**Session 1: Deploying over AWS**

1. Serverless Computing: building and running applications without thinking about servers.
2. Server is encapsulated from the user, hidden from the user, providing services easily.(server abstracted from the user)
3. Cloud computing server – AWS, google cloud, Microsoft etc..
4. Cloud computing servers hides from infrastructure management tasks:
5. Server or cluster provisioning – automatically scale up or scale down systems/users
6. Operating system maintenance – even software at lowest level changes, not affects the other servers which on line.
7. Capacity provisioning – initially one lambda instance per user, according to services requests lambda intsances will become many. So capacity will be managed by the AWS.
8. Downtime – if a server there, goes down means it goes down. But according to lambda there is no down of server because its going to be million lambda clusters. (many backups there if one instance goes down too)
9. The unit of execution here is a function means service provider gives function as a service. Faas – Function as a service., charge for particular function running on a system.
10. Pricing based on how many resources we use there.
11. High availability is always there in AWS, for any application to be built up or any backend service to be built and run up its always available to handle the things.

We are going to use AWS Lambda server

We may use S3 for storage, Topserv, lambda amd sometime sagemaker too.

**AWS Lambda – Serverless Computing platform**

1. Runs code without provisioning or managing servers.
2. No charge when you not running the code, only charges when we consume it.
3. Warm startup will be there, when v try to open some application which we already had in lambda, it will be immediately available when next time we open it.
4. Automatically scales up the application by running code in response to each trigger. Means in one instance if starts, triggers events and ran by next instance followingly. code runs parallelly and process each trigger individually scales with the size of workload.
5. Charged for every 100ms the code executes.
6. Can be triggered by different events. Mostly ours will be http requests means when we try to do its going to send request as a form submission. (Get / Post method)
7. Function code and its dependencies should not be greater than 250 MB. This is going to be challenged how we are going to deal with this we will see.

Example of Lambda use:

Photograph has taken ----🡪 uploaded into Amazon S3 (Storage) -----🡪 triggers into AWS Lambda ----🡪 where it applies many filters and according to our phone configuration it processes and then we download the image (small one) --🡪 means photo is resized into mobile, tablet and etc..

1. Runs on Amazon Linux OS, its not own operating system so to maintain compatability issues, we need Docker.(i.e). inside Lambda runtime container.(The company docker for Amazon)
2. The more RAM we allocate, the faster CPU we will get. From 128 MB to 3008MB we can allocate in AWS in 64 increments.(Means 128, 128+64, 192+64, etc…)
3. Lambda layers where we can have pytorch kind module functions. Maximum 500MB storage can be added whenever extra memory requires to upload or download from S3 container.
4. Free tier. We van have 1 million requests(in the sense something image or object to be stored in S3 conainer) after 1 million requests $0.0000002 per requests thereafter. (3008/128 = 23.5 units(requests) charged accordingly.)
5. 400000GB seconds of compute time per month ($0.000000208 for every 100 ms used thereafter , 128 MB of RAM.

**What is Serverless Framework?**

It is an open source software, free framework and web framework written using Node.js

Its initially only built to work with Lambda but now it can be work with any other Serverless computing service.

Its first framework developed for building applications on AWS Lambda , platform provided by Amazon as a part of AWS, but now can be worked with others as well.

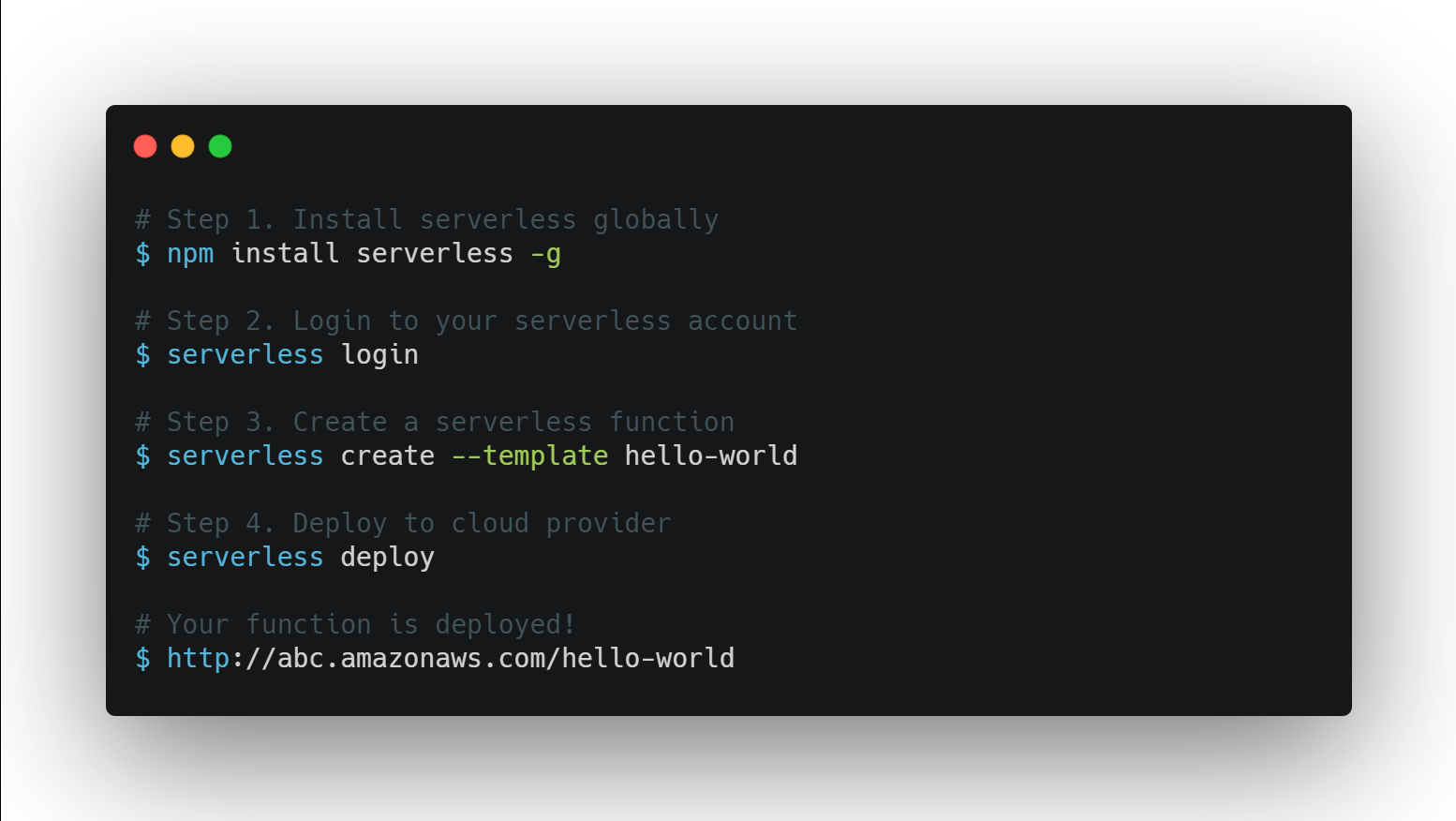
All backend configurations going to be handled by Serverless framework.

* A serverless app can be simply be a couple of lambda functions to do some tasks or an entire backend composed of hundreds of lambda functions.
* Used to build serverless applications via command line and config files.
* Uses Amazon cloud formation under hood which allows to describe and provision all infrastructural resources need using a single JSON file.
* Once we define this file, all resources can be created using this cloud.

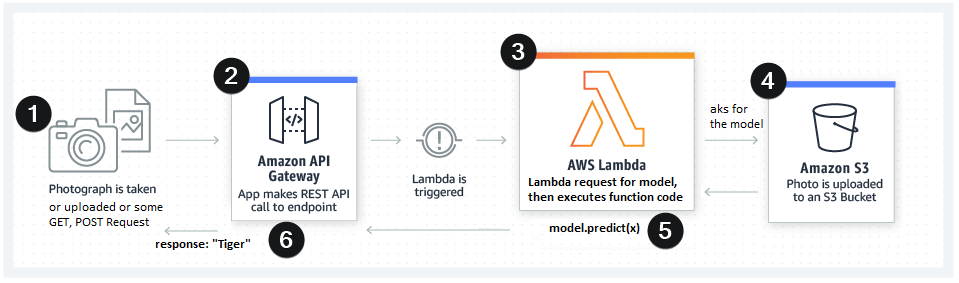
Installing and using serverless is easy.

We can write any code we want in carbon.now.sh we can make a account in it and do it

We can go to snippets in login, and do code snippets in whichever way we want.



What exactly we are going to do?



1. Take a photo, and upload through some request (http get/post).
2. Sent through Amazon API gateway.
3. Lambda is triggered for the event.
4. In AWS Lambda requests for some model to predict about the image.
5. Photo is uploaded into S3 bucket and sent to Lambda
6. Lambda function processes and executes the function code and predicts the Image.
7. Returns response back through Amazon API gateway.
8. We get response of the photo as Tiger.

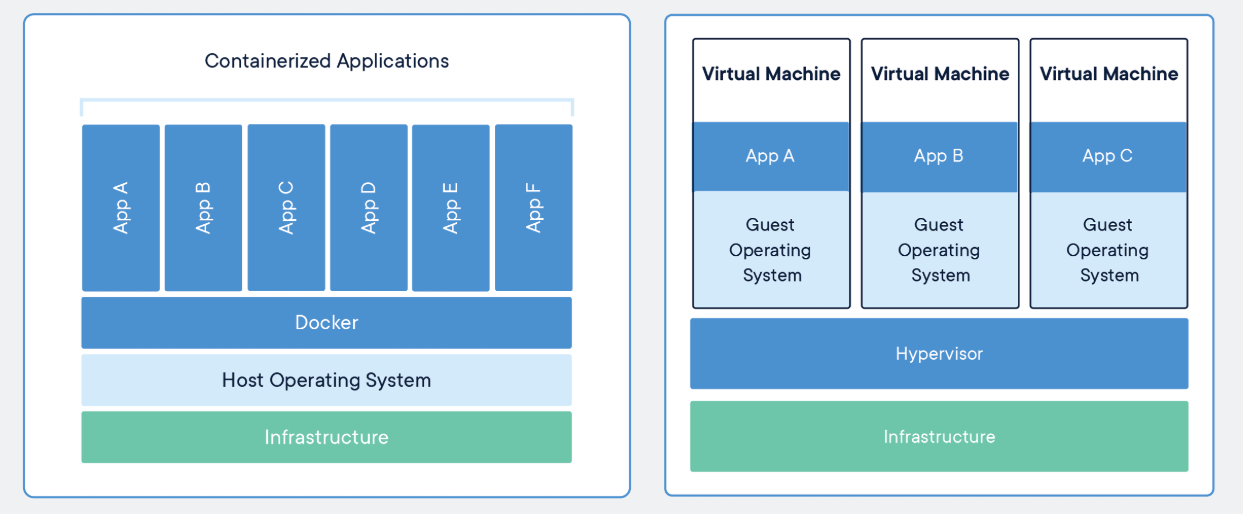
**LINUX:**

Set up is faster

We can easily use in linux OS all features from serverless – python – requirements plugin.

**DOCKER:**

* Docker is a tool for creating, deploying and running applications inside containers
* Containers are standardized software units that contain code, dependencies, runtimes, and settings in a single package. Code will only be looking at the versions of the software are.
* Containers store applications by keeping them isolated from the environment they run on
* They are portable and ensure that application works uniformly on different computing environments
* Container images become containers when they run on Docker Engine (shown as App below in left image)
* Containers virtualize the OS components (ex. Pytorch modules, python APIs) instead of hardware whereas VMs abstract physical hardware
* Each container is an isolated process, but multiple containers can share the same OS kernel with others whereas, Each VM includes a full copy of an OS, apps, and libs, container going to have part of an OS. Not own OS.
* Containers usually take up less space (in tens of MBs) and require fewer resources, whereas, VMs can take up tens of GBs and can be slower.



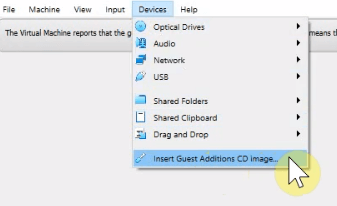
**Docker in context of serverless framework.:**

* Framework plugins can use Docker when installing Python requirements using pip
* Used Docker container is similar to the Lambda environment so the compiled dependencies will be compatible
* Dependencies will be installed using containers and then zipped to the Lambda deployment package.

**Important Steps for Setup:**

**CONFIGURING UBUNTU | NODEJS | SERVERLESS**

1. Install [VirtualBox (Links to an external site.)](https://www.virtualbox.org/wiki/Downloads)  Your password will be osboxes.org
2. Download [Ubuntu 20.04 from Osboxes.org (Links to an external site.)](https://www.osboxes.org/ubuntu/) we will have it as **.vdi file**
3. Create the best possible Virtual Machine on Virtual Box with a minimum
   1. 4GB+ RAM and then
   2. Select Use an existing virtual hard disk file and then point to your extracted Ubuntu 20.04.vdi
   3. Provide as many CPU cores as possible
   4. Your resolution wouldn't be great. Install Guest Editions CD images (Google in case you're still not able to fix it)



* 1. In Ubuntu >> Settings >> Power >> Disable Power savings:

Power Saving - Blank Screen - Never

Automatic Suspend - On Battery Power - OFF

* 1. Create a Snapshot of your Virtual Machine. In case anything goes bad later on, you can always use this snapshot.

**To install Miniconda:**

1. Install Miniconda [https://docs.conda.io/en/latest/miniconda.html (Links to an external site.)](https://docs.conda.io/en/latest/miniconda.html) then Miniconda3 Linux 64-Bit. (going to install 64bit miniconda for Linux)

<https://repo.anaconda.com/miniconda/Miniconda3-latest-Linux-x86_64.sh>

1. Open Terminal and navigate to the Downloads folder, and bash Miniconda3-latest-linux-x86\_64.sh (the sh file we downloaded in the last step)
2. Use Ubuntu Sofware to install the Visual Studio Code (or any other code editor of your choice). Install Python Extension in VS Code.
3. Install pip Run these commands in order:

* sudo apt-get install python3-distutils
* wget http://bootstrap.pypa.io/get-pip.py (Links to an external site.)
* sudo python get-pip.py

1. if you get this error "launchpadlib x.x.x requires testresources, which is not installed" then run

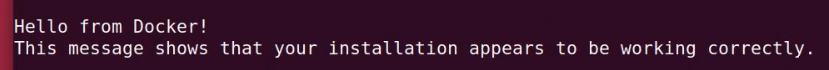
sudo pip3 install testresources

**Installing Docker:**

1. Visit: [https://download.docker.com/linux/ubuntu/dists/bionic/pool/stable/amd64/ (Links to an external site.)](https://download.docker.com/linux/ubuntu/dists/bionic/pool/stable/amd64/)
2. Download these 3 files (you should download latest version, below are the files I used):
   1. docker-ce\_19.03.9~3-0~ubuntu-bionic\_amd64.deb (basic engine)
   2. docker-ce-cli\_19.03.9~3-0~ubuntu-bionic\_amd64.deb (command line interpreter)
   3. containerd.io\_1.2.13-2\_amd64.deb (container)
3. use **sudo dpkg -i containerd.io\_1.2.13-2\_amd64.deb**for example to install containerd... Install all 3
4. Make Docker to start on boot:
   1. **sudo systemctl enable docker**
5. Check if Docker is installed properly

**sudo docker run hello-world**

* 1. If you see below it means everything went fine:



For me, it did not. I got an error "Got permission denied while trying to connect to the Docker daemon socket"

1. this helped: **sudo chmod 666 /var/run/docker.sock on**
2. Refer to this [link (Links to an external site.)](https://www.digitalocean.com/community/questions/how-to-fix-docker-got-permission-denied-while-trying-to-connect-to-the-docker-daemon-socket) in case you also see this error in the future.
3. If you still face this problem (docker doesn't start):
   1. service docker stop  
      cd /var/run/docker/libcontainerd  
      rm -rf containerd/\*  
      rm -f docker-containerd.pid  
      service docker start
4. If everything fails, follow these steps: <https://docs.docker.com/engine/install/ubuntu/>

Have to uninstall and reinstall the docker.

Get repository command as sudo apt repository (above step)

Install docker engine commands.

**Install Node.js: ( we are client side running javasecript)**

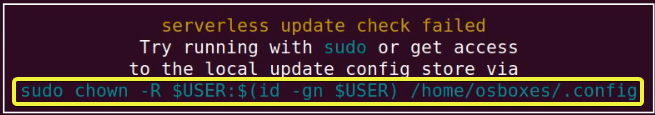
Its helpful for serverless computing framework.

1. Start with installing ***curl*sudo apt install curl**
2. Download an installation script that will add a PPA (Personal Package Archive) to your Ubuntu Package Repositories maintained by Node to get the latest updated node packages.   
   **curl -sL**[**https://deb.nodesource.com/setup\_10.x (Links to an external site.)**](https://deb.nodesource.com/setup_10.x)**-o nodesource\_setup.sh  
   sudo bash nodesource\_setup.sh  
   sudo apt-get install -y nodejs**
3. Check the installation by checking **nodejs** (old version command) (new is node) and **npm** versions  
   node -v   
   npm -v

**Install the Serverless framework**

**sudo npm install -g serverless**

1. you most probably will get this:



just copy the command in yellow: "**sudo chown -R...**" command and execute it.

1. check is serverless is installed properly:  
   **serverless -help**

**AWS Management console:**

Type in google for AWS management console and click on it.

Plenty of services there. Go to IAM when doing sign in IAM service.

Click for IAM service, create a new user.

* Add user option, click on it. Then write user name as session1-user / anything.
* Click on programmatic access type.
* Click on next
* Click on attach existing policies
* There click on administrator access then next. Key not required.
* Create new user button – click on it.
* After success message first download CSV file. And access through the key. There we will have session1 user.

**Setting up SLS with the user credentials:**

* 1. Configure Serverless (using your key and secret): (what we got above)  
     $ **sls config credentials --provider aws --key AKIBWDBSFP1ACHJPAAGQ --secret PZWjxph8vFUkvLinL0frtBk1NijOKjI18DjFMEqm**

**(here we can use sls or serverless command)**

**Message as setting up AWS…**

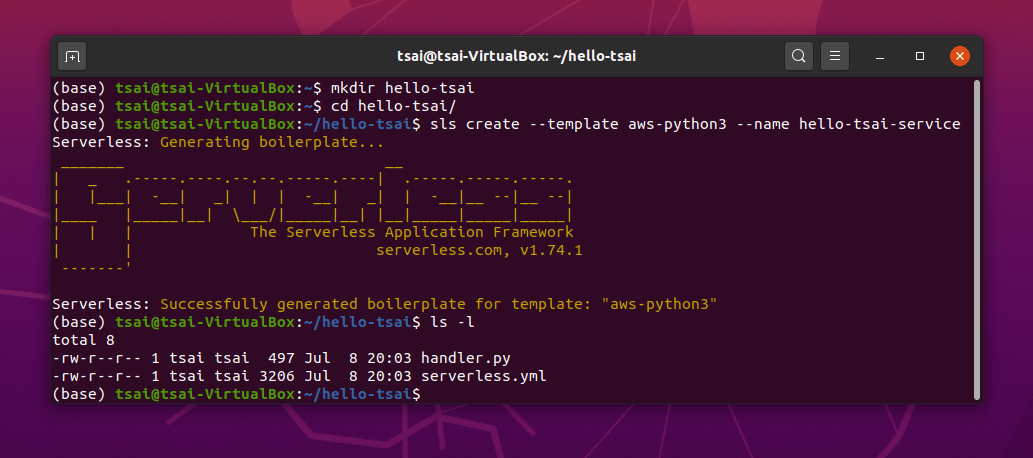
**Saving your AWS profile!**

**Success**

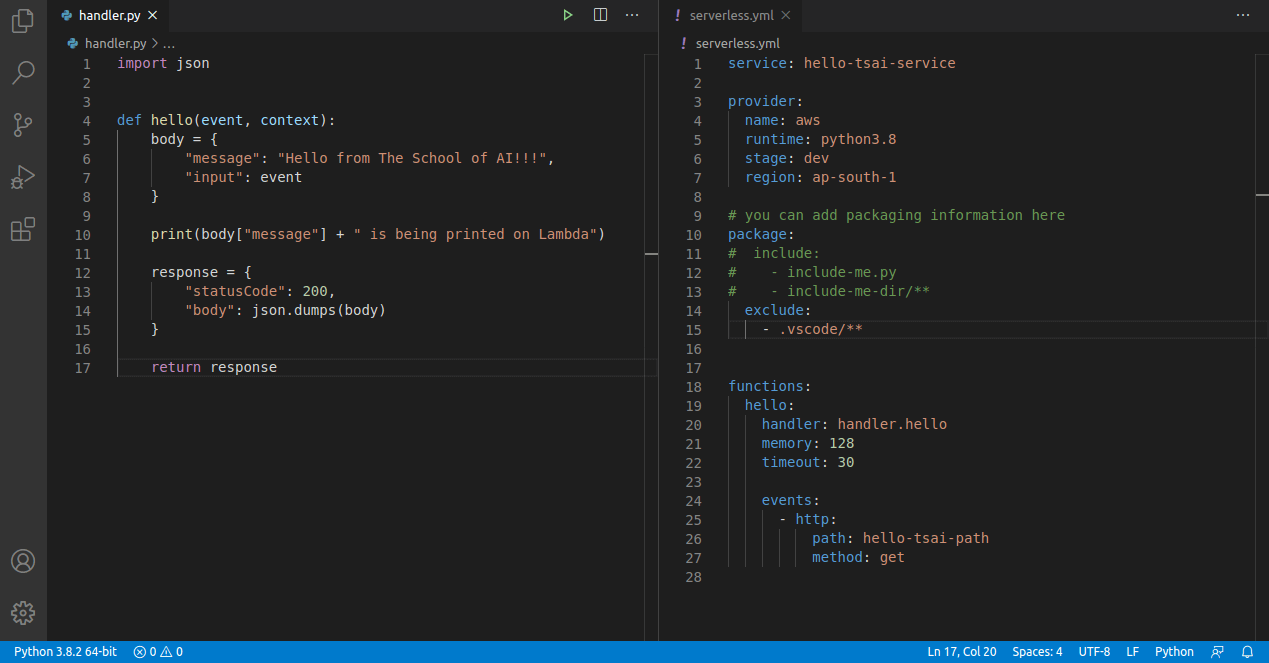
* 1. **Done!**

First setup - Hello TSAI from Lambda

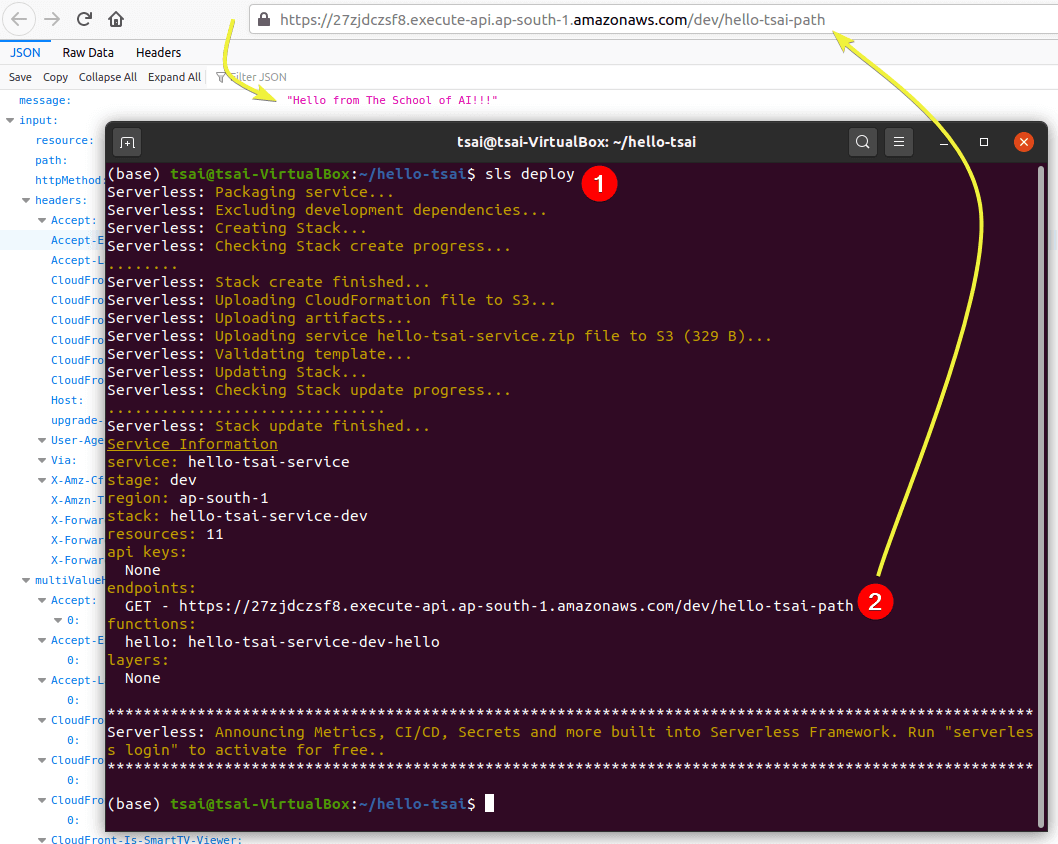
1. Go into ubuntu, Click on terminalMake a new directory for your new project and move into it
2. Execute this command:   
     
   **$ sls create --template aws-python3 --name hello-tsai**



* When you do ls, it shows 2 files. handler.py and serverless.yml
* Go to session1 folder and check these above 2 files.
* handler.py file is the one which is going to handle learning on lambda.
* hello is the main function in handler.py
* in body, type some message Good Morning! Too early for me to be awake.
* 200 status code is good.
* print (body message…. As shown in below figure:

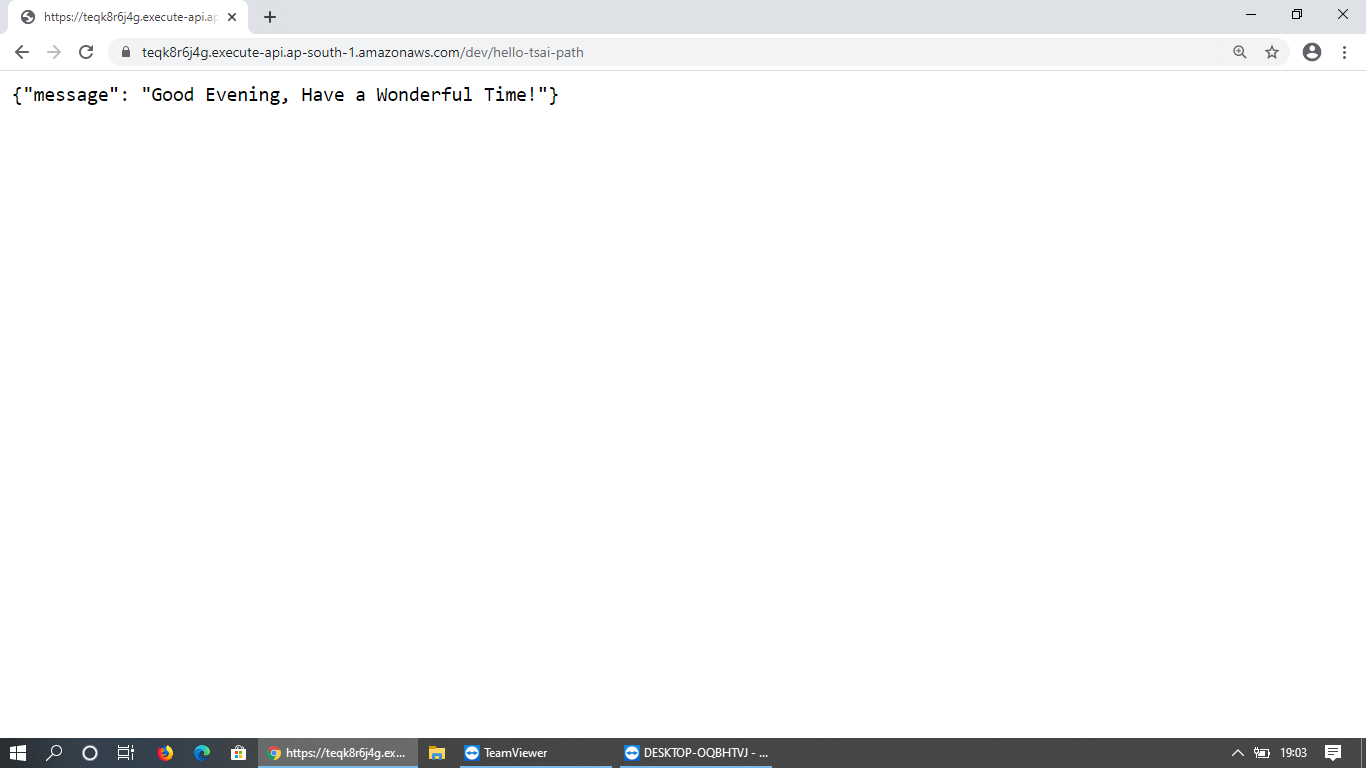


* When u check in serverless.yml file check handler should be same hello.
* We need to specify region ap-south-1
* Package and exclude things to be edited.
* Next **sls deploy command**
* **Click on http link and paste in browser.**
* **Message will be displayed.**

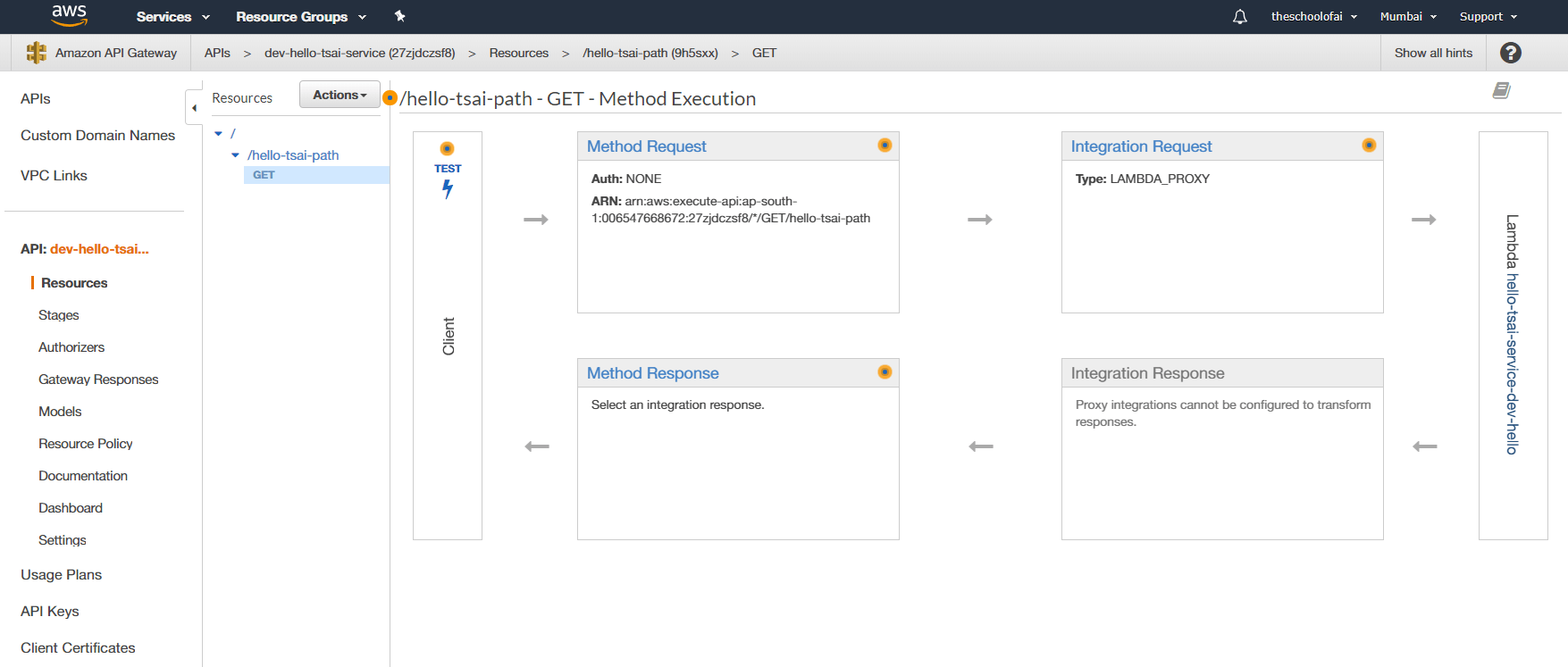


Got message on AWS Lambda. Deployed

Link : <https://teqk8r6j4g.execute-api.ap-south-1.amazonaws.com/dev/hello-tsai-path>



Goto API Gateway in AWS Management Console. Select dev-hello-tsai-service or your differently named service.

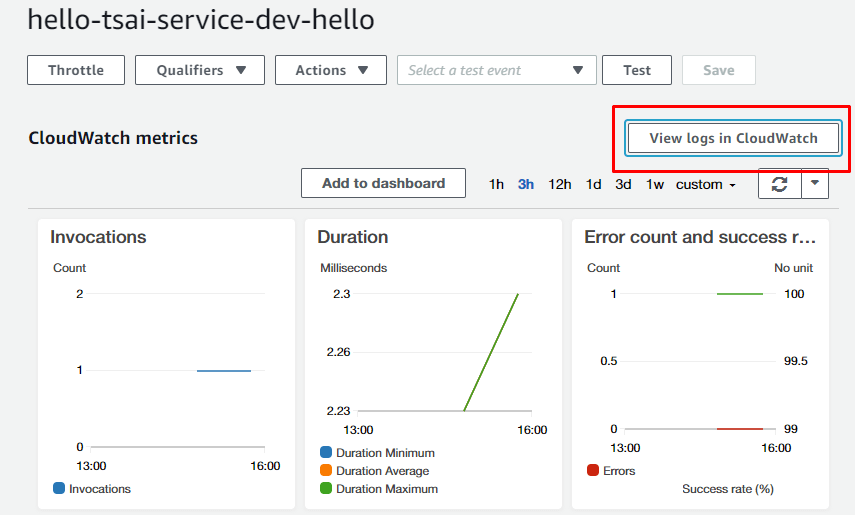


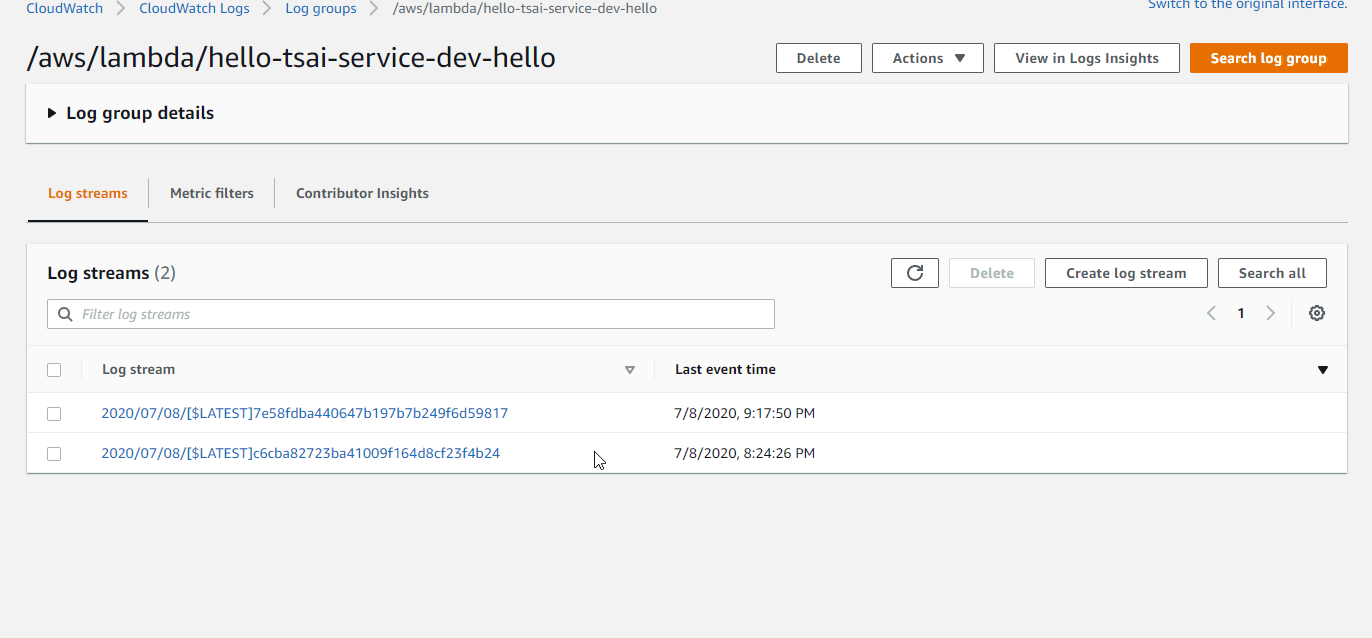
Click on Lambda tab as well

That’s the lambda which we created. Layers will be there in that.

Handler code will be here too.

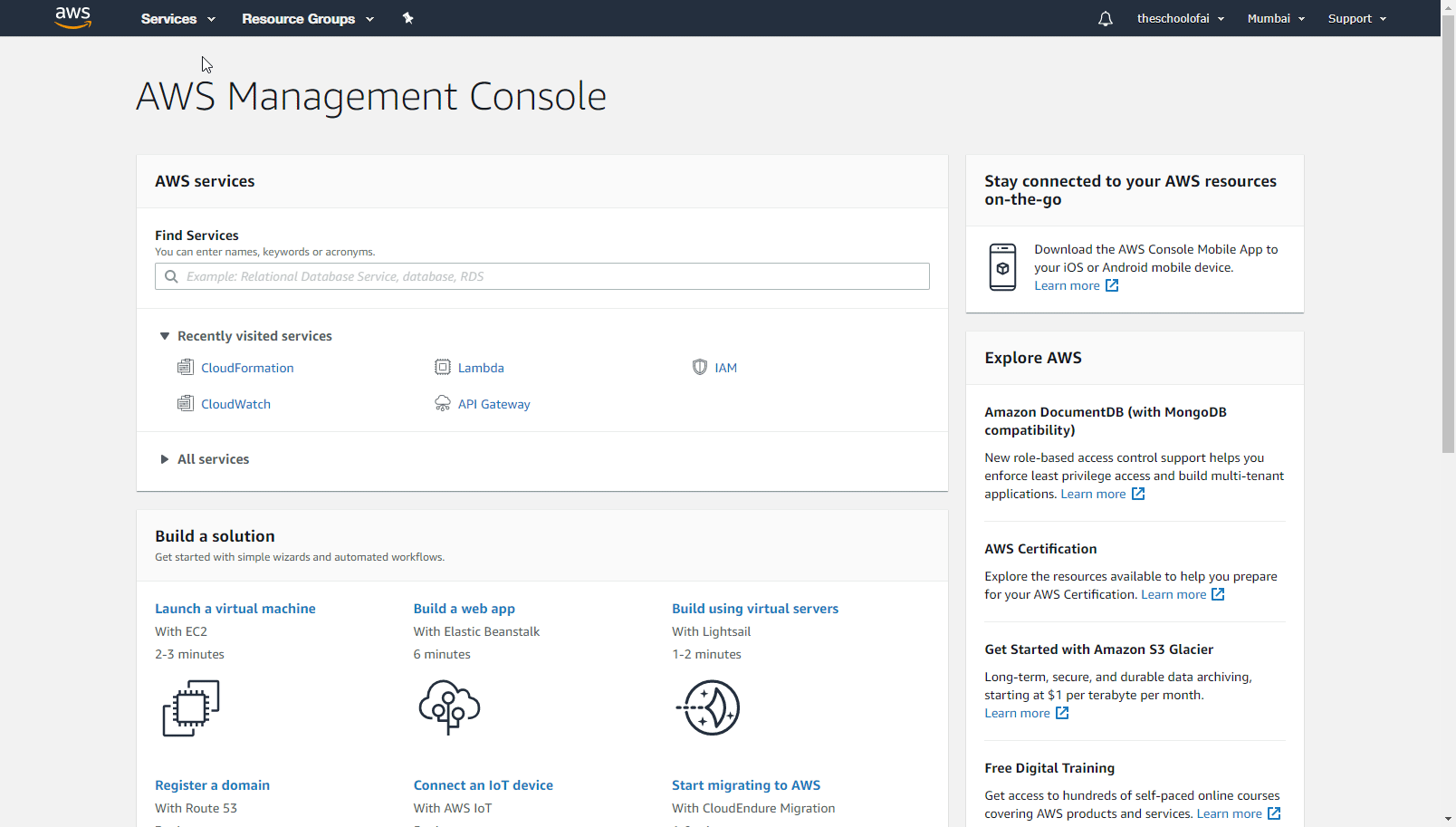
* Click on **hello-tsai-service-dev-hello**on the right end and learn more on that page w.r.t. setting you can change/modify.
* Click on **Monitoring**to access CloudWatch. Here you can see how many times your lambda was accessed and other statistics.





Let us check out Cloud Formation now. Visit CloudFormation on your AWS console.

Check resources in that.

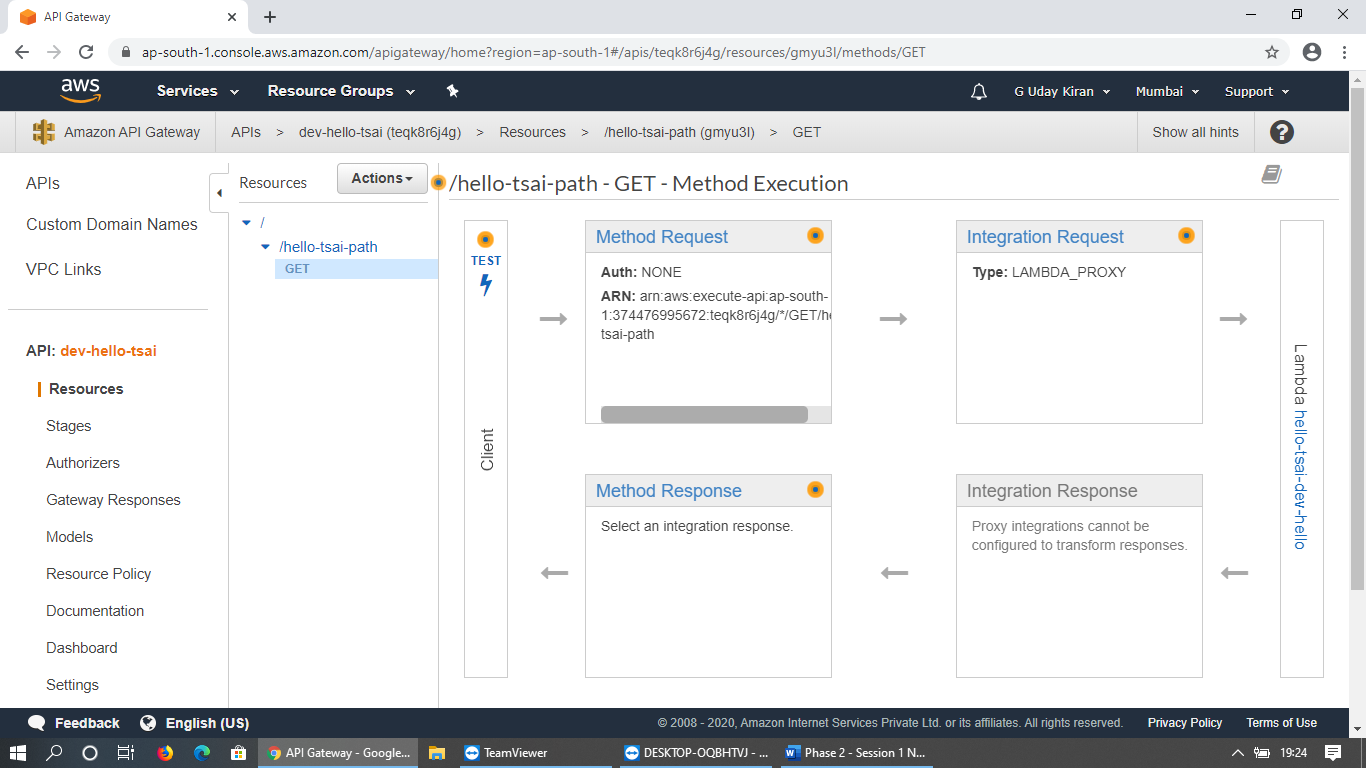


**So, if we do not use serverless, we need to look at all of these services.**

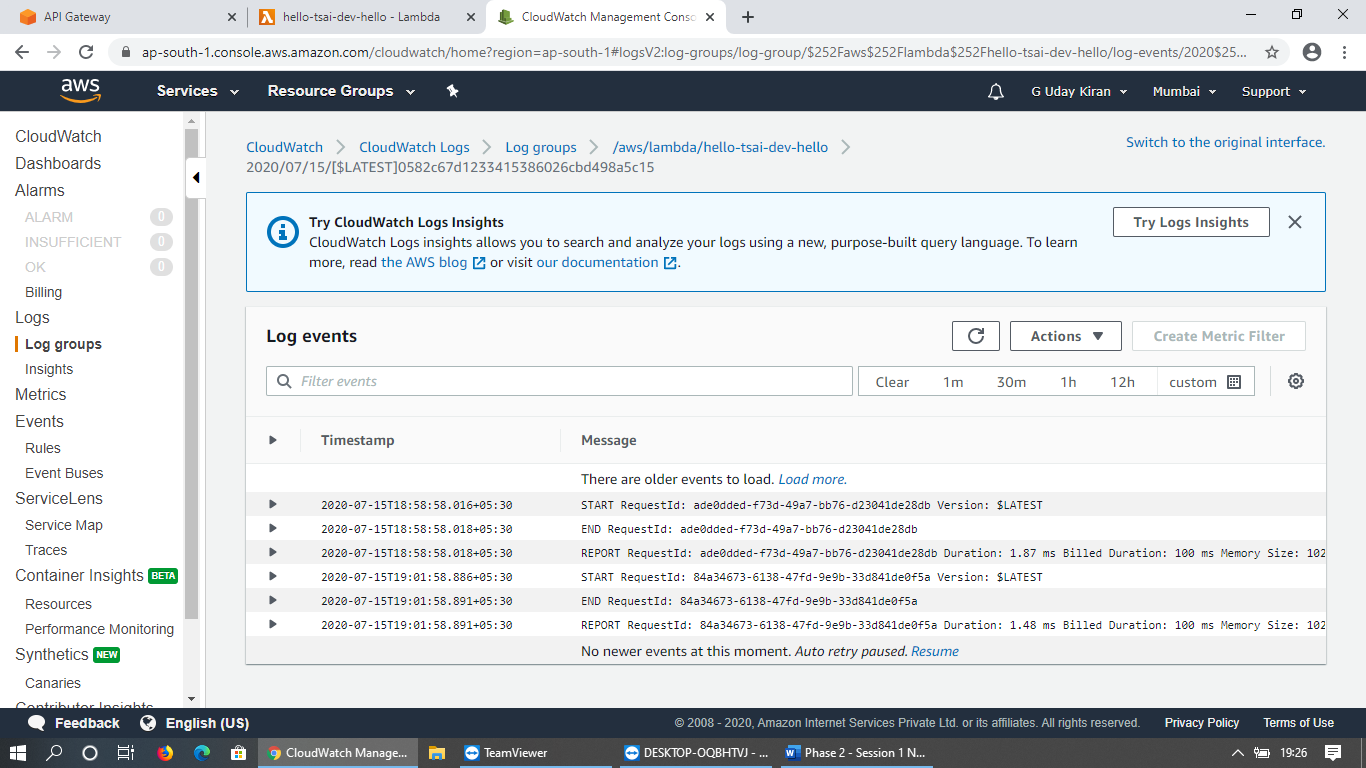
**Event is something in which we click, it triggers and takes action to do the service. Example clicking on user profile is an event in aws, so it opens the profile (action for the event)**

**Only the serverless service provided by AWS is through Lambda.**

**Own Lambda screenshots:**



**Cloudwatch logs:**



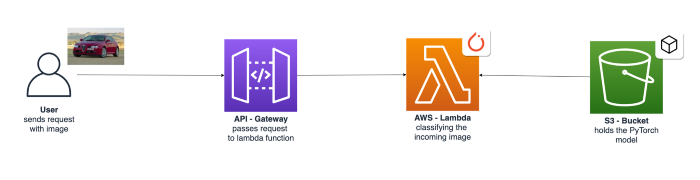
**Deploying a DNN Model:**

Deploying a PyTorch model on Lambda is hard

So here we will use pretrained Resnet model and then process it to classify the images. We are referencing this amazing [blog (Links to an external site.)](https://towardsdatascience.com/scaling-machine-learning-from-zero-to-hero-d63796442526) post,

Points to do:

* download a pre-trained ResNet34 model
* create a Python Lambda function with serverless Framework
* create an S3 bucket, which holds the model
* add a Pytorch to the Lambda Environment (the most difficult task)
  + this is going to be our template moving forward for all Lambda Assignments
* write a prediction function to classify an image
* configure the serverless framework to set up the API gateway for inference.
* test our deployment using Insomnia



We should have a dedicated conda environment

After downloaded and installation, $conda info -e

**Download a Pretrained model:**

In command line itself, we can download pretrained model



**Create a Python Lambda Function:**



This will generate two main files we'd need to edit, handler.py and 'serverless.yml'

Before this step we should have python requirements plugin.

**Python Requirements Plugin**

We need to install a Serverless plugin which will automatically bundle dependencies from a **requirements.txt file**. This is THE main step where we most probably will get stuck (bunding things correctly).

The **serverless-python-requirements** plugin allows you to even bundle non-pure-Python modules. More on that [here (Links to an external site.)](https://github.com/UnitedIncome/serverless-python-requirements#readme).



**Adding Requirements:**

We'll manually create a requirements.txt file on the root level, with all required Python packages. Normally this file is created using this process, **but don't do this**:



We will need to be cautious on how we create this requirement file since our deployment Package on AWS Lambda cannot be greater than 250MB (Pytorch itself can be 470MB or more!).

Instead, we will add a link to python wheel file (.whl) for Pytorch and Lambda will directly install it for us! For a list of all PyTorch and torchvision packages consider [this list (Links to an external site.)](https://download.pytorch.org/whl/torch_stable.html).

The requirements.txt should look like this.



requests\_toolbelt will help to get all functions for base 64 as we have maximum images of base 64.

To make the dependencies even smaller we will employ three techniques available in the serverless-python-requirements plugin:

* **zip** - Compresses the dependencies in the requirements.txt in an additional **.requirements.zip** file and adds **unzip\_requirements.py** in the final bundle.
* **slim** - Removes unneeded files and directories such as \*.so, \*.pyc, dist-info, etc.
* **noDeploy** - Omits certain packages from deployment. We will use the standard list that excludes packages those already built into Lambda, as well as Tensorboard.

You can see the implementation of it in the section where we are “configuring our serverless.yaml " file.

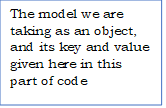
**Handler.py and our main code:**

Previous handler.py was very simple file than this file.

Here all imported packages had been visible plus requests.toolbelt too

**multipart data** form – whenever we want to send media like an image,(both test and image we have here) we need to bundle it and send so we require this multipart decoder data form in lambda resource.

**In handler.py and serverles.yml file it should be same names as S3 Bucket and Model\_path.**

****

Returns if error is there

Classify the image here, (event as http request and the context which we sent)

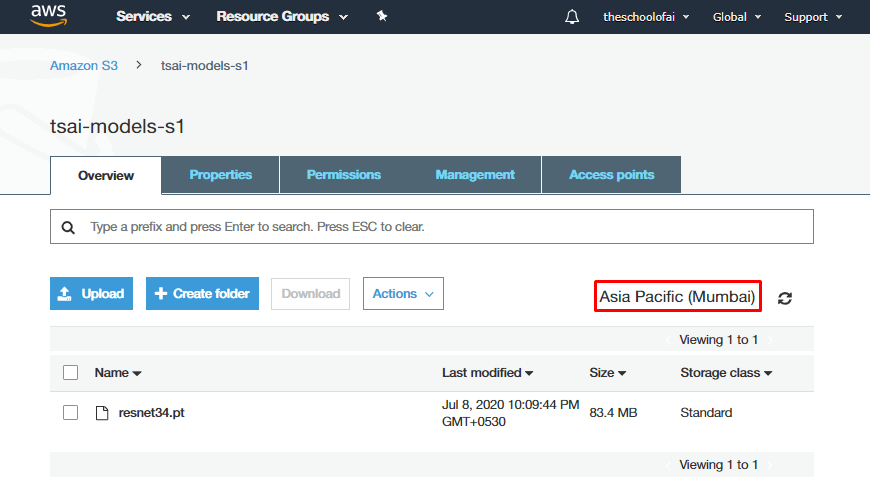
Getting the prediction of model and image here

Here this is image transformations part where the image taken as image bytes, not directly the image.

The model we are taking as an object, and its key and value given here in this part of code

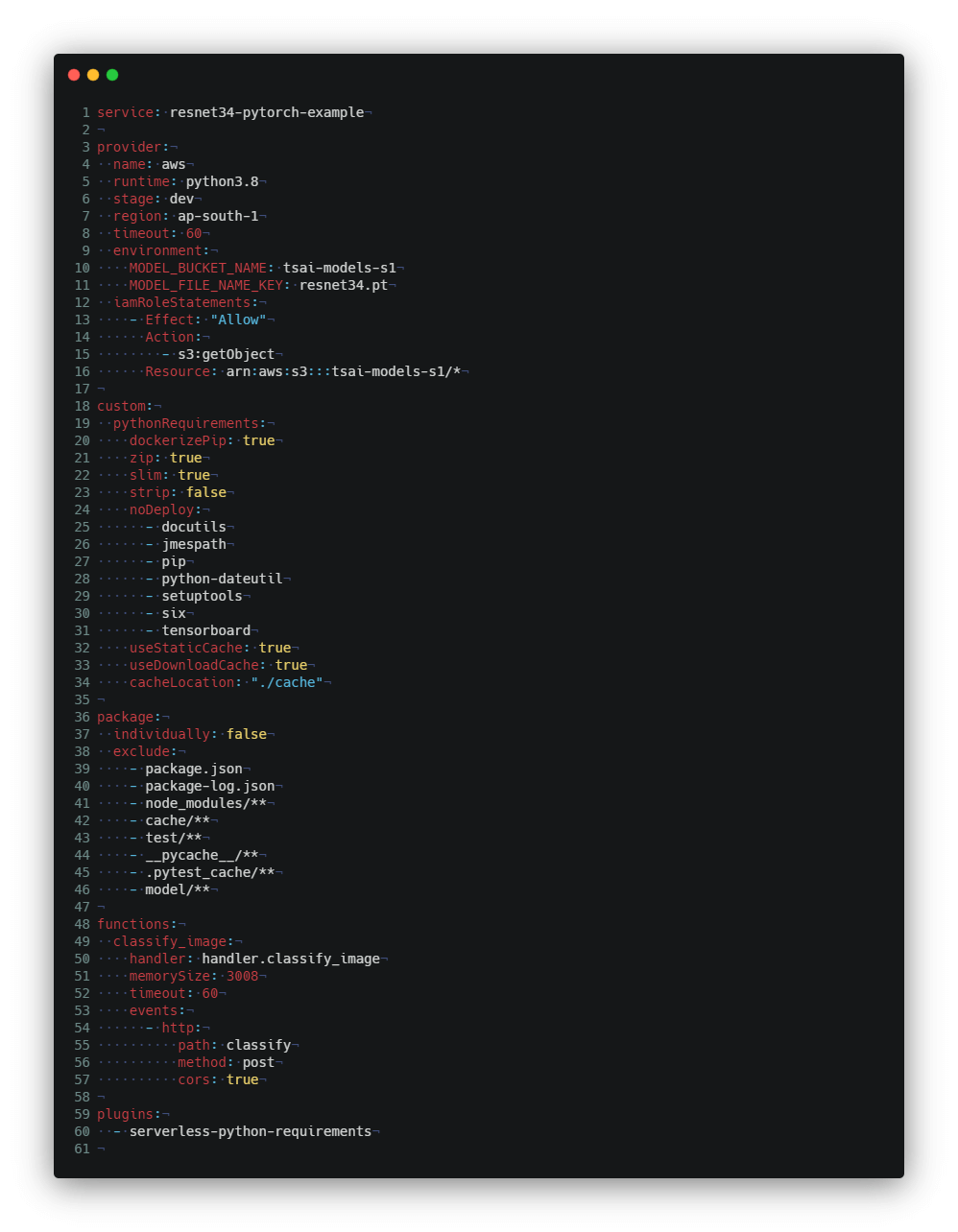
**How to create a S3 bucket:**

* + Go to aws services > S3 > Check once in tsai model > model already exists
  + If we want to create > create bucket > click some name
  + That name of the bucket should be **globally unique. Why because its going to convert that bucket into some url**. And posts. That’s why. And set time zone as Asia Pacific Mumbai)



Go to this bucket and click, upload an image or model into it. Then it will be added.

**Configuring serverless.yml file:**



**Deploying Our Package**

**Add a deploy script to your package.json file.**

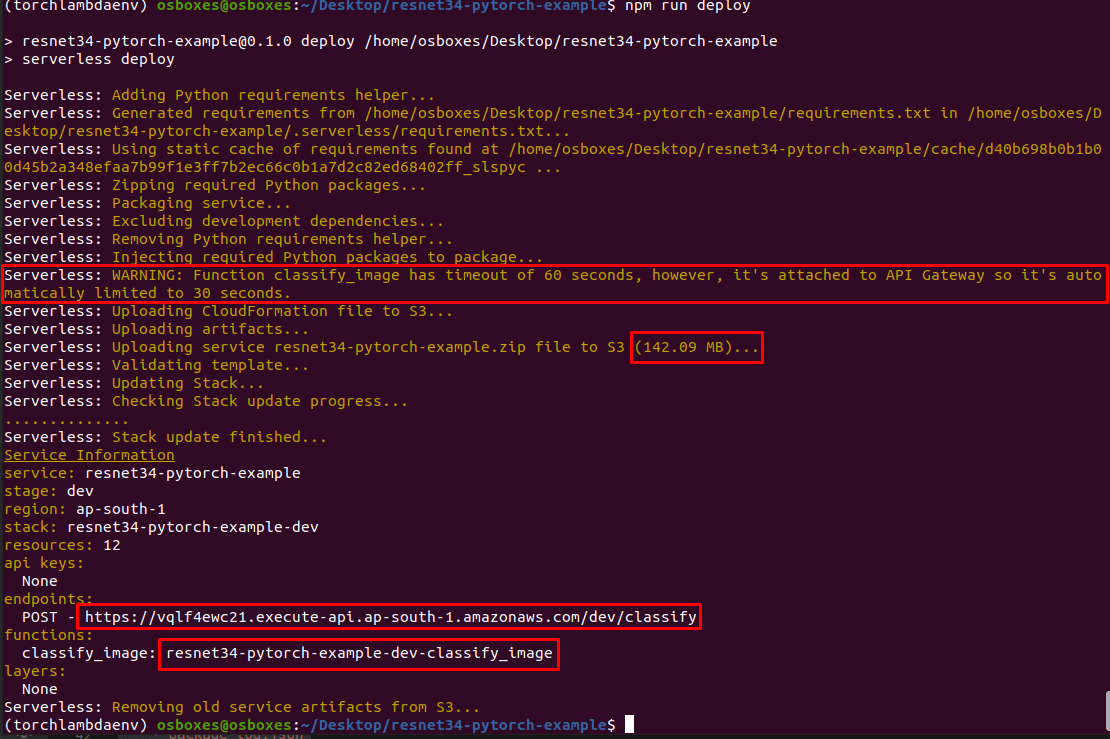


Here we are adding line: serverless deploy

Then:

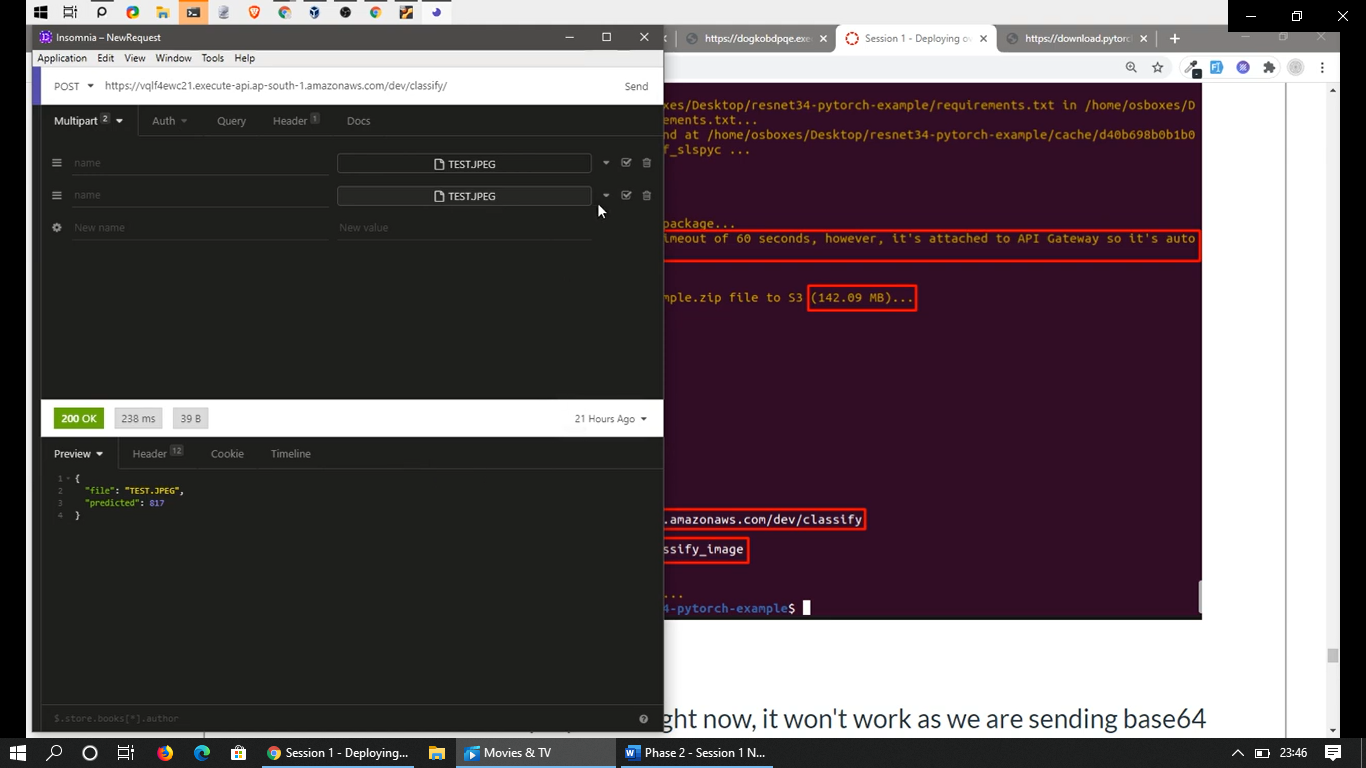


Then we will get a log like this:



**Testing our Deployment:**

**Goto Insomnia App:**



Then click on the **url** which is there above and post here, **upload the file here that’s the image.**

And in header we need to mention it as **multipart form data**.

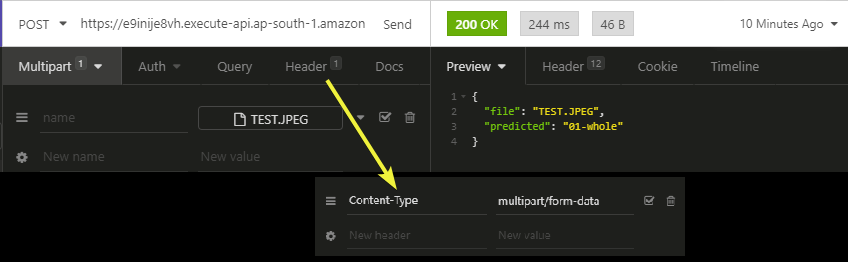
Then send it, **we will get it as file name with class 817.**

***Imagenet class id 817 is Car that’s Sports car*** 😊

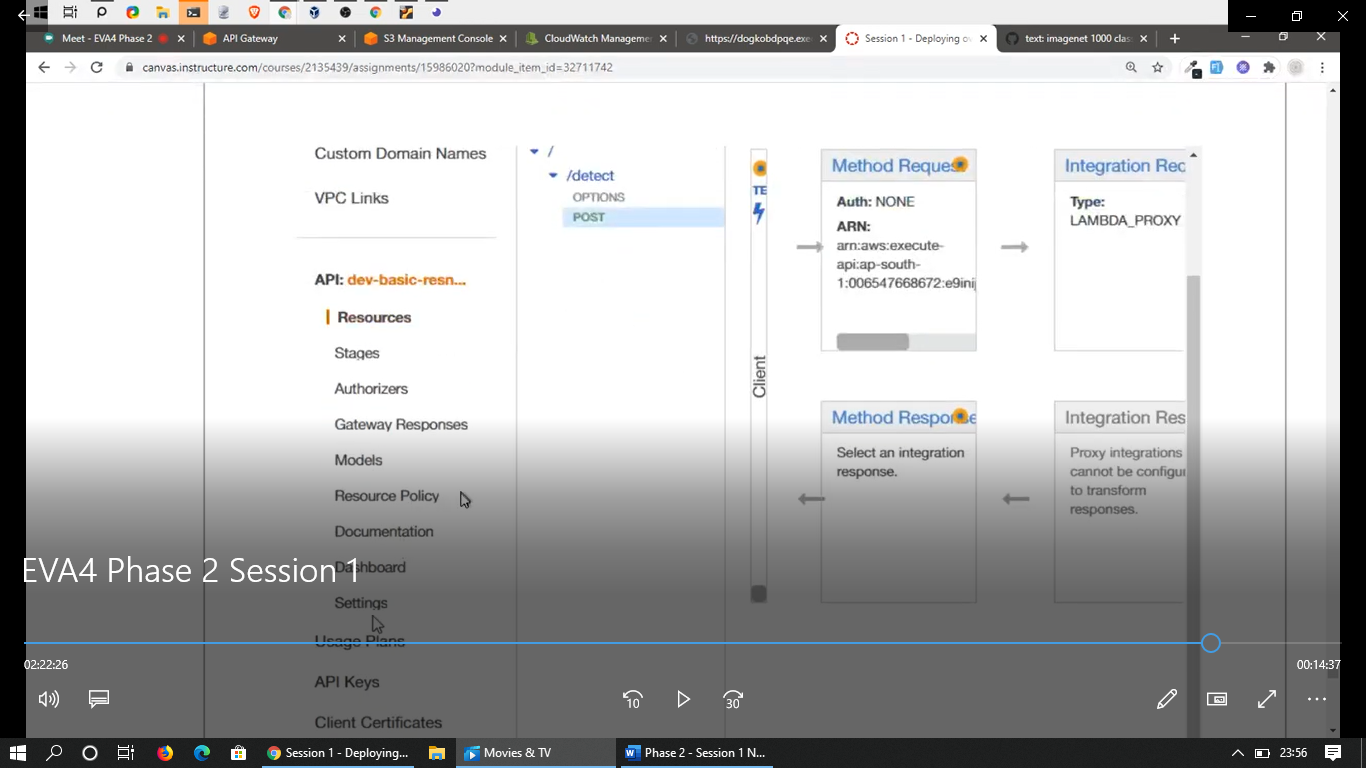
Even if we took care of everything, the first deployment won't show results.

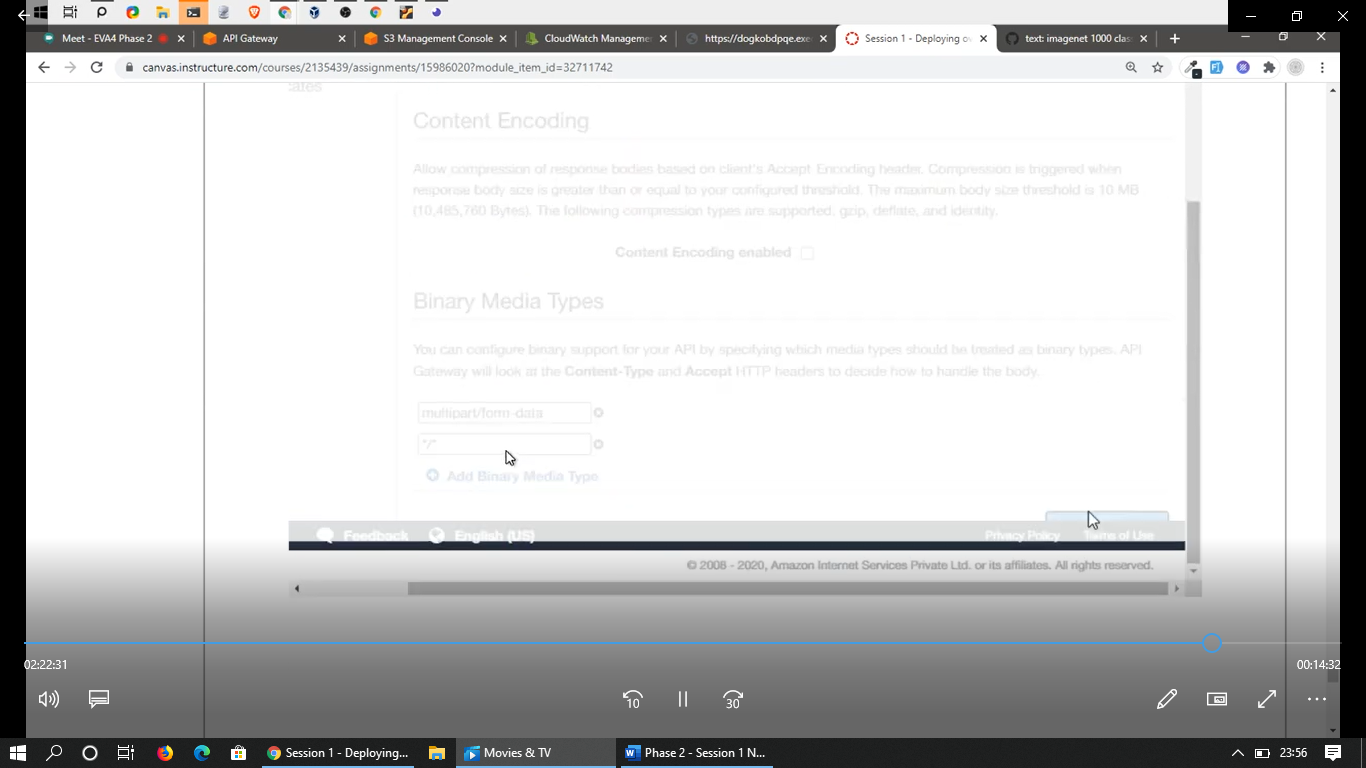
What you are seeing is a COLD start and that takes time, and if it is more than 30 seconds, we won't get the results we are looking for.

But if you test the second time, you'll get the result (warm start) and it would run must faster as well.  Install [Insomnia (Links to an external site.)](https://insomnia.rest/download/) and follow the steps on the right. Results are shown on the left!



**Note:** If we were to check our deployment right now, it won't work as we are sending base64 decoded data using multipart/form-data and AWS API gateway isn't told about that.





**Prepared By: G Uday Kiran & Srilakshmi V**