Introduction to AWS Serverless Framework and Lambda functions

Part1.
Serverless Framework

1. Develop and deploy AWS Lambda functions

2. A CLI that offers:
    a. Structure
    b. Automation

3. Event-driven

4. Comprised of functions and events

5. Different from other application frameworks

    1. Manage code as well as infrastructure

    2. Multiple language supported
        a. Node.js
        b. Python
        c. Java

Core concepts

AWS Lambda function

1. Independent unit of deployment (merely code)

2. Deployed in the cloud

3. To perform a single task like saving a user to a database or process a file in a database

Events

- triggers the executes the Lambda function.
- An AWS API Gateway HTTP endpoint request (e.g., for a REST API)
- An AWS S3 bucket upload (e.g., for an image)
- A CloudWatch timer (e.g., run every 5 minutes)
- An AWS SNS topic (e.g., a message)
- A CloudWatch Alert (e.g., something happened)

Note: necessary setups are automatically created by the framework when you define an event for Lambda functions (e.g. API gateway endpoint). After that, the Lambda functions listen to the event.

Resources

AWS infrastructure components for functions:

1. DynamoDB Table for Saving users, Posts, Comments data.
2. S3 Bucket for saving files or images
3. SNS Topic for sending messages
4. Anything definable in CloudFormation and supported by the framework.

Services

- The framework's unit of organization.
- Can be a project file or can have multiple services for a single application.
- Where you define the functions, events to trigger them and the resources the functions use.
- A service can be described YAML or JSON format using serverless.yaml or *.json at the root directory of the project.
- When deploy using serverless deploy, everything in the service configuration file is deployed at once.

Plugins

- Plugins can overwrite or extend the functionality of the framework.
- Every *.yml can have a plugins property that features multiple plugins.

Amazon API Gateway

- To define http endpoints of a REST API or WebSocket API and connect them with the backend
- Handles:
    - Authentication
    - Access control
    - Monitoring
    - API requests tracing

API Gateway essential for serverless ecosystem:

- Being able to trigger the execution of a serverless function directly in response to an HTTP request

  Integrates with:

  - AWS Lambda: run Lambda functions to generate HTTP API responses
  - AWS SNS: public SNS notification when an endpoint is accessed
  - Amazon Cognito: authentication and authorization

  Works with the Serverless Framework:

  - The framework uses Lambda Proxy integration to make API Gateway events

  Benefits:

  - Map HTTP requests to specific functions in a Serverless application via an API gateway event.
  - Map WebSocket events to Serverless functions
  - Use multiple microservices to serve the same top-level API

  Drawbacks:

  - Added latency in APIs, milliseconds to response times
  - Not suitable for latency-sensitive applications and requests

  Limits:

  - Regional APIs: can only have 600 APIs per account
  - Integration timeouts: shortest about 50ms and longest 29 seconds
  - Payload size: 10MB

Part 2.
AWS Lambda

- Serverless computing service provided by amazon web services.
- Can perform any kind of computing task.
- Not needed to maintain own servers to run these functions.

How it works

- Each function runs in its own container.
- When a function is created, Lambda packages it into a new container.
- Before running the function, each container is allocated its RAM and CPU capacity.
- When running is finished, the RAM allocated at the beginning is multiplied by the amount of time the function spent running.
- Customers get charged based on the allocated memory and the amount of runtime the function took to complete.
- The infrastructure is taken care of. But with this you give up the flexibility of operating your own infrastructure.
- Many instances of the same function, or of different functions can be executed concurrently.
- You only get charged for the compute your functions use, good for creating highly scalable cloud computing solutions.

Why we need AWS Lambda for Serverless architecture

- when building serverless applications, AWS Lambda is one of the main candidates for running application code.
- to complete a serverless stack we need
    - a computing service
    - a database service → DynamoDb and RDS
    - an HTTP gateway service → API gateway

Use cases for Lambda functions

- Scalable APIs: HTTP request, different parts of the API, different parts of the API are automatically scaled according to the demand.
- Data processing: notifications, counters and analytics etc.
- Task automation: when tasks don't require an entire server at all times.
    - running scheduled jobs to perform clean-up in the infrastructure,
    - processing data from submitted forms on your website,
    - moving data around between datastores on demand.

Benefits

- pay per use
  - only for the compute the functions use
  - + any network traffic generated

- Fully managed infrastructure
  - no need to worry about underlying servers and operational tasks (upgrade OS system or managing the network layer)

- Automatic scaling
  - Instances of the function are only created when requested
  - You pay only for each function's runtime

- Tight integration with other AWS products
  - DynamoDB
  - S3
  - API Gateway

Limitations

- Cold start time
  - Small latency when a function is triggered by an event to run.
  - If not used in the last 15 min, it can be as high as 5-10 seconds.
  - Not good for latency critical applications
    - Solution: find out the bottlenecks and use the WarmUp plugin.
    - WarmUP does this by creating a scheduled event Lambda that invokes all the Lambdas you select in a configured time interval (default: 5 minutes) or a specific time, forcing your containers to stay alive.

- Function limits
  - Execution time/runtime
    - The Lambda function runs out after 15 minutes. No way to change this. Not good for long time needed functions.

  - Memory available to the funcion

Part 3.

AWS Step Functions

What is it?

- To orchestrate multiple AWS services to accomplish tasks
- Create steps in a process where the output of one step becomes the input for another step, all using a visual workflow editor.
- Provides:
    - Automatic retry handling
    - Triggering and tracking for each workflow step
    - Ensuring steps are executed in the correct order

How does it work?

- Step functions is a state machine
- Its primary abstractions are called states, configuration constitutes a map of all possible steps and the transitions between them.
- The state and their transitions are defined by the Amazon States Language (a JSON-based and propriety to Amazon.

Why integral to the serverless?

- To scale the application according to their growing workloads
- Keep the cost low and allow multiple teams to work simultaneously on different part of the application
- One way is to separate the business logic into a set of decoupled services
- A challenge when a large number of services need access to various parts of a shared state.
- To operate these services efficiently, orchestrating the flow of data through all application services in a single place is crucial
- Step functions handle this

AWS step functions integration with other AWS services:

- Task execution (wait vs. callback):
    - Invoking Lambda functions
    - Running AWS Batch jobs
    - Running a task in ECS

- Databases:
    - Inserting or fetching an item from Amazon DynamoDB

- Messages and notifications:
    - Publishing a topic n Amazon SNS
    - Sending a message in Amazon SQS

- Invoke another step functions workflow

How does it work with Serverless?

- By using sls step functions plugin
- Use the sls framework to create the AWS Lambda-based services that step functions will orchestrate.

Benefits:

- Quickly create complex sequence of tasks
- Manage state between executions of various stateless functions
- Decouple application workflow logic from business logic
- Efficient workflow with parallel executions

Drawbacks:

- The state language is for machine readability rather than for humans. Learning the language can be challenging.
- Decoupling business logic from task sequencing can make the code harder to understand
- Vendor lock-in, the language can only be based on AWS, migrating to another cloud provider needs re-implementation of the orchestration layer.

It's good for:

- Data ELTs(extract-transform-load), moving the data from the production system to data warehouse can be automated with step functions.
- Data processing, to interconnect multiple data processing steps. Simple queue service or simple notification service.
- Sls workflow orchestration as mentioned.

Limitations:

- The workflow cannot have more than 25,000 state transitions in a single execution.
- 1MB maximum request size, if more, consider using Amazon s3 to store the files and use s3 URIs s inputs to further operations.
- Spikes in AWS API requests caused by a workflow: a sudden flood of requests from the workflow might trigger the API limits. Solvable by grouping requests to the same AWS service into a single API call, or by introducing timeouts between operations.
- Cannot apply more than 50 tags to any of the AWS Step functions resources.

Two types of workflow:

- Standard: ideal for long-running, auditable workflows
- Express: ideal for high-event-rate workloads

Some of the common use cases:

- Function orchestration
- Branching- choice state
- Error handling- Retry and Catch
- Human in the loop- callback and a task token
- Parallel processing- parallel state
- Dynamic parallelism- map state

Useful Links:

- https://www.serverless.com/framework/docs/
- https://www.serverless.com/aws-lambda
- https://www.serverless.com/aws-step-functions
- https://docs.aws.amazon.com/step-functions/latest/dg/welcome.html
- https://www.serverless.com/blog/how-to-manage-your-aws-step-functions-with-serverless
- https://www.serverless.com/blog/serverless-api-gateway-domain
- https://docs.aws.amazon.com/AmazonS3/latest/dev/cors.html
- https://docs.aws.amazon.com/lambda/latest/dg/nodejs-handler.html