Diving into AWS Lambda

An Intro to Serverless for Admins

Penn State MacAdmins 2018

jamf

"Bryson Tyrrell"

Systems Development Engineer II

Jamf Cloud Engineering

@bryson3Gps

@brysontyrrell

Diving into AWS Lambda

An Intro to Serverless for Admins

Today, I’m going to be discussing Lambda - a part of the serverless stack available from Amazon Web Services.

Agenda

- What is Serverless? What is Lambda?
- What AWS services work with Lambda?
- Templating your functions.
- How do IAM permissions work?
- Example code.
- Packaging and deploying your functions.
- Debugging and troubleshooting.
- Q&A

What is serverless, and what part does Lambda play in this stack?
Lambda is a part of serverless, so what other services integrate with it?
How do we template these apps using CloudFormation?
IAM
Examples
How do we package and deploy these apps?
Once deployed, how do we troubleshoot when things aren’t working?

What is Serverless?

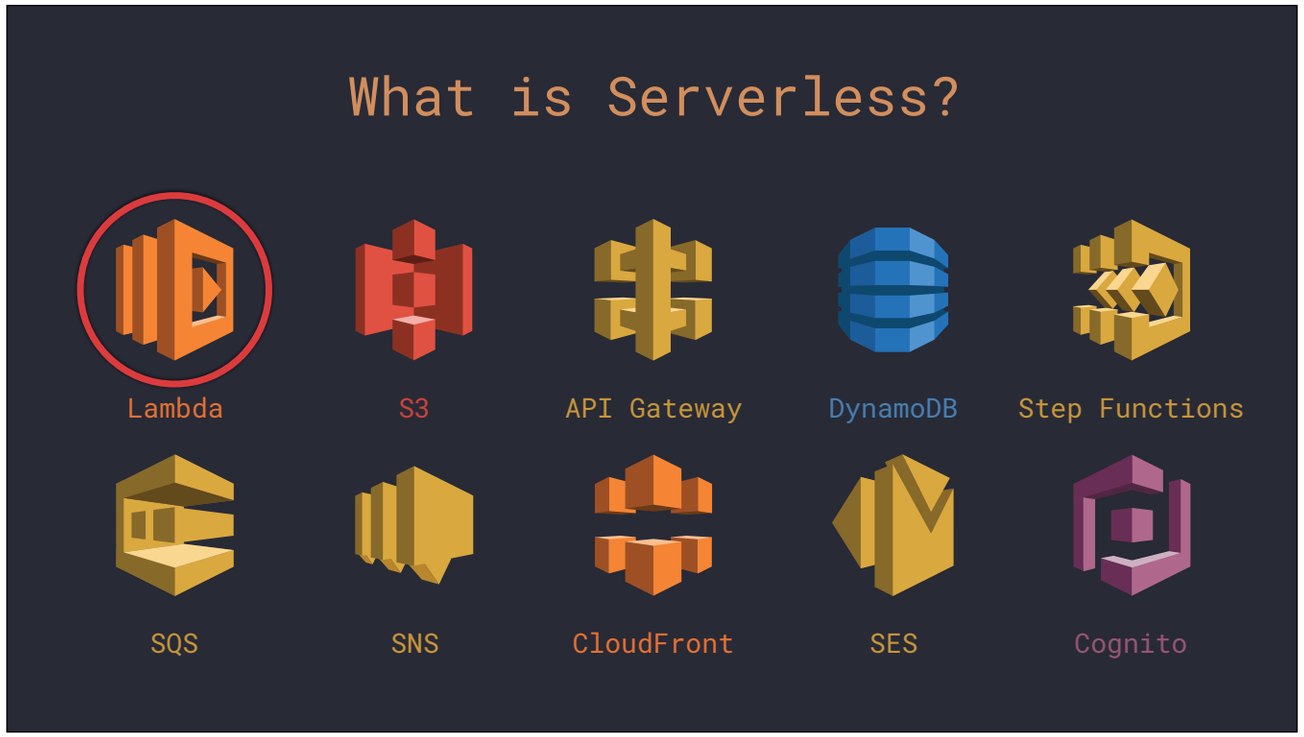
The AWS definition of serverless:

- No servers to provision
- Scales with usage
- Never pay for idle
- Highly available, highly durable

Highly available and highly durable. AWS is broken up into regions, and within those regions are availability zones. AZs provide resilency within a region. If one goes down, your service continues to work as long as it’s deployed in another. Serverless resources exist at a regional level and run in all availability zones.

What is Serverless?

Serverless abstracts the infrastructure from your code.



AWS services are grouped into categories in the console.
S3 is a Storage service.
DynamoDB is a Database service.
API Gateway and CloudFront are Networking services.
Step Functions, SQS, and SNS are Integration services.
SES is a Customer Engatement service.
Cognito is an Identity service.
Lambda is a Computer service: the same group as EC2.

What is Lambda?

Running code without servers

Also known as FaaS
(Functions as a Service)

Microsoft and Google both have their own versions of FaaS.

What is Lambda?

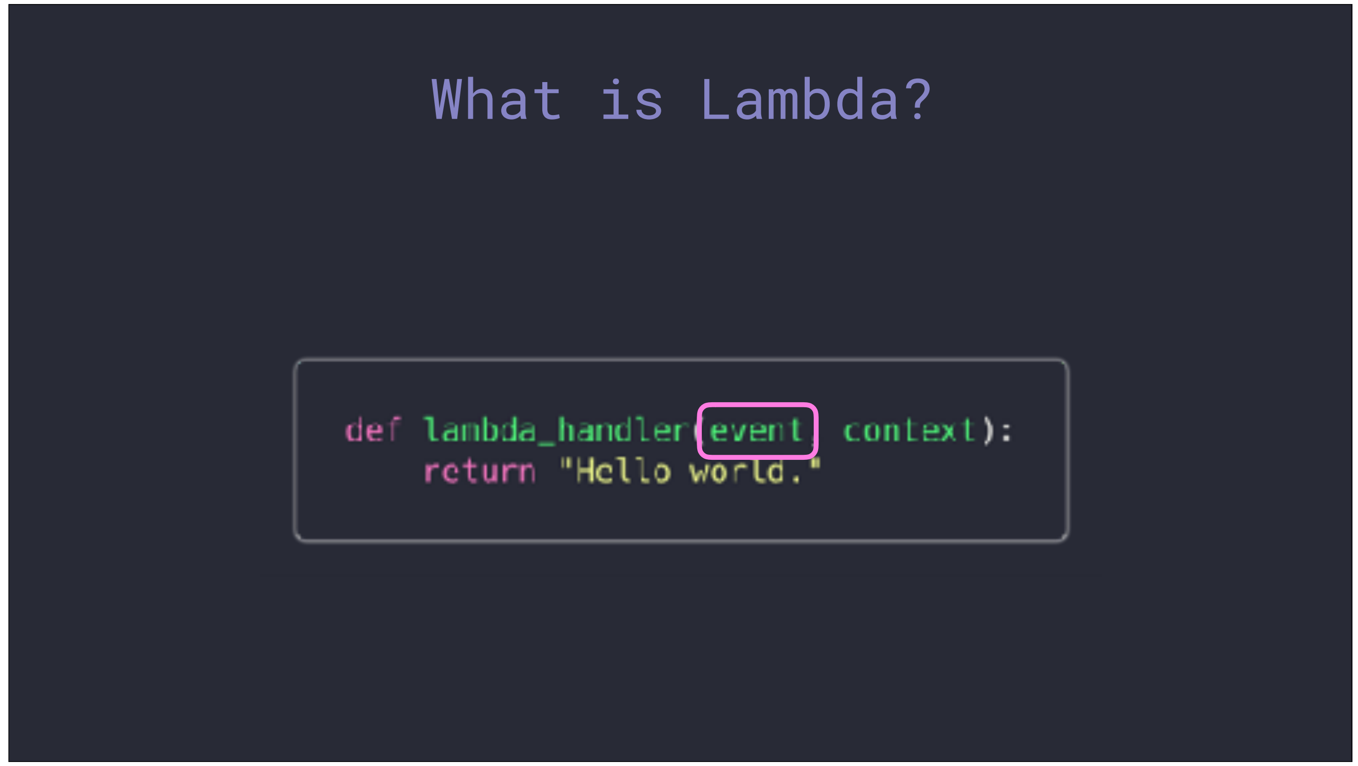
Supported Languages:

- C# .NET Core 1.0 / 2.0
- Go 1.x
- Java 8
- Node.js 4.3 / 6.10 / 8.10
- Python 2.7 / 3.6

What is Lambda?

```
def lambda_handler(event, context):
    return "Hello world."
```

This is the most basic version of a Lambda in Python.
When invoked, Lambda will execute a function within your code - called a handler.
In examples you'll normally see the function labeled 'lambda_handler'.



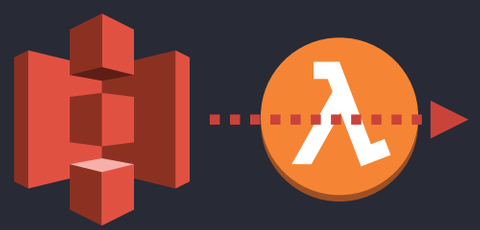
The two arguments to the function are passed by Lambda. The ‘event’ contains a dictionary payload with details about the event that invoked the function.

What is Lambda?

```
def lambda_handler(event, context):
    return "Hello world."
```

The ‘context’ contains an object with details about the runtime environment - like the amount of execution time remaining. We’ll be covering the ‘event’ object more later.

What Services work with Lambda?

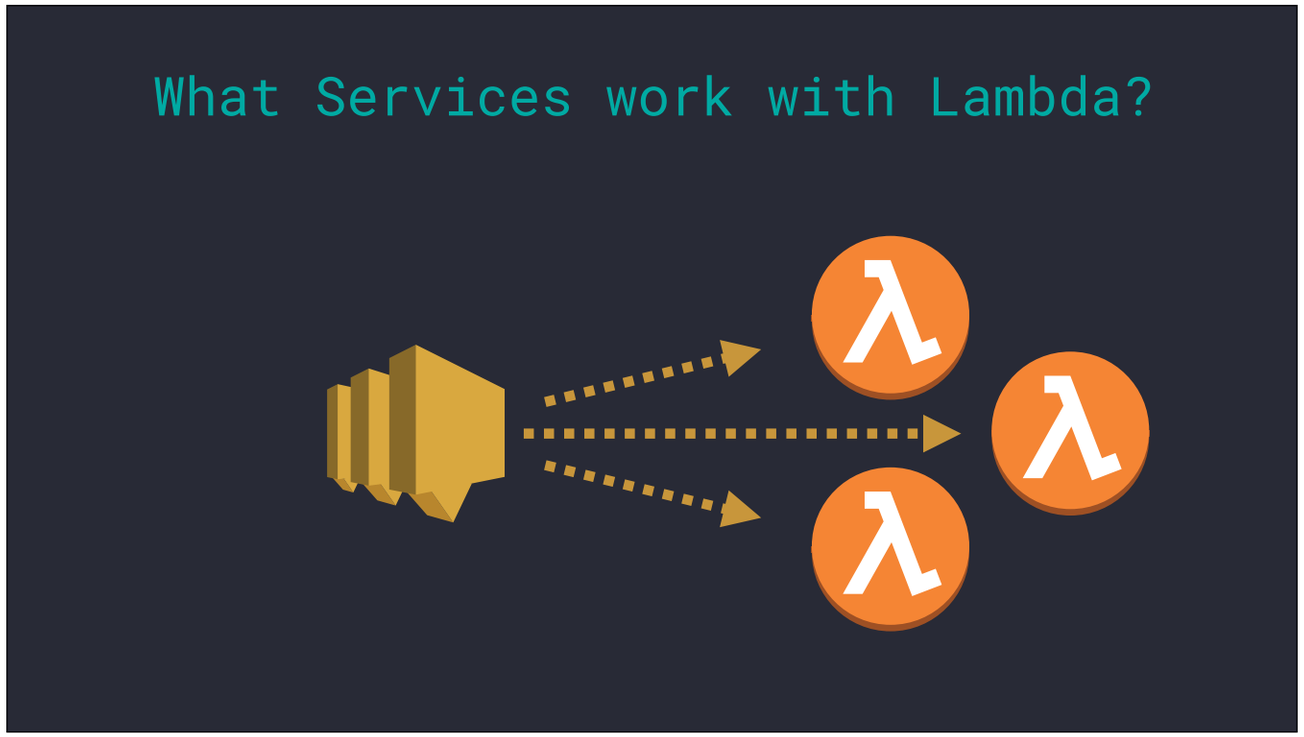


Lambdas can be executed manually, but they are really meant to be triggered off events from a number of AWS services.

S3: file events like creation and deletion.

What Services work with Lambda?

API Gateway allows you to create an HTTP endpoint you can make requests to and invoke your function.



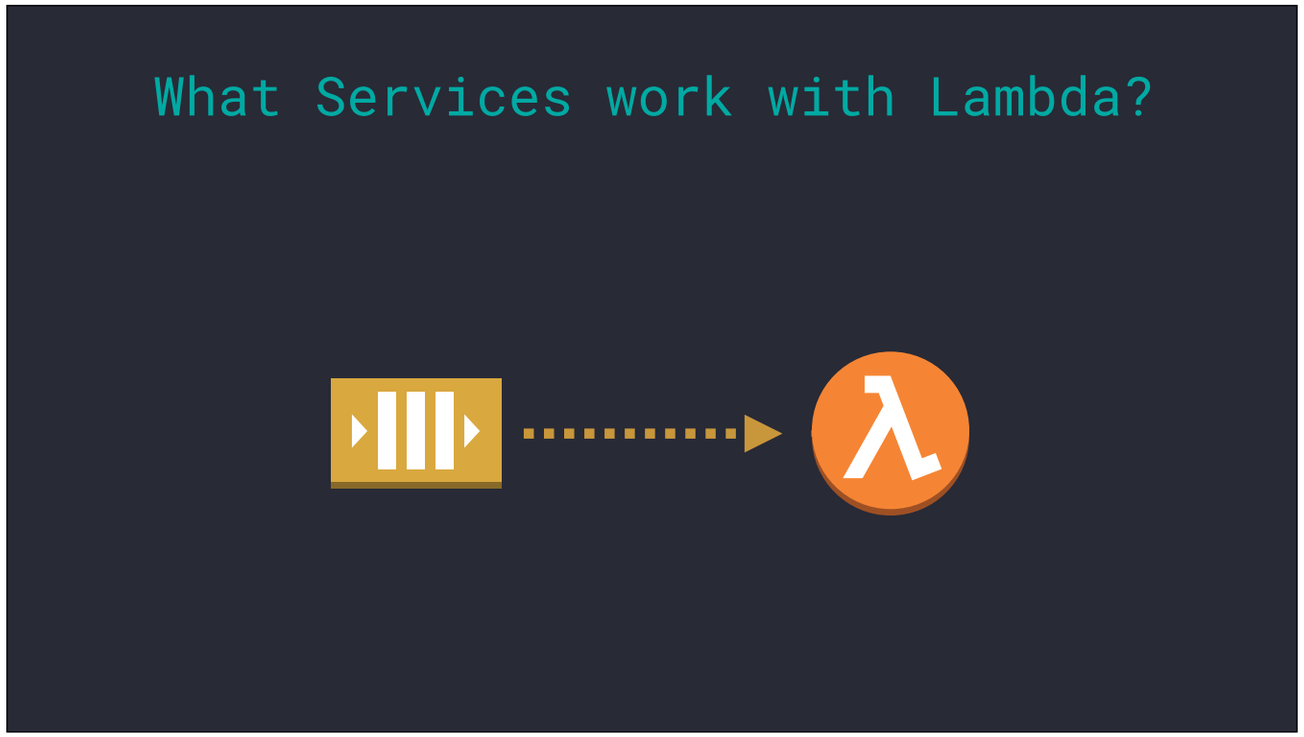
SNS, or simple notification service, is a one-to-many type of event. If you have multiple Lambdas that should execute in parallel off a single event, thing employee lifecycle actions, you can put them behind SNS and trigger all with a single call.

What Services work with Lambda?

CloudWatch has different methods for triggering Lambda functions.

CloudWatch events can execute Lambdas on a variety of different events from other AWS services. You could, for example, trigger automation on EC2 instance changes, or providing a trigger on health check failures.

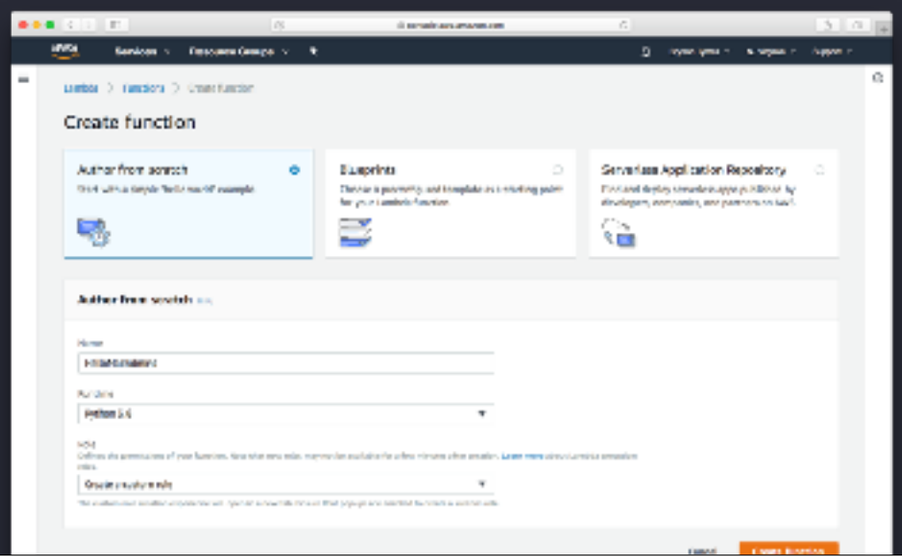
Schedules provide the ability to execute your Lambdas like a cron job



And now, as of just a couple weeks ago, we have SQS - Simple Queue Service - as an event source.

Before this, if you wanted to use a queue with your Lambdas, you had to write a poller to periodically read the queue and then start launching your Lambdas to process those messages. Now, as soon as messages drop onto the queue, the service will start invoking the associated Lambdas. No need to poll!

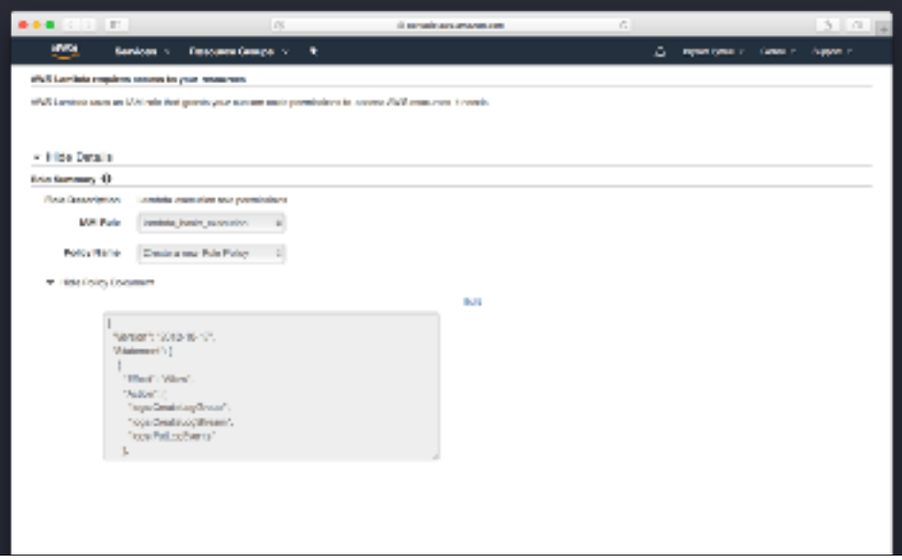
Creating Lambda Functions



If you've explored using Lambda functions before, you might be familiar with creating one in the AWS console.

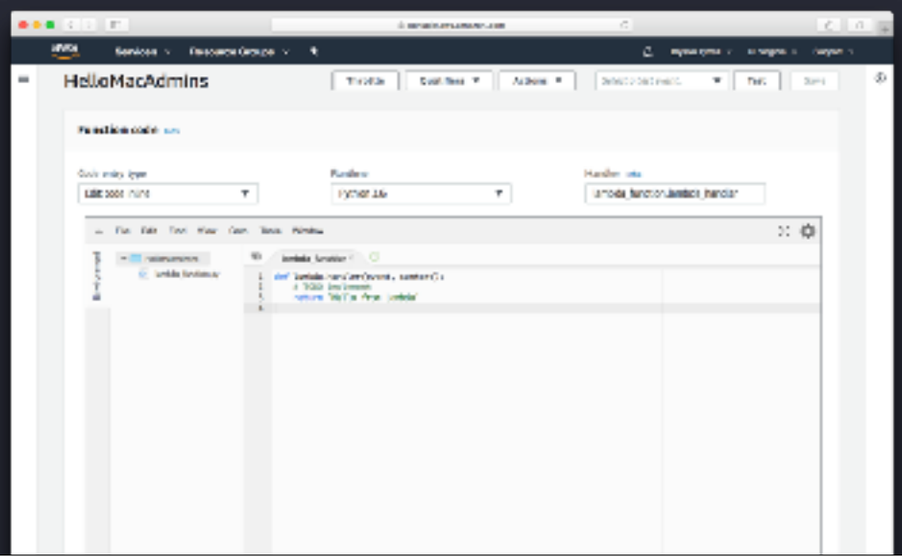
You need to create a new Lambda object either from scratch, meaning completely empty, or by using a pre-existing blueprint as a starting base.

Creating Lambda Functions



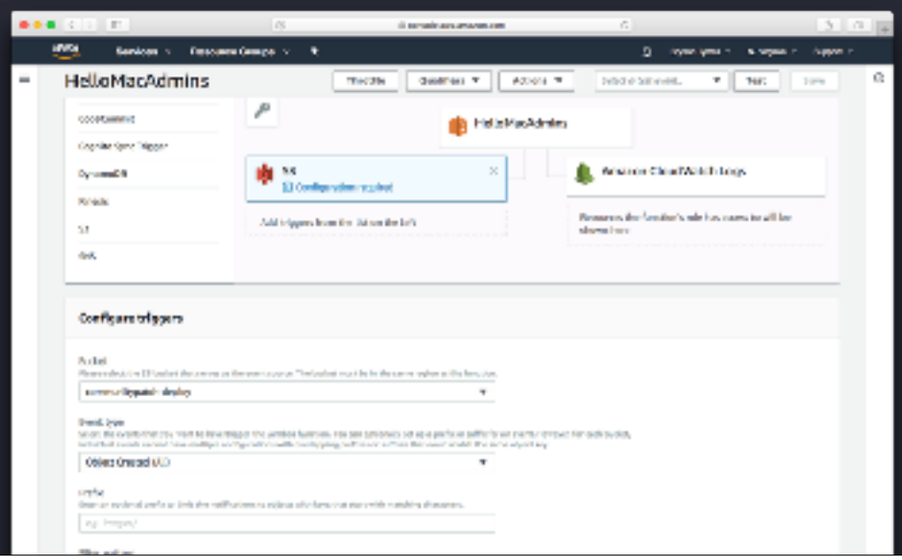
We need to create or associate an IAM role to grant all the required permissions.

Creating Lambda Functions



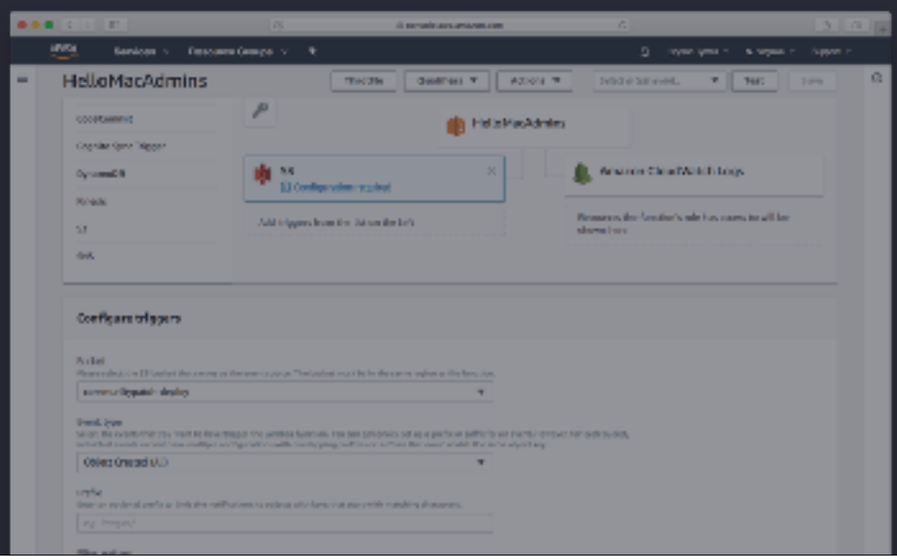
Once those steps are done, we can use the fancy new editor to write our code.

Creating Lambda Functions



Then we need to configure the triggering events.

Creating Lambda Functions



This can works for exploring and experimenting, but relying on the console to create functions is not the right way to do it.
Build a function, its roles, its permissions, linking the events, doing all of this by hand is not reproducable and can be error prone.

Templating Lambda Functions

```yaml
AWSTemplateFormatVersion: 2010-09-09
Transform: AWS::Serverless-2016-10-31

Resources:

  Bucket:
    Type: AWS::S3::Bucket
    Properties:
      BucketName: !Sub '${AWS::StackName}-bucket'

  S3Lambda:
    Type: AWS::Serverless::Function
    Description: Process events from an S3 bucket.
    Properties:
      Runtime: python3.6
      Handler: lambda_function.lambda_handler
      CodeUri: ./src
      Events:
        ObjectCreatedEvent:
          Type: S3
          Properties:
            Bucket: !Ref Bucket
            Events: s3:ObjectCreated:*
```

This is where we start discussing CloudFormation templates.

CloudFormation is a YAML format for defining AWS infrastructure as code. If it’s something you see in the console, you can, most of the time, define it in a CloudFormation template.

Normally, these templates can be very verbose because you need to define every part of these services. This resource defines a Lambda function. Normally, there would be individual resources for the function, the function’s permissions, and an IAM role, but we only have one here.

Templating Lambda Functions

```yaml
AWSTemplateFormatVersion: 2010-09-09
Transform: AWS::Serverless-2016-10-31
Resources:

  Bucket:
    Type: AWS::S3::Bucket
    Properties:
      BucketName: !Sub '${AWS::StackName}-bucket'

  S3Lambda:
    Type: AWS::Serverless::Function
    Description: Process events from an S3 bucket.
    Properties:
      Runtime: python3.6
      Handler: lambda_function.lambda_handler
      CodeUri: ./src
      Events:
        ObjectCreatedEvent:
          Type: S3
          Properties:
            Bucket: !Ref Bucket
            Events: s3:ObjectCreated:*
```

This simplified syntax, which gets rid of a lot of the complexity in defining the function, is owed to this key ‘Transform’ key at the top.

The ‘Serverless’ transform is an extension that provides special models to use in CloudFormation templates.

Templating Lambda Functions

```yaml
AWSTemplateFormatVersion: 2010-09-09
Transform: AWS::Serverless-2016-10-31

Resources:

  Bucket:
    Type: AWS::S3::Bucket
    Properties:
      BucketName: !Sub '${AWS::StackName}-bucket'

  S3Lambda:
    Type: AWS::Serverless::Function
    Description: Process events from an S3 bucket.
    Properties:
      Runtime: python3.6
      Handler: lambda_function.lambda_handler
      CodeUri: ./src
      Events:
        ObjectCreatedEvent:
          Type: S3
          Properties:
            Bucket: !Ref Bucket
            Events: s3:ObjectCreated:*
```

This is one of those models.

When CloudFormation processes this template, it will translate this model into all of the underlying resources that make up a complete Lamda function.

Templating Lambda Functions

```
AWSTemplateFormatVersion: 2010-09-09
Transform: AWS::Serverless-2016-10-31

Resources:

  Bucket:
    Type: AWS::S3::Bucket
    Properties:
      BucketName: !Sub '${AWS::StackName}-bucket'

  S3Lambda:
    Type: AWS::Serverless::Function
    Description: Process events from an S3 bucket.
    Properties:
      Runtime: python3.6
      Handler: lambda_function.lambda_handler
      CodeUri: ./src
      Events:
        ObjectCreatedEvent:
          Type: S3
          Properties:
            Bucket: !Ref Bucket
            Events: s3:ObjectCreated:*
```

Let’s explore they keys we have defined. The Runtime states the language.

Templating Lambda Functions

```
AWSTemplateFormatVersion: 2010-09-09
Transform: AWS::Serverless-2016-10-31

Resources:

  Bucket:
    Type: AWS::S3::Bucket
    Properties:
      BucketName: !Sub '${AWS::StackName}-bucket'

  S3Lambda:
    Type: AWS::Serverless::Function
    Description: Process events from an S3 bucket.
    Properties:
      Runtime: python3.6
      Handler: lambda_function.lambda_handler
      CodeUri: ./src
      Events:
        ObjectCreatedEvent:
          Type: S3
          Properties:
            Bucket: !Ref Bucket
            Events: s3:ObjectCreated:*
```

The Handler specifies the filename and the inner function that are executed by Lambda. The part before the period is the file and the part after is the function.

Templating Lambda Functions

```yaml
AWSTemplateFormatVersion: 2010-09-09
Transform: AWS::Serverless-2016-10-31

Resources:

  Bucket:
    Type: AWS::S3::Bucket
    Properties:
      BucketName: !Sub '${AWS::StackName}-bucket'

  S3Lambda:
    Type: AWS::Serverless::Function
    Description: Process events from an S3 bucket.
    Properties:
      Runtime: python3.6
      Handler: lambda_function.lambda_handler
      CodeUri: ./src
      Events:
        ObjectCreatedEvent:
          Type: S3
          Properties:
            Bucket: !Ref Bucket
            Events: s3:ObjectCreated:*
```

CodeUri contains either a directory path to where the source code is, or an S3 address for a deployable Lambda artifact. We'll cover the packaging and deployment steps later in the presentation.

Templating Lambda Functions

```
AWSTemplateFormatVersion: 2010-09-09
Transform: AWS::Serverless-2016-10-31

Resources:

  Bucket:
    Type: AWS::S3::Bucket
    Properties:
      BucketName: !Sub '${AWS::StackName}-bucket'

  S3Lambda:
    Type: AWS::Serverless::Function
    Description: Process events from an S3 bucket.
    Properties:
      Runtime: python3.6
      Handler: lambda_function.lambda_handler
      CodeUri: ./src
      Events:
        ObjectCreatedEvent:
          Type: S3
          Properties:
            Bucket: !Ref Bucket
            Events: s3:ObjectCreated:*
```

Under Events we can define all of the triggers for this function. In this example, the Lambda will be invoked every time a file is written into the bucket that's created in this template.

At the moment, the function won't be able to do much. While it will be triggered by write operations into the bucket, it won't be able to do anything with those files. In order to perform our processing we need to grant permissions for the function to access the bucket.

IAM Permissions

```
S3Lambda:
  Type: AWS::Serverless::Function
  Description: Process events from an S3 bucket.
  Properties:
    Runtime: python3.6
    Handler: lambda_function.lambda_handler
    CodeUri: ./src
    Policies:
      Statement:
      - Effect: Allow
        Action: s3:GetObject
        Resource: !Sub 'arn:aws:s3:::${AWS::StackName}-bucket/*'
    Events:
      ObjectCreatedEvents:
        Type: S3
        Properties:
          Bucket: !Ref Bucket
          Events: s3:ObjectCreated:*
```

Those permissions are granted through IAM policies using a Policies key in our template.

Show of hands; who here has taken a look at IAM in the console and rethought their careers?

If you’re searching online for examples of how to do anything in AWS when it comes to permissions, you will likely find some instructions on how to create an IAM role in the console, and the JSON for it might look something like this.

This is wrong. This is bad. Best practice with IAM is the least-permissive model. All permissions in AWS start off as no permissions. You should only grant the permissions that are required and nothing more. When it comes to S3, I have seen this example way too much, and it is a really bad idea.

IAM Permissions

```
{
    "Statement": [
        {
            'Action": "s3:*",
            'Resource": "*",
            'Effect": "Allow"
        }
    ]
}
```

This example, which I have seem many times, breaks the least-permissive model. First, the actions you are granting as a part of this policy are all actions available in S3. That means not just create, read, and delete to files within the bucket, but the configuration of the bucket itself.

IAM Permissions

```
{
    "Statement": [
        {
            "Action": "s3:*",
            "Resource": "*",
            "Effect": "Allow"
        }
    ]
}
```

But it isn’t limited to a single bucket. This policy allows any action for ANY bucket in the whole of your account.

The Lambda function we're working on is going to process files as they are uploaded. The only permission the function needs is to be able to download files, and only from the bucket that is triggering it.

The first change is to be explicit about the actions we want to grant.

IAM Permissions

```
{
    "Statement": [
        {
            "Action": [
                "s3:GetObject",
                "s3:PutObject"
            ],
            "Resource": "arn:aws:s3:::macadmins18-s3-bucket/*",
            "Effect": "Allow"
        }
    ]
}
```

If we need to grant permissions to more than one action, this becomes a list.

IAM Permissions

```
{
    "Statement": [
        {
            "Action": "s3:GetObject",
            "Resource": "arn:aws:s3:::macadmins18-s3-bucket/*",
            "Effect": "Allow"
        }
    ]
}
```

The next is to be explicit about which bucket we're allowing that action against. All AWS resources have what is called an ARN (stands for Amazon Resource Name).

IAM Permissions

```
'arn:aws:${Service}:${AWS::Region}:${AWS::AccountId}:${ResourceType}/${Resource}'
```

ARNs aren’t actually that complex. They’re similar to reverse domain naming.

First, we state it is an ARN, and it’s for AWS.

IAM Permissions

```
'arn:aws:${Service}:${AWS::Region}:${AWS::AccountId}:${ResourceType}/${Resource}'
```

“Service” is going to be the AWS service the resource belongs to. This could be dynamodb, apigateway, s3, and so on.

IAM Permissions

```
'arn:aws:${Service}:${AWS::Region}:${AWS::AccountId}:${ResourceType}/${Resource}'
```

Next we state which region the resource is in and the acount it belongs to. The majority of AWS services are region based. Some are global and deviate from this, like S3, but most will require these values.

This syntax you see here with the bash style variable declaration and “AWS::” are called pseudo parameters in CloudFormation. These are special values that CloudFormation will populate for you so you don’t have to hard code them. These are very powerful for making templates portable across regions and accounts.

IAM Permissions

```
'arn:aws:${Service}:${AWS::Region}:${AWS::AccountId}:${ResourceType}/${Resource}'
```

Lastly, we have the resource itself. This also can very a little between services. Some, like DynamoDB, follow this style here. The “ResourceType” would be “table” and then after a forward slash the “Resource” will be the name of the table.

I was using DynamoDB as an example. Here's how the ARN to a table would look.

I mentioned S3 is different because it's a global resource. In the case of S3, global means global in the sense that there are no concepts of regional or account specific assets. Bucket names in S3 can only be used once for any region in any account, which is why those values are absent from our ARN.

Permissions on buckets also are different depending on the ARN you pass. With just the bucket name, I'm granting permissions for actions on the bucket itself, but not the contents.

Adding a foward slash, I'm providing an ARN for the bucket's contents. Here it's an asterisk saying all objects in this bucket, but you can do ARNs for specific paths within the bucket and prevent reade, write, or delete access to certain contents.

Example Code Walkthrough

```
AWSTemplateFormatVersion: 2010-09-09
Transform: AWS::Serverless-2016-10-31

Resources:

  Bucket:
    Type: AWS::S3::Bucket
    Properties:
      BucketName: !Sub '${AWS::StackName}-bucket'

  S3Lambda:
    Type: AWS::Serverless::Function
    Description: Process events from an S3 bucket.
    Properties:
      Handler: lambda_function.lambda_handler
      CodeUri: ./src
      Policies:
        Statement:
          - Effect: Allow
            Action: s3:GetObject
            Resource: !Sub  arn:aws:s3:::${AWS::StackName}-bucket/*
      Events:
        ObjectCreated:s3:
          Type: S3
          Properties:
            Bucket: !Ref Bucket
            Events: s3:ObjectCreated:*
```

With the addition of our IAM policy onto the Lambda function, our template is now complete. This will create an S3 bucket to load data into, and a Lambda function with permissions to that bucket to read the files from and perform processing.

Let’s walk though the code that actually does the work.

Example Code Walkthrough

Here is the code for our “lambda_function.py” file. Let’s walk through what happens when our S3 bucket triggers it.

First, there’s going to be what’s known as a “cold start”. On the first incovation, the Lambda service downloads the code from storage, starts a new container for it, and the code is loaded. Everything outside of the handler function - such as import statements and global variable setting - is performed at this step.

Once past the cold start phase there is now a ready to use container for subsequent invocations up until a certain point at which the container is unloaded. Then the next invocation after that will go through the cold start again.

Example Code Walkthrough

Now that we’re past the cold start, when it does happen, our handler function is executed. Lambda is going to pass an event object, which is always a Python dictionary, and a context object which we won’t focus on for these examples.

Example Code Walkthrough

In my handler you can see here that I’m extracting some information from the event. There is a “Records” key that contains a list of events. On each entry, I’m reading from an “s3” key that contains information about the bucket and the object.

How did I know to write my code this way?

Example Code Walkthrough

There are documented examples for every type of Lambda event. This is the one for S3 that I copied straight from their page. I'll have a link to this resource at the end. It takes most of the guesswork out of writing your Lambdas as you know exactly what to expect for the event's data structure.

Next step in the code is to download the file that triggered our Lambda in the first place. I split that out into a separate function. In my example here, I'm expecting the data to be JSON data. I didn't get into this, but one option I could have put into the template for the S3 event is a filter so that only files that match a provided pattern will start the execution.

Now, in my "get_file" function, you can see boto3 being used to download the file, but nowhere in our code do we have any AWS access keys. How is boto3 able to do this if we aren't providing credentials? The answer is the IAM policy we attached to the function. The Lambda service loaded temporary credentials into the environment variables for our function's container when it spun up. The AWS SDKs all know how to read in those environment variables to be able to make authenticated requests.

The BytesIO bit you see in here is a handy tool to perform the file download and keep it in memory instead of writing the file to disk.

Example Code Walkthrough

Once the download is complete the file can now be processed. I don't have any code here showing any parsing or manipulation, but I do have a piece to POST that JSON to a remote HTTP service. Maybe something like Splunk.

You might have noticed I'm using requests here. If you aren't familliar with requests, it's a third party module for Python for making HTTP requests. Here's another handy trick for you if you are writing Lambdas that you want to use requests in: it's an embedded module inside botocore which is a lower level module used by boto3. It's an older version of requests - current is 2.19, the embedded is 2.7 - but unless you're missing features you can skip the process of needing to package an external dependency.

Packaging and Deploying

```
src/
|   └── lambda_function.py
└── template.yaml
```

Here is a view of the serverless app. This is a simple app with only one function so we only have two working files.

Packaging and Deploying

```
$ pip install awscli --upgrade --user
```

```
$ pip3 install awscli --upgrade --user
```

We are going to package and then deploy our serverless application. The good news is that the AWS command line tool can perform both steps for us.

If you don’t have the AWSCLI, you can install it using Pip. Yes, this tool is written in Python and uses the Boto3.

I will say, I strongly recommend installing this using a Python environment you’ve installed separately from the system Python. Even better: install Python 3 on your Mac and then install the CLI there.

Packaging and Deploying

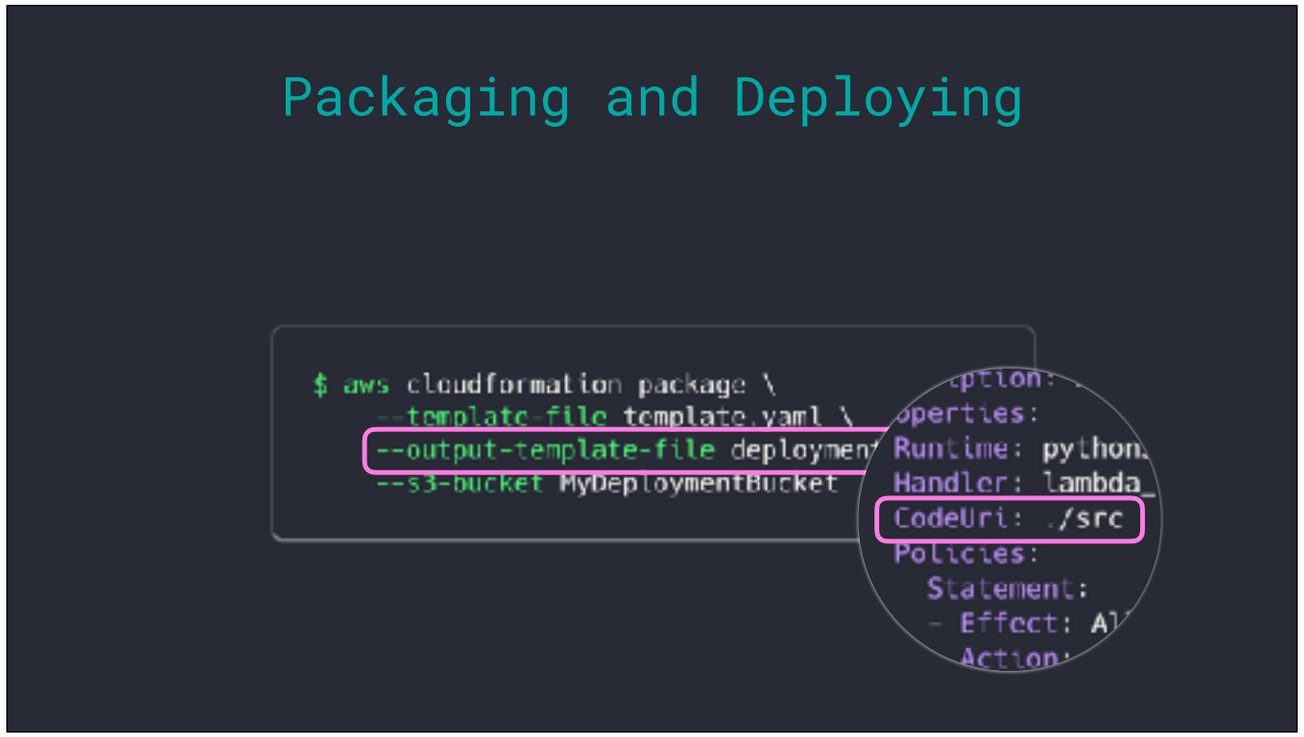
```
$ aws cloudformation package \
    --template-file template.yaml \
    --output-template-file deployment.yaml \
    --s3-bucket MyDeploymentBucket
```

Once you have the AWSCLI, from the app’s directory you can run the “cloudformation package” command.

Packaging and Deploying

```
$ aws cloudformation package \
    --template-file template.yaml \
    --output-template-file deployment.yaml \
    --s3-bucket MyDeploymentBucket
```

Our first argument passes the name of the femplate file we are packaging.



The second argument is going to be where we write deployment ready template. This command outputs a template because the packaging process will iterate over every function we have defined and zip up the contents of the directories specified in the CodeURIs.

Packaging and Deploying

```
$ aws cloudformation package \
    --template-file template.yaml \
    --output-template-file deployment.yaml \
    --s3-bucket MyDeploymentBucket
```

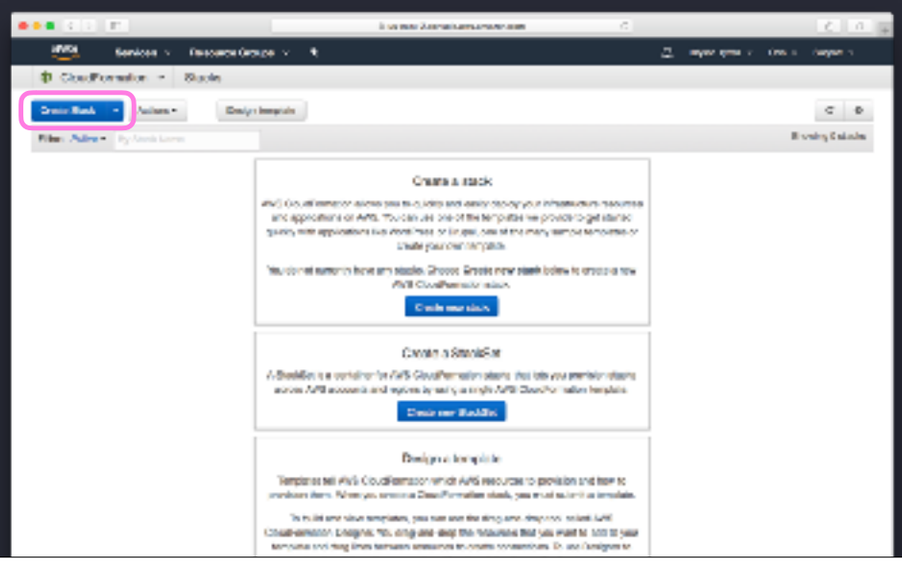
Those zipped files, called deployment packages in the AWS docs, are going to be uploaded to an S3 bucket which you provide here.

Packaging and Deploying

```
AWSTemplateFormatVersion: 2010-09-09
Transform: AWS::Serverless-2015-10-31

Resources:

  Bucket:
    Type: AWS::S3::Bucket
    Properties:
      BucketName: !Sub '${AWS::StackName}-bucket'

  S3Lambda:
    Type: AWS::Serverless::Function
    Description: Process events from an S3 bucket.
    Properties:
      Runtime: python3.6
      Handler: index.function_handler.handler
      CodeUri: s3://MyDeployment/Bucket/d8d3f48dc1394d8d8f6f5544502c1144
      Policies:
        Statement:
          - Effect: Allow
            Action: s3:GetObject
            Resource: !Sub 'arn:aws:s3:::${AWS::StackName}-bucket/*'
      Events:
        ObjectCreateEvents:
          Type: S3
          Properties:
            Bucket: !Ref Bucket
            Events: s3:ObjectCreated:*
```

The deployment template is going to contain all the same information as our original template, except the CodeUris will be updated with the S3 locations for each Lambda's deployment package. This file is now ready for us to use in CloudFormation and create deploy our app.

Packaging and Deploying

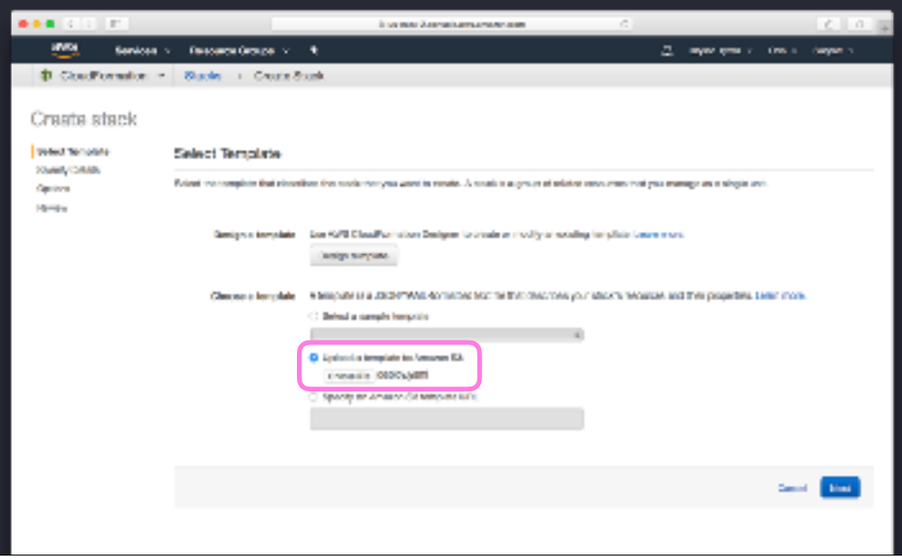


At this point, we can go to the AWS console and deploy there.

It's important to point out that you need to create your Stack in the same region as the S3 bucket that the Lambda packages were uploaded to. If you need to deploy across different regions you will need to perform packaging using a bucket for each.

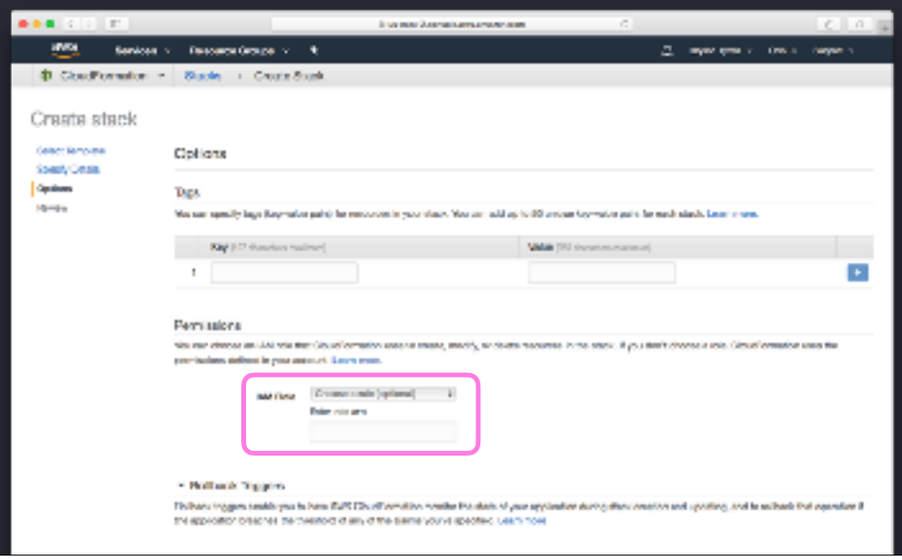
At the CloudFormation screen we're going to click "Create Stack". A template you deploy in CloudFormation creates what is called a Stack: a collection of related resources.

Packaging and Deploying



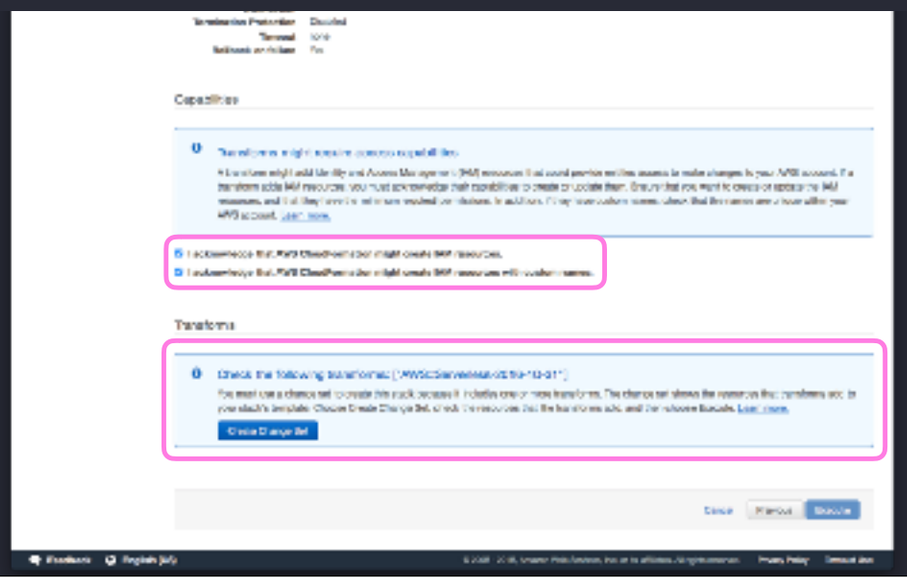
Select the option to upload a template and select the deployment template.

Packaging and Deploying



Click through this Options screen. It allows you do some change like selecting an IAM role which CloudFormation will use to perform all the API calls for creating the resources. If you don’t change this, CloudFormation will use the permissions of the logged in use.

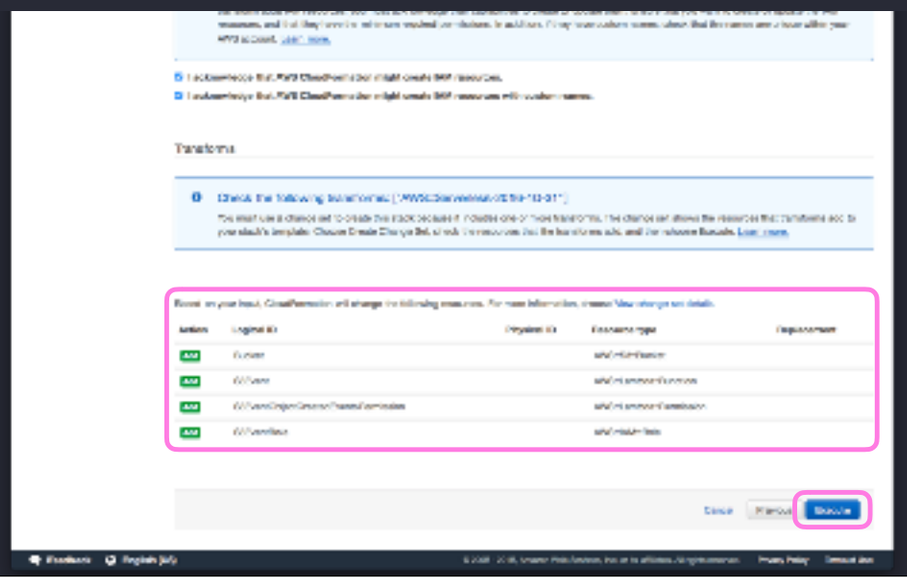
Packaging and Deploying



After that, we have to check a couple of boxes acknoledging that CloudFormation is going to create IAM resources.

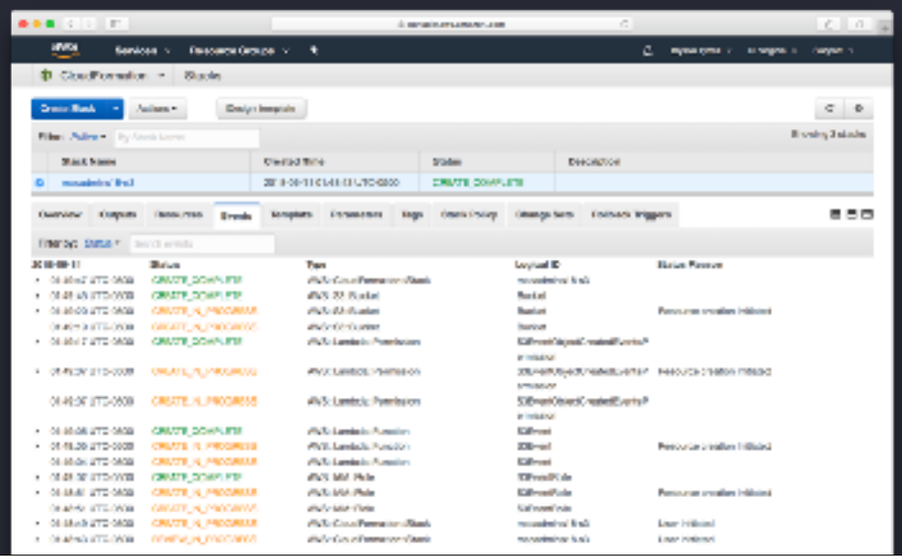
Then, we have to create what is called a “Change Set”. Because our template has a Transform key, CloudFormation needs to convert the template’s Serverless resources to native resources.

Packaging and Deploying



The results of the “Change Set” will be displayed. We can then click “Execute” and launch the Stack.

Packaging and Deploying



After a few minutes, as long as no errors were encountered - like permissions errors when trying to create resources - the Stack will finish with a CREATE_COMPLETE status.

That’s a lot of clicking and a lot of steps. There are advantages to using the GUI, like leveraging this Events tab to see what is happening, and what the problems were should a create fail, but the alternative method that cuts out many of these steps is to go back to the AWSCLI.

Packaging and Deploying

```
$ aws cloudformation deploy \
    --template-file deployment.yaml \
    --stack-name uploader-test \
    --capabilities CAPABILITY_IAM
```

Back at the command line, instead of the “cloudformation package” command we will use “cloudformation deploy”.

Packaging and Deploying

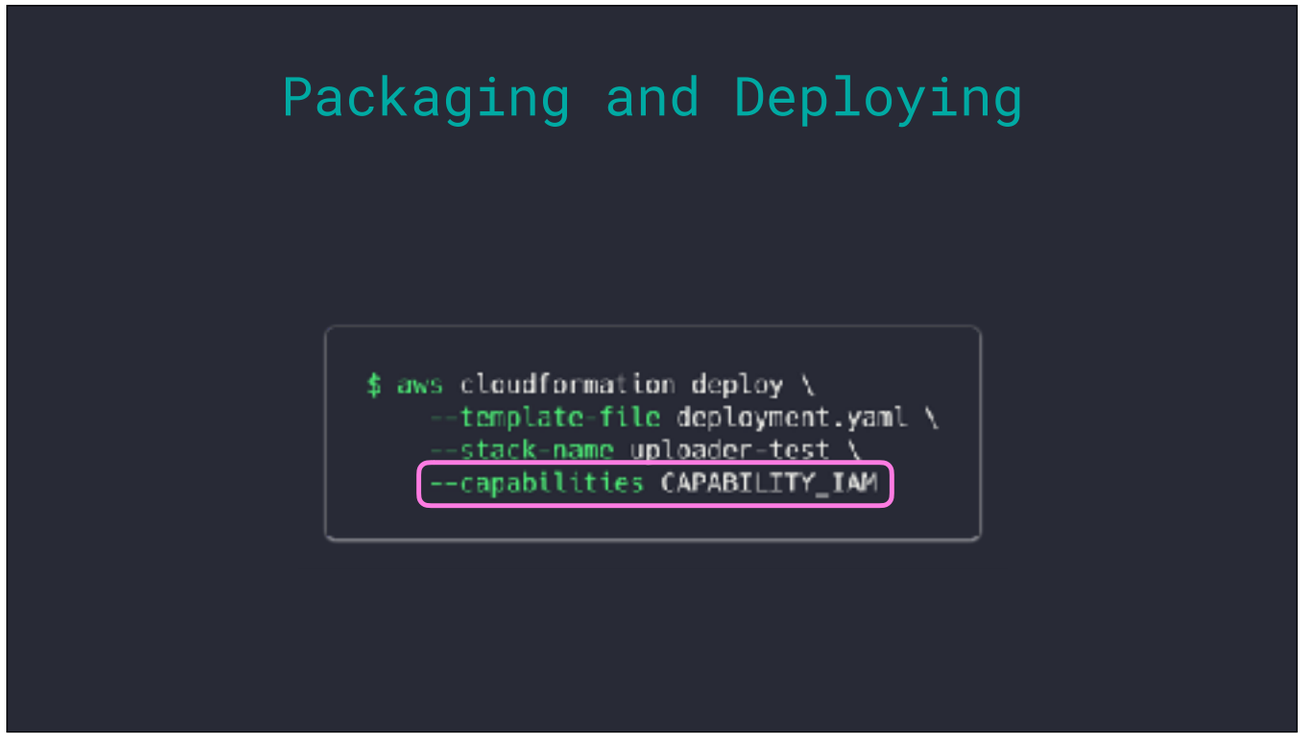
```
$ aws cloudformation deploy \
    --template-file deployment.yaml \
    --stack-name uploader-test \
    --capabilities CAPABILITY_IAM
```

In the first argument we will pass the template file that will be used for the Stack.

Packaging and Deploying

```
$ aws cloudformation deploy \
    --template-file deployment.yaml \
    --stack-name uploader-test \
    --capabilities CAPABILITY_IAM
```

Next we will give a name for the Stack



And last, if you remember from the console we had to check a couple of boxes saying we understood that IAM resources were going to be created, this is that same thing.

Packaging and Deploying

```
$ aws cloudformation deploy \
    --template-file deployment.yaml \
    --stack-name uploader-test \
    --capabilities CAPABILITY_IAM
```

This command will perform all of the steps from our console walkthrough in a single action. The Change Set is created and then executed in sequence.

Packaging and Deploying

```
$ aws cloudformation deploy \
    --region us-east-1 \
    --template-file deployment.yaml \
    --stack-name uploader-test \
    --capabilities CAPABILITY_IAM
```

Normally, when you're using the AWSCLI you're working within a default region that's listed in your credentials file with your access keys. If you are deploying a template to a different region than your default, or you are prompted that no region was specified, you can pass it using the region argument.

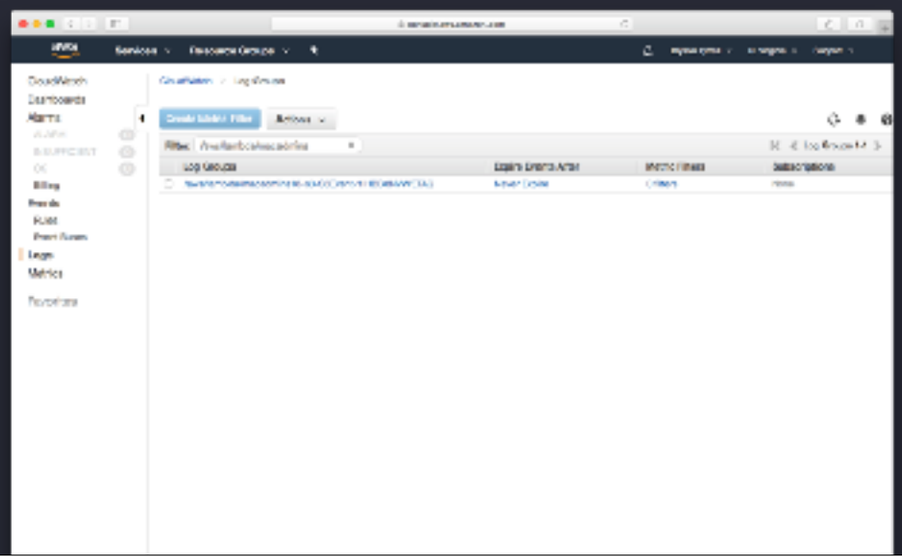
Debugging and Troubleshooting

“Why isn’t this working?”

I’m going to wrap up this talk with everyone’s favorite question:

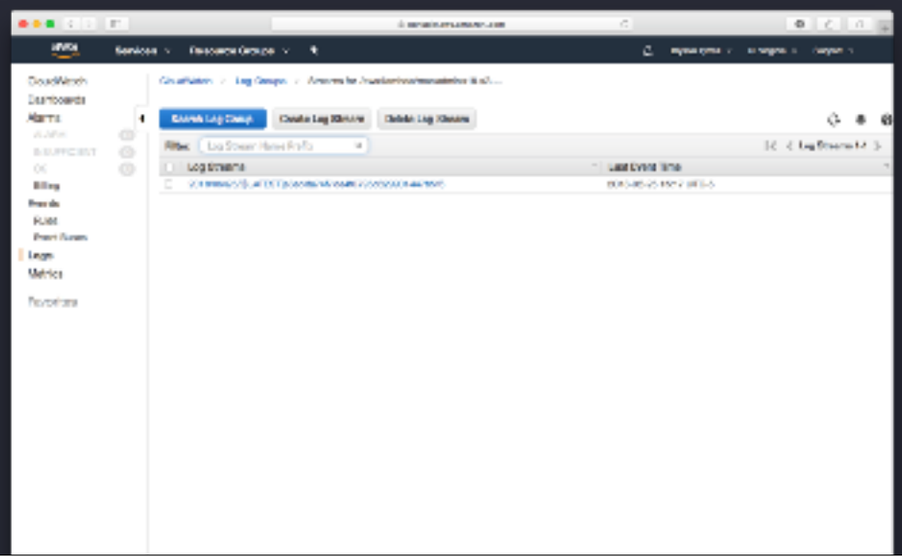
Once we’ve deployed, and our functions are running, problems at a code level can and do come up.

Debugging and Troubleshooting



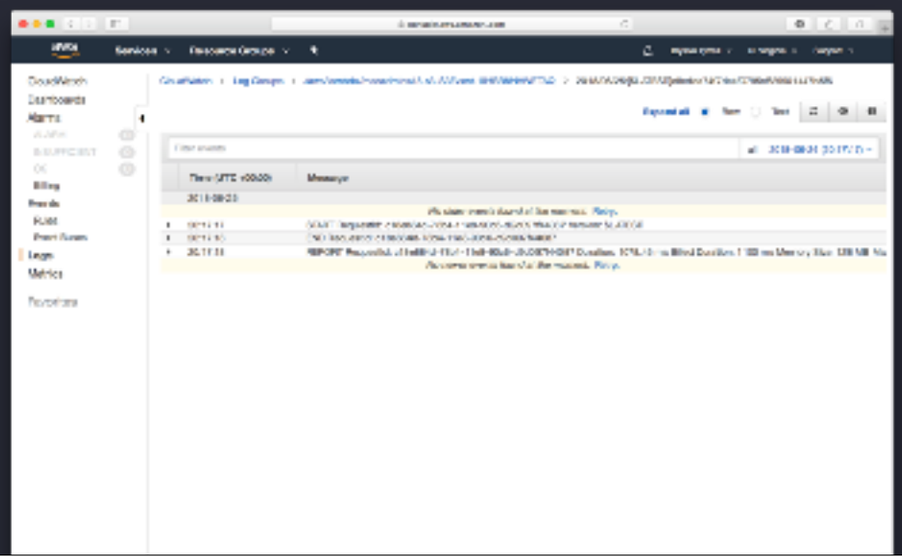
For logging, we have CloudWatch. Like everyone, logs are regional so be sure you’re looking for your Lambda’s logs in the same region it was deployed. Each log group will be prefixed with /aws/lambda/ and then the name of the Lambda.

Debugging and Troubleshooting



Inside the log group there will be a list of streams. Each stream will contain all of the logged information for each container. Whenever a new container starts there will be an new log stream.

Debugging and Troubleshooting



In the stream will be the printed output from each invocation.

Debugging and Troubleshooting

Without any additional statement, each successful invocation will have a START, END, and REPORT. The REPORT will contain information about execution time and the amount of memory used.

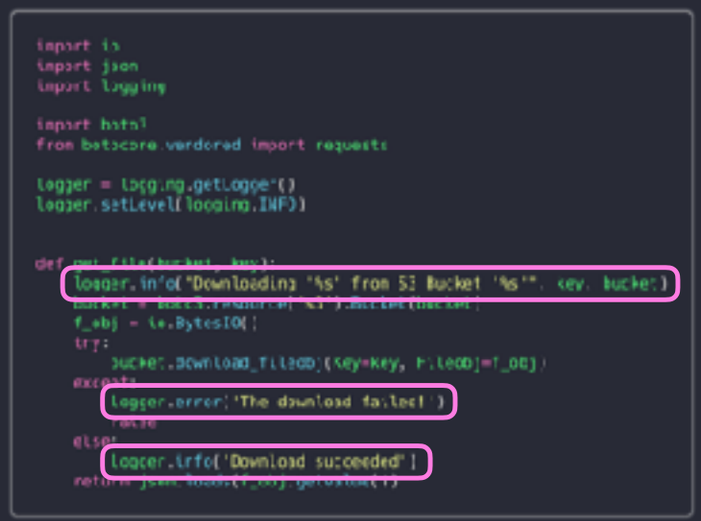
In Python, anything in your code that uses print() or the logging module will appear between the START and END lines.

Debugging and Troubleshooting

```
import io
import json
import logging

import botocore
from botocore.vendored import requests

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


def get_file(bucket, key):
    logger.info("Downloading '%s' from S3 Bucket '%s'", key, bucket)
    bucket = boto3.resource('s3').Bucket(bucket)
    f_obj = io.BytesIO()
    try:
        bucket.download_fileobj(Key=key, Fileobj=f_obj)
    except:
        logger.error('The download failed')
        raise
    else:
        logger.info('Download succeeded')
    return json.loads(f_obj.getvalue())
```

Best practice is to use the logging module so you can use leverage the level options.

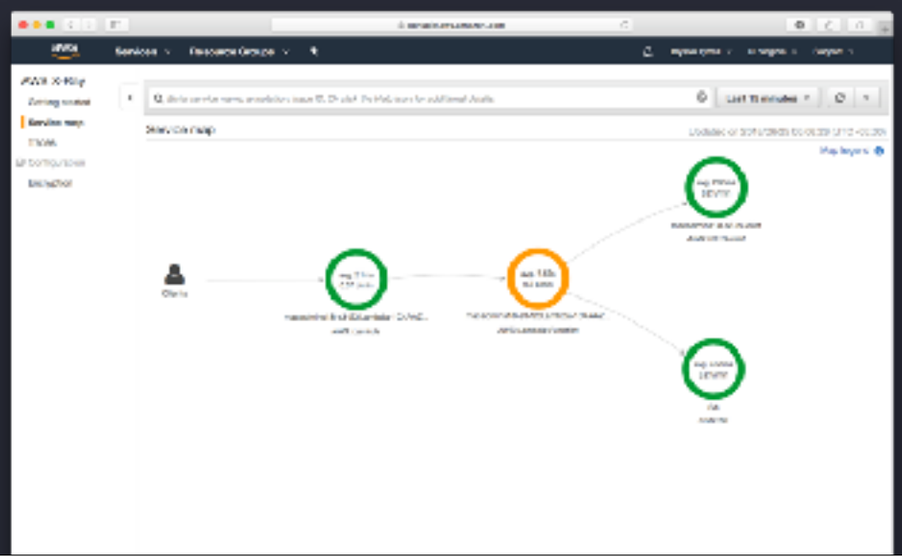
Debugging and Troubleshooting



The better the logging in your Lamdbas the easier time you'll have in troubleshooting issues.

If any uncaught exceptions occur the tracebacks will also be logged.

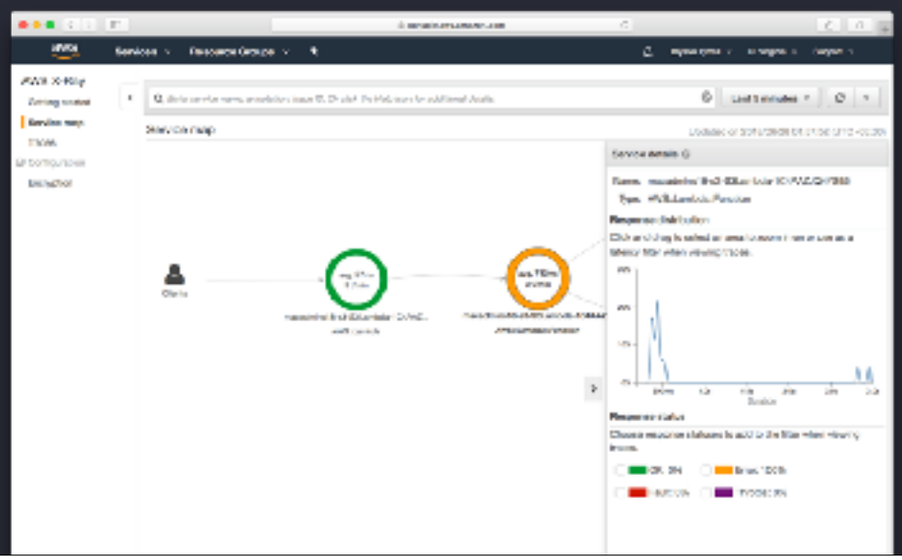
Debugging and Troubleshooting



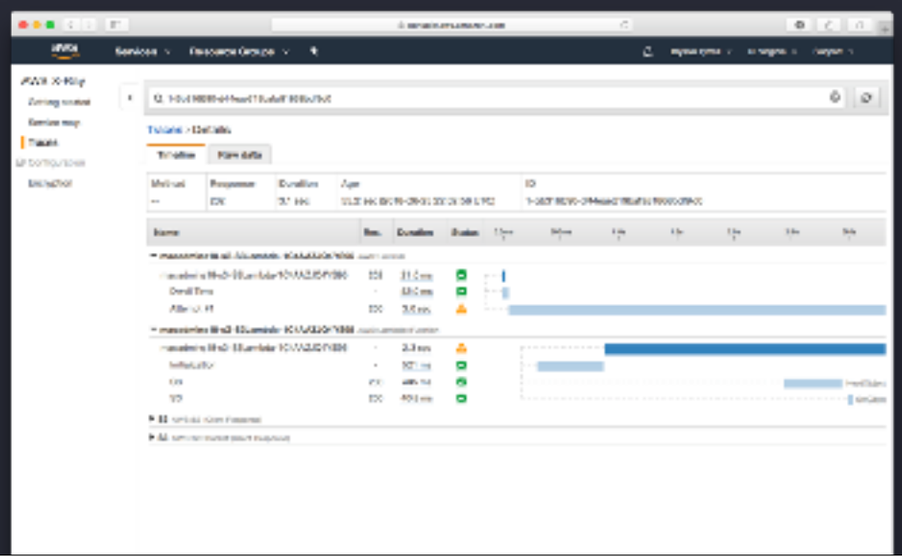
On top of logging, there's also X-Ray. X-Ray is a tracing tool. When enabled in a Lambda function, it allows you to see data concerning the invocation, execution times, and latency in requests. The map view shows averages over the course a a time period. The rings are color coded pie charts:

Green for OK
Yellow for Errors
Red for faults
Purple for throttle actions - which can happen with AWS APIs.

Debugging and Troubleshooting



Debugging and Troubleshooting



Debugging and Troubleshooting

```
S3Lambda:
  Type: AWS::Serverless::Function
  Description: Process events from an S3 bucket.
  Properties:
    Runtime: python3.6
    Handler: lambda_function.lambda_handler
    CodeUri: ./src
    Tracing: Active
    Policies:
      Statement:
        - Effect: Allow
          Action: s3:GetObject
          Resource: !Sub 'arn:aws:s3:::${AWS::StackName}-bucket/*'
    Events:
      ObjectCreatedEvents:
        Type: S3
        Properties:
          Bucket: !Ref Bucket
          Events: s3:ObjectCreated:*
```

For each Lambda function in our template, we need to add a Tracing key. This will take care of adding the needed IAM permissions to allow the function to write tracing data to X-Ray.

Debugging and Troubleshooting

```
import io
import json
import logging

from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.core import patch
import boto3
from botocore.vendored import requests

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

xray_recorder.configure()
patch(['boto3', 'requests'])
```

The X-Ray SDKs are not included in the Python environment like Boto3 is. They need to installed and bundled with the Lambda package before upload.

But after that, you can enable tracing in the code by adding these imports…

Debugging and Troubleshooting

```
import io
import json
import logging

from aws_xray_sdk.core import xray_recorder
from aws_xray_sdk.core import patch
import boto3
from botocore.vendored import requests

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

xray_recorder.configure()
patch(['boto3', 'requests'])
```

…and enabling the recorder. X-Ray supports patching a few of the most popular Python modules and frameworks. Boto3 and Requests are among them. There is also a patch_all() method that will attempt to patch every supported module that is found in the running function.

Would you like to know more?

AWS SAM (Serverless Application Model)
https://github.com/awslabs/serverless-application-model/

CloudFormation Psuedo Parameters
https://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/pseudo-parameter-reference.html

Lambda Event Examples
https://docs.aws.amazon.com/lambda/latest/dg/eventsources.html

X-Ray for Python
https://docs.aws.amazon.com/xray/latest/devguide/xray-sdk-python.html

https://github.com/brysontyrrell/Serverless-Examples/

About Cost...

Free Tier (per month):

- Lambda: 400,000 seconds (@1GB)
- S3: 5 GB Storage, 20,000 GET, 2,000 PUT, 15 GB Transfer
- SQS: 1,000,000 Requests
- API Gateway: 1,000,000 Requests*

Free Tier Stats (per month):
Lambda: 128MB - 3,200,000 seconds execution time / 1 GB - 400,000
S3: 5 GB storage, 20,000 GET, 2,000 PUT, 15 GB transfer out
SQS: 1,000,000 requests
API Gateway: 1,000,000 requests 12 months, $3.50 per

Thank you!

Q&A?