An Intro to Serverless with Go: Using AWS Lambda

linkedin.com/pulse/intro-serverless-go-using-aws-lambda-pat-alwell/

Abstract: This post covers the basics of using GoLang with AWS Lambda. We will be using a text editor, functioning code, and screenshots to review the essential ingredients.

Cost information about AWS resources can be controlled using tags. I will specifically explore how to automatically tag EC2 instances using API events as our source of information and a GoLang client as our source of interaction. With this solution, your instances, volumes, and snapshots will automatically be tagged with a creation date and username. You'll also be able to enforce a tagging policy and politely remind colleagues to tend to their resources or shut down those resources no longer in use.

The inspiration for this blog post originally came from Alessandro Martini. I've essentially converted his Python over to Go. You can read the original auto-tagging post here: Alessandro Martini's AWS Blog Post

What is GoLang? Why bother learning it?

GoLang (Go) is an open source statically typed compiled programming language developed by the team at Google. Go was originally designed to improve code productivity in Google's massive high-concurrency networking environment. Developers with a C or Python background will see similarities. Go isn't a perfect match for every application pattern, but the language is fun, lightweight and easy to learn. What’s more, Go will become very popular as engineering teams get tired of maintaining a legacy code base. Follow this link to learn more: Go at Google

In short, Go provides

> Static typing and run-time efficiency (like C or C++)

> Readability and usability (like Python)

> High-performance networking and concurrency

What is Serverless? And what is AWS Lambda?

Serverless computing is an executional framework that allows a developer to deploy code to a service provider, like AWS or Google, without having to maintain, build, or patch a runtime environment. AWS Lambda is one example of a serverless framework.

Lambda makes it incredibly easy to execute code in response to an event. Some examples include responding to messages posted to a queue, reacting to custom API events fired from an HTTP client, or ingesting custom JSON generated by an application or device. For a full list of event sources, you can visit Amazon's official documentation here: Lambda Event Sources

Installing Go and Setting up the appropriate Environment Variables

For the sake of simplicity, I am going to install Go using Homebrew. One can alternatively install the Go binaries by visiting golang.org and following the instructions. In the end, we'll still have a functioning Go environment by setting up the GOPATH and GOROOT.

Let's start by running *brew update* and *brew install go.* If you already have Go installed feel free to skip to the next section.

```
$ brew update
$ brew install go
```

After we've installed Go we can go ahead and create our working directory and set the GOPATH env variable. For a full listing of Go's environment vars run *go env*.

```
$ go env
```

If we've used Brew to install Go, our GOROOT will be located in a predefined brew path. In this case, the Go libraries were installed under a local directory by the name of Cellar.

```
GOROOT="/usr/local/Cellar/go/1.12.6/libexec"
```

Next, we'll want to set our GOPATH. Create a directory in your development space called Go. *Note: Brew does this for you as well, however, I wanted to move my work to a devoted development directory to keep things organized.* Then make three directories inside of the parent Go directory, /src, /pkg, and /bin. These child directories will serve to encapsulate our source code, packages, and executables.

In essence, you should have a Go project structure that looks like this:

```
$ /GoDev
  -/src
  -/bin
  -/pkg
```

Now we can point Go to the root folder of this predefined project location by exporting GOPATH. The Go compiler now has the information it needs to successfully compile, package, and pull our code.

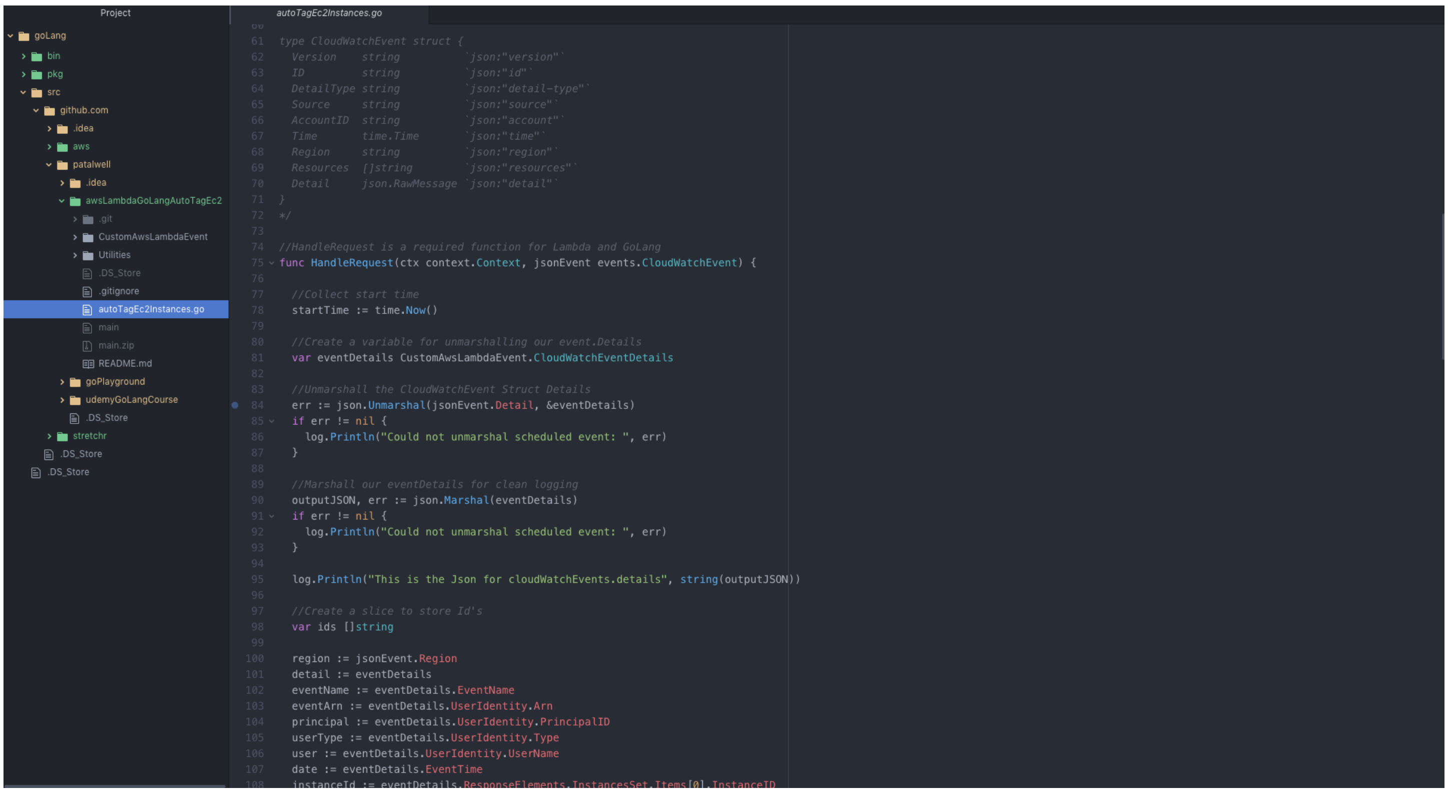
```
$ vi .bash_profile

  export GOPATH=$HOME/Development/Go
```

Retrieving the Go Packages for This Post

After we've set up Go, we can pull some code for the sake of understanding my post. Navigate to your terminal and run go get <package> for several packages that we'll need to build our executable:

```
$ go get github.com/patalwell/awsLambdaGoLangAutoTagEc2
$ go get github.com/aws/aws-lambda-go/events
$ go get github.com/aws/aws-lambda-go/lambda
$ go get github.com/aws/aws-sdk-go/aws
$ go get github.com/aws/aws-sdk-go/aws/session
$ go get github.com/aws/aws-sdk-go/service/ec2
```

If you've successfully pulled my Go package from GitHub you'll see a directory called */awsLambdaGoLangAutTagEc2* this is the Go package containing our code. Here is an example of what you should see:



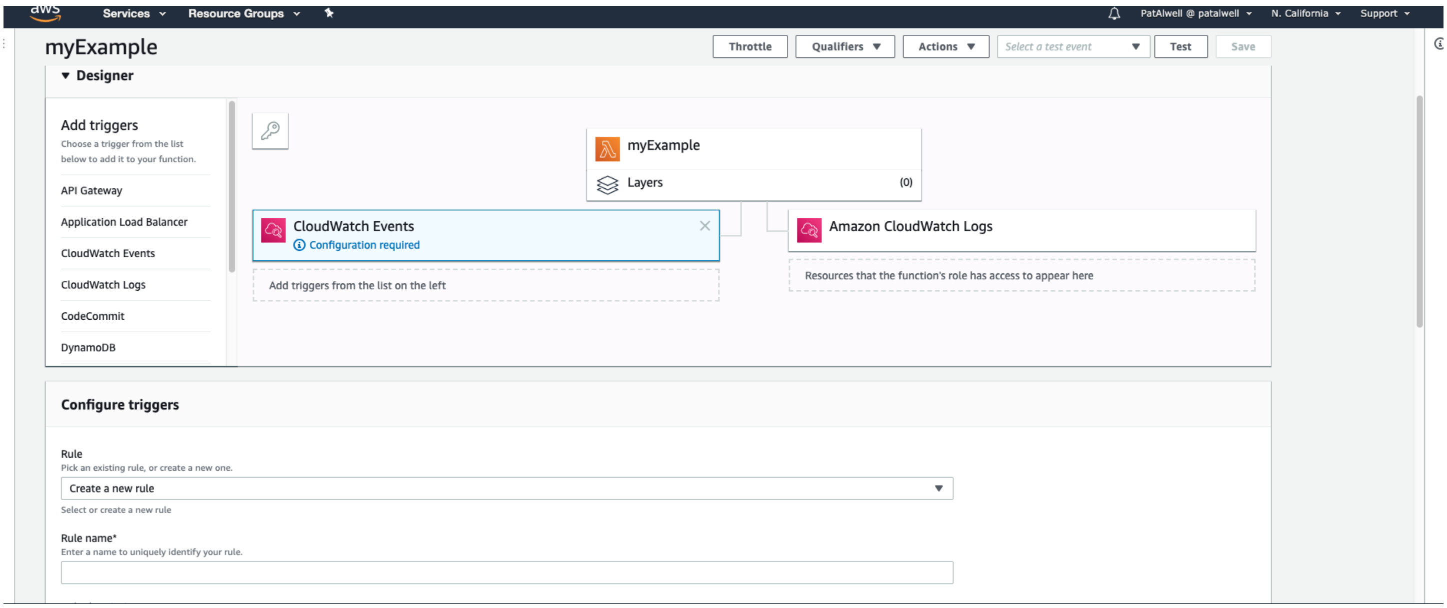
Setup AWS Lambda for Serverless Execution

To setup Lambda for Go, we'll need to create several AWS resources. First, we'll create a Lambda function and assign it the appropriate permissions. Then, we'll create an event rule in CloudWatch and finally, we'll enable CloudTrail in our given region.

Log onto the AWS console, navigate to compute services, select Lambda, and push the create function button. You'll be prompted to name the function, select a template, and assign permissions. Name the function something practical for the sake of simplicity and choose to author the function from scratch. Next, select the Go runtime environment and opt to create new permissions for Lambda. After that's done, you'll be taken to a blank function canvas.

From left to right you'll notice several options. The first option is to add a trigger to the function, the next option is to set up the code, and the final option is showing function permissions. In this case, let's add and configure a trigger for CloudWatch events. Click on CloudWatch events in the add trigger menu and navigate to the result modal at the bottom named configure triggers (*Figure 2a*).

Figure 2a: *The Lambda User Interface*

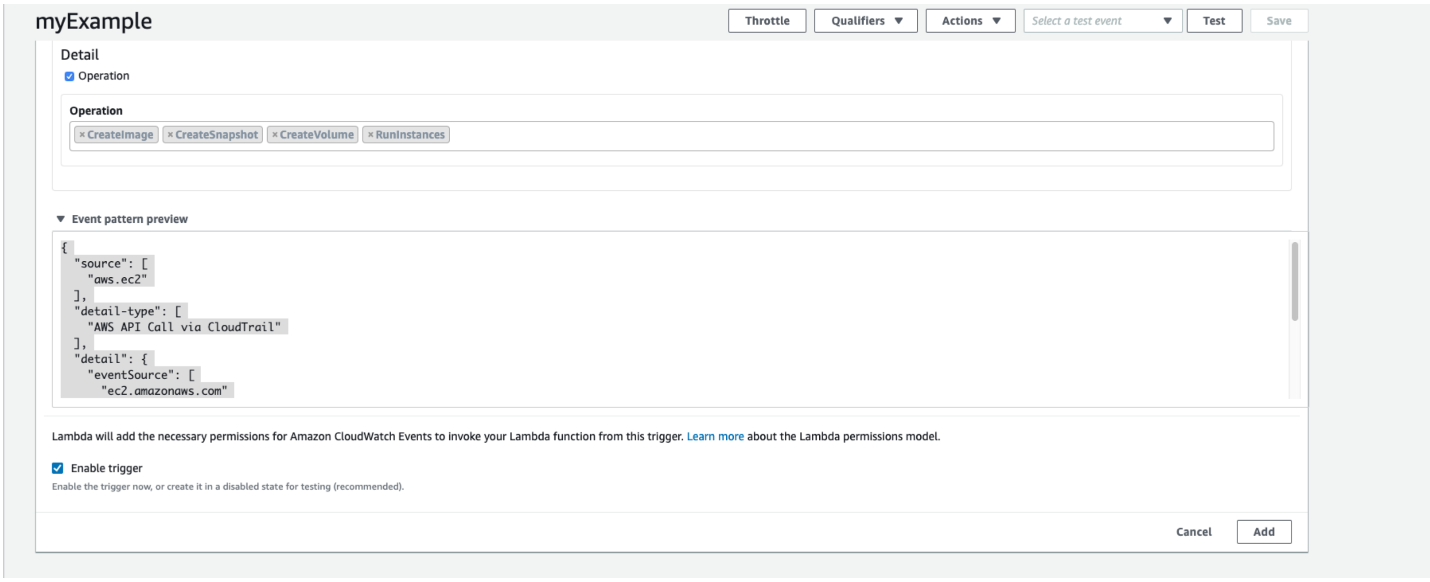


Add a name for the new event rule and provide a brief description. In this case, we want something that will remind us of our objective, so lets use filterEc2ApiEvents. Check the operation box and search for the following API operations:

```
CreateImage
CreateSnapshot
CreateVolume
RunInstances
```

Add them to our rule, enable the trigger, and add the custom event to our function (*Figure 2b*).

Figure 2b: *Loading our Event Rules and Enabling the Trigger*

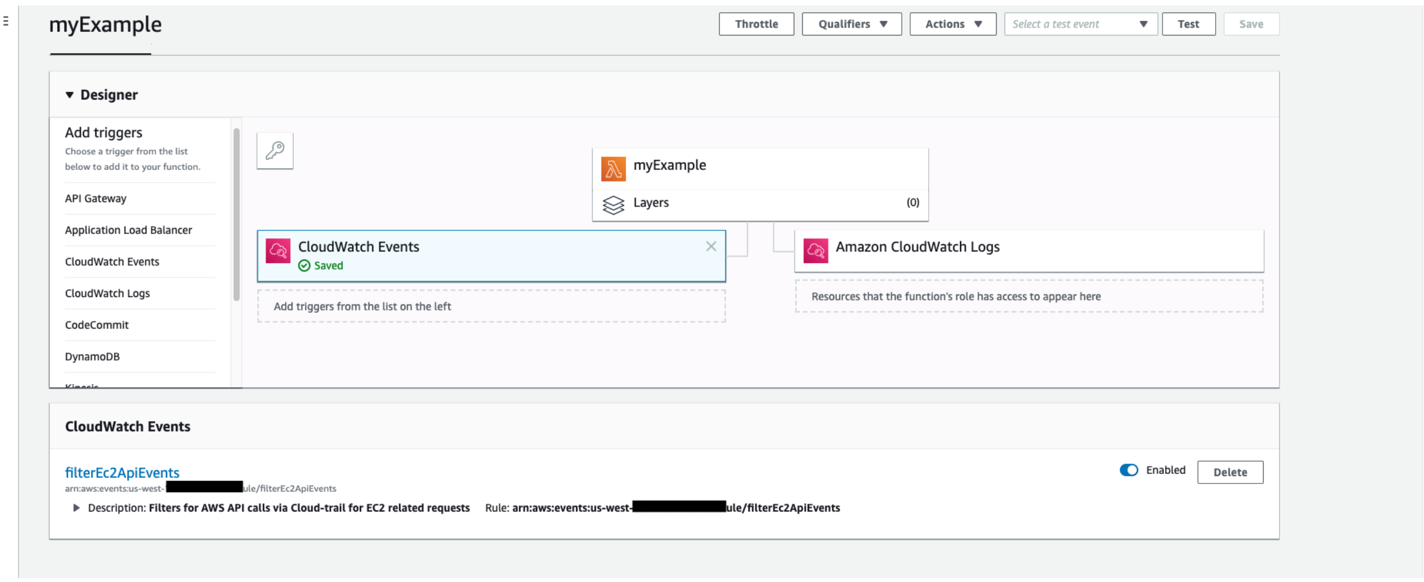


Our event pattern should now resemble the following JSON:

```
{
  "source": [
    "aws.ec2"
  ],
  "detail-type": [
    "AWS API Call via CloudTrail"
  ],
  "detail": {
    "eventSource": [
      "ec2.amazonaws.com"
    ],
    "eventName": [
      "CreateImage",
      "CreateSnapshot",
      "CreateVolume",
      "RunInstances"
    ]
  }
}
```

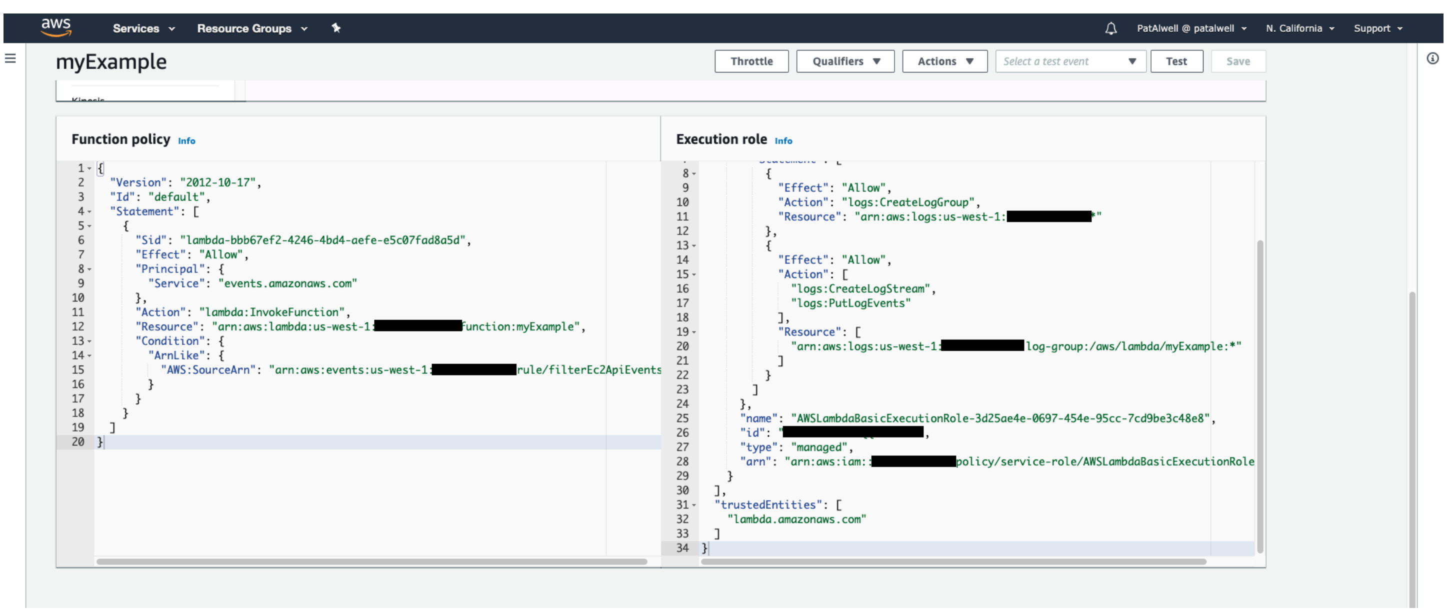
Once we've added the event, navigate to the top right of the Lambda user interface and click save. We should now see green text appear in our new event (*Figure 2c*).

Figure 2c: *Saving our Initial Configurations and Triggers*



Now that we've added an event to our function we can assign permissions. Click on the key icon right above the CloudWatch events panel to view the function's current policy and execution role (*Figure 2d*). On the left, you'll notice we are allowing *events.amazon.aws* to invoke a function with the resource ARN for our function. You'll also notice that there is a source ARN for the action. In this case, the source ARN is the event we just created for filtering EC2 API events via CloudTrail. We are essentially granting our function permission to invoke itself after finding API events of the types we previously established.

Figure 2d: *Function Policy and Execution Role*

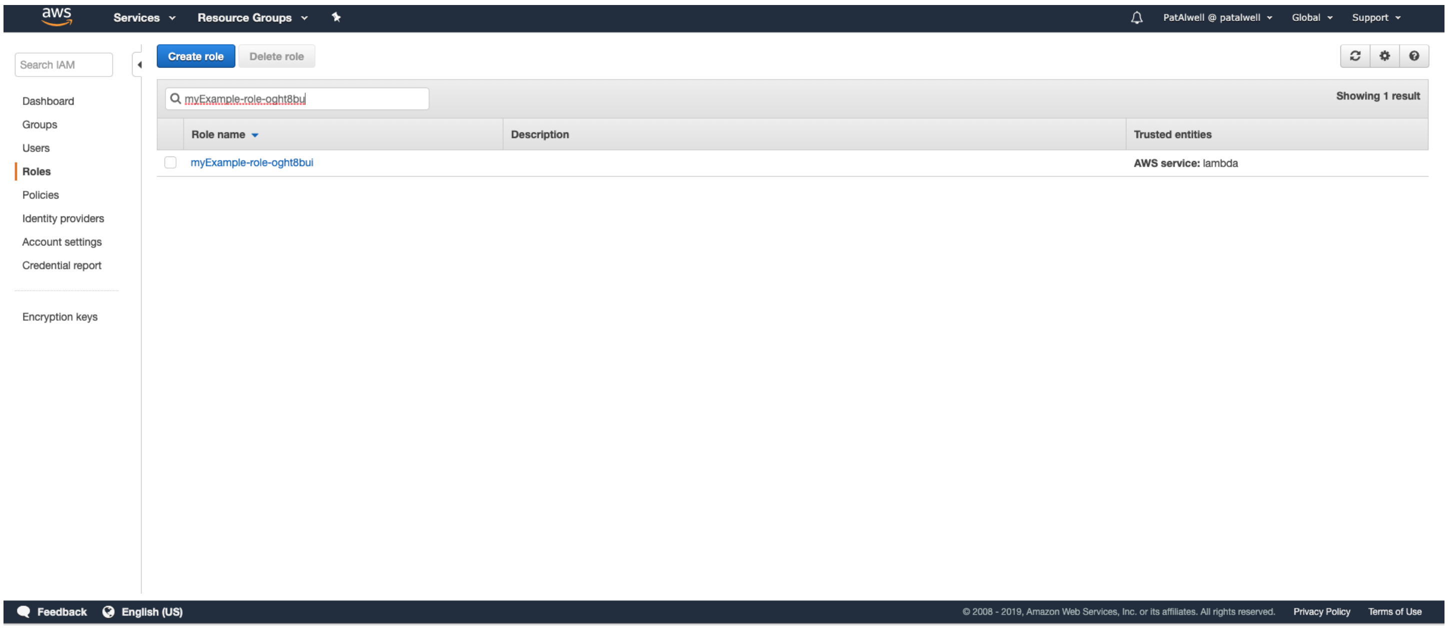


On the right, you'll notice the function's execution role. Let's keep track of the role's name and open another tab for AWS IAM to edit the role and grant access to the services we'll need to leverage in the function.

Set Up an IAM Role for Lambda

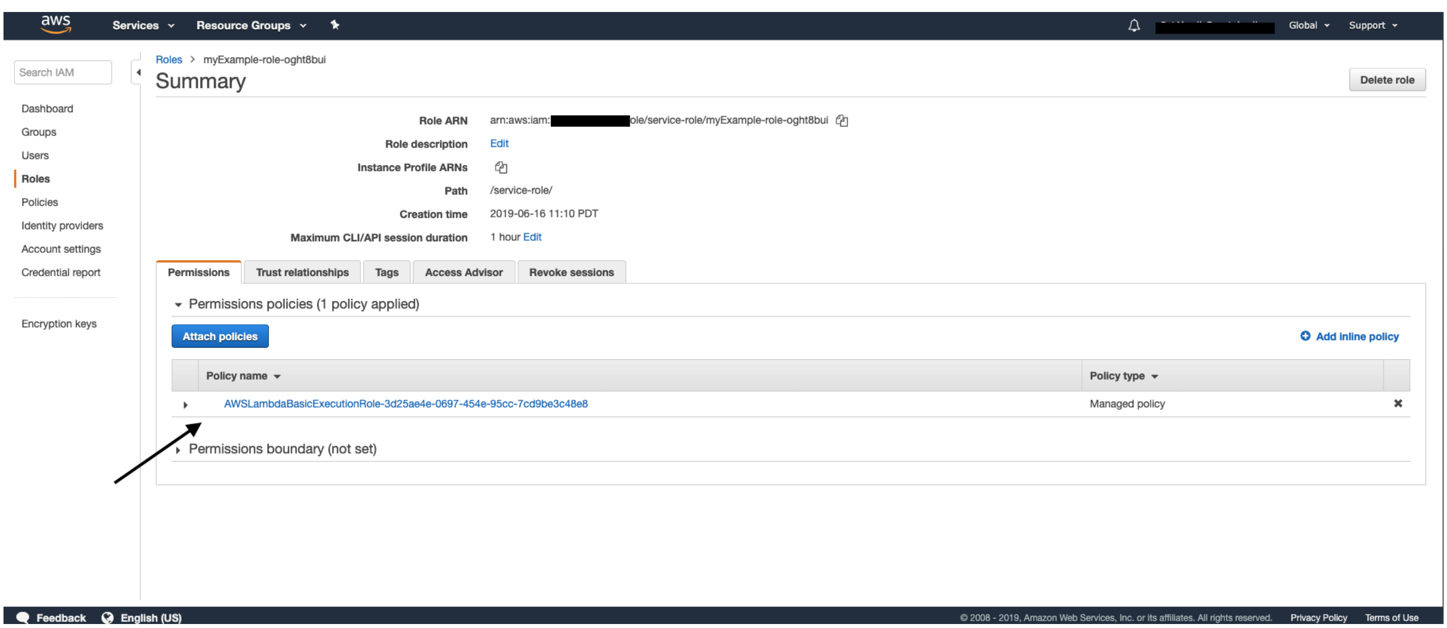
In the spirit of application security and best practices, we'll want to keep the execution role configured with the least amount of privileges the function needs to successfully launch our code. After we've opened another browser tab and navigated to the Identity and Access Management (IAM) portal, select roles from the right-hand menu and search for our role in the search box. In this case *myExample-role-oght8bui* (*Figure 3a*).

Figure 3a: *The IAM Roles Menu*



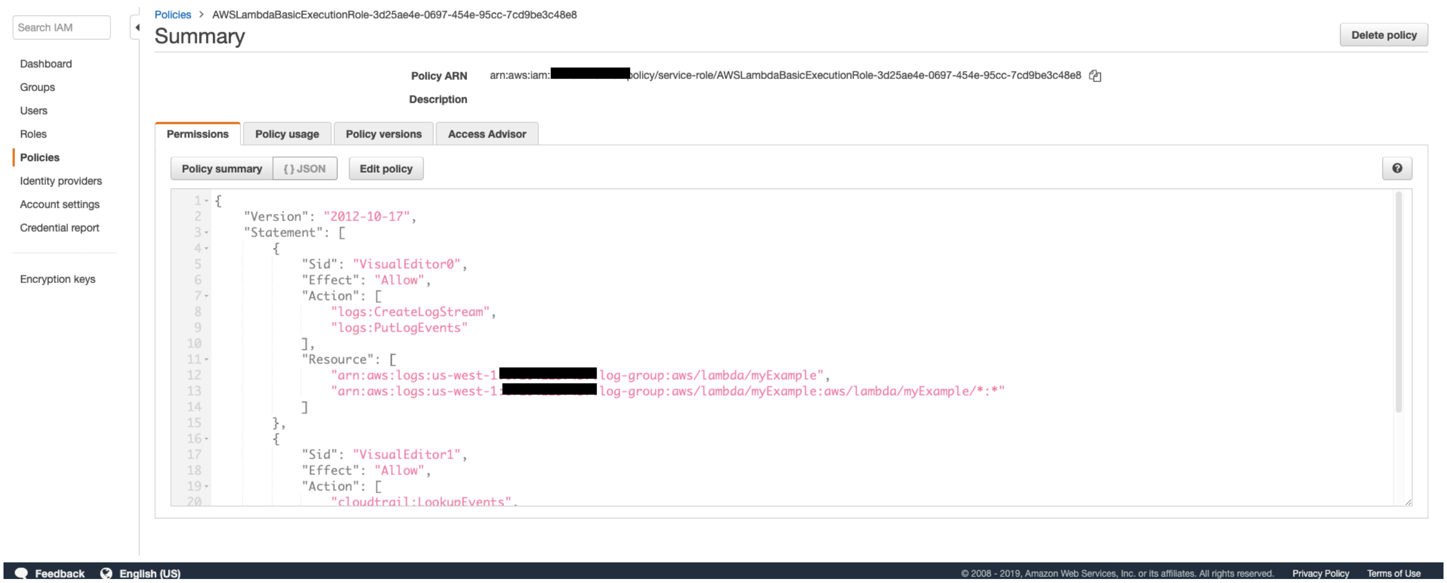
Click on the role and then click on the policy generated by Lambda so we can make the appropriate edits (*Figure 3b*)

Figure 3b: *Our Lambda Execution Role*



On the next page, click on the edit policy button (*Figure 3c*).

Figure 3c: *Our Lambda Execution Role's Policy*

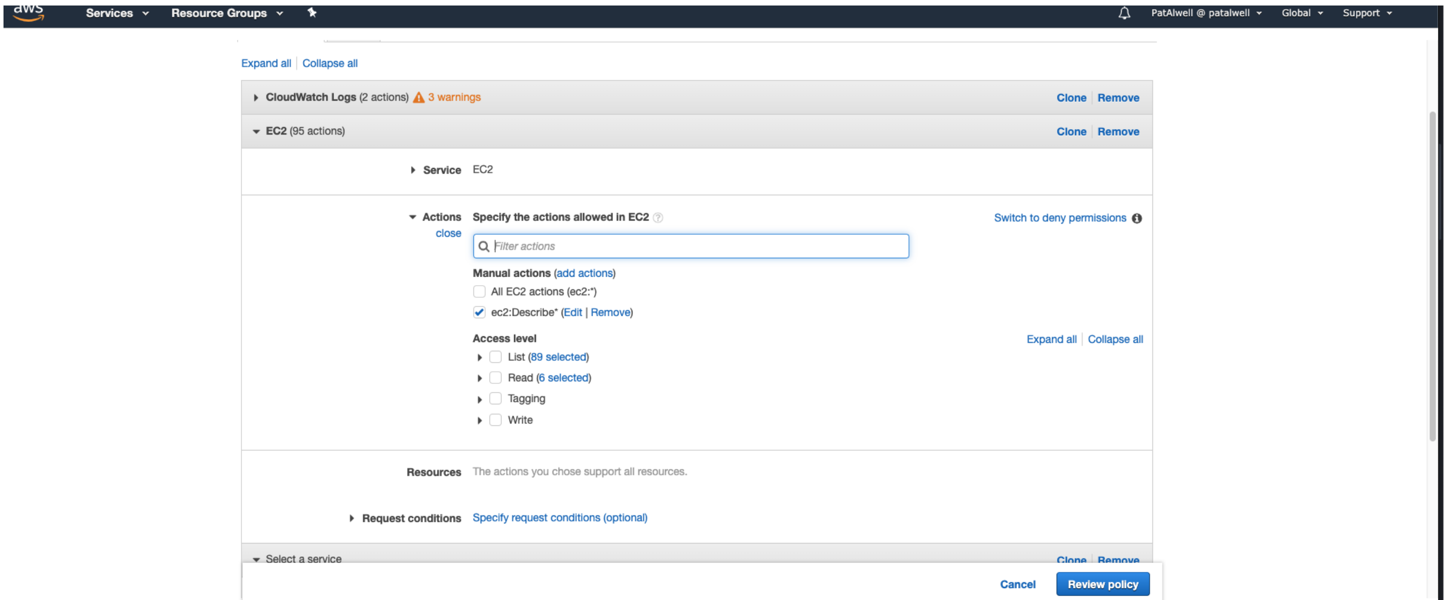


With IAM, you have the option to create your own policies using the AWS console or making the appropriate edits to the resulting JSON map that serves as a policy struct. In this case, let's use the user interface to author an appropriate policy for our execution role after hitting the edit policy button.

The default execution role only has permissions for CloudWatch Logs, so we'll need to add additional permissions for Amazon's EC2 and CloudTrail services. Click add additional services, choose a service, and select EC2.

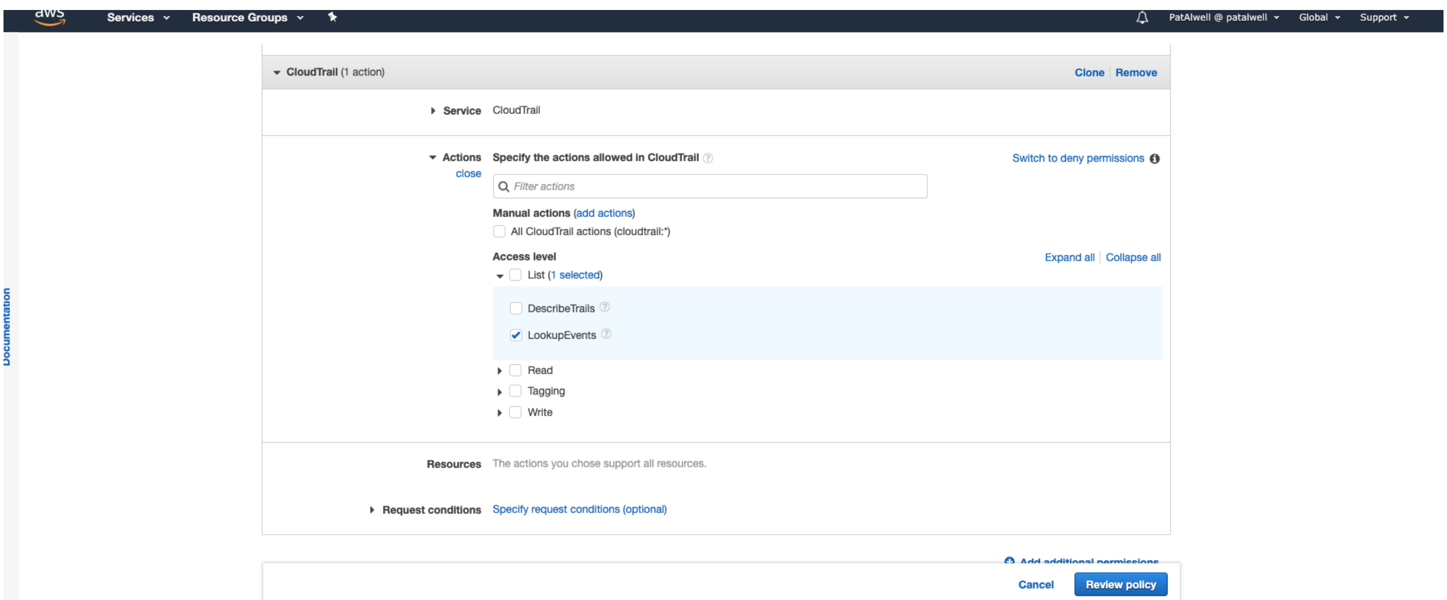
Next, specify a manual action by clicking add actions. Type Describe* into the prompt and click the add additional permission link again. In this case, we are giving the execution role the permission to invoke read-only methods on Amazon's EC2 API service so we are able to capture state after an instance, image, or volume has been created using the AWS API (*Figure 3d*).

Figure 3d: *Editing our Policy with the User Interface*



After clicking the add additional permissions link again, select the CloudTrail service and add permissions for the LookupEvents API method under the List option (*Figure 3e*).

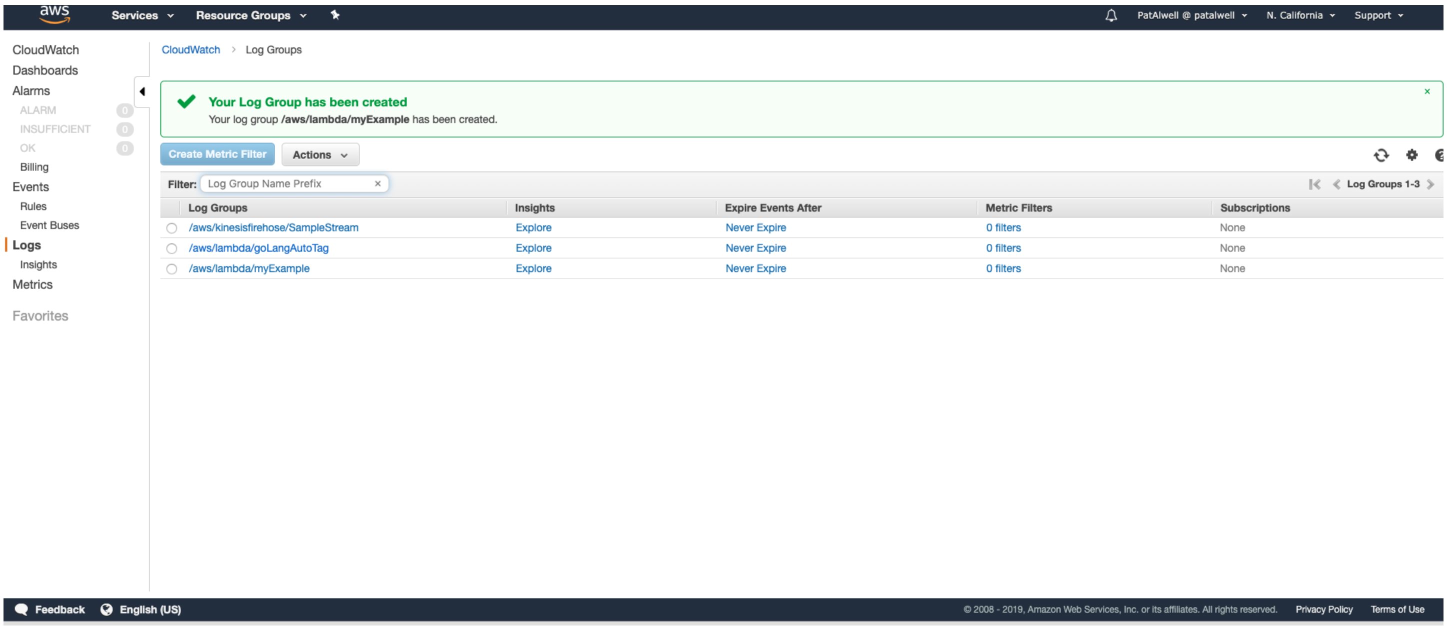
Figure 3e: *Editing our Policy with the User Interface*



Close the boxes by clicking on the grey headers and navigate back up top to edit the CloudWatch logging actions by clicking on the grey header box that says CloudWatch. You'll notice we have preconfigured write actions. These are the only write actions we'll need for our functions execution role. In essence, we are giving our function the ability to write logs to a devoted path in CloudWatch. What's more, this is the default location for logging Lambda functions, so we'll want to make sure we label our log-groups and log-streams accordingly. In that spirit, let's set up our log-groups and streams and then enter the paths and resource ARNs into our function policy.

Keeping our other tabs open, open yet another tab and navigate to CloudWatch. Click on logs in the right-hand side of the screen and the actions button to create a new log-group. I'm going to label my logGroup myExample, but you'll notice I've created two other log groups for other projects and services (*Figure 3f*).

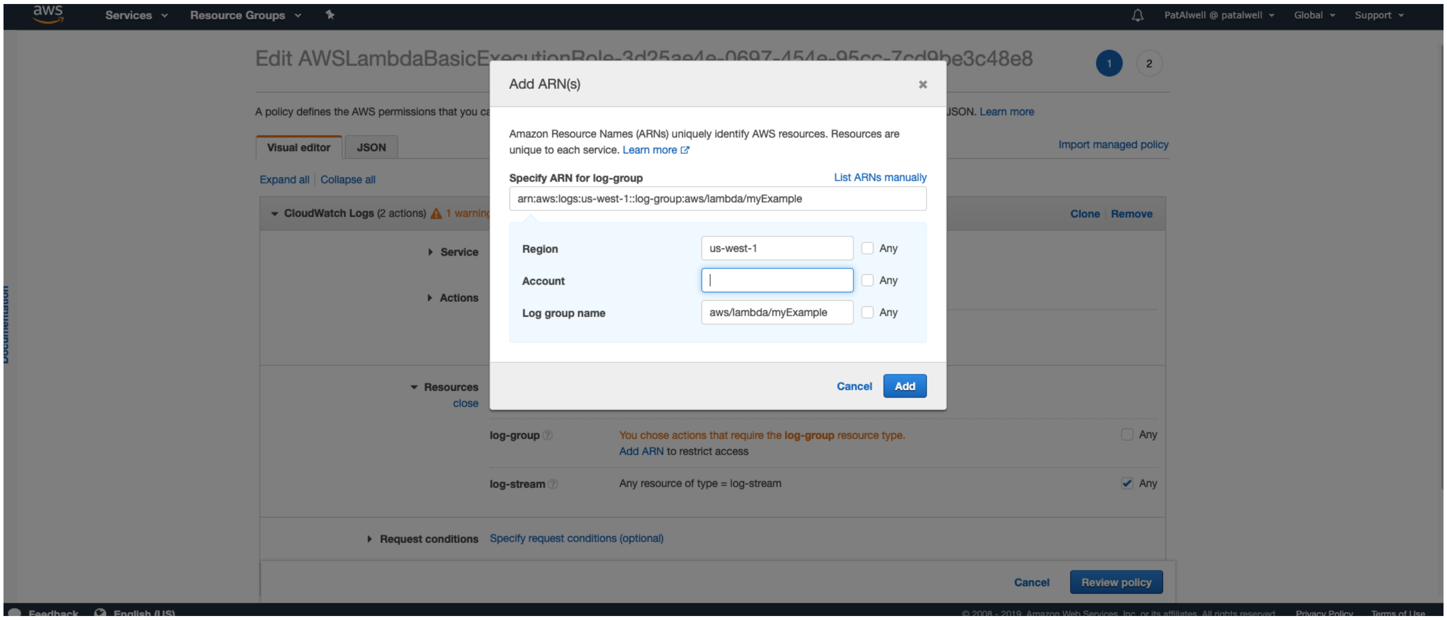
Figure 3f: *Creating Cloud Watch Log-Groups and Log-Streams*



We won't need to create log streams, as Lambda will do this for us upon execution of our function, but we will need to grab the ARN for the log-group in question. In this case, *aws/lambda/myExample.*

Switching tabs to IAM and our attention to the CloudWatch Logs box, you'll see warnings for log-groups and log-stream resources. We'll take care of these warnings by setting an ARN for the log-group. Click Add ARN to restrict access and enter the appropriate information into the dialogue box. See (*Figure 3g*) for an example.

Figure 3g: *Adding a Role ARN to CloudWatch Resource Policy*



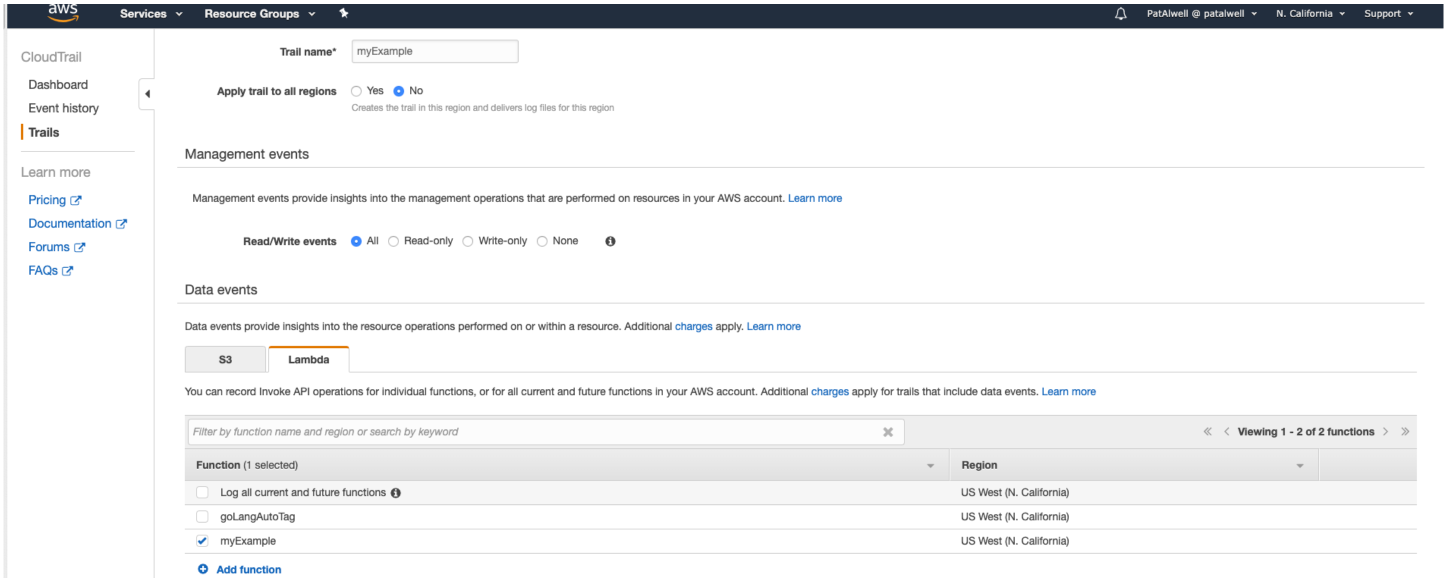
Next, check the any box next to log-stream to enable Lambda to write to any log-stream within the log-group. Click review policy and save changes. Close the browser tabs for IAM and CloudWatch logs.

Enabling CloudTrail for your Region

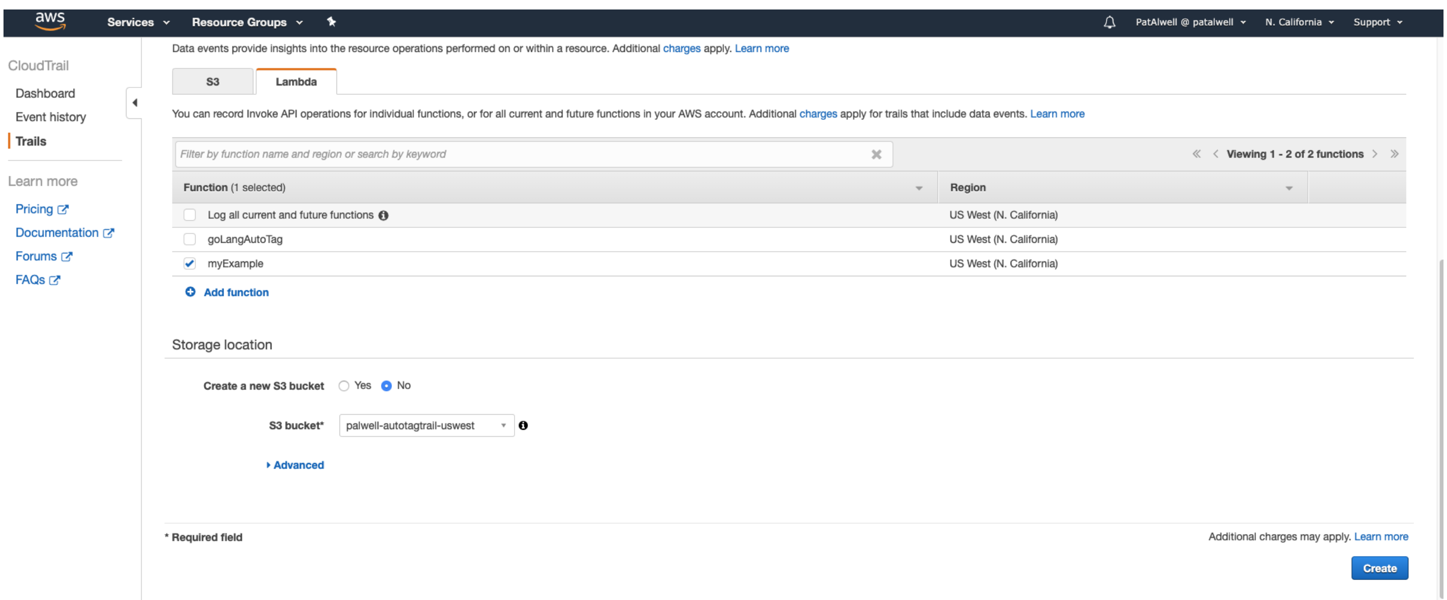
Now that we’ve set up our functions' event rules and permissions we can enable the CloudTrail service so our function has permission to lookup API events. CloudTrail is a billable AWS service that allows administrators to log AWS API events to S3. For more information see CloudTrail Pricing. I have this currently enabled and haven't been slammed with any crazy bills; e.g. nothing more than $10 per month.

Navigate to AWS services, Management and Governance, and finally CloudTrail. Click trails and proceed to add a new trail. Let’s assign our trail a name, opt to not apply the trail to all regions, and select all read/write events (*Figure 4a*).

Figure 4a: *Enabling CloudTrail for our Lambda Function*



Under data events select the Lambda tab and find the function we created, e.g. myExample. Finally, select a bucket to store our events. We can either create a new bucket for CloudTrail or select a bucket we've already created. In this case, I already had a devoted bucket for logging API events (*Figure 4b*). Click create and close the tab. The trail will automatically be enabled for your function.



Review the Go Code to Understand the AWS SDK Client Requirements

Now that we've created our serverless execution stack, we can review our application logic. Every lambda function in Go requires the use of the Handler function, which is a generic function that accepts type context and an event of type *[]byte* from AWS' github.com/aws/aws-lambda-go/lambda package. The handler function must be placed in the main package of our Lambda application and invoked at the bottom of our application as an argument to lambda.Start(). Lambda's start method must be located within our application's driver, e.g. *func main(){}.* You'll see this towards the end of our code examples.

In Lambda, a context type is an interface that provides one with methods and properties about our function and its execution environment. According to AWS, "*Lambda functions have access to metadata about their environment and the invocation request. This can be accessed via the context package. Should your handler include context.Context as a parameter, Lambda will insert information about your function into the context's value propert*y."

In other words, we can treat context like an object and use it to access metadata about our function and its state. See AWS: Go Lambda Context for more details.

```
package main

import (
    "context"
    "encoding/json"
    "fmt"
    "github.com/aws/aws-lambda-go/events"
    "github.com/aws/aws-lambda-go/lambda"
    "github.com/aws/aws-sdk-go/aws"
    "github.com/aws/aws-sdk-go/aws/session"
    "github.com/aws/aws-sdk-go/service/ec2"
    "github.com/patalwell/awsLambdaGoLangAutoTagEc2/CustomAwsLambdaEvent"
    "log"
    "strings"
    "time"
)


/*
Sample Event Type we need to unmarshall
We'll also need to unMarshall the Detail field into a custom Struct
See CustomAwsLambdaEvent package for struct fields

type CloudWatchEvent struct {
    Version    string          `json:"version"`
    ID         string          `json:"id"`
    DetailType string          `json:"detail-type"`
    Source     string          `json:"source"`
    AccountID  string          `json:"account"`
    Time       time.Time       `json:"time"`
    Region     string          `json:"region"`
    Resources  []string        `json:"resources"`
    Detail     json.RawMessage `json:"detail"`
}
*/


//HandleRequest is a required function for Lambda and GoLang
func HandleRequest(ctx context.Context, jsonEvent events.CloudWatchEvent) {
```

Since we're filtering for CloudWatch events in our Lambda execution stack, we'll want to use the CloudWatchEvent struct from AWS's Lambda Events Go package. https://github.com/aws/aws-lambda-go/tree/master/events. I've commented out a sample event for CloudWatch in the code sample above.

As you can see, this event is the "outer structure" of any particular CloudWatch event, so we'll need to define a custom struct to unmarshall the details portion of the CloudWatchEvent which is referenced as type *json.RawMessage,* a raw encoded JSON *[]byte* value we'll need to unmarshall into a custom struct.

To unmarshal the Details attribute in our event, we could potentially use a generic Go type like a *map[string]interface{}* , but this will make it difficult to manage types later down the line, so we should really define a custom struct for the details portion of the event as I've defined below.

Note: Structs are very similar to Classes in Java or Python and contain references to various attributes and their respective primitive types.

```
package CustomAwsLambdaEvent


import (
	"time"
)


//Custom Struct to Manage the Details Portion of the AWS CloudWatchEvent
//via events generated from API calls logged to CloudTrail
type CloudWatchEventDetails struct {
	EventVersion     string    `json:"eventVersion"`
	EventID          string    `json:"eventID"`
	EventTime        time.Time `json:"eventTime"`
	EventType        string    `json:"eventType"`
	ResponseElements *struct {
		OwnerID      string `json:"ownerId"`
		ImageId      string `json:"imageId"`
		SnapShotId   string `json:"snapShotId"`
		VolumeId     string `json:"volumeId"`
		InstancesSet struct {
			Items []struct {
				InstanceID string `json:"instanceId"`
			} `json:"items"`
		} `json:"instancesSet"`
	} `json:"responseElements"`
	AwsRegion    string `json:"awsRegion"`
	EventName    string `json:"eventName"`
	UserIdentity struct {
		UserName    string `json:"userName"`
		PrincipalID string `json:"principalId"`
		AccessKeyID string `json:"accessKeyId"`
		InvokedBy   string `json:"invokedBy"`
		Type        string `json:"type"`
		Arn         string `json:"arn"`
		AccountID   string `json:"accountId"`
	} `json:"userIdentity"`
	EventSource string `json:"eventSource"`
	ErrorCode   string `json:"errorCode"`
	ErrorMessage string `json:"errorMessage"`
}
}
```

Now that we have a means to unmarshalling our event, we can use Go’s JSON package to unmarshall our raw *bytes[]* into a pointer of type CloudWatchEventDetails.We'll also incorporate some error handling into the process to log any serialization errors during the unmarshalling process. After we've unmarshalled our event and event details, we can start to capture some relevant information for our function.

```
//Collect start time
        startTime := time.Now()


        //Create a variable for unmarshalling our event.Details
        var eventDetails CustomAwsLambdaEvent.CloudWatchEventDetails


        //Unmarshall the CloudWatchEvent Struct Details
        err := json.Unmarshal(jsonEvent.Detail, &eventDetails)
        if err != nil {
                log.Println("Could not unmarshal scheduled event: ", err)
        }


        //Marshall our eventDetails for clean logging
        outputJSON, err := json.Marshal(eventDetails)
        if err != nil {
                log.Println("Could not unmarshal scheduled event: ", err)
        }

        log.Println("This is the Json for cloudWatchEvents.details", string(outputJSON))
```

During runtime, our code will be scanning our eventDetails object for various event attributes like region, eventName, eventArn, principalId, userType, userName, EventTime, and InstanceId. We'll eventually want to store instanceIds, volumeIds, or imageIds so we'll also want to create a composite structure for persistence. In other languages, such as Java, we could have used the list interface *List<String> myList = new ArrayList*<>; to create a composite type and capture similar details, however things are a little different in Go. In fact, the official Go programming specification recommends using a slice instead of an array.

Arrays in go are actually values as opposed to reference types. Slices, on the other hand, are references and far more flexible than arrays. Seems counter-intuitive, but you don't want to pass an array to a function as you'd have to pass by copy (*super inefficient*) or use a pointer. In either case, you'd want to use a slice as managing pointers to array's defeats the purpose of having a dynamic data structure in the first place. For more details on slices and arrays in Go see: Using Slices in Go

Long story short, the ids[] slice will be used to capture the instance, volume, or image ids for use during the tagging portion of our application.

```
//Create a slice to store Id's
        var ids []string


        region := jsonEvent.Region
        detail := eventDetails
        eventName := eventDetails.EventName
        eventArn := eventDetails.UserIdentity.Arn
        principal := eventDetails.UserIdentity.PrincipalID
        userType := eventDetails.UserIdentity.Type
        user := eventDetails.UserIdentity.UserName
        date := eventDetails.EventTime
        instanceId := eventDetails.ResponseElements.InstancesSet.Items[0].InstanceID


        if userType == "IAMUser" {
                user = eventDetails.UserIdentity.UserName
        } else {
                //split the principal
                user = strings.Split(principal, ":")[5]
        }


        log.Println("principalId: ", principal)
        log.Println("eventArn", eventArn)
        log.Println("awsRegion: ", region)
        log.Println("eventName: ", eventName)
        log.Println("eventTime: ", date)
        log.Println("userName: ", user)
        log.Println("instanceId: ", instanceId)


        //if not detail['responseElements']:
        if detail.ResponseElements == nil {
                log.Println("No Response Elements Found")
                if detail.ErrorCode == "errorCode" {
                        log.Println("errorCode: ", detail.ErrorCode)
                }
                if detail.ErrorMessage == "errorMessage" {
                        log.Println("errorMessage: ", detail.ErrorMessage)
                }
        }
```

Next, we'll use eventName to filter through the cloudWatch eventDetails we are collecting. We'll also create an EC2 client in order to invoke the CreateTags method. What's more, we'll capture some metadata about the event by using the AWS API to describe our current instance by filtering on the instanceIds we've captured from the "RunInstances" event type.

Specifically, we are looking to see if we can tag any network interfaces or EBS volumes attached to the instance.

```
    // Create a EC2 client
        mySession := session.Must(session.NewSession())
        ec2Client := ec2.New(mySession)


        //Filter by eventName
        switch eventName {


        case "CreateVolume":
                ids = append(ids, detail.ResponseElements.VolumeId)
                log.Println("Here are the volume Ids: ", ids)


        case "RunInstances":
                items := detail.ResponseElements.InstancesSet.Items
                for _, v := range items {
                        ids = append(ids, v.InstanceID)
                }
                log.Println("Here are the instance Ids: ", ids)
                log.Println("Number of Instances ", len(ids))


                //Check filtering by instance Id
                base, err := ec2Client.DescribeInstances(&ec2.DescribeInstancesInput{
                        Filters: []*ec2.Filter{
                                {

//https://docs.aws.amazon.com/cli/latest/reference/ec2/describe-instances.html
                                        Name: aws.String("instance-id"),
                                        Values: []*string{
                                                aws.String(strings.Join(ids, ",")),
                                        },
                                },
                        },
                })


                if err != nil {
                        log.Println("There was an issue getting our instances by Id :", err.Error())
                }


                //fmt.Println("Here is the base: ", base.Reservations)
                fmt.Println("Here are the ids: ", ids)


                for i, res := range base.Reservations {
                        log.Println("Number of instances: ", len(res.Instances))
```

```
                        for idx, inst := range base.Reservations[i].Instances {
                                //The ID of the network interface attachment.
                                log.Println("Here is the Id of the network interace attachment: ",
                                        *inst.NetworkInterfaces[idx].Attachment.AttachmentId)
                                ids = append(ids,
*inst.NetworkInterfaces[idx].Attachment.AttachmentId)
                                //The ID of the EBS volume attached to an instance.
                                log.Println("Here are the EBS Volume Ids ",
*inst.BlockDeviceMappings[idx].Ebs.VolumeId)
                                ids = append(ids, *inst.BlockDeviceMappings[idx].Ebs.VolumeId)
                        }
                }


        case "CreateImage":
                ids = append(ids, detail.ResponseElements.ImageId)
                log.Println("Here is the Image Id: ", ids)


        case "CreateSnapshot":
                ids = append(ids, detail.ResponseElements.SnapShotId)
                log.Println("Here is the Snapshot Id: ", ids)


        default:
                log.Println("Not a Supported Action")


        }
```

Afterwords, we'll unload our slice and tag the objects with a predefined set of key pair values {Key,Value}. We can also use some of the variables we referenced at the top of our application like user and date. The rest of the keys will populate empty strings so our users can customize the tags the way they see fit.

```
	//If our Id's Slice isn't null,
	// iterate over the Ids and tag our instances per id
	if len(ids) != 0 {
		for _, v := range ids {
			log.Println("Tagging Resource", v)


			_, errTag := ec2Client.CreateTags(&ec2.CreateTagsInput{
				Resources: []*string{aws.String(v)},
				Tags: []*ec2.Tag{
					{Key: aws.String("Name"),
						Value: aws.String(" ")},
					{Key: aws.String("Service"),
						Value: aws.String(" ")},
					{Key: aws.String("Application"),
						Value: aws.String(" ")},
					{Key: aws.String("Team"),
						Value: aws.String(" ")},
					{Key: aws.String("Owner"),
						Value: aws.String(user)},
					{Key: aws.String("CreateDate"),
						Value: aws.String(date.String())},
				},
			})


			if errTag != nil {
				log.Println("Could not create tags for instances with Id: ", v,
errTag)
			}
		}
	}

	//Get execution time
	executionTime := time.Since(startTime)
	log.Printf("Remaining time: %s\n", executionTime)
}

func main() {
	lambda.Start(HandleRequest)
}
```

Finally, we’ll log any errors from the execution of the function and print our execution time. You'll also notice that we close the original func HandleRequest() and include it in our package driver main() at the bottom of the application.

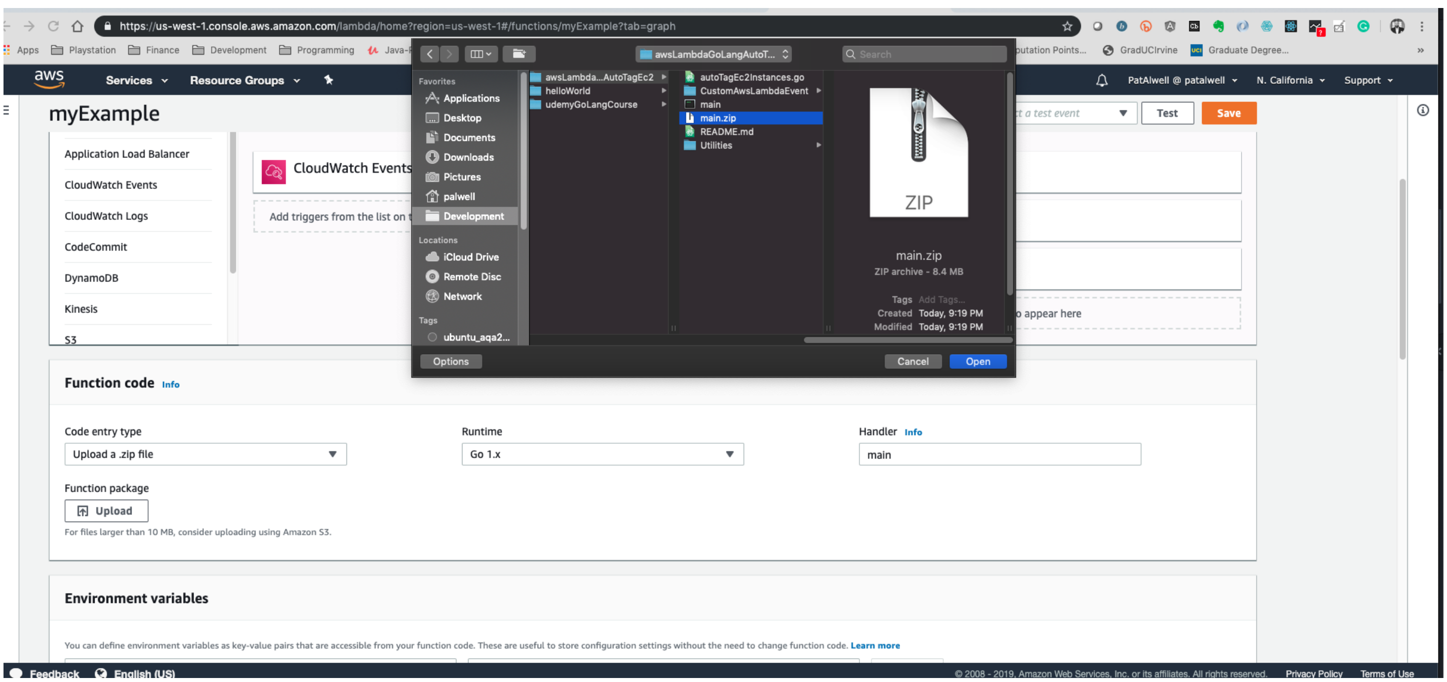
Compile the App and Launch a Test Event

Now that we've reviewed the code, we can compile our application and run a test event. Navigate to the awsLambdaGoLangAutoTagEc2 package, compile our application, and compress it with a GZIP client. In order to have Lambda run our Go executable, we must use certain flags for the Go compiler as outlined below.

```
$cd ~/DevDir/goLang/src/github.com/patalwell/awsLambdaGoLangAutoTagEc2

$GOOS=linux GOARCH=amd64 go build -o main autoTagEc2Instances.go

$zip main.zip main
```

Next, we'll want to load this to the Lambda user interface and provide details on its name. Navigate over to the Lambda user interface and click on the lambda icon in the designer template. This will load a footer modal called Function code. Click the upload button and select the Go executable we just compiled and zipped main.zip. Next, type main in the Handler dialogue and click the orange save button in the upper right corner (*Figure 5a*).

Figure 5a: *Loading our Go Executable to Lambda*



Navigate to the same menu as the save button and click the faded select a test event menu button and configure test event. This will launch a modal that allows you to upload a test event for our function. I've provided a sample test event for our function in the goPackage. Simply navigate to the package's *Utilities/* directory and open the *sampleEvent.json* file. Copy the JSON and paste it into the configure events modal. Name the test event and click create (*Figure 5b-5c*).

Figure 5b: *Our Sample CloudWatch Event*

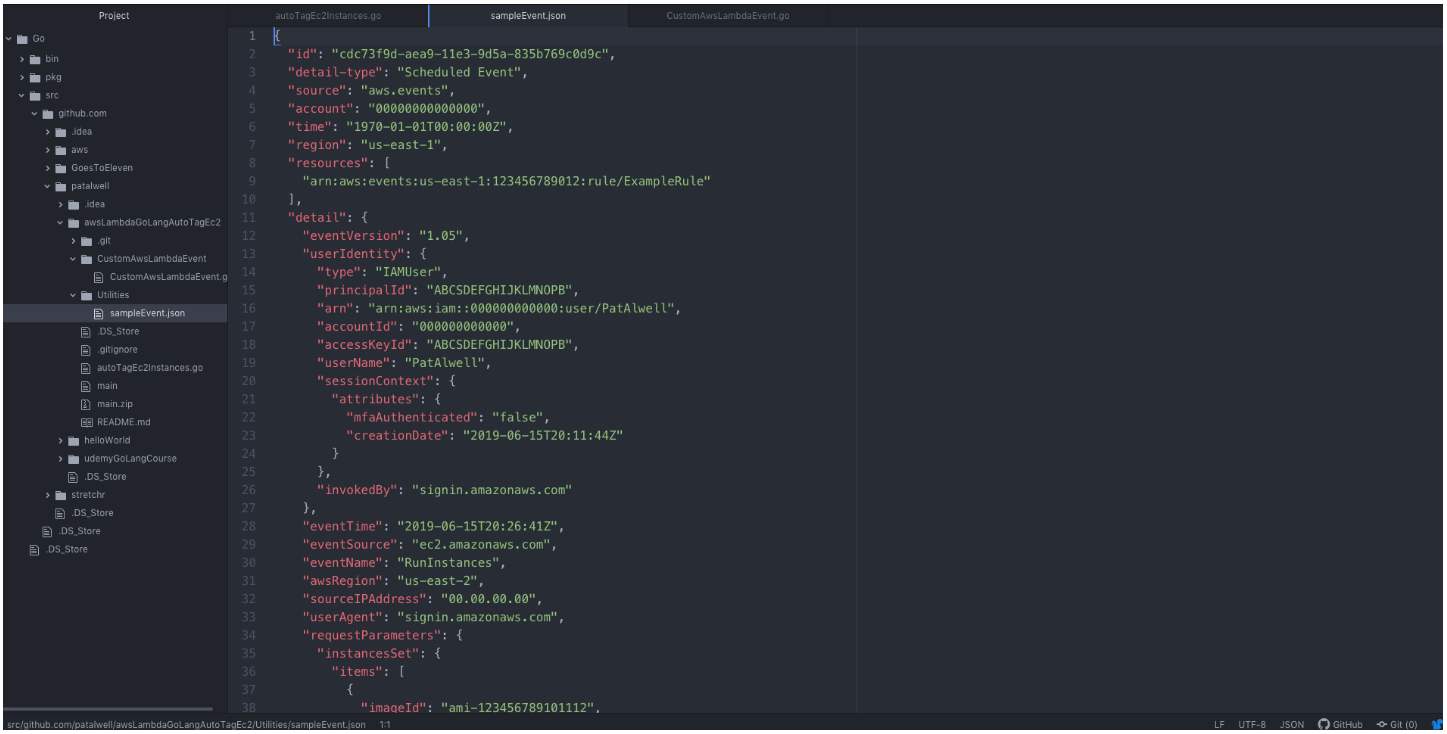
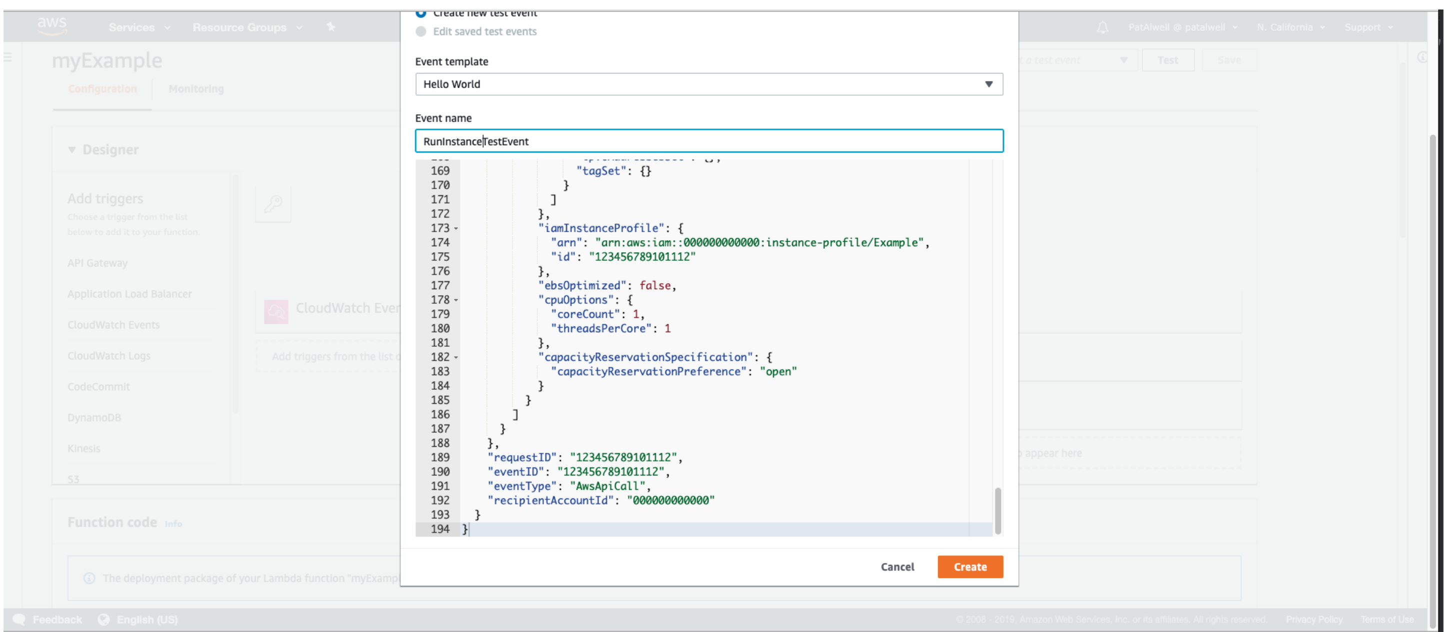
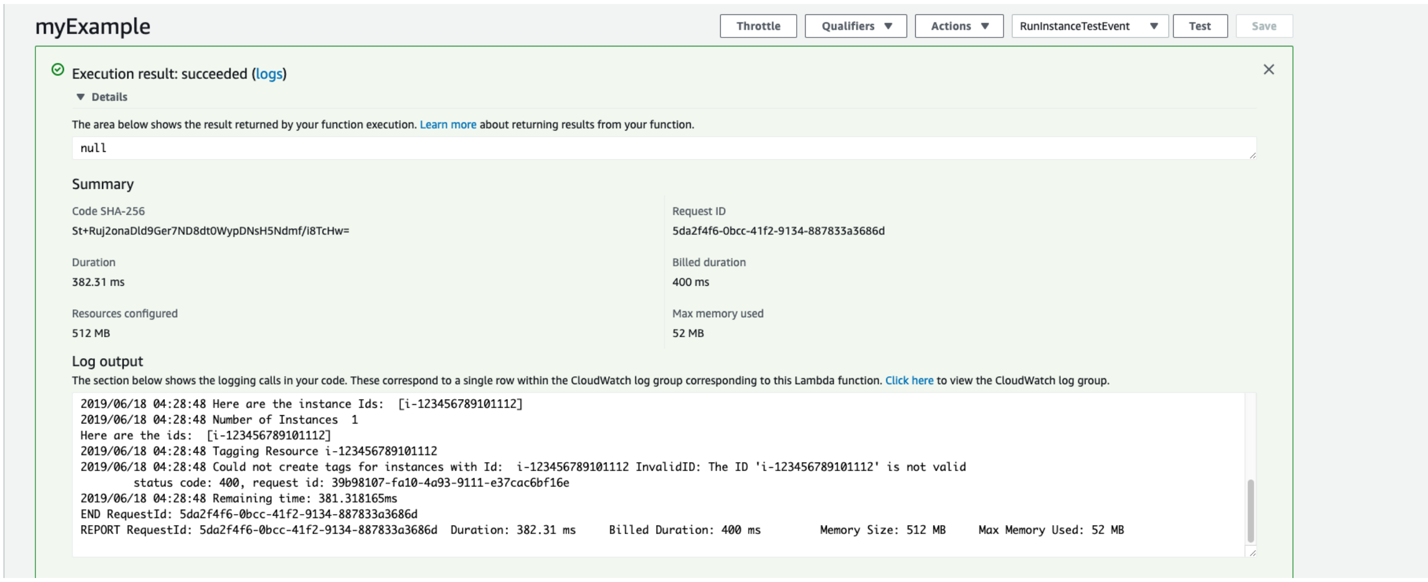


Figure 5c: *Uploading our Event as a Test to the Lambda Function*



Click test and you should see results within a few seconds. You'll run into an error as I've shown below. This is because the function isn't actually able to tag an instance with that Id because it doesn't exist. You should still be able to run an instance and have it automatically tagged regardless (*Figure 5d*).

Figure 5d: *Console Output From our Test*



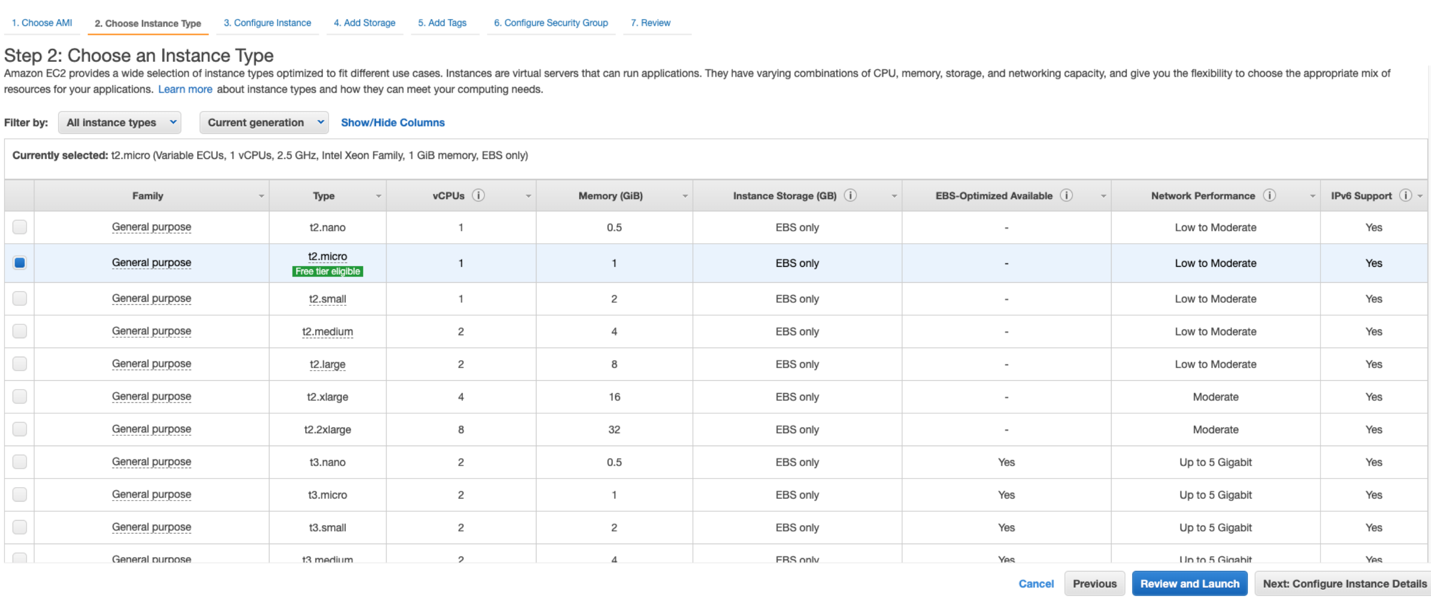
Running an Instance and Following the Flow of Events

Now that we've setup Lambda, reviewed our code, compiled our application, and performed a test, we can review the flow of events. Moreover, we'll want to understand how the API call triggers and logs the event in order to troubleshoot any issues with our code.

Log onto the AWS console and navigate to services, compute, EC2. You'll be taken to the EC2 user interface and provided with an option to create instances via a button located at the top left of the screen.

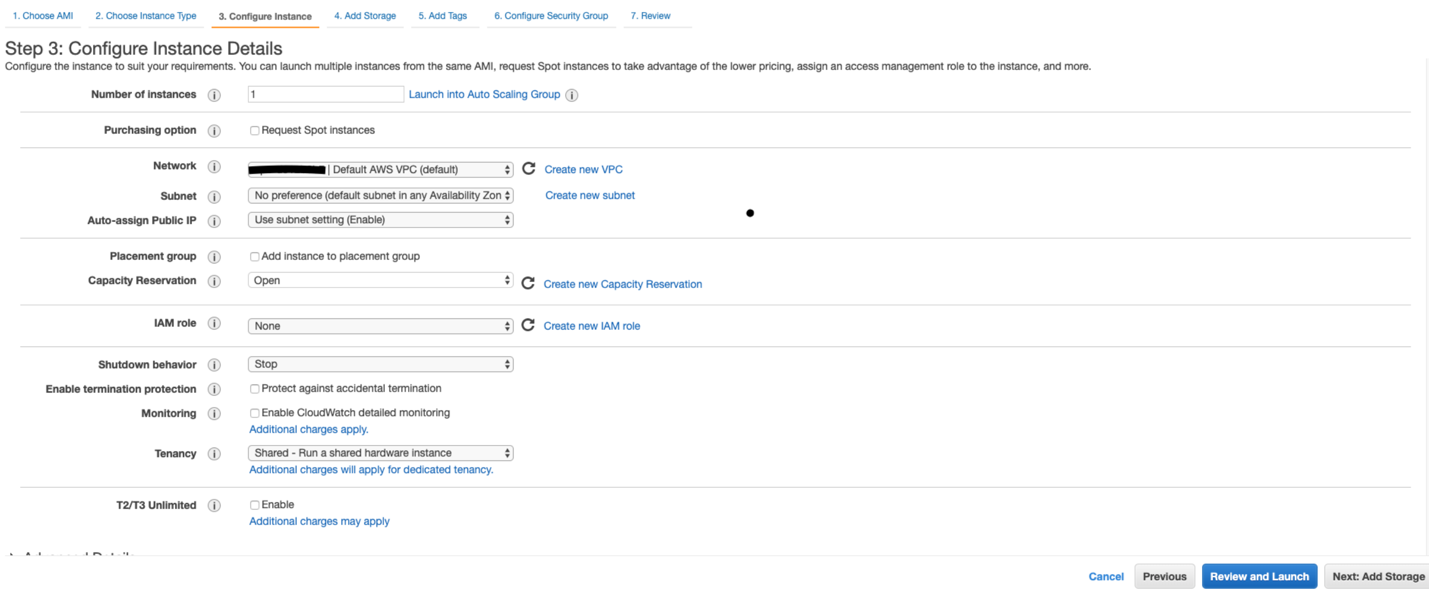
Click the run instances button and select an Amazon Linux image from the quick start menu. Let's make it free tier eligible by keeping the default selection of t2.micro for instance type (*Figure 7a)*.

Figure 7a: *Selecting our Image Type*



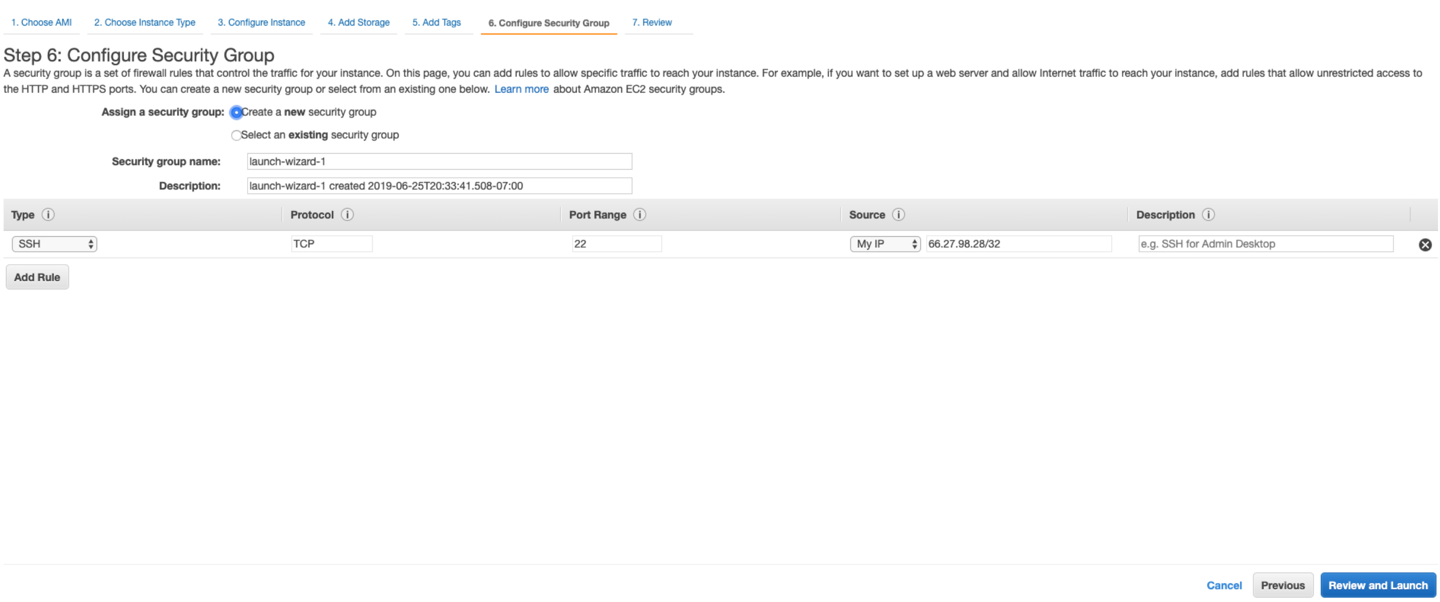
Next, let's click configure details and make sure it's in our default VPC or a VPC we previously created that supports some kind of subnet. We won't need to assign a role and we can keep the IP assignment for the instance to the default (Figure 7b).

Figure 7b: *Configuring Instance Details*



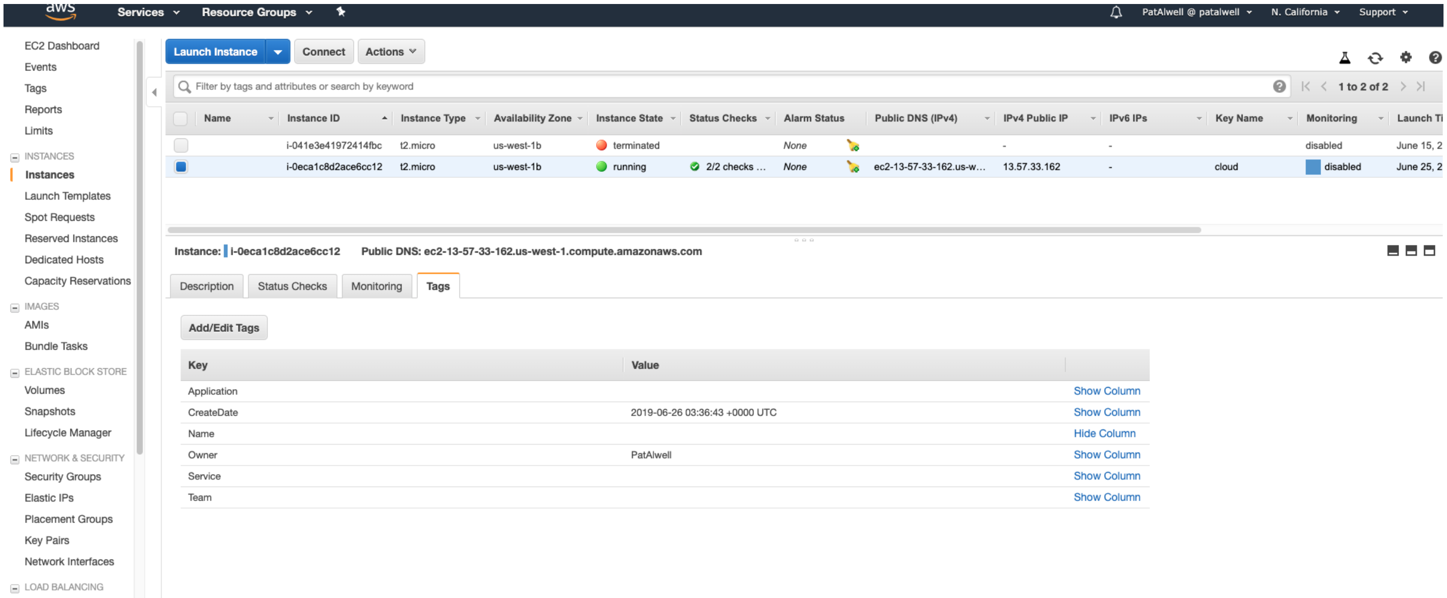
Click next: *add storage* and leave the defaults. Click next: *add tags* and leave these blank. Click next: *configure security group*. You can select a security group you previously configured on this screen or simply create a new one that tightens down inbound network access to your IP address (*Figure 7c*).

Figure 7c: *Configuring Security Groups*



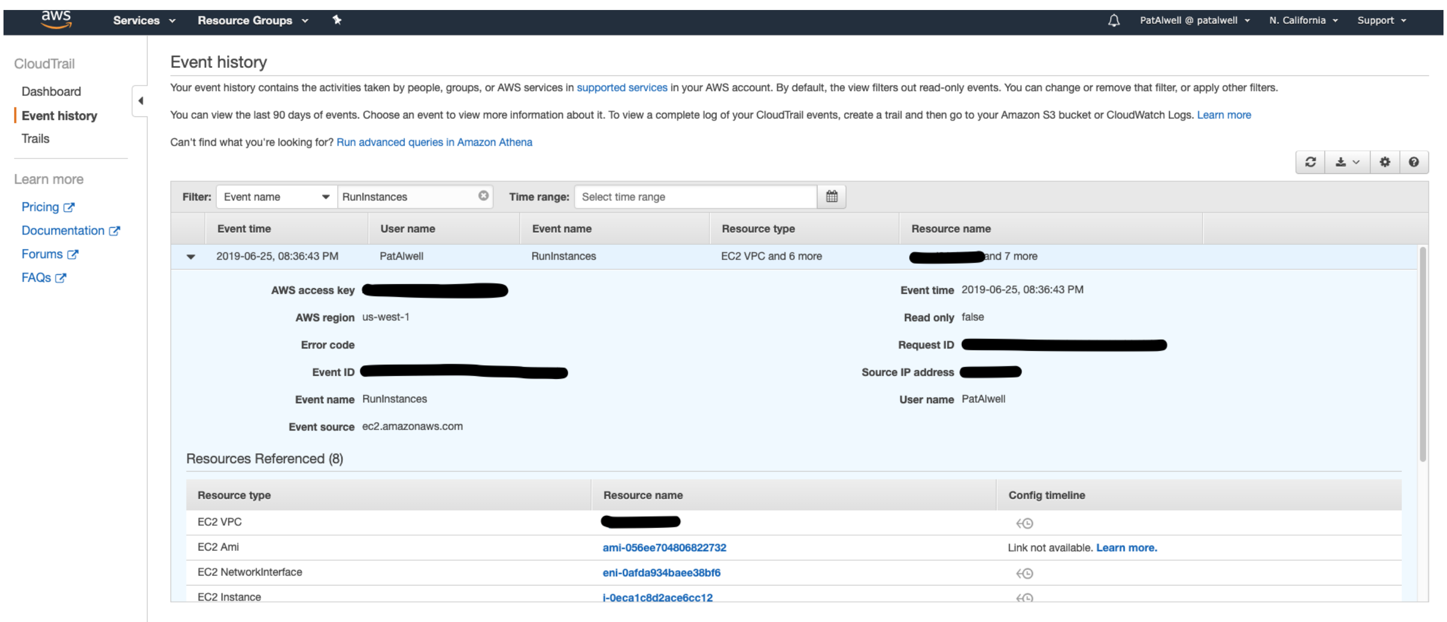
Finally, click review and launch. We don't need a key for this instance since it's serving as a deployment test. Now that we've started our instance, we should see our tags within a few minutes *(Figure 7d).*

Figure 7d: Tags Applied to our Instance



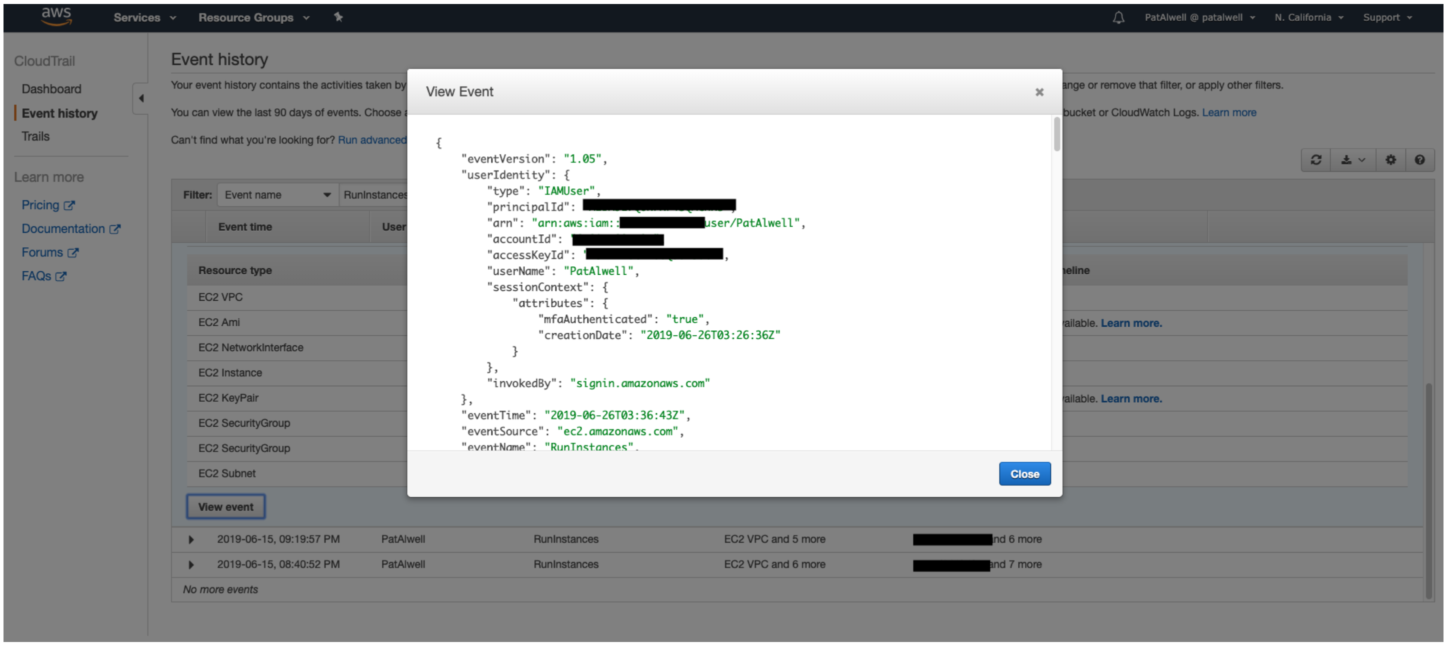
It will still take upwards of 15 min for the event to be populated to CloudTrail, so take a break, open a new tab, and navigate to cloud trail. After 15mins perform a search on the runInstances() method by filtering on event name. You should see the event we triggered by starting an instance. (*Figure 7e*).

Figure 7e: *Cloud Trail Event History Management Console*



If we open the line item for our event in question, we'll notice the resource details of our event and the option to view our event at the bottom of the details section (*Figure 7f*). Click view event. You now have the event details to our Lambda function e.g. the custom struct we authored and packaged via CustomAwsLambdaEvent and the details of the event we are passing into the Handler function.

Figure 7f: *Event Details of our API Event for RunInstances*



Note: The event we are viewing in (*Figure 7e*) is ***not*** the full event, but rather the event details section of the AWS CloudWatch event below. This is *also* why I thought it was necessary to show the API event process. In the future, if we want to create functions for different events, we'll need to find examples of how the parent CloudWatch event creates it's subsequent child event details. We can use this cloud trail management console to give us an idea of how to unmarshal event details our json.RawMessage below.

```
type CloudWatchEvent struct {
        Version    string          `json:"version"`
        ID         string          `json:"id"`
        DetailType string          `json:"detail-type"`
        Source     string          `json:"source"`
        AccountID  string          `json:"account"`
        Time       time.Time       `json:"time"`
        Region     string          `json:"region"`
        Resources  []string        `json:"resources"`
        Detail     json.RawMessage `json:"detail"`
}
```

Below is a sample of the event I provided in the */utilities* directory of our project, shortened for brevity. As you can see, the detail{} portion of our cloudwatch event from the management console contains the same fields.

```
{
  "id": "cdc73f9d-aea9-11e3-9d5a-835b769c0d9c",
  "detail-type": "Scheduled Event",
  "source": "aws.events",
  "account": "00000000000000",
  "time": "1970-01-01T00:00:00Z",
  "region": "us-east-1",
  "resources": [
    "arn:aws:events:us-east-1:123456789012:rule/ExampleRule"
  ],
  "detail": {
    "eventVersion": "1.05",
    "userIdentity": {
      "type": "IAMUser",
      "principalId": "ABCSDEFGHIJKLMNOPB",
      "arn": "arn:aws:iam::000000000000:user/PatAlwell",
      "accountId": "000000000000",
      "accessKeyId": "ABCSDEFGHIJKLMNOPB",
      "userName": "PatAlwell",
      "sessionContext": {
        "attributes": {
          "mfaAuthenticated": "false",
          "creationDate": "2019-06-15T20:11:44Z"
        }
      },
      ...
    "requestID": "123456789101112",
    "eventID": "123456789101112",
    "eventType": "AwsApiCall",
    "recipientAccountId": "000000000000"
  }
}
```

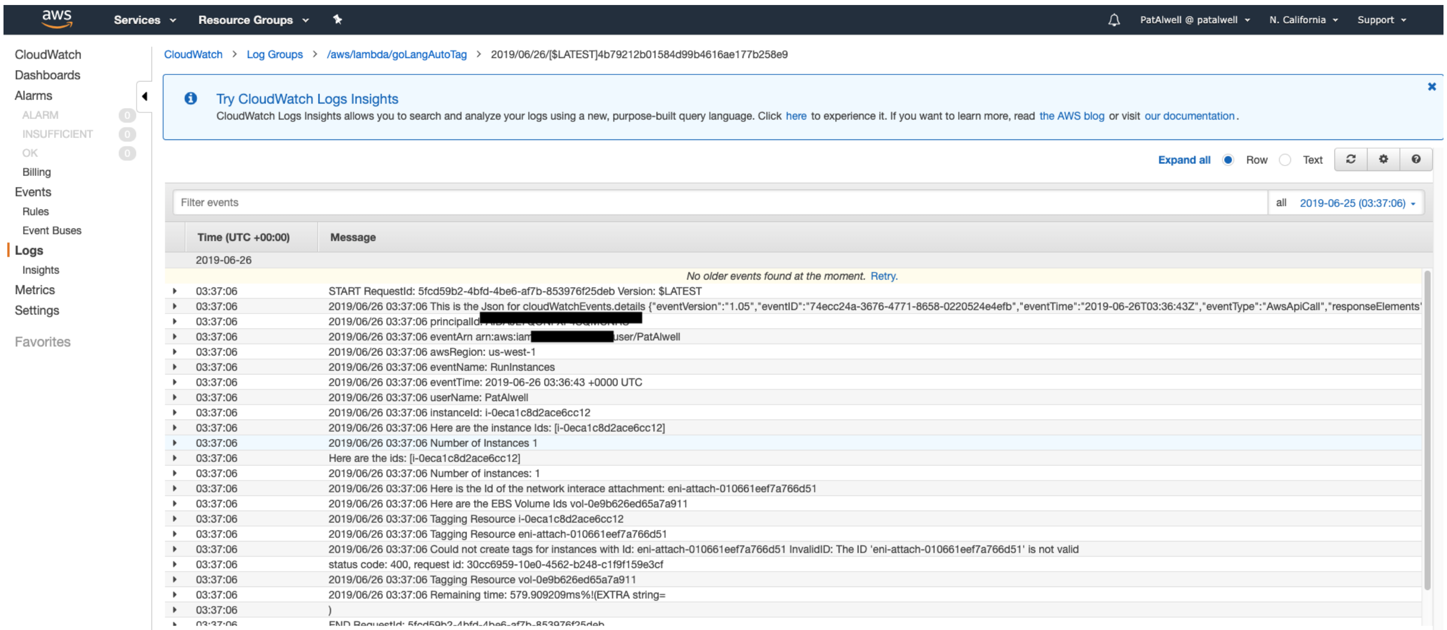

Note: the event details often contain *sensitive information* about your account e.g. access keys, role ARNs, etc. If you want to make a testing framework public, be sure to clear or randomize the sensitive fields as I have done in the project's utilities directory.

```
{
  "id": "cdc73f9d-aea9-11e3-9d5a-835b769c0d9c",
  "detail-type": "Scheduled Event",
  "source": "aws.events",
  "account": "00000000000000",
  "time": "1970-01-01T00:00:00Z",
  "region": "us-east-1",
  "resources": [
    "arn:aws:events:us-east-1:123456789012:rule/ExampleRule"
  ],
  "detail": {event from cloudtrail event modal goes here}
}
```

Function Logging During Execution

To view the functions' logs simply navigate to services, cloudwatch, logs. Select the appropriate log group and select the most recent stream. You'll see the output of our logging and runtime displayed in the user interface (*Figure 7g)*.

Figure 7g: *Lambda Function Logs via Cloudwatch*



In essence, the flow of events starts with a user interacting with the AWS console, our function fires when our API event is posted to CloudWatch, and our flow ends when tags are added to the instances, volumes, or images we've created.

Parting Words

In this post we learned about Go, Lambda, and ran some GoLang on a serverless execution stack. We also learned how we could leverage Lambda to take advantage of automating some of our development tasks, like tagging instances. Some other folks, myself included, were struggling to understand how to properly unmarshall the event details portion of a given event payload, (https://github.com/aws/aws-lambda-go/issues/51) so I also wanted to make sure other developers understood the process. Please feel free to shoot me an email or direct message me with questions or concerns.

Published by

#GoLang #AWS #Serverless #Lambda #microservices

I've taken the liberty to write a brief tutorial on using GoLang with AWS lambda. There were a few folks struggling to make sense of the unmarshalling process, so I wanted to make this was a non issue in the future :)