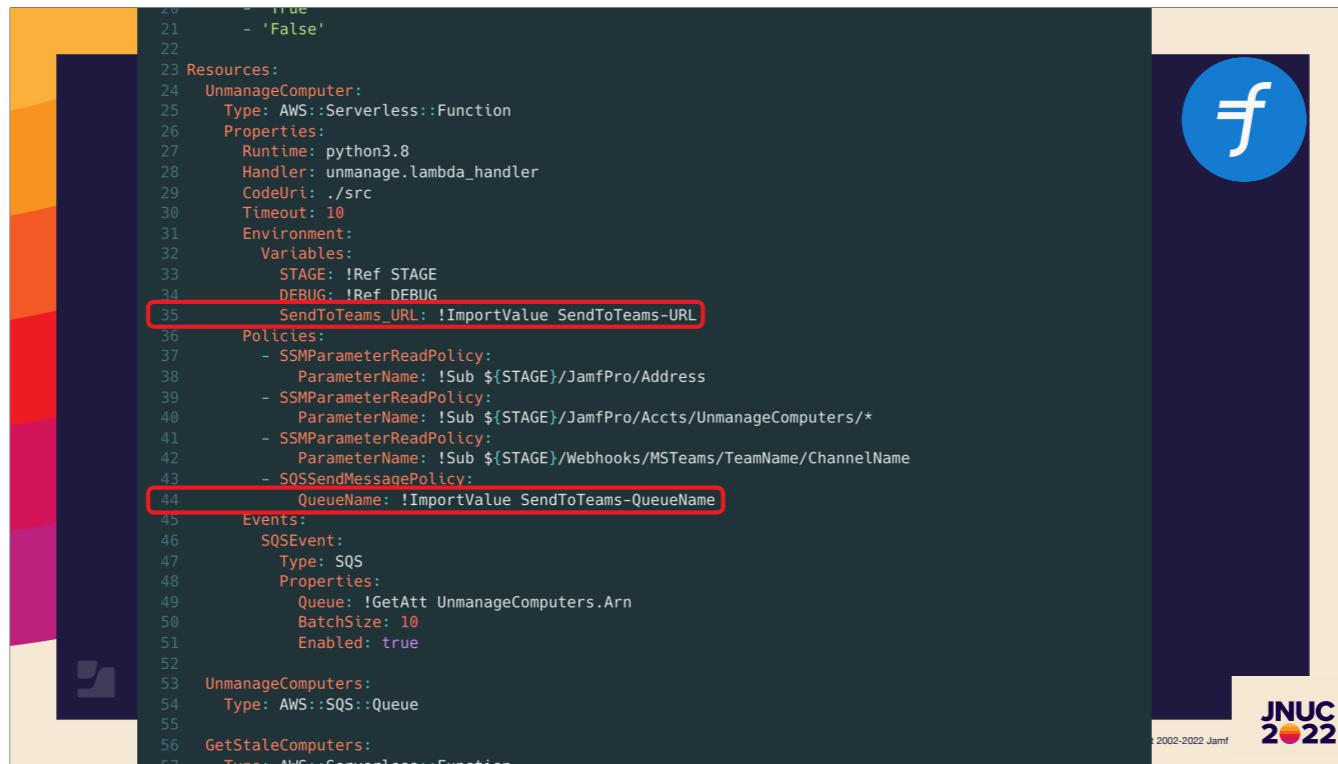
© copyright 2002-2022 Jamf

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.



```
20      - True
21      - 'False'
22
23 Resources:
24   UnmanageComputer:
25     Type: AWS::Serverless::Function
26     Properties:
27       Runtime: python3.8
28       Handler: unmanage.lambda_handler
29       CodeUri: ./src
30       Timeout: 10
31       Environment:
32         Variables:
33           STAGE: !Ref STAGE
34           DEBUG: !Ref DEBUG
35           SendToTeams_URL: !ImportValue SendToTeams-URL
36       Policies:
37         - SSMPolicy:
38             ParameterName: !Sub ${STAGE}/JamfPro/Address
39         - SSMPolicy:
40             ParameterName: !Sub ${STAGE}/JamfPro/Accts/UnmanageComputers/*
41         - SSMPolicy:
42             ParameterName: !Sub ${STAGE}/Webhooks/MSTeams/TeamName/ChannelName
43         - SQSSendMessagePolicy:
44             QueueName: !ImportValue SendToTeams-QueueName
45       Events:
46         SQSEvent:
47           Type: SQS
48           Properties:
49             Queue: !GetAtt UnmanageComputers.Arn
50             BatchSize: 10
51             Enabled: true
52
53   UnmanageComputers:
54     Type: AWS::SQS::Queue
55
56   GetStaleComputers:
57     Type: AWS::Serverless::Function
```

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

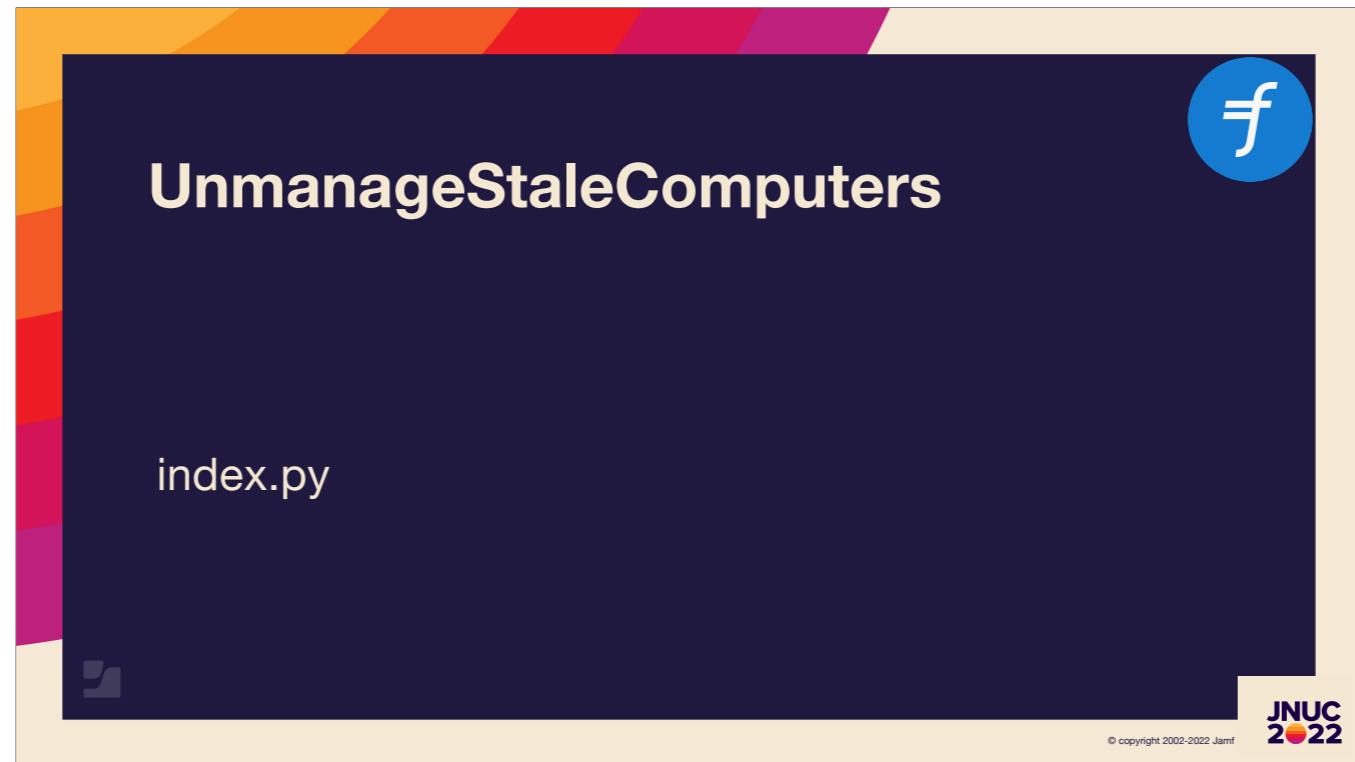


```
20     - True
21     - 'False'
22
23 Resources:
24   UnmanageComputer:
25     Type: AWS::Serverless::Function
26     Properties:
27       Runtime: python3.8
28       Handler: unmanage.lambda_handler
29       CodeUri: ./src
30       Timeout: 10
31       Environment:
32         Variables:
33           STAGE: !Ref STAGE
34           DEBUG: !Ref DEBUG
35           SendToTeams_URL: !ImportValue SendToTeams-URL
36   Policies:
37     - SSMPolicy:
38       ParameterName: !Sub ${STAGE}/JamfPro/Address
39     - SSMPolicy:
40       ParameterName: !Sub ${STAGE}/JamfPro/Accts/UnmanageComputers/*
41     - SSMPolicy:
42       ParameterName: !Sub ${STAGE}/Webhooks/MSTeams/TeamName/ChannelName
43     - SQSSendMessagePolicy:
44       QueueName: !ImportValue SendToTeams-QueueName
45   Events:
46     SQSEvent:
47       Type: SQS
48       Properties:
49         Queue: !GetAtt UnmanageComputers.Arn
50         BatchSize: 10
51         Enabled: true
52
53 UnmanageComputers:
54   Type: AWS::SQS::Queue
55
56 GetStaleComputers:
57   Type: AWS::Serverless::Function
```



JNUC  
2022

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

```
1 """
2 Given a Jamf Pro Advanced Search name, generate a list of computer IDs
3 and add them to an AWS SQS queue. The Advanced Search should contain
4 computers that have not checked in within a designated timeframe.
5 """
6
7 import os
8 import json
9 import urllib
10 import requests
11 import logging
12 import boto3
13 from botocore.exceptions import ClientError
14 from http.client import HTTPConnection
15
16
17 def api_get_advancedcomputersearch_by_name(name):
18     """
19     Retrieve saved search data given a group name.
20
21     :param name: String The name of a Jamf Pro Advanced Search
22     :return: JSON Object containing information about the request's get
23             action. If error, returns None.
24     """
25
```

JNUC  
2022

© 2002-2022 Jamf



```
102     return
103
104 computer_ids = get_computer_ids_from_json(json_result)
105 if computer_ids is None:
106     LOGGER.error('Failed to retrieve Computer IDs from the returned JSON.')
107     return
108 elif computer_ids == []:
109     LOGGER.info('No Computer IDs returned. The search appears to be empty.')
110     return
111
112 for computer_id in computer_ids:
113     LOGGER.info(f'Sending Computer ID \'{computer_id}\' to the queue to be unmanaged.')
114     send_to_sqs(sqs_queue_url, computer_id)
115
116
117 SOS = boto3.client('sns')
118 SSM = boto3.client('ssm')
119 STAGE = os.getenv("STAGE", "dev").lower()
120 DEBUG = os.getenv("DEBUG", "False").lower()
121
122 logging.basicConfig(format='%(levelname)s: %(asctime)s: %(message)s')
123 LOGGER = logging.getLogger(__name__)
124 if DEBUG == 'true':
125     LOGGER.setLevel(logging.DEBUG)
126     HTTPConnection.debuglevel = 1
127 else:
128     LOGGER.setLevel(logging.INFO)
129
130 API_URL = get_ssm_secret_value(f'/{STAGE}/JamfPro/Address')
131 API_USER = get_ssm_secret_value(f'/{STAGE}/JamfPro/Accts/UnmanageComputers/Username')
132 API_PASS = get_ssm_secret_value(f'/{STAGE}/JamfPro/Accts/UnmanageComputers>Password')
```



© copyright 2002-2022 Jamf

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.



```
52     return output
53
54
55 def get_ssm_secret_value(parameter_name):
56     """
57     Retreive a stored parameter from the AWS Systems Manager parameter store.
58
59     :param parameter_name: String The Parameter Key to retreive
60     :return: String containing the Parameter Value
61     """
62     return SSM.get_parameter(
63         Name=parameter_name,
64         WithDecryption=True
65     ).get("Parameter").get("Value")
66
67
68 def send_to_sqs(sqs_queue_url, message):
69     """
70     Publish a message to SQS.
71
72     :param sqs_queue_url: String URL of SQS Queue
73     :param message: String The message body
74     :return: Dictionary containing information about the sent message. If
75             error, returns None.
76     """
77     try:
78         response = SQS.send_message(
79             QueueUrl=sqs_queue_url,
80             MessageBody=str(message)
81         )
82     except ClientError as e:
83         LOGGER.error(e)
84         return None
85     return response
86
87
88 def lambda_handler(event, context):
```

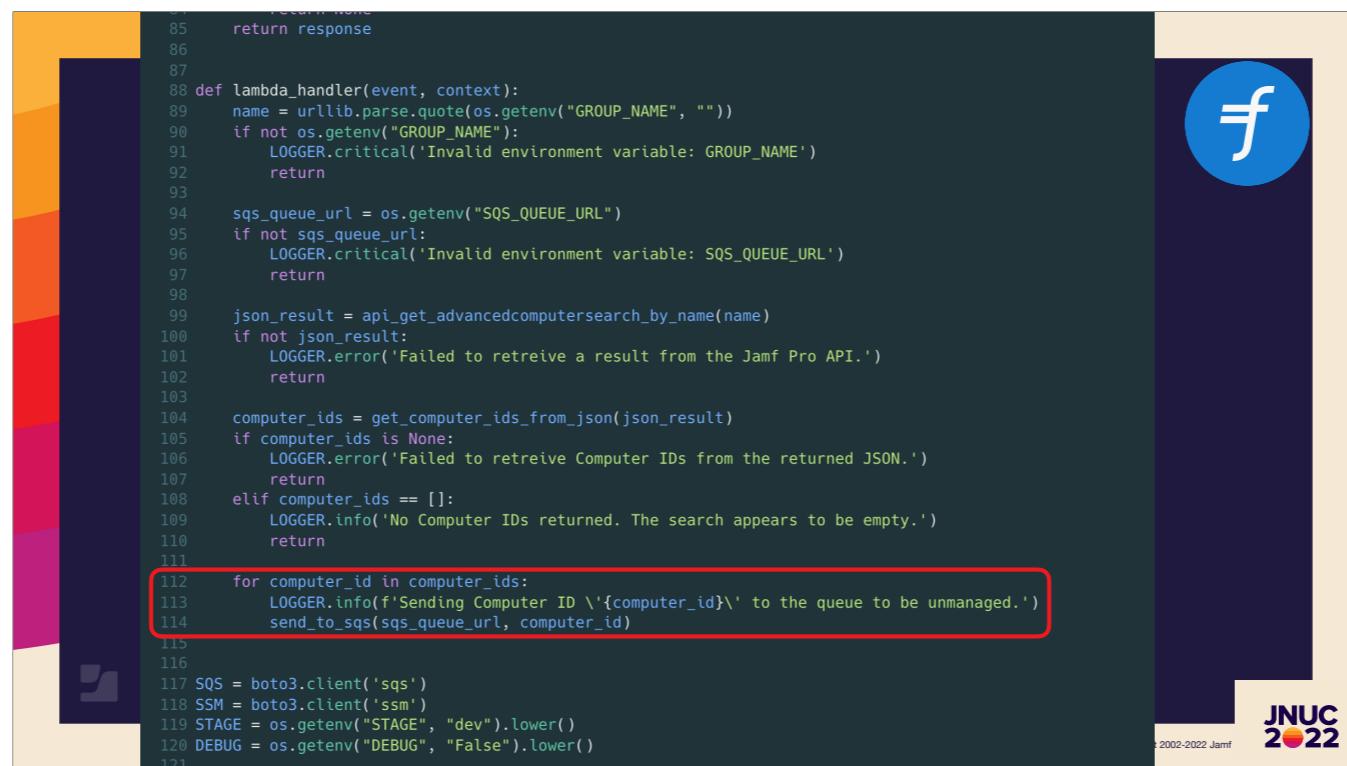
© 2002-2022 Jamf

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.



```
    return None
85    return response
86
87
88 def lambda_handler(event, context):
89     name = urllib.parse.quote(os.getenv("GROUP_NAME", ""))
90     if not os.getenv("GROUP_NAME"):
91         LOGGER.critical('Invalid environment variable: GROUP_NAME')
92         return
93
94     sqs_queue_url = os.getenv("SQS_QUEUE_URL")
95     if not sqs_queue_url:
96         LOGGER.critical('Invalid environment variable: SQS_QUEUE_URL')
97         return
98
99     json_result = api_get_advancedcomputersearch_by_name(name)
100    if not json_result:
101        LOGGER.error('Failed to retrieve a result from the Jamf Pro API.')
102        return
103
104    computer_ids = get_computer_ids_from_json(json_result)
105    if computer_ids is None:
106        LOGGER.error('Failed to retrieve Computer IDs from the returned JSON.')
107        return
108    elif computer_ids == []:
109        LOGGER.info('No Computer IDs returned. The search appears to be empty.')
110        return
111
112    for computer_id in computer_ids:
113        LOGGER.info(f'Sending Computer ID \'{computer_id}\' to the queue to be unmanaged.')
114        send_to_sqs(sqs_queue_url, computer_id)
115
116
117 SQS = boto3.client('sns')
118 SSM = boto3.client('ssm')
119 STAGE = os.getenv("STAGE", "dev").lower()
120 DEBUG = os.getenv("DEBUG", "False").lower()
121
```

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.



```
    return None
85    return response
86
87
88 def lambda_handler(event, context):
89     name = urllib.parse.quote(os.getenv("GROUP_NAME", ""))
90     if not os.getenv("GROUP_NAME"):
91         LOGGER.critical('Invalid environment variable: GROUP_NAME')
92         return
93
94     sqs_queue_url = os.getenv("SQS_QUEUE_URL")
95     if not sqs_queue_url:
96         LOGGER.critical('Invalid environment variable: SQS_QUEUE_URL')
97         return
98
99     json_result = api_get_advancedcomputersearch_by_name(name)
100    if not json_result:
101        LOGGER.error('Failed to retrieve a result from the Jamf Pro API.')
102        return
103
104    computer_ids = get_computer_ids_from_json(json_result)
105    if computer_ids is None:
106        LOGGER.error('Failed to retrieve Computer IDs from the returned JSON.')
107        return
108    elif computer_ids == []:
109        LOGGER.info('No Computer IDs returned. The search appears to be empty.')
110        return
111
112    for computer_id in computer_ids:
113        LOGGER.info(f'Sending Computer ID \'{computer_id}\' to the queue to be unmanaged.')
114        send_to_sqs(sqs_queue_url, computer_id)
115
116
117 SQS = boto3.client('sns')
118 SSM = boto3.client('ssm')
119 STAGE = os.getenv("STAGE", "dev").lower()
120 DEBUG = os.getenv("DEBUG", "False").lower()
121
```

© 2002-2022 Jamf  
JNUC 2022

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.



```
6
7 import os
8 import json
9 import urllib
10 import requests
11 import logging
12 import boto3
13 from botocore.exceptions import ClientError
14 from http.client import HTTPConnection
15
16
17 def api_get_advancedcomputersearch_by_name(name):
18     """
19         Retrieve saved search data given a group name.
20
21         :param name: String The name of a Jamf Pro Advanced Search
22         :return: JSON Object containing information about the request's get
23                 action. If error, returns None.
24     """
25     auth_tuple = (API_USER, API_PASS)
26     headers = { 'Accept': 'application/json' }
27     url = f'{API_URL}/JSSResource/advancedcomputersearches/name/{name}'
28
29     LOGGER.debug(f'URL generated: {url}')
30
31     try:
32         r = requests.get(url, auth=auth_tuple, headers=headers)
33     except requests.exceptions.RequestException as e:
34         LOGGER.error(e)
35     return None
36     return r.json()
37
38
39 def get_computer_ids_from_json(json_result):
40     """
41         Parse the Jamf Pro API json output and return a list of computer IDs.
42     """

```

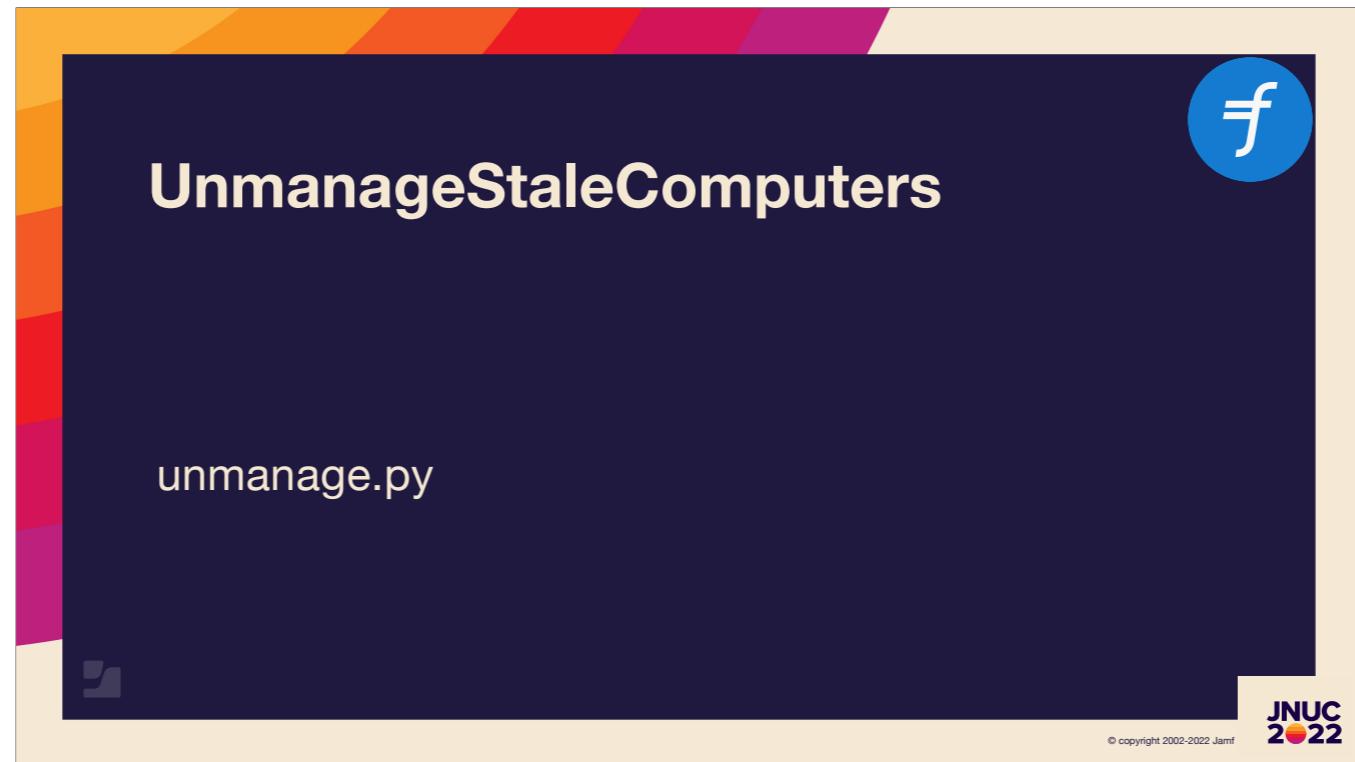


The Jamf Pro Classic API integration relies on the values from the secrets to make the connection, but this way keeps those values secure. Otherwise, this function is pretty standard stuff when you are working with a vendor API. Set some headers, define your URL, and invoke the requests module.



```
55     get_ssm_secret_value(parameter_name):
56     """
57     Retreive a stored parameter from the AWS Systems Manager parameter store.
58
59     :param parameter_name: String The Parameter Key to retreive
60     :return: String containing the Parameter Value
61     """
62     return SSM.get_parameter(
63         Name=parameter_name,
64         WithDecryption=True
65     ).get("Parameter").get("Value")
66
67
68 def send_to_sqs(sqs_queue_url, message):
69     """
70     Publish a message to SQS.
71
72     :param sqs_queue_url: String URL of SQS Queue
73     :param message: String The message body
74     :return: Dictionary containing information about the sent message. If
75             error, returns None.
76     """
77     try:
78         response = SQS.send_message(
79             QueueUrl=sqs_queue_url,
80             MessageBody=str(message)
81         )
82     except ClientError as e:
83         LOGGER.error(e)
84     return None
85     return response
86
87
88 def lambda_handler(event, context):
89     name = urllib.parse.quote(os.getenv("GROUP_NAME", ""))
90     if not os.getenv("GROUP_NAME"):
91         LOGGER.critical('Invalid environment variable: GROUP_NAME')
92     return
```

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.

# UnmanageStaleComputers

```
1 """
2 Accepts Jamf Pro Computer IDs from an AWS SQS queue and unmanages them.
3 The list of computers should include machines that have not checked in
4 within a designated timeframe.
5 """
6
7 import os
8 import json
9 import requests
10 import logging
11 import boto3
12 from botocore.exceptions import ClientError
13 from http.client import HTTPConnection
14
15
16 def api_get_request(url):
17     """
18     Make a GET request to the Jamf Pro API.
19
20     :param url: String The url to connect to.
21     :return: JSON Object containing information about the request's get
22             action. If error, returns None.
23     """
24     auth_tuple = (APT_USER, APT_PASS)
```

2002-2022 Jamf

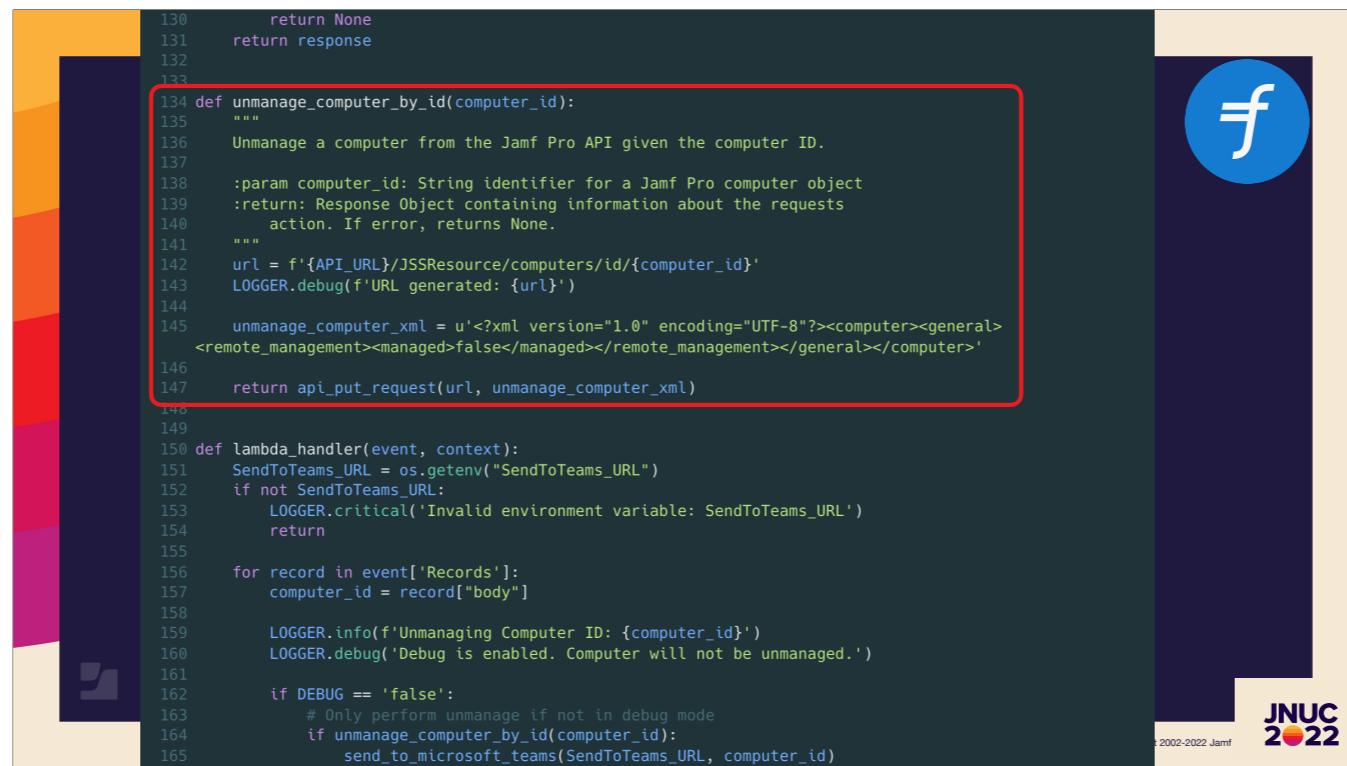
JNUC  
2022

This file is mostly the same stuff, but there are 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.



```
41     :return: Dictionary containing each cookie from the PUT request. If error, returns None.
42     """
43     auth_tuple = (API_USER, API_PASS)
44
45     try:
46         r = requests.put(url, auth=auth_tuple, data=data)
47         LOGGER.debug(str(r.content))
48     except requests.exceptions.RequestException as e:
49         LOGGER.error(e)
50         return None
51
52     return r.cookies
53
54
55 def get_computer_name_by_id(computer_id):
56     """
57     Get a computer record from a Jamf Pro API given the computer ID.
58
59     :param computer_id: String identifier for a Jamf Pro computer object
60     :return: Response Object containing information about the requests get
61             action. If error, returns None.
62     """
63     url = f'{API_URL}/JSSResource/computers/id/{computer_id}/subset/General'
64     LOGGER.debug(f'URL generated: {url}')
65
66     computer_record = api_get_request(url)
67
68     return computer_record["computer"]["general"]["name"]
69
70
71 def get_ssm_secret_value(parameter_name):
72     """
73     Retreive a stored parameter from the AWS Systems Manager parameter store.
74
75     :param paramater_name: String The Parameter Key to retreive
76     :return: String containing the Parameter Value
77     """
```

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.



```
130     return None
131     return response
132
133
134 def unmanage_computer_by_id(computer_id):
135     """
136     Unmanage a computer from the Jamf Pro API given the computer ID.
137
138     :param computer_id: String identifier for a Jamf Pro computer object
139     :return: Response Object containing information about the requests
140             action. If error, returns None.
141     """
142     url = f'{API_URL}/JSSResource/computers/id/{computer_id}'
143     LOGGER.debug(f'URL generated: {url}')
144
145     unmanage_computer_xml = u'<?xml version="1.0" encoding="UTF-8"?><computer><general>
146         <remote_management><managed>false</managed></remote_management></general></computer>'
147
148     return api_put_request(url, unmanage_computer_xml)
149
150 def lambda_handler(event, context):
151     SendToTeams_URL = os.getenv("SendToTeams_URL")
152     if not SendToTeams_URL:
153         LOGGER.critical('Invalid environment variable: SendToTeams_URL')
154     return
155
156     for record in event['Records']:
157         computer_id = record["body"]
158
159         LOGGER.info(f'Unmanaging Computer ID: {computer_id}')
160         LOGGER.debug('Debug is enabled. Computer will not be unmanaged.')
161
162         if DEBUG == 'false':
163             # Only perform unmanage if not in debug mode
164             if unmanage_computer_by_id(computer_id):
165                 send_to_microsoft_teams(SendToTeams_URL, computer_id)
```



© 2002-2022 Jamf

The `unmanage_computer_by_id` function defines the XML necessary to unmanage 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.

```
153     LOGGER.critical('Invalid environment variable: SendToTeams_URL')
154     return
155
156     for record in event['Records']:
157         computer_id = record["body"]
158
159         LOGGER.info(f'Unmanaging Computer ID: {computer_id}')
160         LOGGER.debug('Debug is enabled. Computer will not be unmanaged.')
161
162         if DEBUG == 'false':
163             # Only perform unmanage if not in debug mode
164             if unmanage_computer_by_id(computer_id):
165                 send_to_microsoft_teams(SendToTeams_URL, computer_id)
166
167
168 SOS = boto3.client('sns')
169 SSM = boto3.client('ssm')
170 STAGE = os.getenv("STAGE", "dev").lower()
171 DEBUG = os.getenv("DEBUG", "False").lower()
172
173 logging.basicConfig(format='%(levelname)s: %(asctime)s: %(message)s')
174 LOGGER = logging.getLogger(__name__)
175 if DEBUG == 'true':
176     LOGGER.setLevel(logging.DEBUG)
177     HTTPConnection.debuglevel = 1
178 else:
179     LOGGER.setLevel(logging.INFO)
180
181 API_URL = get_ssm_secret_value(f'/{STAGE}/JamfPro/Address')
182 API_USER = get_ssm_secret_value(f'/{STAGE}/JamfPro/Accts/UnmanageComputers/Username')
183 API_PASS = get_ssm_secret_value(f'/{STAGE}/JamfPro/Accts/UnmanageComputers>Password')
```



JNUC  
2022

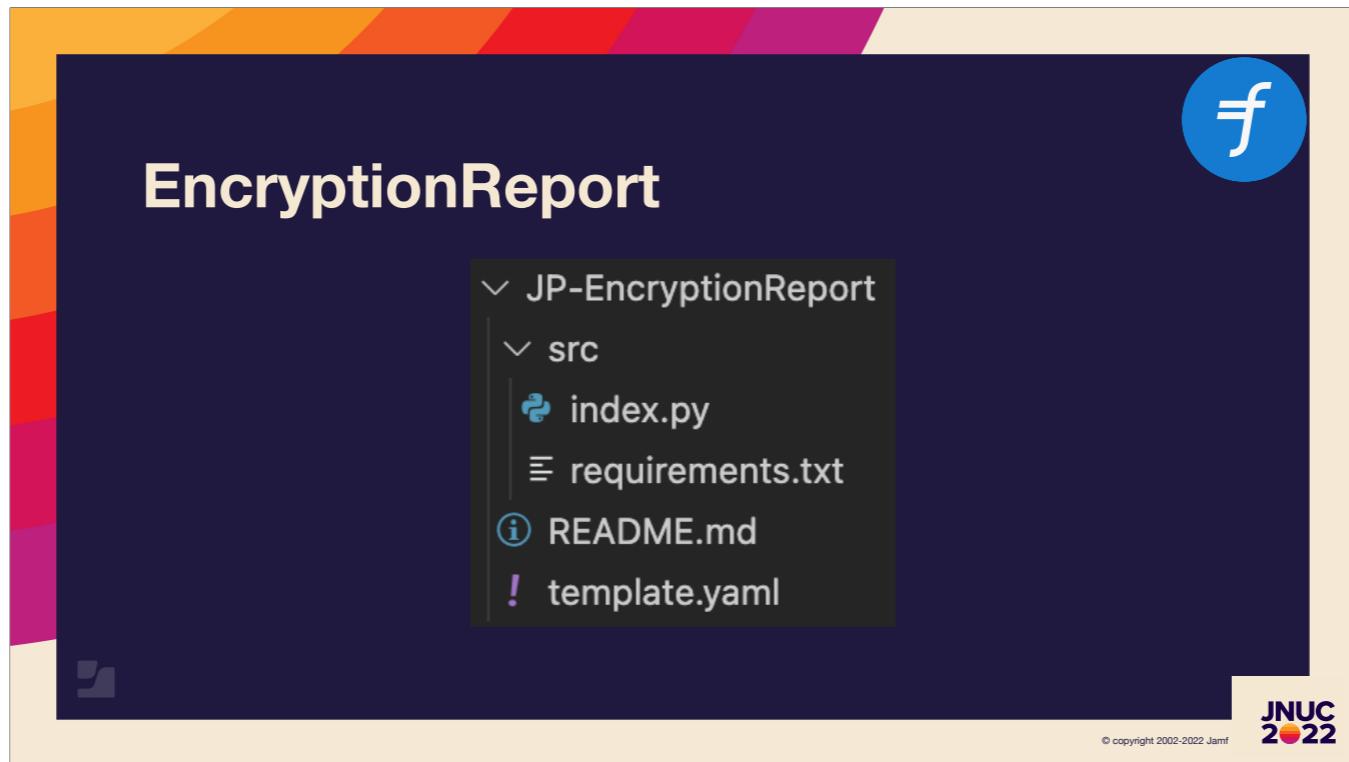
The `unmanage_computer_by_id` function defines the XML necessary to unmanage a computer in Jamf Pro. It takes a Computer ID as input and sends that XML to the [API](#). If it fails, it doesn't send the message to Teams.



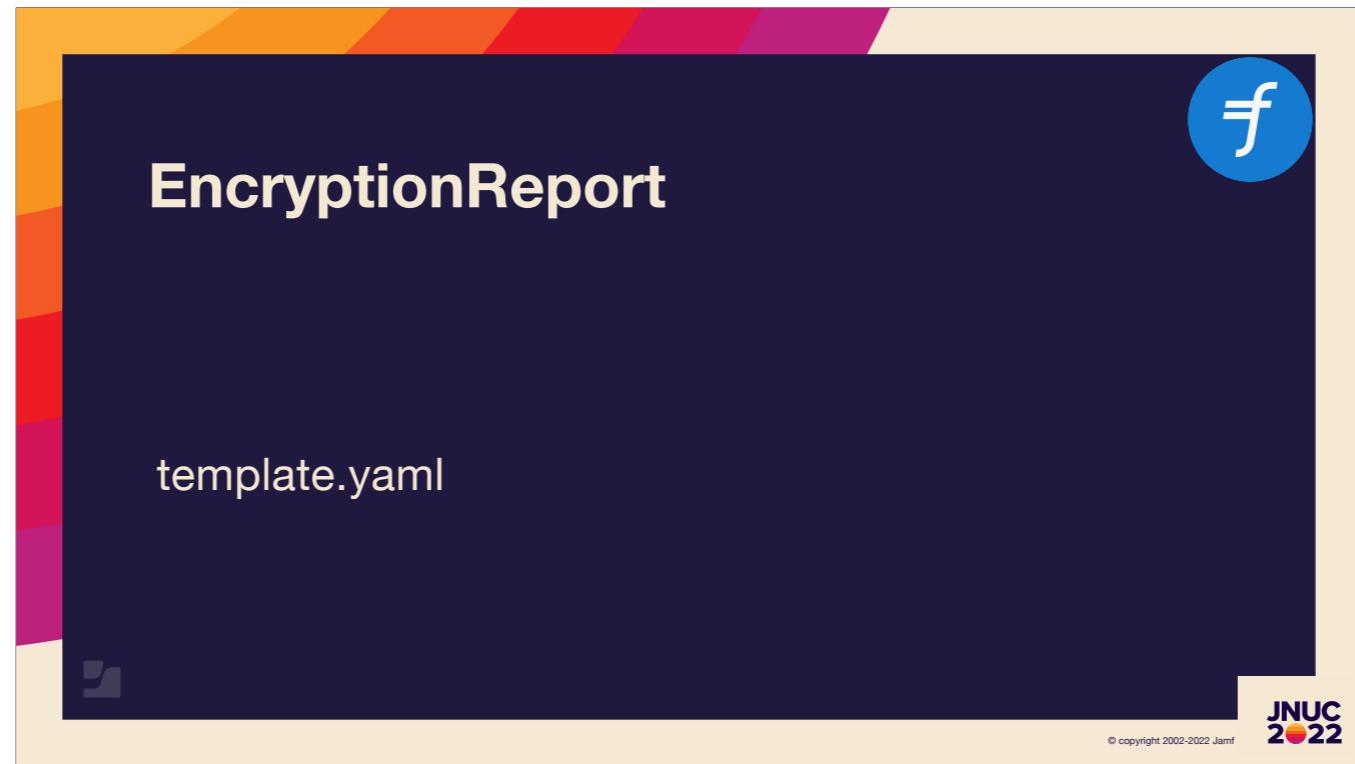
Have you ever wanted to run a report in Jamf Pro, but it wasn't quite able to do what you needed? I had a situation where I needed to share a report with my Security team that was split by Site. In Jamf Pro, site isn't something you can add to the scope, so I had to script a [solution](#).



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](#).

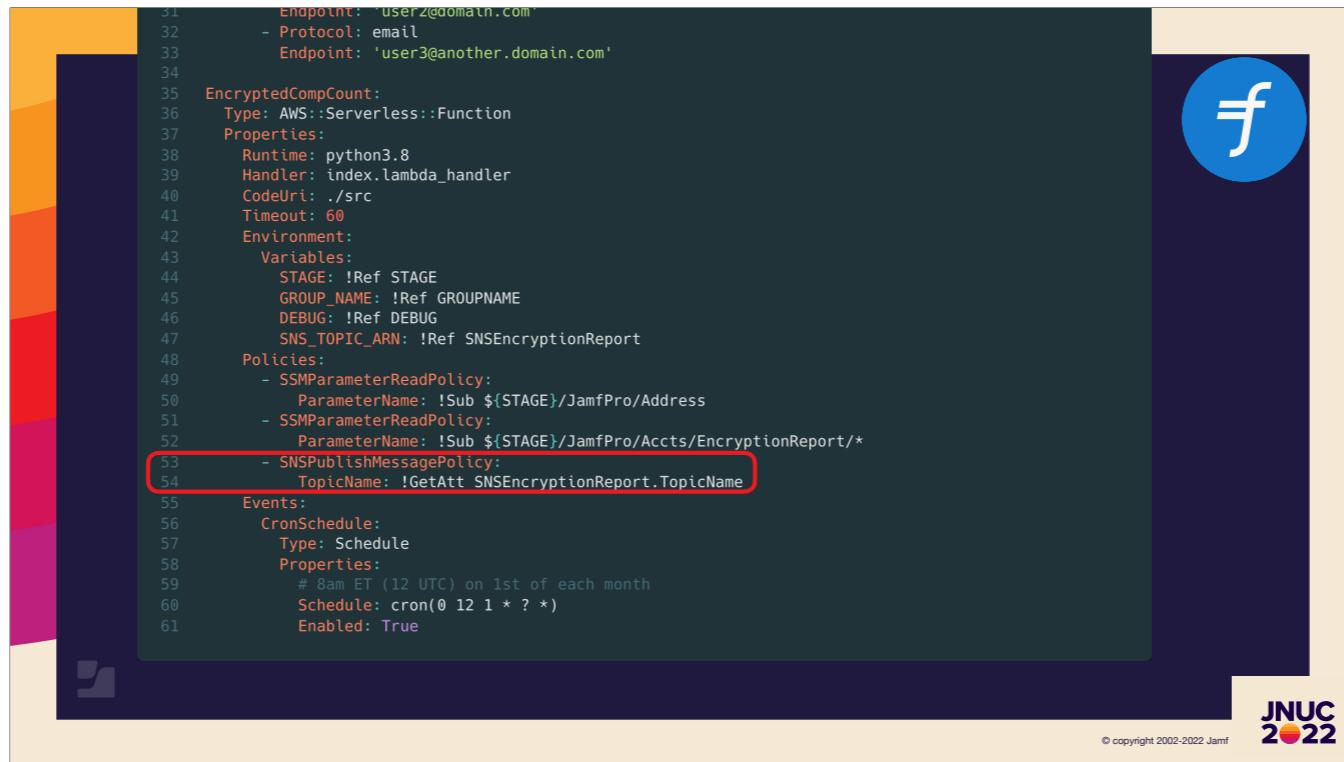


```
31     Endpoint: 'user2@domain.com'
32   - Protocol: email
33     Endpoint: 'user3@another.domain.com'
34
35 EncryptedCompCount:
36   Type: AWS::Serverless::Function
37   Properties:
38     Runtime: python3.8
39     Handler: index.lambda_handler
40     CodeUri: ./src
41     Timeout: 60
42   Environment:
43     Variables:
44       STAGE: !Ref STAGE
45       GROUP_NAME: !Ref GROUPNAME
46       DEBUG: !Ref DEBUG
47       SNS_TOPIC_ARN: !Ref SNSEncryptionReport
48   Policies:
49     - SSMPParameterReadPolicy:
50       ParameterName: !Sub ${STAGE}/JamfPro/Address
51     - SSMPParameterReadPolicy:
52       ParameterName: !Sub ${STAGE}/JamfPro/Accts/EncryptionReport/*
53     - SNSPublishMessagePolicy:
54       TopicName: !GetAtt SNSEncryptionReport.TopicName
55   Events:
56     CronSchedule:
57       Type: Schedule
58       Properties:
59         # 8am ET (12 UTC) on 1st of each month
60         Schedule: cron(0 12 1 * ? *)
61         Enabled: True
```



© copyright 2002-2022 Jamf

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



```
31     Endpoint: 'user2@domain.com'
32   - Protocol: email
33     Endpoint: 'user3@another.domain.com'
34
35 EncryptedCompCount:
36   Type: AWS::Serverless::Function
37   Properties:
38     Runtime: python3.8
39     Handler: index.lambda_handler
40     CodeUri: ./src
41     Timeout: 60
42   Environment:
43     Variables:
44       STAGE: !Ref STAGE
45       GROUP_NAME: !Ref GROUPNAME
46       DEBUG: !Ref DEBUG
47       SNS_TOPIC_ARN: !Ref SNSEncryptionReport
48   Policies:
49     - SSMPParameterReadPolicy:
50       ParameterName: !Sub ${STAGE}/JamfPro/Address
51     - SSMPParameterReadPolicy:
52       ParameterName: !Sub ${STAGE}/JamfPro/Accts/EncryptionReport/*
53     - SNSPublishMessagePolicy:
54       TopicName: !GetAtt SNSEncryptionReport.TopicName
55   Events:
56     CronSchedule:
57       Type: Schedule
58       Properties:
59         # 8am ET (12 UTC) on 1st of each month
60         Schedule: cron(0 12 1 * ? *)
61         Enabled: True
```

© copyright 2002-2022 Jamf  
JNUC 2022

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.



```
8   Default: dev
9   AllowedValues:
10    - dev
11    - prod
12 GROUPNAME:
13   Description: 'Required. The name of an Advanced Search within Jamf Pro.'
14   Type: String
15 DEBUG:
16   Description: 'Optional. Enable debug logging.'
17   Type: String
18   Default: 'False'
19   AllowedValues:
20    - 'True'
21    - 'False'
22
23 Resources:
24 SNSEncryptionReport:
25   Type: AWS::SNS::Topic
26   Properties:
27     Subscription:
28      - Protocol: email
29        Endpoint: 'user1@domain.com'
30      - Protocol: email
31        Endpoint: 'user2@domain.com'
32      - Protocol: email
33        Endpoint: 'user3@another.domain.com'
34
35 EncryptedCompCount:
36   Type: AWS::Serverless::Function
37   Properties:
38     Runtime: python3.8
39     Handler: index.lambda_handler
40     CodeUri: ./src
41     Timeout: 60
42     Environment:
43       Variables:
44         STAGE: !Ref STAGE
```

© 2002-2022 Jamf

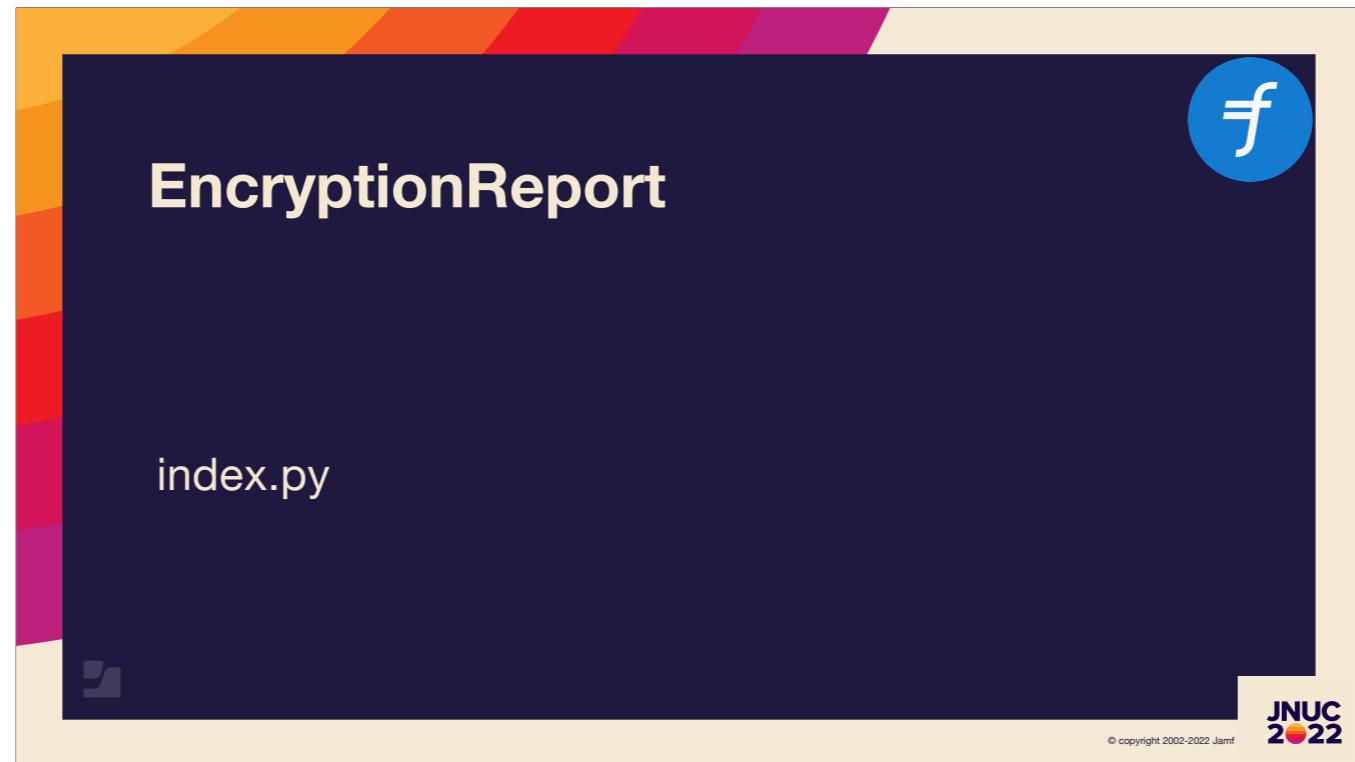
Just make sure it matches this.



```
8  Default: dev
9  AllowedValues:
10 - dev
11 - prod
12 GROUPNAME:
13 Description: 'Required. The name of an Advanced Search within Jamf Pro.'
14 Type: String
15 DEBUG:
16 Description: 'Optional. Enable debug logging.'
17 Type: String
18 Default: 'False'
19 AllowedValues:
20 - 'True'
21 - 'False'
22
23 Resources:
24 SNSEncryptionReport:
25   Type: AWS::SNS::Topic
26   Properties:
27     Subscription:
28       - Protocol: email
29         Endpoint: 'user1@domain.com'
30       - Protocol: email
31         Endpoint: 'user2@domain.com'
32       - Protocol: email
33         Endpoint: 'user3@another.domain.com'
34
35 EncryptedCompCount:
36   Type: AWS::Serverless::Function
37   Properties:
38     Runtime: python3.8
39     Handler: index.lambda_handler
40     CodeUri: ./src
41     Timeout: 60
42     Environment:
43       Variables:
44         STAGE: !Ref STAGE
```

© 2002-2022 Jamf

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.



```
11 import boto3
12 from botocore.exceptions import ClientError
13
14
15 def api_get_sites():
16     """
17     Get a dictionary listing each site from a Jamf Pro API.
18
19     :return: Dictionary containing each site ID and Name. If error, returns None.
20     """
21     auth_tuple = (API_USER, API_PASS)
22     url = f'{API_URL}/JSSResource/sites'
23     LOGGER.debug(f'URL generated: {url}')
24
25     try:
26         xml = requests.get(url, auth=auth_tuple).content
27         LOGGER.debug(f'XML Returned: {xml}')
28     except requests.exceptions.RequestException as e:
29         LOGGER.error(e)
30         return None
31
32     try:
33         sites = ElementTree.fromstring(xml).findall('site')
34         LOGGER.debug(f'Sites list: {sites}')
35     except ElementTree.ParseError as e:
36         LOGGER.error(e)
37         return None
38
39     output = {}
40     for site in sites:
41         output[site.find('id').text] = site.find('name').text
42
43     LOGGER.debug(f'api_get_sites() output: {output}')
44     return output
45
46
47 def api_put_request(url, data):
48     """
49     Put request to Jamf Pro API
50
51     :param url: URL to put to
52     :param data: Data to put
53     :return: Response object
54     """
55     auth_tuple = (API_USER, API_PASS)
56     headers = {'Content-Type': 'application/json'}
57
58     response = requests.put(url, auth=auth_tuple, json=data, headers=headers)
59
60     if response.status_code == 200:
61         return response
62     else:
63         LOGGER.error(f'PUT request failed with status code {response.status_code}: {response.text}')
64         return None
```



© 2002-2022 Jamf

JNUC  
2022

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.



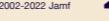
```
64
65
66 def get_encrypted_count_by_site_id(site_id):
67     """
68     Gets the number of encrypted devices within a given site.
69
70     :param site_id: Integer The ID of a Jamf Pro site.
71     :return: Integer containing the number of computers that are encrypted. If error, returns None.
72     """
73     change_site_xml = f'<advanced_computer_search><site><id>{site_id}</id></site>
74     </advanced_computer_search>'
75
76     auth_tuple = (API_USER, API_PASS)
77     name = urllib.parse.quote(os.getenv("GROUP_NAME", ""))
78
79     url = f'{API_URL}/JSSResource/advancedcomputersearches/name/{name}'
80     LOGGER.debug(f'URL generated: {url}')
81
82     r = api_put_request(url, change_site_xml)
83     cookies = dict(APBALANCEID=r.get('APBALANCEID'))
84
85     try:
86         xml = requests.get(url, auth=auth_tuple, cookies=cookies).content
87     except requests.exceptions.RequestException as e:
88         LOGGER.error(e)
89         return None
90
91     try:
92         encrypted_count = ElementTree.fromstring(xml).findtext('computers/size')
93     except ElementTree.ParseError as e:
94         LOGGER.debug('Value of the \'xml\' parameter: '+str(xml))
95         LOGGER.error(e)
96         return None
97
98     return encrypted_count
99
```

© 2002-2022 Jamf

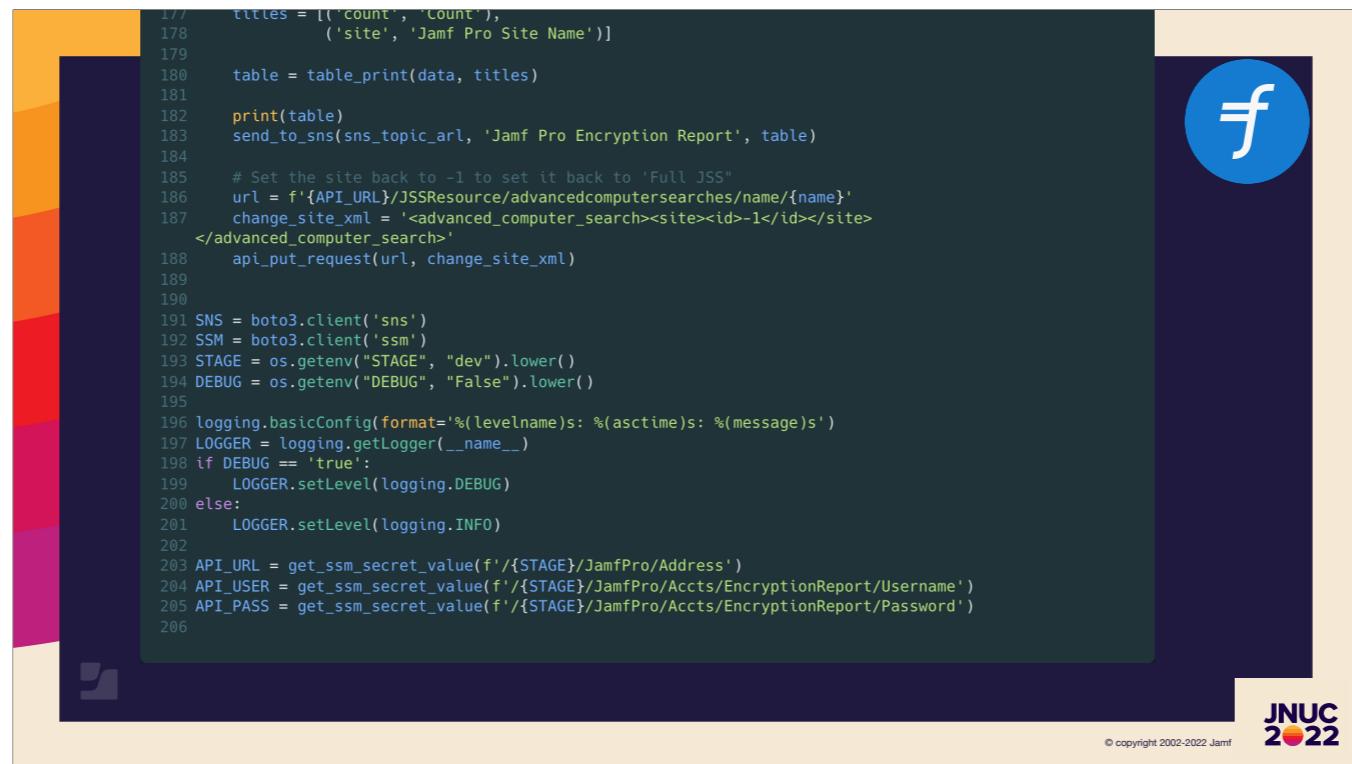
It then loops through each site and calls this function. It changes the Site ID on a specific Advanced Search, gets the number of computers returned, and then moves onto the next site.



```
125     LOGGER.error(e)
126     return None
127     return response
128
129
130 def table_print(data, title_row):
131     """
132     :param data: list of dicts,
133     :param title_row: e.g. [('name', 'Programming Language'), ('type', 'Language Type')]
134     :return: String containing an ASCII table view of the given data.
135     """
136     max_widths = {}
137     data_copy = [dict(title_row)] + list(data)
138     for col in data_copy[0].keys():
139         max_widths[col] = max([len(str(row[col])) for row in data_copy])
140     cols_order = [tuple[0] for tuple in title_row]
141     underline = '-' * max_widths[col] for col in cols_order)
142
143     def custom_just(col, value):
144         if type(value) == int:
145             return str(value).rjust(max_widths[col])
146         else:
147             return value.ljust(max_widths[col])
148
149     output = f'{underline}{os.linesep}'
150     for row in data_copy:
151         row_str = ' | '.join([custom_just(col, row[col]) for col in cols_order])
152         output += f'{row_str}{os.linesep}'
153         if data_copy.index(row) == 0:
154             output += f'{underline}{os.linesep}'
155         output += f'{underline}{os.linesep}'
156
157     return output
158
159
160 def lambda_handler(event, context):
161     name = urllib.parse.quote(os.getenv("GROUP_NAME", ""))
```



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.

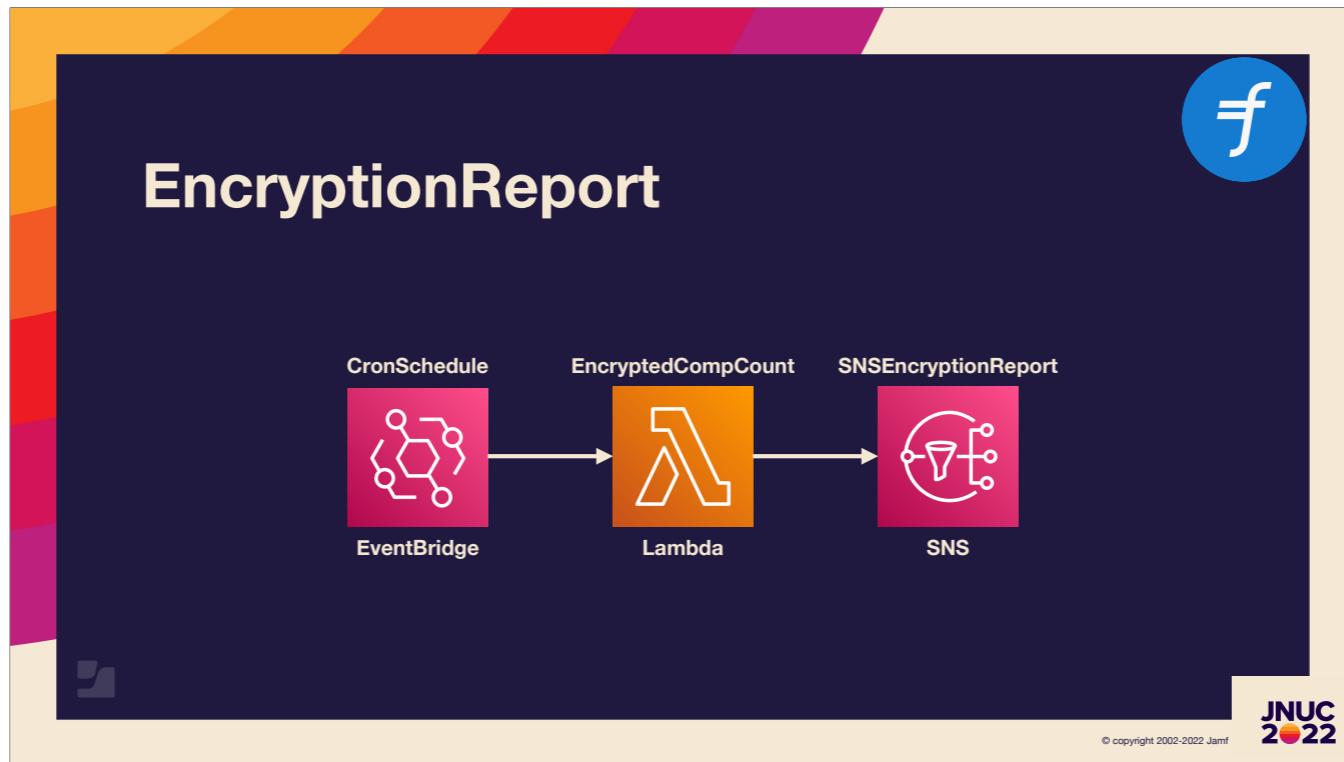


```
177     titles = [('count', 'Count'),
178                ('site', 'Jamf Pro Site Name')]
179
180     table = table_print(data, titles)
181
182     print(table)
183     send_to_sns(sns_topic_arn, 'Jamf Pro Encryption Report', table)
184
185     # Set the site back to -1 to set it back to 'Full JSS'
186     url = f'{API_URL}/JSSResource/advancedcomputersearches/name/{name}'
187     change_site_xml = '<advanced_computer_search><site><id>-1</id></site>' \
188         '</advanced_computer_search>'
189     api_put_request(url, change_site_xml)
190
191 SNS = boto3.client('sns')
192 SSM = boto3.client('ssm')
193 STAGE = os.getenv("STAGE", "dev").lower()
194 DEBUG = os.getenv("DEBUG", "False").lower()
195
196 logging.basicConfig(format='%(levelname)s: %(asctime)s: %(message)s')
197 LOGGER = logging.getLogger(__name__)
198 if DEBUG == 'true':
199     LOGGER.setLevel(logging.DEBUG)
200 else:
201     LOGGER.setLevel(logging.INFO)
202
203 API_URL = get_ssm_secret_value(f'/{STAGE}/JamfPro/Address')
204 API_USER = get_ssm_secret_value(f'/{STAGE}/JamfPro/Accts/EncryptionReport/Username')
205 API_PASS = get_ssm_secret_value(f'/{STAGE}/JamfPro/Accts/EncryptionReport>Password')
206
```



© copyright 2002-2022 Jamf

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

© copyright 2002-2022 Jamf



## 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

© copyright 2002-2022 Jamf



## Boto3 Resources

- [AWS SDK for Python \(Boto3\)](#)
- [Boto3 Documentation](#)
- [Boto3 API Reference](#)
- [Code Examples](#)

JNUC  
2022

© copyright 2002-2022 Jamf



## Other Resources

- [Lambda runtimes](#)
- [Office 365 connector card](#)
- [AWS Schedule Rules](#)

JNUC  
2022

© copyright 2002-2022 Jamf



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.