Siddhant Sathe **D15A/51**

EXPERIMENT 11

Aim: To understand AWS Lambda, its workflow, various functions and create your first Lambda functions using Python / Java / Nodejs.

Theory:

AWS Lambda

AWS Lambda is a serverless computing service provided by Amazon Web Services (AWS). Users of AWS Lambda create functions, self-contained applications written in one of the supported languages and runtimes, and upload them to AWS Lambda, which executes those functions in an efficient and flexible manner.The Lambda functions can perform any kind of computing task, from serving web pages and processing streams of data to calling APIs and integrating with other AWS services.

The concept of “serverless” computing refers to not needing to maintain your own servers to run these functions. AWS Lambda is a fully managed service that takes care of all the infrastructure for you. And so “serverless” doesn’t mean that there are no servers involved: it just means that the servers, the operating systems, the network layer and the rest of the infrastructure have already been taken care of so that you can focus on writing application code.

Features of AWS Lambda

● AWS Lambda easily scales the infrastructure without any additional configuration. It reduces the operational work involved.
● It offers multiple options like AWS S3, CloudWatch, DynamoDB, API Gateway, Kinesis, CodeCommit, and many more to trigger an event.
● You don’t need to invest upfront. You pay only for the memory used by the lambda function and minimal cost on the number of requests hence cost-efficient.
● AWS Lambda is secure. It uses AWS IAM to define all the roles and security policies.
● It offers fault tolerance for both services running the code and the function. You do not have to worry about the application down.

Packaging Functions

Lambda functions need to be packaged and sent to AWS. This is usually a process of compressing the function and all its dependencies and uploading it to an S3 bucket. And letting AWS know that you want to use this package when a specific event takes place. To help us with this process we use the Serverless Stack Framework (SST).

Execution Model

The container (and the resources used by it) that runs our function is managed completely by AWS. It is brought up when an event takes place and is turned off if it is not being used. If additional requests are made while the original event is being served, a new container is brought
up to serve a request. This means that if we are undergoing a usage spike, the cloud provider

simply creates multiple instances of the container with our function to serve those requests. This has some interesting implications. Firstly, our functions are effectively stateless. Secondly, each request (or event) is served by a single instance of a Lambda function. This means that you are not going to be handling concurrent requests in your code. AWS brings up a container whenever there is a new request. It does make some optimizations here. It will hang on to the container for a few minutes (5 - 15mins depending on the load) so it can respond to subsequent requests without a cold start.

Stateless Functions

The above execution model makes Lambda functions effectively stateless. This means that every time your Lambda function is triggered by an event it is invoked in a completely new environment. You don't have access to the execution context of the previous event.

However, due to the optimization noted above, the actual Lambda function is invoked only once per container instantiation. Recall that our functions are run inside containers. So when a function is first invoked, all the code in our handler function gets executed and the handler function gets invoked. If the container is still available for subsequent requests, your function will get invoked and not the code around it.

For example, the createNewDbConnection method below is called once per container instantiation and not every time the Lambda function is invoked. The myHandler function on the other hand is called on every invocation.

Common Use Cases for Lambda

Due to Lambda's architecture, it can deliver great benefits over traditional cloud computing setups for applications where:

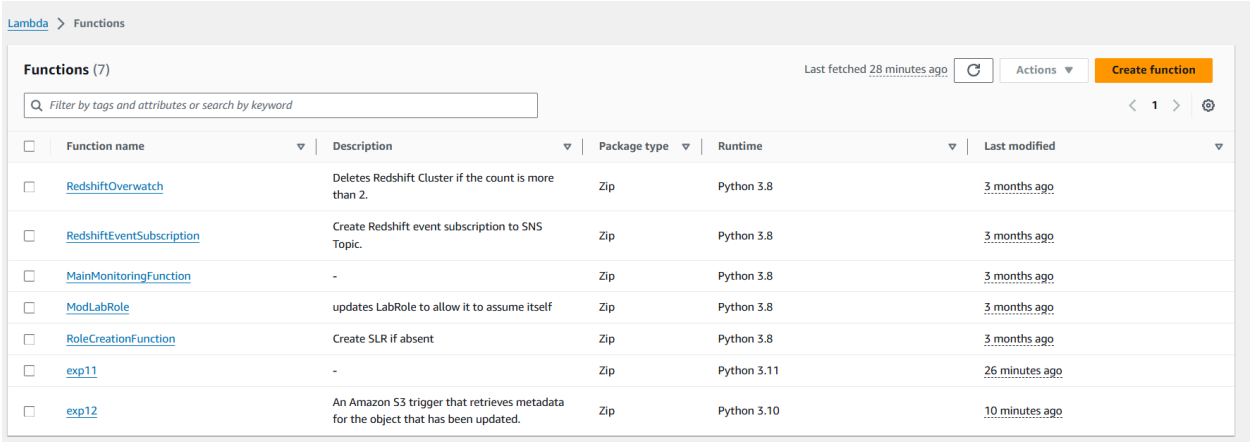
1. Individual tasks run for a short time;
2. Each task is generally self-contained;
3. There is a large difference between the lowest and highest levels in the workload of the application.

Some of the most common use cases for AWS Lambda that fit these criteria are: Scalable APIs. When building APIs using AWS Lambda, one execution of a Lambda function can serve a single HTTP request. Different parts of the API can be routed to different Lambda functions via Amazon API Gateway. AWS Lambda automatically scales individual functions according to the demand for them, so different parts of your API can scale differently according to current usage levels. This allows for cost-effective and flexible API setups.

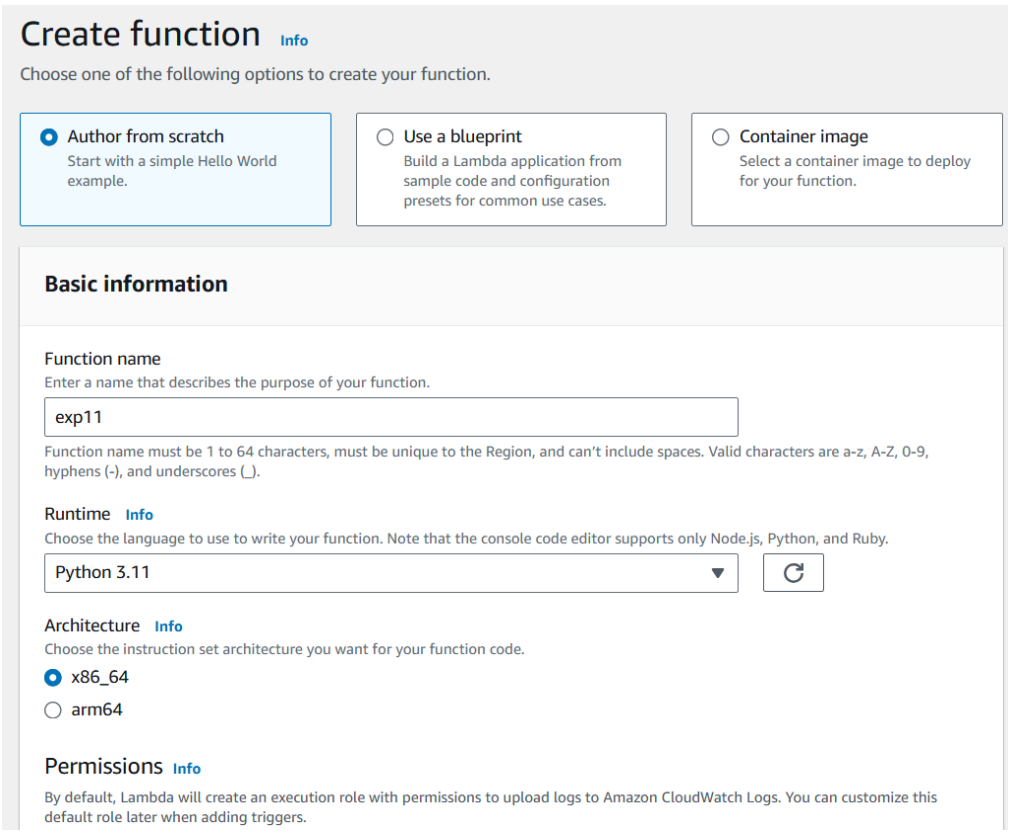
Data processing. Lambda functions are optimized for event-based data processing. It is easy to integrate AWS Lambda with data sources like Amazon DynamoDB and trigger a Lambda function for specific kinds of data events. For example, you could employ Lambda to do some work every time an item in DynamoDB is created or updated, thus making it a good fit for things like notifications, counters and analytics.

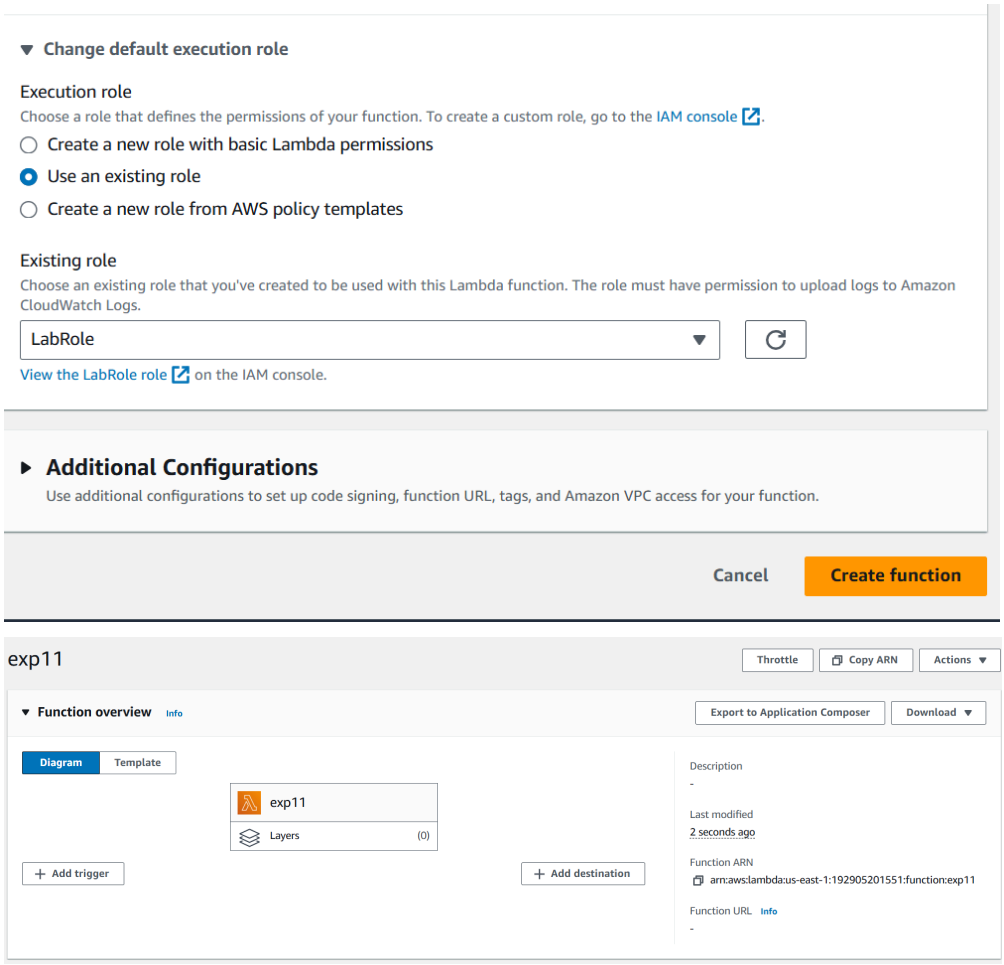
Steps to create an AWS Lambda function

Step 1: Open up the Lambda Console and click on the Create button. Be mindful of where you create your functions since Lambda is region-dependent.

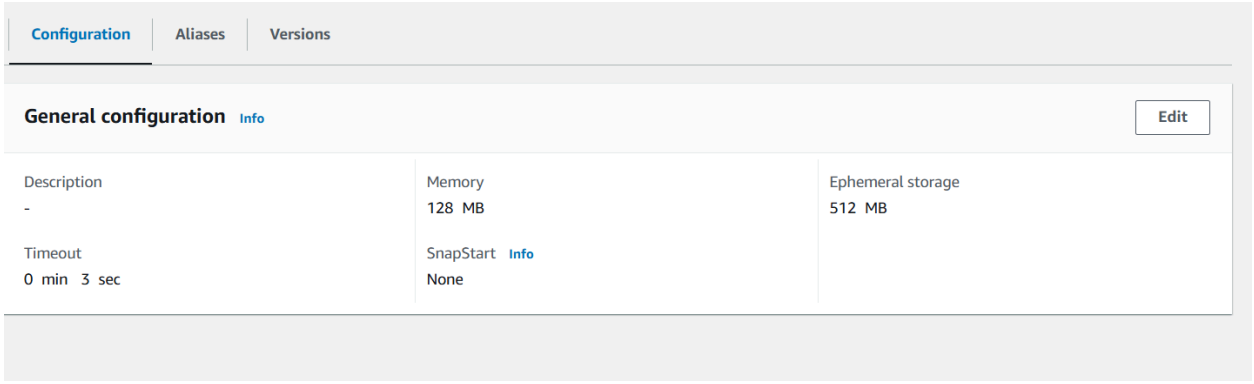


Step 2: Choose to create a function from scratch or use a blueprint, i.e templates defined by AWS for you with all configuration presets required for the most common use cases.Then, choose a runtime env for your function, under the dropdown, you can see all the options AWS supports, Python, Nodejs, .NET and Java being the most popular ones. After that, choose to create a new role with basic Lambda permissions if you don't have an existing one.

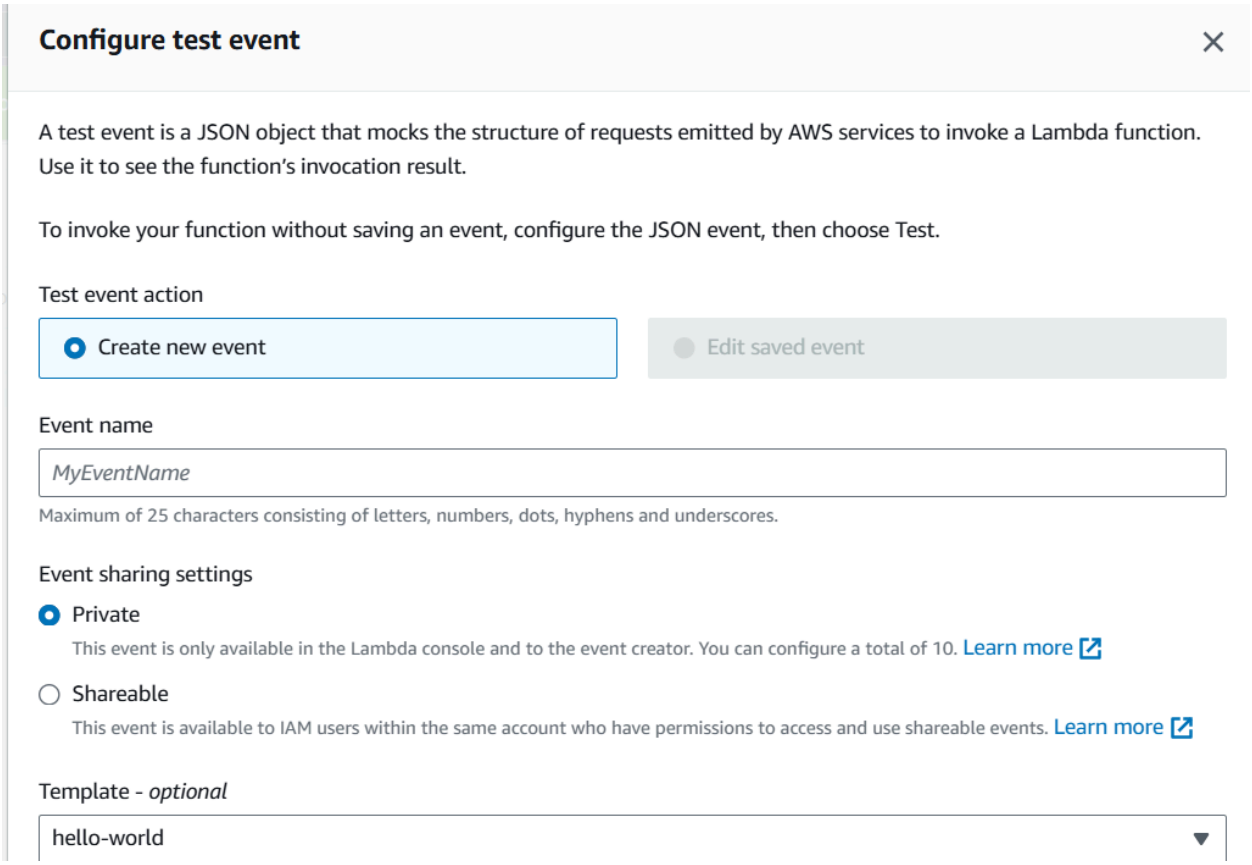




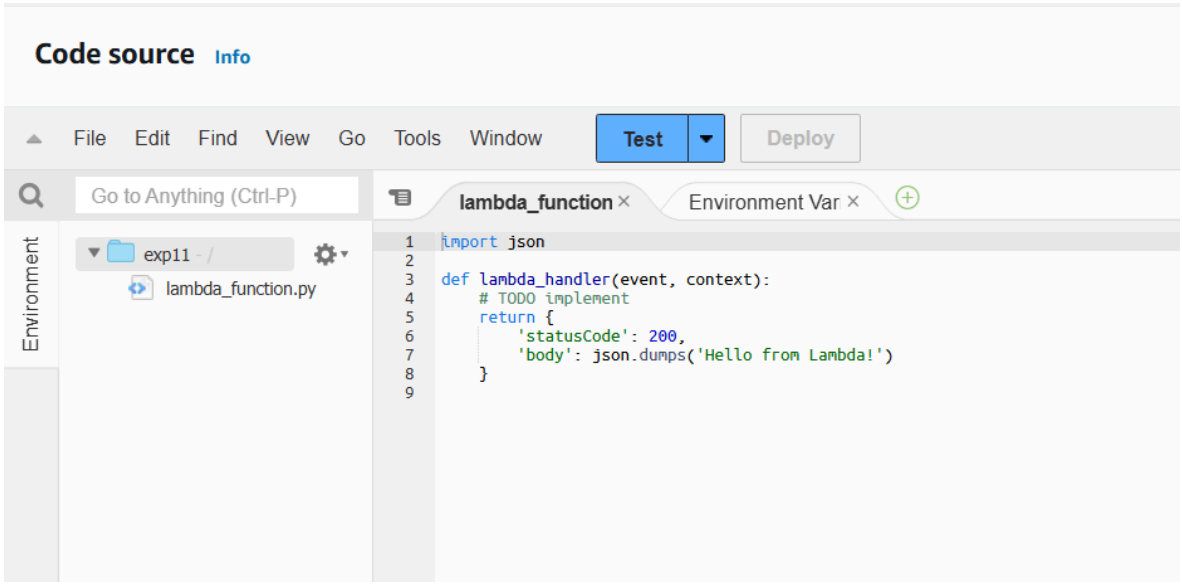
Step 3:. To change the configuration, open up the Configuration tab and under General Configuration, choose Edit. Here, you can enter a description and change Memory and Timeout.



Step 4: Now Click on the Test tab then select Create a new event, give a name to the event and select Event Sharing to private, and select hello-world template.

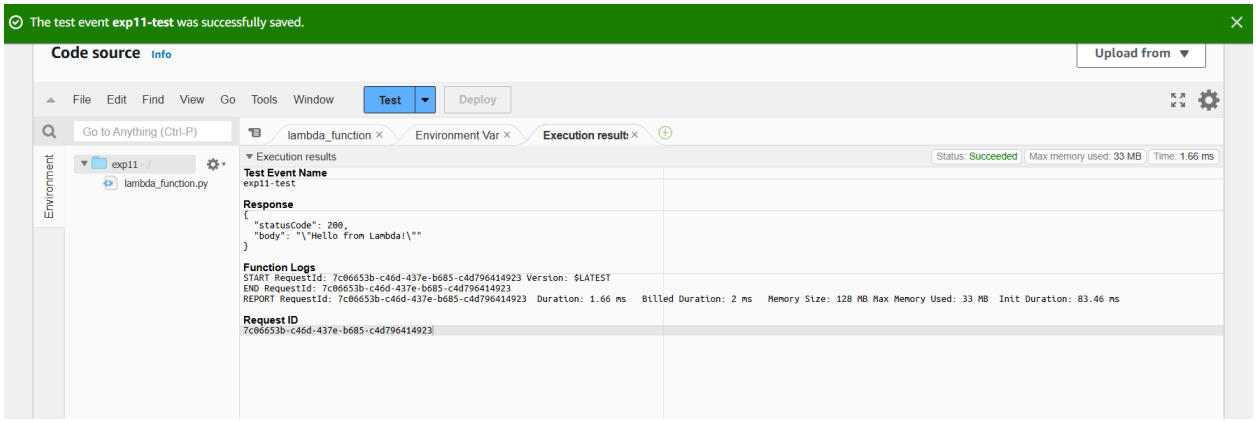


Step 5: Now In the Code section select the created event from the dropdown of test then click on test . You will see the below output.




Event JSON
Format JSON
1 {
2 "key1": "value1",
3 "key2": "value2",
4 "key3": "value3"
5 }
1:1 JSON Spaces: 2


Cancel
Invoke
Save


The test event exp11-test was successfully saved.
Code source Info
Upload from
File Edit Find View Go Tools Window
Test
Deploy
Go to Anything (Ctrl-P)
lambda_function
Environment Var
Execution result
Environment
exp11
lambda_function.py
Execution results
Status: Succeeded
Max memory used: 33 MB
Time: 1.66 ms
Test Event Name
exp11-test
Response
{
"statusCode": 200,
"body": "\"Hello from Lambda!\""
}
Function Logs
START RequestId: 7c06653b-c46d-437e-b685-c4d796414923 Version: $LATEST
END RequestId: 7c06653b-c46d-437e-b685-c4d796414923
REPORT RequestId: 7c06653b-c46d-437e-b685-c4d796414923 Duration: 1.66 ms Billed Duration: 2 ms Memory Size: 128 MB Max Memory Used: 33 MB Init Duration: 83.46 ms
Request ID
7c06653b-c46d-437e-b685-c4d796414923