AWS Interview Prep:

**API composition in AWS**

API composition in AWS involves combining multiple APIs or services to create a unified API endpoint or workflow that provides the functionality needed by your application. This approach allows you to leverage existing AWS services to build powerful, scalable, and modular applications without having to build everything from scratch. Here’s how you can achieve API composition in AWS:

**Approach and Components:**

1. **AWS Lambda Functions**:
   * **Serverless Functions**: Use AWS Lambda to create small, focused functions that perform specific tasks or operations.
   * **Integration**: Lambda functions can be integrated with other AWS services like API Gateway, S3, DynamoDB, etc., to process data and trigger actions based on events or API calls.
2. **Amazon API Gateway**:
   * **API Management**: API Gateway allows you to create, publish, maintain, monitor, and secure APIs.
   * **Integration with Lambda**: API Gateway can directly integrate with AWS Lambda functions, enabling you to expose serverless APIs that run your backend logic.
3. **AWS Step Functions**:
   * **Orchestration**: Use Step Functions to coordinate multiple AWS services into serverless workflows.
   * **State Machines**: Define state machines that can call Lambda functions sequentially or in parallel, wait for tasks to complete, and handle errors gracefully.
4. **Amazon EventBridge**:
   * **Event-Driven Architecture**: EventBridge allows you to connect different services using events.
   * **Integration**: You can create rules that match events and route them to targets such as Lambda functions, SNS topics, SQS queues, or even other EventBridge buses.
5. **AWS AppSync**:
   * **GraphQL API**: AppSync provides GraphQL APIs to access and combine data from multiple sources.
   * **Integration**: Connect to data sources like DynamoDB, Elasticsearch, Lambda functions, HTTP endpoints, etc., and define resolvers to fetch and combine data as per GraphQL queries.

**Example Scenario:**

Let’s say you want to build an application where users can upload images, which are then processed (e.g., resized, watermarked) and stored in S3, with metadata stored in DynamoDB. Here’s how you could compose APIs using AWS services:

* **API Gateway**: Expose RESTful endpoints for uploading images and retrieving metadata.
* **Lambda Functions**: Implement functions for image processing, metadata storage/retrieval, and business logic.
* **S3**: Store original and processed images.
* **DynamoDB**: Store metadata such as image IDs, timestamps, and processing status.
* **Step Functions**: Orchestrate the workflow: trigger Lambda functions for image processing, store metadata in DynamoDB, and handle errors or retries.
* **EventBridge**: Trigger workflows based on events, such as image upload completion or DynamoDB updates.

**Benefits:**

* **Scalability**: AWS services like Lambda, API Gateway, and S3 scale automatically based on demand.
* **Flexibility**: Compose APIs and workflows tailored to specific use cases using a variety of AWS services.
* **Cost-Effective**: Pay only for the resources used without managing infrastructure.
* **Integration**: Easily integrate with other AWS services or third-party APIs.

**Considerations:**

* **Security**: Implement IAM roles, policies, and encryption to secure data and APIs.
* **Monitoring**: Use AWS CloudWatch for monitoring and logging to track API usage, performance, and errors.
* **Error Handling**: Design workflows to handle failures gracefully and implement retries where necessary.

**Event-driven architecture in AWS**

Event-driven architecture (EDA) in AWS leverages AWS services to build applications where the flow of information and processing is driven by events. Here are some key components and considerations for implementing event-driven architecture in AWS:

1. **Event Sources**: These are services or systems that generate events. Examples include AWS services like S3 (for object creation events), DynamoDB (for data changes), SNS (for notifications), SQS (for message queues), and custom applications integrated using AWS SDKs.
2. **Event Consumers**: These are services or functions that react to events. In AWS, event consumers are typically AWS Lambda functions, which are serverless functions that can execute code in response to events. However, event consumers can also be EC2 instances, containers, or other services depending on the architecture.
3. **Event Routing and Processing**: AWS provides various services to route events from sources to consumers:
   * **Amazon EventBridge**: A serverless event bus that makes it easy to connect applications using data from AWS services, integrated SaaS applications, and custom applications.
   * **Amazon SNS**: A fully managed pub/sub messaging service that enables distributed systems to communicate with each other.
   * **Amazon SQS**: A fully managed message queuing service for decoupling and scaling microservices, distributed systems, and serverless applications.
   * **AWS Lambda**: Execute code in response to events. Lambda can be triggered directly by many AWS services, or indirectly via EventBridge or SNS/SQS.
4. **Data Integration**: Often, events carry data payloads. AWS services like AWS Glue, Amazon Kinesis, or custom Lambda functions can be used for data transformation, enrichment, or filtering before passing events to downstream consumers.
5. **Scalability and Resilience**: AWS inherently provides scalability and fault tolerance for event-driven architectures:
   * **Auto-scaling**: Services like Lambda and SQS automatically scale based on workload.
   * **Resilience**: Built-in redundancy and availability zones ensure high availability of services.
6. **Monitoring and Observability**: AWS CloudWatch can be used to monitor and log events, metrics, and alarms for EDA components. AWS X-Ray provides distributed tracing for understanding performance bottlenecks.

**Example Use Cases:**

* **Real-time Analytics**: Processing streaming data from IoT devices or clickstream data.
* **Asynchronous Workflows**: Handling file uploads, batch processing, or long-running tasks.
* **Microservices Communication**: Decoupling and scaling of microservices through event-driven communication.
* **Automated Responses**: Triggering actions based on changes in AWS infrastructure or external systems.

**Design Patterns:**

* **Fan-out**: Distributing events to multiple consumers for parallel processing.
* **Chained**: Sequentially processing events where output of one function triggers another.
* **Filters and Routing**: Using EventBridge rules or SNS/SQS filtering to route events based on content.

**AWS SNS Services**

AWS SNS (Amazon Simple Notification Service) is a fully managed pub/sub messaging service provided by Amazon Web Services. It enables you to send messages or notifications to a large number of subscribers (endpoints) simultaneously, such as mobile devices, email addresses, Amazon SQS queues, AWS Lambda functions, and more. SNS simplifies the process of sending messages to distributed systems and applications, making it a versatile tool for building event-driven architectures and communication workflows.

**Key Features and Concepts:**

1. **Topics**:
   * **Logical Channels**: SNS uses topics as communication channels to distribute messages to multiple subscribers.
   * **Publishing Messages**: You publish messages to topics, and SNS delivers them to subscribers asynchronously.
2. **Subscriptions**:
   * **Endpoints**: Subscribers receive messages from topics through subscriptions.
   * **Types of Subscribers**: Subscribers can be various AWS services (like SQS, Lambda), HTTP endpoints, email addresses, SMS endpoints, mobile push notifications (iOS, Android), and more.
   * **Subscription Protocols**: SNS supports protocols such as HTTP, HTTPS, Email, Email-JSON, SQS, Application (for mobile push notifications), Lambda, and SMS.
3. **Message Filtering**:
   * **Message Attributes**: You can use message attributes to filter messages based on specific criteria.
   * **Subscription Filter Policies**: Define filter policies on subscriptions to receive only messages that match certain attributes.
4. **Delivery Retries and Dead-Letter Queues**:
   * **Retry Mechanism**: SNS retries message delivery attempts multiple times (configurable).
   * **Dead-Letter Queues**: Optionally, you can configure a dead-letter queue (DLQ) to capture messages that SNS cannot deliver after retries.
5. **Message Delivery**:
   * **Fanout**: Messages published to a topic are delivered to each subscribed endpoint concurrently.
   * **Ordering**: SNS ensures best-effort message ordering when messages are published to topics.
6. **Message Attributes**:
   * **Metadata**: You can attach metadata to messages using message attributes (key-value pairs).

**Use Cases:**

* **Notifications and Alerts**: Send notifications and alerts to users or administrators via email, SMS, or mobile push notifications.
* **Application Integration**: Integrate applications and services by using SNS to trigger actions or workflows in response to events.
* **Distributed Systems**: Build event-driven architectures where services react to changes or events published to SNS topics.
* **Decoupling Systems**: Decouple message producers from consumers by using SNS as an intermediary to distribute messages.

**Example Workflow:**

1. **Create a Topic**: Define an SNS topic for a specific category of messages, e.g., "OrderNotifications".
2. **Subscribe Endpoints**: Subscribe various endpoints (e.g., email addresses, SQS queues, Lambda functions) to receive messages from the "OrderNotifications" topic.
3. **Publish Messages**: Publish messages to the "OrderNotifications" topic whenever an order status changes.
4. **Delivery**: SNS delivers the messages to all subscribed endpoints, ensuring that each endpoint receives the message according to its subscription protocol.

**Integration with Other AWS Services:**

* **Amazon SQS**: Use SNS to send messages to SQS queues for asynchronous processing by multiple consumers.
* **AWS Lambda**: Trigger Lambda functions directly from SNS messages, allowing for serverless event-driven architectures.
* **Amazon EC2**: Notify instances or applications running on EC2 using SNS to handle configuration changes, alerts, or notifications.

**Security and Monitoring:**

* **IAM Policies**: Control access to SNS topics and subscriptions using AWS IAM policies.
* **Amazon CloudWatch**: Monitor SNS usage, message delivery, and failure metrics using CloudWatch.

AWS SNS provides a reliable and scalable solution for messaging and notification needs in cloud-based applications, facilitating communication between distributed components and enabling event-driven architectures.

**SNS With Lambda [Node Ts Code Snipest]:**

Top of Form

Bottom of Form

|  |
| --- |
| import { Stack, StackProps, CfnParameter } from 'aws-cdk-lib'; |
|  | import { Construct } from 'constructs'; |
|  | import \* as iam from 'aws-cdk-lib/aws-iam'; |
|  | import \* as lambda from 'aws-cdk-lib/aws-lambda'; |
|  | import \* as subscriptions from 'aws-cdk-lib/aws-sns-subscriptions'; |
|  | import \* as sns from 'aws-cdk-lib/aws-sns'; |
|  | import \* as dotenv from 'dotenv'; |
|  |  |
|  | dotenv.config(); |
|  | export class CdkLambdaSnsStack extends Stack { |
|  | constructor(scope: Construct, id: string, props?: StackProps) { |
|  | super(scope, id, props); |
|  |  |
|  | // 1. Set SNS Topic |
|  | // 1-1 Use existing SNS topic: Hard code the topic arn |
|  | // const myTopic = sns.Topic.fromTopicArn(this, 'MyTopic', <topic-arn>); |
|  |  |
|  | // 1-2 Use existing SNS topic: Read arn from .env file |
|  | // const myTopic = sns.Topic.fromTopicArn(this, 'MyTopic', process.env.SNS\_ARN ?? ''); |
|  |  |
|  | // 1-3 Create a new SNS topic |
|  | const myTopic = new sns.Topic(this, 'MyTopic'); |
|  |  |
|  | // 2.Set SNS subscription |
|  | // 2-1 Hard code email address |
|  | // myTopic.addSubscription(new subscriptions.EmailSubscription('mail-at-address')); |
|  |  |
|  | // 2-2 Read email address from .env file |
|  | // myTopic.addSubscription(new subscriptions.EmailSubscription(process.env.MY\_EMAIL ?? '')); |
|  |  |
|  | // 2-3 Read Context: cdk deploy --context email=aaaa@example.com |
|  | // const email = this.node.tryGetContext('email'); |
|  | // myTopic.addSubscription(new subscriptions.EmailSubscription(email)); |
|  |  |
|  | // 2-4 Read from Cfn parameter for email address. cdk deploy --parameters emailparam=aaaa@example.com |
|  | // const email = new CfnParameter(this, 'email-param'); |
|  | // myTopic.addSubscription(new subscriptions.EmailSubscription(email.valueAsString)); |
|  |  |
|  | const helloFn = new lambda.Function(this, 'HelloHandler', { |
|  | runtime: lambda.Runtime.PYTHON\_3\_9, |
|  | code: lambda.Code.fromAsset('lambda'), |
|  | handler: 'my\_lambda.handler', |
|  | environment: {'SNS\_ARN': myTopic.topicArn,}, |
|  | }); |
|  |  |
|  | const snsTopicPolicy = new iam.PolicyStatement({ |
|  | actions: ['sns:publish'], |
|  | resources: ['\*'], |
|  | }); |
|  |  |
|  | helloFn.addToRolePolicy(snsTopicPolicy); |
|  | } |
|  | }  AWS SNS  AWS SNS (Amazon Simple Notification Service) is a fully managed pub/sub messaging service provided by Amazon Web Services. It enables you to send messages or notifications to a large number of subscribers (endpoints) simultaneously, such as mobile devices, email addresses, Amazon SQS queues, AWS Lambda functions, and more. SNS simplifies the process of sending messages to distributed systems and applications, making it a versatile tool for building event-driven architectures and communication workflows. Key Features and Concepts:  1. **Topics**:    * **Logical Channels**: SNS uses topics as communication channels to distribute messages to multiple subscribers.    * **Publishing Messages**: You publish messages to topics, and SNS delivers them to subscribers asynchronously. 2. **Subscriptions**:    * **Endpoints**: Subscribers receive messages from topics through subscriptions.    * **Types of Subscribers**: Subscribers can be various AWS services (like SQS, Lambda), HTTP endpoints, email addresses, SMS endpoints, mobile push notifications (iOS, Android), and more.    * **Subscription Protocols**: SNS supports protocols such as HTTP, HTTPS, Email, Email-JSON, SQS, Application (for mobile push notifications), Lambda, and SMS. 3. **Message Filtering**:    * **Message Attributes**: You can use message attributes to filter messages based on specific criteria.    * **Subscription Filter Policies**: Define filter policies on subscriptions to receive only messages that match certain attributes. 4. **Delivery Retries and Dead-Letter Queues**:    * **Retry Mechanism**: SNS retries message delivery attempts multiple times (configurable).    * **Dead-Letter Queues**: Optionally, you can configure a dead-letter queue (DLQ) to capture messages that SNS cannot deliver after retries. 5. **Message Delivery**:    * **Fanout**: Messages published to a topic are delivered to each subscribed endpoint concurrently.    * **Ordering**: SNS ensures best-effort message ordering when messages are published to topics. 6. **Message Attributes**:    * **Metadata**: You can attach metadata to messages using message attributes (key-value pairs).  Use Cases:  * **Notifications and Alerts**: Send notifications and alerts to users or administrators via email, SMS, or mobile push notifications. * **Application Integration**: Integrate applications and services by using SNS to trigger actions or workflows in response to events. * **Distributed Systems**: Build event-driven architectures where services react to changes or events published to SNS topics. * **Decoupling Systems**: Decouple message producers from consumers by using SNS as an intermediary to distribute messages.  Example Workflow:  1. **Create a Topic**: Define an SNS topic for a specific category of messages, e.g., "OrderNotifications". 2. **Subscribe Endpoints**: Subscribe various endpoints (e.g., email addresses, SQS queues, Lambda functions) to receive messages from the "OrderNotifications" topic. 3. **Publish Messages**: Publish messages to the "OrderNotifications" topic whenever an order status changes. 4. **Delivery**: SNS delivers the messages to all subscribed endpoints, ensuring that each endpoint receives the message according to its subscription protocol.  Integration with Other AWS Services:  * **Amazon SQS**: Use SNS to send messages to SQS queues for asynchronous processing by multiple consumers. * **AWS Lambda**: Trigger Lambda functions directly from SNS messages, allowing for serverless event-driven architectures. * **Amazon EC2**: Notify instances or applications running on EC2 using SNS to handle configuration changes, alerts, or notifications.  Security and Monitoring:  * **IAM Policies**: Control access to SNS topics and subscriptions using AWS IAM policies. * **Amazon CloudWatch**: Monitor SNS usage, message delivery, and failure metrics using CloudWatch.   AWS SNS provides a reliable and scalable solution for messaging and notification needs in cloud-based applications, facilitating communication between distributed components and enabling event-driven architectures.  **SQS AWS**  Amazon SQS (Simple Queue Service) is a fully managed message queuing service provided by AWS. It enables you to decouple and scale microservices, distributed systems, and serverless applications by allowing messages to be queued and processed asynchronously. Here’s an overview of SQS, its key features, use cases, and integration options: Key Features:  1. **Message Queues**:    * SQS provides message queues where messages can be stored temporarily until they are processed by consumers (receivers). 2. **Distributed Architecture**:    * SQS is designed to be highly available and scalable. It redundantly stores messages across multiple availability zones within a region to ensure message durability. 3. **Message Visibility Timeout**:    * Messages are not immediately deleted from the queue after being received. Instead, SQS allows you to set a visibility timeout during which the message is invisible to other consumers. If the message processing fails, it becomes visible again and can be processed by another consumer. 4. **Delay Queues**:    * You can configure SQS to delay the delivery of messages up to 15 minutes, allowing you to control the delivery time of messages. 5. **Dead-Letter Queues (DLQ)**:    * SQS supports dead-letter queues, which capture messages that cannot be processed successfully after a certain number of attempts. DLQ helps in debugging and handling message processing errors. 6. **Long Polling**:    * SQS supports long polling, where the polling request waits until a message is available in the queue before returning a response. This reduces the number of empty responses and improves efficiency. 7. **Message Attributes**:    * Messages in SQS can have associated metadata (attributes) in the form of key-value pairs, providing additional context or instructions to consumers.  Use Cases:  * **Decoupling and Scaling**: SQS allows components of a distributed system to operate independently, enabling seamless scaling of applications. * **Asynchronous Communication**: Use SQS to decouple components that produce and consume data, ensuring reliable message delivery and reducing coupling between services. * **Batch Processing**: SQS supports batch operations, allowing you to send or receive multiple messages in a single API call, which is useful for processing messages in bulk. * **Event-Driven Architecture**: SQS integrates well with AWS Lambda, Amazon EC2, and other AWS services, enabling event-driven architectures where services react to messages in the queue.  Integration with Other AWS Services:  * **AWS Lambda**: SQS can directly trigger AWS Lambda functions to process messages asynchronously, enabling serverless architectures. * **Amazon EC2**: Integrate SQS with EC2 instances to handle background processing tasks or job queues. * **Amazon SNS**: Combine SQS with SNS to fan out messages from a topic to multiple SQS queues or other endpoints.  Security and Monitoring:  * **IAM Policies**: Control access to SQS queues using AWS IAM policies to specify who can send, receive, or manage messages. * **Amazon CloudWatch**: Monitor SQS queues using CloudWatch metrics, such as the number of messages sent, received, or deleted, queue depth, and message age.  Example Workflow:  1. **Create a Queue**: Define an SQS queue with a unique name, such as "OrderProcessingQueue". 2. **Send Messages**: Producers (e.g., applications, services) send messages to the "OrderProcessingQueue" containing order details. 3. **Receive Messages**: Consumers (e.g., worker processes, Lambda functions) poll the queue to retrieve and process messages. 4. **Processing**: Consumers process each message, perform required actions (e.g., process order, update database), and delete the message from the queue upon successful completion. 5. **Error Handling**: Configure DLQ to capture messages that cannot be processed successfully after retries, allowing for investigation and reprocessing.   Amazon SQS is a fundamental building block for building scalable and reliable applications that need to handle messaging and task-queuing scenarios in a distributed environment. It provides high availability, durability, and flexibility while offloading the complexity of managing message queues to AWS.  **SQS CDK (Node TS):**  import \* as cdk from '@aws-cdk/core';  import \* as sqs from '@aws-cdk/aws-sqs';  export class SqsCdkDemoStack extends cdk.Stack {  constructor(scope: cdk.Construct, id: string, props?: cdk.StackProps) {  super(scope, id, props);  // Define an SQS queue  const queue = new sqs.Queue(this, 'MyQueue', {  visibilityTimeout: cdk.Duration.seconds(300), // Message visibility timeout  retentionPeriod: cdk.Duration.days(7) // Message retention period  });  // Output the Queue's URL  new cdk.CfnOutput(this, 'QueueUrl', {  value: queue.queueUrl,  });  }  }  **Accessing the SQS:**  const AWS = require('aws-sdk');  // Configure AWS SDK with your region  AWS.config.update({ region: 'your-region' });  // Create SQS service object  const sqs = new AWS.SQS();  // Example message payload  const messageParams = {  MessageBody: 'Hello from AWS CDK!',  QueueUrl: 'your-sqs-queue-url'  };  // Send message to SQS queue  sqs.sendMessage(messageParams, (err, data) => {  if (err) {  console.log('Error', err);  } else {  console.log('Message sent:', data.MessageId);  }  });  Top of Form  Bottom of Form |

**CRUD AWS CDK Node JS (Typescript)**

Creating a CRUD (Create, Read, Update, Delete) application using AWS CDK with Node.js involves setting up AWS Lambda functions, API Gateway endpoints, and DynamoDB tables. This setup allows you to build a serverless REST API where data is stored and retrieved from DynamoDB through Lambda functions triggered by API Gateway. Here’s a step-by-step guide:

**Step 1: Set Up Your AWS CDK Project**

Ensure you have Node.js and npm installed. Initialize a new AWS CDK project if you haven't already:

bash

Copy code

mkdir crud-cdk-demo

cd crud-cdk-demo

cdk init app --language=typescript # Initialize CDK project (can choose TypeScript or JavaScript)

Install necessary CDK modules for DynamoDB, Lambda, API Gateway, and IAM (for permissions):

bash

Copy code

npm install @aws-cdk/aws-dynamodb @aws-cdk/aws-lambda @aws-cdk/aws-apigateway @aws-cdk/aws-iam

**Step 2: Define the DynamoDB Table**

Modify your lib/crud-cdk-demo-stack.ts (or .js for JavaScript) to define the DynamoDB table:

typescript

Copy code

import \* as cdk from '@aws-cdk/core';

import \* as dynamodb from '@aws-cdk/aws-dynamodb';

export class CrudCdkDemoStack extends cdk.Stack {

constructor(scope: cdk.Construct, id: string, props?: cdk.StackProps) {

super(scope, id, props);

// Define DynamoDB table

const table = new dynamodb.Table(this, 'ItemsTable', {

partitionKey: { name: 'itemId', type: dynamodb.AttributeType.STRING },

removalPolicy: cdk.RemovalPolicy.DESTROY, // Only for demo purposes; consider retention policy

});

}

}

**Step 3: Define the Lambda Functions**

Define Lambda functions for CRUD operations (createItem, getItem, updateItem, deleteItem). Each function interacts with DynamoDB through the AWS SDK for Node.js (aws-sdk).

typescript

Copy code

import \* as lambda from '@aws-cdk/aws-lambda';

// Create Lambda functions

const createItemLambda = new lambda.Function(this, 'CreateItemFunction', {

runtime: lambda.Runtime.NODEJS\_14\_X,

handler: 'index.handler',

code: lambda.Code.fromAsset('lambda'), // Directory containing Lambda code

environment: {

TABLE\_NAME: table.tableName,

PRIMARY\_KEY: 'itemId',

},

});

const getItemLambda = new lambda.Function(this, 'GetItemFunction', {

runtime: lambda.Runtime.NODEJS\_14\_X,

handler: 'index.handler',

code: lambda.Code.fromAsset('lambda'),

environment: {

TABLE\_NAME: table.tableName,

PRIMARY\_KEY: 'itemId',

},

});

const updateItemLambda = new lambda.Function(this, 'UpdateItemFunction', {

runtime: lambda.Runtime.NODEJS\_14\_X,

handler: 'index.handler',

code: lambda.Code.fromAsset('lambda'),

environment: {

TABLE\_NAME: table.tableName,

PRIMARY\_KEY: 'itemId',

},

});

const deleteItemLambda = new lambda.Function(this, 'DeleteItemFunction', {

runtime: lambda.Runtime.NODEJS\_14\_X,

handler: 'index.handler',

code: lambda.Code.fromAsset('lambda'),

environment: {

TABLE\_NAME: table.tableName,

PRIMARY\_KEY: 'itemId',

},

});

**Step 4: Define API Gateway with Lambda Integrations**

Define an API Gateway REST API with endpoints that trigger the Lambda functions:

typescript

Copy code

import \* as apigateway from '@aws-cdk/aws-apigateway';

// Define API Gateway REST API

const api = new apigateway.RestApi(this, 'itemsApi', {

restApiName: 'Items Service',

description: 'CRUD operations for Items',

});

// Define API resources and integrate with Lambda functions

const items = api.root.addResource('items');

items.addMethod('POST', new apigateway.LambdaIntegration(createItemLambda));

items.addMethod('GET', new apigateway.LambdaIntegration(getItemLambda));

const singleItem = items.addResource('{itemId}');

singleItem.addMethod('GET', new apigateway.LambdaIntegration(getItemLambda));

singleItem.addMethod('PUT', new apigateway.LambdaIntegration(updateItemLambda));

singleItem.addMethod('DELETE', new apigateway.LambdaIntegration(deleteItemLambda));

**Step 5: Lambda Function Code**

Create Lambda function code (lambda/index.js) for CRUD operations using the AWS SDK for Node.js (aws-sdk):

javascript

Copy code

const AWS = require('aws-sdk');

const dynamoDB = new AWS.DynamoDB.DocumentClient();

exports.handler = async (event) => {

const tableName = process.env.TABLE\_NAME;

const primaryKey = process.env.PRIMARY\_KEY;

switch (event.httpMethod) {

case 'POST':

// Create item

// Example: const item = JSON.parse(event.body);

// await dynamoDB.put({ TableName: tableName, Item: item }).promise();

break;

case 'GET':

// Get item(s)

// Example: const itemId = event.pathParameters.itemId;

// const data = await dynamoDB.get({ TableName: tableName, Key: { [primaryKey]: itemId } }).promise();

// return { statusCode: 200, body: JSON.stringify(data.Item) };

break;

case 'PUT':

// Update item

// Example: const updatedItem = JSON.parse(event.body);

// await dynamoDB.update({ TableName: tableName, Key: { [primaryKey]: updatedItem[primaryKey] }, UpdateExpression: 'SET ...', ExpressionAttributeValues: { ... } }).promise();

break;

case 'DELETE':

// Delete item

// Example: const itemIdToDelete = event.pathParameters.itemId;

// await dynamoDB.delete({ TableName: tableName, Key: { [primaryKey]: itemIdToDelete } }).promise();

break;

default:

return { statusCode: 404, body: 'Invalid HTTP Method' };

}

};

Replace the placeholder code in the Lambda functions (POST, GET, PUT, DELETE) with actual DynamoDB operations using the AWS SDK.

**Step 6: Deploy Your CDK Stack**

Deploy the CDK stack using the AWS CDK CLI:

bash

Copy code

cdk deploy

Follow the prompts to confirm deployment. After deployment, AWS CDK will provide you with the API Gateway endpoint URL (itemsApi.EndpointURL) that you can use to interact with your CRUD API.

**Step 7: Clean Up**

To avoid ongoing charges, clean up resources by destroying the CDK stack:

bash

Copy code

cdk destroy

Confirm the deletion when prompted.

**Summary**

Using AWS CDK with Node.js allows you to define and deploy a serverless CRUD API with DynamoDB as the data store, Lambda functions for business logic, and API Gateway for RESTful endpoints. This approach enables you to manage infrastructure as code, automate deployments, and leverage the scalability and reliability of AWS services for your applications. Adjustments can be made based on specific requirements or additional functionalities such as authentication, authorization, and error handling.

AWS DynamoDB Quires Using Node js

Querying Amazon DynamoDB using Node.js involves using the AWS SDK for JavaScript (aws-sdk). DynamoDB provides both the DocumentClient and low-level DynamoDB interfaces for interacting with the database. Here’s a guide on how to perform various queries such as GetItem, Query, Scan, and UpdateItem using Node.js:

**Prerequisites**

1. **AWS SDK for JavaScript**: Make sure you have installed the aws-sdk package in your Node.js project.

bash

Copy code

npm install aws-sdk

1. **AWS Credentials**: Configure AWS SDK with your AWS credentials (access key ID and secret access key) either through environment variables or directly in your Node.js code. It's recommended to use AWS IAM roles or environment variables for security reasons.

javascript

Copy code

const AWS = require('aws-sdk');

// Set the region

AWS.config.update({ region: 'your-region' });

// Create DynamoDB DocumentClient

const docClient = new AWS.DynamoDB.DocumentClient();

**1. GetItem**

Retrieve a single item from DynamoDB based on its primary key:

javascript

Copy code

const params = {

TableName: 'YourTableName',

Key: {

partitionKey: 'value1',

sortKey: 'value2'

}

};

docClient.get(params, (err, data) => {

if (err) {

console.error('Unable to read item. Error JSON:', JSON.stringify(err, null, 2));

} else {

console.log('GetItem succeeded:', JSON.stringify(data, null, 2));

}

});

**2. Query**

Query items in DynamoDB based on a partition key and optionally a sort key condition:

javascript

Copy code

const params = {

TableName: 'YourTableName',

KeyConditionExpression: 'partitionKey = :pk',

ExpressionAttributeValues: {

':pk': 'value1'

}

};

docClient.query(params, (err, data) => {

if (err) {

console.error('Unable to query. Error JSON:', JSON.stringify(err, null, 2));

} else {

console.log('Query succeeded:', JSON.stringify(data, null, 2));

}

});

**3. Scan**

Scan the entire table or a segment of the table:

javascript

Copy code

const params = {

TableName: 'YourTableName',

};

docClient.scan(params, (err, data) => {

if (err) {

console.error('Unable to scan. Error JSON:', JSON.stringify(err, null, 2));

} else {

console.log('Scan succeeded:', JSON.stringify(data, null, 2));

}

});

**4. UpdateItem**

Update an existing item in DynamoDB:

javascript

Copy code

const params = {

TableName: 'YourTableName',

Key: {

partitionKey: 'value1',

sortKey: 'value2'

},

UpdateExpression: 'set #attrName = :attrValue',

ExpressionAttributeNames: {

'#attrName': 'attributeNameToUpdate'

},

ExpressionAttributeValues: {

':attrValue': 'newValue'

},

ReturnValues: 'UPDATED\_NEW'

};

docClient.update(params, (err, data) => {

if (err) {

console.error('Unable to update item. Error JSON:', JSON.stringify(err, null, 2));

} else {

console.log('UpdateItem succeeded:', JSON.stringify(data, null, 2));

}

});

**Notes:**

* Replace 'YourTableName' with the name of your DynamoDB table.
* Adjust partitionKey, sortKey, and other attribute names/values according to your table schema and query requirements.
* Handle errors and exceptions (err) returned by DynamoDB operations to ensure robust error handling in your Node.js application.
* DynamoDB operations (get, query, scan, update, etc.) are asynchronous, so you can use callbacks as shown or promises (promise() method) for handling asynchronous operations in a more modern Node.js code style.

This guide provides a basic overview of how to perform common DynamoDB operations using Node.js. Depending on your application's specific requirements, you may need to implement additional features such as pagination, condition expressions, batch operations, and more complex queries.

### Authenticating API Gateway Requests to Lambda Functions with AWS CDK and TypeScript

To implement authentication for API Gateway requests that invoke Lambda functions using AWS CDK and TypeScript, you can use various strategies, including JWT authorizers, AWS Cognito, or custom authorizers. Here’s a detailed guide to setting up JWT-based authentication with a custom authorizer.

### Steps

1. **Set Up Your CDK Project**
2. **Create Lambda Functions**
3. **Define CDK Stack**
4. **Deploy and Test**

### Step 1: Set Up Your CDK Project

Initialize a new CDK project:

Bash || Terminal

mkdir cdk-lambda-auth

cd cdk-lambda-auth

cdk init app --language=typescript

### Step 2: Create Lambda Functions

#### Lambda Function for Business Logic

Create a directory for the Lambda function and add the code:

**lambda/hello/index.ts**:

typescript

Copy code

import { APIGatewayProxyHandler } from 'aws-lambda';

export const handler: APIGatewayProxyHandler = async (event) => {

return {

statusCode: 200,

body: JSON.stringify({ message: "Hello from Lambda!" }),

};

};

#### Lambda Function for Custom JWT Authorizer

**lambda/auth/index.ts**:

typescript

Copy code

import { APIGatewayTokenAuthorizerHandler } from 'aws-lambda';

import jwt from 'jsonwebtoken';

const SECRET\_KEY = 'your\_secret\_key'; // Replace with your JWT secret or public key

export const handler: APIGatewayTokenAuthorizerHandler = async (event) => {

const token = event.authorizationToken?.split(' ')[1]; // Extract token

if (!token) {

return generatePolicy('user', 'Deny', event.methodArn);

}

try {

const decoded = jwt.verify(token, SECRET\_KEY);

// Add your logic to get user info if needed

return generatePolicy(decoded.sub, 'Allow', event.methodArn);

} catch (err) {

return generatePolicy('user', 'Deny', event.methodArn);

}

};

// Helper function to generate IAM policy

const generatePolicy = (principalId: string, effect: string, resource: string) => {

return {

principalId,

policyDocument: {

Version: '2012-10-17',

Statement: [{

Action: 'execute-api:Invoke',

Effect: effect,

Resource: resource

}]

}

};

};

**Install JWT dependency**:

bash

cd lambda/auth

npm init -y

npm install jsonwebtoken @types/jsonwebtoken

cd ../../

### Step 3: Define CDK Stack

Edit the CDK stack to include Lambda functions and API Gateway setup.

**lib/cdk-lambda-auth-stack.ts**:

typescript

Copy code

import \* as cdk from 'aws-cdk-lib';

import { Duration } from 'aws-cdk-lib';

import \* as lambda from 'aws-cdk-lib/aws-lambda';

import \* as apigateway from 'aws-cdk-lib/aws-apigateway';

import \* as path from 'path';

export class CdkLambdaAuthStack extends cdk.Stack {

constructor(scope: cdk.App, id: string, props?: cdk.StackProps) {

super(scope, id, props);

// Lambda for the main function

const helloLambda = new lambda.Function(this, 'HelloLambda', {

runtime: lambda.Runtime.NODEJS\_18\_X,

code: lambda.Code.fromAsset(path.join(\_\_dirname, '../lambda/hello')),

handler: 'index.handler',

});

// Lambda for the authorizer

const authLambda = new lambda.Function(this, 'AuthLambda', {

runtime: lambda.Runtime.NODEJS\_18\_X,

code: lambda.Code.fromAsset(path.join(\_\_dirname, '../lambda/auth')),

handler: 'index.handler',

});

// API Gateway

const api = new apigateway.RestApi(this, 'HelloApi', {

restApiName: 'Hello Service',

description: 'This service serves hello.',

});

// Custom Authorizer

const auth = new apigateway.TokenAuthorizer(this, 'HelloAuthorizer', {

handler: authLambda,

resultsCacheTtl: Duration.seconds(0), // Disable cache for simplicity

});

// Define API method

const hello = api.root.addResource('hello');

hello.addMethod('GET', new apigateway.LambdaIntegration(helloLambda), {

authorizer: auth,

});

}

}

### Step 4: Deploy and Test

#### Deploy CDK Stack

bash

Copy code

cdk bootstrap

cdk deploy

#### Test the API

1. **Generate JWT**: Obtain a JWT from your authentication provider.
2. **Make Request**:
   * Use curl or Postman to test the endpoint.

bash

Copy code

curl -H "Authorization: Bearer YOUR\_JWT\_TOKEN" <https://your-api-id.execute-api.your-region.amazonaws.com/prod/hello>

### Project Structure

python

Copy code

cdk-lambda-auth/

│

├── lambda/

│ ├── hello/

│ │ └── index.ts

│ └── auth/

│ └── index.ts

│

├── lib/

│ └── cdk-lambda-auth-stack.ts

│

└── bin/

└── cdk-lambda-auth.ts

### Notes

1. **Secret Management**: In production, use secure ways to manage secrets, such as AWS Secrets Manager.
2. **Environment Variables**: Pass sensitive data using environment variables or parameter stores.
3. **CORS**: Configure CORS settings in API Gateway if needed.

### Summary

This setup demonstrates how to authenticate API Gateway requests using a custom JWT authorizer in AWS CDK with TypeScript. You can extend this by integrating with AWS Cognito or other identity providers for more complex use cases.

**Configure CORS settings in API Gateway CDK typescript**

Configuring CORS (Cross-Origin Resource Sharing) in API Gateway using AWS CDK and TypeScript involves setting up the necessary headers and options to allow cross-origin requests. Here's a step-by-step guide on how to configure CORS settings for your API Gateway in a CDK stack written in TypeScript.

**Steps to Configure CORS in API Gateway using AWS CDK**

1. **Set Up CDK Project**
2. **Create Lambda Functions**
3. **Define CDK Stack with CORS Configuration**
4. **Deploy and Test**

**Step 1: Set Up CDK Project**

If you haven’t already, initialize a new CDK project:

bash

Copy code

mkdir cdk-lambda-cors

cd cdk-lambda-cors

cdk init app --language=typescript

**Step 2: Create Lambda Functions**

Create the Lambda function that will handle requests:

**lambda/hello/index.ts**:

typescript

Copy code

import { APIGatewayProxyHandler } from 'aws-lambda';

export const handler: APIGatewayProxyHandler = async (event) => {

return {

statusCode: 200,

body: JSON.stringify({ message: "Hello from Lambda!" }),

};

};

**Step 3: Define CDK Stack with CORS Configuration**

Edit the CDK stack to include Lambda functions and configure CORS for the API Gateway.

**lib/cdk-lambda-cors-stack.ts**:

typescript

Copy code

import \* as cdk from 'aws-cdk-lib';

import \* as lambda from 'aws-cdk-lib/aws-lambda';

import \* as apigateway from 'aws-cdk-lib/aws-apigateway';

import { Duration } from 'aws-cdk-lib';

import \* as path from 'path';

export class CdkLambdaCorsStack extends cdk.Stack {

constructor(scope: cdk.App, id: string, props?: cdk.StackProps) {

super(scope, id, props);

// Lambda function

const helloLambda = new lambda.Function(this, 'HelloLambda', {

runtime: lambda.Runtime.NODEJS\_18\_X,

code: lambda.Code.fromAsset(path.join(\_\_dirname, '../lambda/hello')),

handler: 'index.handler',

});

// API Gateway

const api = new apigateway.RestApi(this, 'HelloApi', {

restApiName: 'Hello Service',

description: 'This service serves hello.',

defaultCorsPreflightOptions: {

allowOrigins: ['\*'], // Allow all origins

allowMethods: ['GET', 'POST', 'PUT', 'DELETE', 'OPTIONS'], // Allow all HTTP methods

allowHeaders: ['Content-Type', 'Authorization'], // Allow specific headers

allowCredentials: true,

},

});

// Add /hello resource

const hello = api.root.addResource('hello');

hello.addMethod('GET', new apigateway.LambdaIntegration(helloLambda));

// Additional resources can be added here

}

}

**Key Configuration Points:**

* **defaultCorsPreflightOptions**: This configures CORS for the entire API Gateway.
  + **allowOrigins**: Specifies allowed origins. Use ['\*'] to allow all origins, or list specific domains like ['https://example.com'].
  + **allowMethods**: Specifies allowed HTTP methods.
  + **allowHeaders**: Specifies allowed headers.
  + **allowCredentials**: Specifies if credentials (cookies, authorization headers, etc.) are allowed.

**Step 4: Deploy and Test**

**Deploy CDK Stack**

bash

Copy code

cdk bootstrap

cdk deploy

**Test the API**

1. **Use curl or Postman** to test the endpoint, including preflight (OPTIONS) requests and actual API calls:

bash

Copy code

# Test CORS Preflight

curl -i -X OPTIONS https://your-api-id.execute-api.your-region.amazonaws.com/prod/hello -H "Origin: https://example.com" -H "Access-Control-Request-Method: GET"

# Test GET request

curl -i https://your-api-id.execute-api.your-region.amazonaws.com/prod/hello -H "Origin: <https://example.com>"

**Project Structure**

python

Copy code

cdk-lambda-cors/

│

├── lambda/

│ └── hello/

│ └── index.ts

│

├── lib/

│ └── cdk-lambda-cors-stack.ts

│

└── bin/

└── cdk-lambda-cors.ts

**Additional Configurations**

* **Custom CORS Settings**: Customize defaultCorsPreflightOptions as needed.
* **Multiple Resources**: Apply CORS settings on individual resources if you need different configurations per endpoint.
* **Specific Origins**: Replace ['\*'] with specific domain names for more restrictive CORS settings.

**Example for More Granular Control**

If you need different CORS settings for specific resources:

**lib/cdk-lambda-cors-stack.ts**:

typescript

Copy code

const hello = api.root.addResource('hello');

hello.addMethod('GET', new apigateway.LambdaIntegration(helloLambda), {

methodResponses: [{

statusCode: '200',

responseParameters: {

'method.response.header.Access-Control-Allow-Origin': true,

},

}],

});

hello.addCorsPreflight({

allowOrigins: ['https://example.com'], // Only allow specific origin

allowMethods: ['GET', 'OPTIONS'],

allowHeaders: ['Content-Type'],

allowCredentials: true,

});

In this configuration, CORS settings are applied only to the /hello resource.

**Summary**

This guide shows how to set up CORS for your API Gateway using AWS CDK and TypeScript, allowing controlled access from different origins. Adjust the CORS configuration based on your security and application needs.

**CI\CD Code pipeline setup for AWS CDK Typescript App.**

Integrating GitHub and GitHub Actions CI/CD with AWS CDK and TypeScript for deploying infrastructure using AWS CodePipeline can streamline your development and deployment workflows. This approach leverages GitHub Actions for continuous integration and AWS CodePipeline for continuous deployment.

### Overview

1. **Set Up AWS CodePipeline with CDK**
2. **Create GitHub Actions Workflow**
3. **Deploy Infrastructure**

### Prerequisites

* An AWS account
* A GitHub repository
* AWS CLI and CDK installed
* A GitHub Personal Access Token with repo and workflow permissions

### Step 1: Set Up AWS CodePipeline with CDK

First, you need to define an AWS CodePipeline with AWS CDK to deploy your infrastructure.

**1. Create CDK Project**

bash

Copy code

mkdir cdk-pipeline

cd cdk-pipeline

cdk init app --language=typescript

**2. Define CDK Stack for CodePipeline**

Create a new stack that includes CodePipeline, CodeBuild, and integrates with GitHub.

**lib/cdk-pipeline-stack.ts**:

typescript

Copy code

import \* as cdk from 'aws-cdk-lib';

import \* as codebuild from 'aws-cdk-lib/aws-codebuild';

import \* as codepipeline from 'aws-cdk-lib/aws-codepipeline';

import \* as codepipeline\_actions from 'aws-cdk-lib/aws-codepipeline-actions';

import { Construct } from 'constructs';

import { SecretValue } from 'aws-cdk-lib';

export class CdkPipelineStack extends cdk.Stack {

constructor(scope: Construct, id: string, props?: cdk.StackProps) {

super(scope, id, props);

const sourceOutput = new codepipeline.Artifact();

const buildOutput = new codepipeline.Artifact();

const pipeline = new codepipeline.Pipeline(this, 'Pipeline', {

pipelineName: 'CDKPipeline',

restartExecutionOnUpdate: true,

});

// GitHub Source Action

const githubSource = new codepipeline\_actions.GitHubSourceAction({

actionName: 'GitHub\_Source',

owner: '<your-github-username>',

repo: '<your-repo-name>',

branch: 'main', // Replace with your branch name

oauthToken: SecretValue.secretsManager('github-token'),

output: sourceOutput,

});

// CodeBuild Project

const buildProject = new codebuild.PipelineProject(this, 'BuildProject', {

buildSpec: codebuild.BuildSpec.fromObject({

version: '0.2',

phases: {

install: {

commands: [

'npm install -g aws-cdk',

'npm install'

],

},

build: {

commands: [

'npx cdk synth',

],

},

},

artifacts: {

'base-directory': 'cdk.out',

files: '\*\*/\*',

},

}),

environment: {

buildImage: codebuild.LinuxBuildImage.STANDARD\_5\_0,

},

});

// Add source and build stages to the pipeline

pipeline.addStage({

s stageName: 'Source',

actions: [githubSource],

});

pipeline.addStage({

stageName: 'Build',

actions: [

new codepipeline\_actions.CodeBuildAction({

actionName: 'CodeBuild',

project: buildProject,

input: sourceOutput,

outputs: [buildOutput],

}),

],

});

// Deploy stage (self-mutation or application deployment)

pipeline.addStage({

stageName: 'Deploy',

actions: [

new codepipeline\_actions.CloudFormationCreateUpdateStackAction({

actionName: 'CDK\_Deploy',

templatePath: buildOutput.atPath('CdkPipelineStack.template.json'),

stackName: 'CdkPipelineStack',

adminPermissions: true,

}),

],

});

}

}

**3. Define Secrets for GitHub Token**

Store your GitHub Personal Access Token in AWS Secrets Manager:

* **AWS Secrets Manager Console**: Create a new secret.
  + Secret name: github-token
  + Secret value: <your-github-token>

**4. Bootstrap and Deploy CDK Stack**

bash

Copy code

cdk bootstrap

cdk deploy

### Step 2: Create GitHub Actions Workflow

Create a GitHub Actions workflow file to trigger the pipeline on pushes to the repository.

**1. Define GitHub Actions Workflow**

Create a workflow file in your GitHub repository:

**.github/workflows/deploy.yml**:

yaml

Copy code

name: Deploy to AWS

on:

push:

branches:

- main

jobs:

deploy:

runs-on: ubuntu-latest

steps:

- name: Checkout code

uses: actions/checkout@v3

- name: Set up Node.js

uses: actions/setup-node@v3

with:

node-version: '18.x'

- name: Install dependencies

run: npm install

- name: Set up AWS credentials

uses: aws-actions/configure-aws-credentials@v2

with:

aws-access-key-id: ${{ secrets.AWS\_ACCESS\_KEY\_ID }}

aws-secret-access-key: ${{ secrets.AWS\_SECRET\_ACCESS\_KEY }}

aws-region: us-east-1 # Adjust to your region

- name: Deploy CDK

run: npx cdk deploy --all --require-approval never

**2. Add Secrets to GitHub Repository**

* Go to your GitHub repository.
* Click on **Settings** > **Secrets and variables** > **Actions**.
* Add the following secrets:
  + **AWS\_ACCESS\_KEY\_ID**
  + **AWS\_SECRET\_ACCESS\_KEY**

### Step 3: Deploy Infrastructure

**Commit and Push Changes**

* Commit your changes to the repository and push them to GitHub:

bash

Copy code

git add .

git commit -m "Setup CDK pipeline and GitHub Actions"

git push origin main

### Summary

By following these steps, you’ve set up a CI/CD pipeline using GitHub Actions and AWS CodePipeline with AWS CDK in TypeScript. This pipeline automates the deployment of your CDK stack whenever changes are pushed to your GitHub repository, providing an efficient workflow for managing and deploying AWS infrastructure.

### Detailed Explanation

#### AWS CodePipeline CDK Stack

**lib/cdk-pipeline-stack.ts**:

* **Source Stage**: Uses GitHubSourceAction to pull code from GitHub.
* **Build Stage**: Uses PipelineProject to build the CDK application.
* **Deploy Stage**: Uses CloudFormationCreateUpdateStackAction to deploy the CDK stack.

#### GitHub Actions Workflow

**.github/workflows/deploy.yml**:

* **Checkout**: Uses actions/checkout@v3 to pull code.
* **Setup Node.js**: Uses actions/setup-node@v3 to set up Node.js environment.
* **Install Dependencies**: Installs NPM dependencies.
* **Configure AWS**: Uses aws-actions/configure-aws-credentials@v2 to set up AWS credentials.
* **Deploy CDK**: Uses npx cdk deploy to deploy the CDK application.

### Enhancements

1. **Additional Stages**: You can add more stages in AWS CodePipeline for testing, manual approval, etc.
2. **Secure Secrets**: Consider using AWS IAM roles and policies for more secure management of secrets.
3. **Notifications**: Integrate notifications for build and deploy status using Amazon SNS or other tools.

This setup provides a robust framework for deploying and managing AWS infrastructure through a modern CI/CD pipeline integrated with GitHub.

**AWS CDK S3 Pre-singed URL:**

Creating a pre-signed URL for Amazon S3 in a Node.js application using AWS CDK involves several steps:

1. **Set up an AWS CDK project**
2. **Define an S3 bucket and Lambda function in your CDK stack**
3. **Create a Lambda function to generate presigned URLs**
4. **Deploy the CDK stack**
5. **Test the Lambda function**

Here’s a step-by-step guide to achieve this:

### Step 1: Set Up an AWS CDK Project

First, initialize a new CDK project:

bash

Copy code

mkdir cdk-presigned-url

cd cdk-presigned-url

cdk init app --language=typescript

### Step 2: Define S3 Bucket and Lambda in CDK

Edit the CDK stack to define the S3 bucket and the Lambda function that will generate the presigned URL.

**lib/cdk-presigned-url-stack.ts**:

typescript

Copy code

import \* as cdk from 'aws-cdk-lib';

import \* as s3 from 'aws-cdk-lib/aws-s3';

import \* as lambda from 'aws-cdk-lib/aws-lambda';

import \* as apigateway from 'aws-cdk-lib/aws-apigateway';

import \* as path from 'path';

export class CdkPresignedUrlStack extends cdk.Stack {

constructor(scope: cdk.App, id: string, props?: cdk.StackProps) {

super(scope, id, props);

// Create S3 bucket

const bucket = new s3.Bucket(this, 'PresignedUrlBucket', {

removalPolicy: cdk.RemovalPolicy.DESTROY, // Adjust this for production use

autoDeleteObjects: true, // Adjust this for production use

});

// Create Lambda function

const presignedUrlLambda = new lambda.Function(this, 'PresignedUrlLambda', {

runtime: lambda.Runtime.NODEJS\_18\_X,

handler: 'index.handler',

code: lambda.Code.fromAsset(path.join(\_\_dirname, '../lambda/presigned-url')),

environment: {

BUCKET\_NAME: bucket.bucketName,

},

});

// Grant Lambda permissions to read from the bucket

bucket.grantReadWrite(presignedUrlLambda);

// Create API Gateway

const api = new apigateway.RestApi(this, 'PresignedUrlApi', {

restApiName: 'PresignedUrlService',

});

// Integrate Lambda with API Gateway

const getPresignedUrl = api.root.addResource('presigned-url');

getPresignedUrl.addMethod('GET', new apigateway.LambdaIntegration(presignedUrlLambda));

}

}

### Step 3: Create Lambda Function for Presigned URL

Create the Lambda function that generates the presigned URL.

**1. Create Lambda Directory**:

bash

Copy code

mkdir -p lambda/presigned-url

cd lambda/presigned-url

npm init -y

npm install aws-sdk

**2. Create Lambda Code**:

**lambda/presigned-url/index.ts**:

typescript

Copy code

import { APIGatewayProxyHandler } from 'aws-lambda';

import \* as AWS from 'aws-sdk';

const s3 = new AWS.S3();

export const handler: APIGatewayProxyHandler = async (event) => {

const bucketName = process.env.BUCKET\_NAME;

if (!bucketName) {

throw new Error('BUCKET\_NAME environment variable is not set');

}

const objectKey = event.queryStringParameters?.key || 'default-key.txt'; // File key

const expires = parseInt(event.queryStringParameters?.expires || '3600'); // URL expiration in seconds

const params = {

Bucket: bucketName,

Key: objectKey,

Expires: expires,

};

try {

const url = s3.getSignedUrl('putObject', params);

return {

statusCode: 200,

body: JSON.stringify({ url }),

};

} catch (error) {

return {

statusCode: 500,

body: JSON.stringify({ error: error.message }),

};

}

};

### Step 4: Deploy the CDK Stack

Deploy your CDK stack to create the resources:

bash

Copy code

cdk bootstrap

cdk deploy

### Step 5: Test the Lambda Function

After deployment, test the Lambda function by invoking the API Gateway endpoint to generate a presigned URL.

**1. Get the API URL**: Find the API endpoint in the CDK output.

**2. Generate Presigned URL**: Use curl or Postman to get a presigned URL.

Example using curl:

bash

Copy code

curl "https://<your-api-id>.execute-api.<your-region>.amazonaws.com/prod/presigned-url?key=test-file.txt&expires=3600"

### Project Structure

python

Copy code

cdk-presigned-url/

│

├── lambda/

│ └── presigned-url/

│ ├── index.ts

│ └── package.json

│

├── lib/

│ └── cdk-presigned-url-stack.ts

│

└── bin/

└── cdk-presigned-url.ts

### Full CDK Stack Example

Here’s the full lib/cdk-presigned-url-stack.ts for reference:

typescript

Copy code

import \* as cdk from 'aws-cdk-lib';

import \* as s3 from 'aws-cdk-lib/aws-s3';

import \* as lambda from 'aws-cdk-lib/aws-lambda';

import \* as apigateway from 'aws-cdk-lib/aws-apigateway';

import \* as path from 'path';

export class CdkPresignedUrlStack extends cdk.Stack {

constructor(scope: cdk.App, id: string, props?: cdk.StackProps) {

super(scope, id, props);

// Create S3 bucket

const bucket = new s3.Bucket(this, 'PresignedUrlBucket', {

removalPolicy: cdk.RemovalPolicy.DESTROY,

autoDeleteObjects: true,

});

// Create Lambda function

const presignedUrlLambda = new lambda.Function(this, 'PresignedUrlLambda', {

runtime: lambda.Runtime.NODEJS\_18\_X,

handler: 'index.handler',

code: lambda.Code.fromAsset(path.join(\_\_dirname, '../lambda/presigned-url')),

environment: {

BUCKET\_NAME: bucket.bucketName,

},

});

// Grant Lambda permissions to read from the bucket

bucket.grantReadWrite(presignedUrlLambda);

// Create API Gateway

const api = new apigateway.RestApi(this, 'PresignedUrlApi', {

restApiName: 'PresignedUrlService',

});

// Integrate Lambda with API Gateway

const getPresignedUrl = api.root.addResource('presigned-url');

getPresignedUrl.addMethod('GET', new apigateway.LambdaIntegration(presignedUrlLambda));

}

}

### Summary

By following this guide, you’ve set up a CDK stack that includes an S3 bucket, a Lambda function to generate presigned URLs, and an API Gateway to expose this functionality. This allows users to securely upload files to S3 using presigned URLs generated by the Lambda function.

When deploying AWS resources using the AWS Cloud Development Kit (CDK) with Node.js, managing permissions and security is essential. AWS Identity and Access Management (IAM) roles are crucial for controlling permissions for your AWS resources. Here’s a comprehensive guide on how to create and manage IAM roles in an AWS CDK application using Node.js.

### Creating IAM Roles in AWS CDK.

When deploying AWS resources using the AWS Cloud Development Kit (CDK) with Node.js, managing permissions and security is essential. AWS Identity and Access Management (IAM) roles are crucial for controlling permissions for your AWS resources. Here’s a comprehensive guide on how to create and manage IAM roles in an AWS CDK application using Node.js.

#### 1. **Basic IAM Role**

You can create a simple IAM role and attach policies to it. This role can be assumed by services like Lambda or EC2.

**Example**: Create an IAM role for an EC2 instance.

**lib/cdk-stack.ts**:

typescript

Copy code

import \* as cdk from 'aws-cdk-lib';

import \* as iam from 'aws-cdk-lib/aws-iam';

import { Construct } from 'constructs';

export class IamRoleStack extends cdk.Stack {

constructor(scope: Construct, id: string, props?: cdk.StackProps) {

super(scope, id, props);

// IAM Role

const ec2Role = new iam.Role(this, 'EC2Role', {

assumedBy: new iam.ServicePrincipal('ec2.amazonaws.com'),

description: 'Role for EC2 instance to access S3 and CloudWatch',

});

// Attach policies

ec2Role.addManagedPolicy(iam.ManagedPolicy.fromAwsManagedPolicyName('AmazonS3ReadOnlyAccess'));

ec2Role.addManagedPolicy(iam.ManagedPolicy.fromAwsManagedPolicyName('CloudWatchAgentServerPolicy'));

}

}

### 2. ****IAM Role with Inline Policies****

Sometimes you might need more granular control over permissions. You can define inline policies directly within the IAM role.

**Example**: Create an IAM role with an inline policy for a Lambda function to access a specific S3 bucket.

**lib/cdk-stack.ts**:

typescript

Copy code

import \* as cdk from 'aws-cdk-lib';

import \* as iam from 'aws-cdk-lib/aws-iam';

import \* as lambda from 'aws-cdk-lib/aws-lambda';

import { Construct } from 'constructs';

import \* as path from 'path';

export class IamRoleStack extends cdk.Stack {

constructor(scope: Construct, id: string, props?: cdk.StackProps) {

super(scope, id, props);

// IAM Role

const lambdaRole = new iam.Role(this, 'LambdaRole', {

assumedBy: new iam.ServicePrincipal('lambda.amazonaws.com'),

description: 'Role for Lambda function to access specific S3 bucket',

});

// Inline policy for specific S3 bucket

const s3BucketPolicy = new iam.Policy(this, 'S3BucketPolicy', {

statements: [

new iam.PolicyStatement({

actions: ['s3:GetObject', 's3:PutObject'],

resources: ['arn:aws:s3:::my-bucket/\*'],

}),

],

});

// Attach inline policy to the role

lambdaRole.attachInlinePolicy(s3BucketPolicy);

// Create Lambda function

new lambda.Function(this, 'MyLambdaFunction', {

runtime: lambda.Runtime.NODEJS\_18\_X,

handler: 'index.handler',

code: lambda.Code.fromAsset(path.join(\_\_dirname, '../lambda')),

role: lambdaRole,

});

}

}

### 3. ****IAM Role for Different AWS Services****

IAM roles can be created for different services like Lambda, EC2, ECS, etc., each with specific permissions required by the service.

#### **Lambda Function with IAM Role**

**Example**: Create a role for a Lambda function to read from DynamoDB and write logs to CloudWatch.

**lib/cdk-stack.ts**:

typescript

Copy code

import \* as cdk from 'aws-cdk-lib';

import \* as lambda from 'aws-cdk-lib/aws-lambda';

import \* as iam from 'aws-cdk-lib/aws-iam';

import { Construct } from 'constructs';

import \* as path from 'path';

export class LambdaRoleStack extends cdk.Stack {

constructor(scope: Construct, id: string, props?: cdk.StackProps) {

super(scope, id, props);

// IAM Role for Lambda

const lambdaRole = new iam.Role(this, 'LambdaExecutionRole', {

assumedBy: new iam.ServicePrincipal('lambda.amazonaws.com'),

managedPolicies: [

iam.ManagedPolicy.fromAwsManagedPolicyName('service-role/AWSLambdaBasicExecutionRole'),

iam.ManagedPolicy.fromAwsManagedPolicyName('AmazonDynamoDBReadOnlyAccess'),

],

});

// Lambda function

new lambda.Function(this, 'MyLambda', {

runtime: lambda.Runtime.NODEJS\_18\_X,

handler: 'index.handler',

code: lambda.Code.fromAsset(path.join(\_\_dirname, '../lambda')),

role: lambdaRole,

});

}

}

#### **EC2 Instance with IAM Role**

**Example**: Create a role for an EC2 instance to access S3 and CloudWatch.

**lib/cdk-stack.ts**:

typescript

Copy code

import \* as cdk from 'aws-cdk-lib';

import \* as ec2 from 'aws-cdk-lib/aws-ec2';

import \* as iam from 'aws-cdk-lib/aws-iam';

import { Construct } from 'constructs';

export class Ec2RoleStack extends cdk.Stack {

constructor(scope: Construct, id: string, props?: cdk.StackProps) {

super(scope, id, props);

// IAM Role for EC2

const ec2Role = new iam.Role(this, 'EC2InstanceRole', {

assumedBy: new iam.ServicePrincipal('ec2.amazonaws.com'),

managedPolicies: [

iam.ManagedPolicy.fromAwsManagedPolicyName('AmazonS3ReadOnlyAccess'),

iam.ManagedPolicy.fromAwsManagedPolicyName('CloudWatchAgentServerPolicy'),

],

});

// VPC

const vpc = new ec2.Vpc(this, 'VPC');

// EC2 Instance

new ec2.Instance(this, 'Instance', {

vpc,

instanceType: new ec2.InstanceType('t3.micro'),

machineImage: new ec2.AmazonLinuxImage(),

role: ec2Role,

});

}

}

#### **ECS Task with IAM Role**

**Example**: Create a role for an ECS task to access S3.

**lib/cdk-stack.ts**:

typescript

Copy code

import \* as cdk from 'aws-cdk-lib';

import \* as ecs from 'aws-cdk-lib/aws-ecs';

import \* as iam from 'aws-cdk-lib/aws-iam';

import { Construct } from 'constructs';

export class EcsTaskRoleStack extends cdk.Stack {

constructor(scope: Construct, id: string, props?: cdk.StackProps) {

super(scope, id, props);

// IAM Role for ECS Task

const taskRole = new iam.Role(this, 'ECSTaskRole', {

assumedBy: new iam.ServicePrincipal('ecs-tasks.amazonaws.com'),

managedPolicies: [

iam.ManagedPolicy.fromAwsManagedPolicyName('AmazonS3ReadOnlyAccess'),

],

});

// ECS Cluster

const cluster = new ecs.Cluster(this, 'Cluster', {

clusterName: 'MyCluster',

});

// ECS Task Definition

const taskDefinition = new ecs.FargateTaskDefinition(this, 'TaskDef', {

taskRole,

});

// Add container to the task definition

taskDefinition.addContainer('MyContainer', {

image: ecs.ContainerImage.fromRegistry('amazon/amazon-ecs-sample'),

memoryLimitMiB: 512,

});

// ECS Service

new ecs.FargateService(this, 'Service', {

cluster,

taskDefinition,

});

}

}

### 4. ****Best Practices for IAM Roles****

* **Principle of Least Privilege**: Always grant the minimum permissions necessary.
* **Use Managed Policies**: Use AWS managed policies where possible to simplify permission management.
* **Rotate Credentials**: Regularly rotate access keys and other credentials.
* **Monitor IAM Roles**: Use AWS CloudTrail to monitor IAM role activities.

### 5. ****Advanced IAM Role Configurations****

#### **Cross-Account Access**

To grant access to a role in a different AWS account, you can define trust relationships.

**Example**: Role that allows another AWS account to assume it.

**lib/cdk-stack.ts**:

typescript

Copy code

import \* as cdk from 'aws-cdk-lib';

import \* as iam from 'aws-cdk-lib/aws-iam';

import { Construct } from 'constructs';

export class CrossAccountRoleStack extends cdk.Stack {

constructor(scope: Construct, id: string, props?: cdk.StackProps) {

super(scope, id, props);

// IAM Role for cross-account access

new iam.Role(this, 'CrossAccountRole', {

assumedBy: new iam.AccountPrincipal('123456789012'), // Replace with the actual AWS account ID

inlinePolicies: {

allowS3Read: new iam.PolicyDocument({

statements: [

new iam.PolicyStatement({

actions: ['s3:ListBucket', 's3:GetObject'],

resources: ['arn:aws:s3:::my-bucket', 'arn:aws:s3:::my-bucket/\*'],

}),

],

}),

},

});

}

}

### Summary

By using IAM roles in AWS CDK with Node.js, you can manage and secure access to your AWS resources efficiently. This guide provided examples for creating IAM roles for Lambda, EC2, ECS, and cross-account access, along with best practices for managing these roles. These configurations help ensure that your AWS applications follow the principle of

#### 1. **Basic IAM Role**

You can create a simple IAM role and attach policies to it. This role can be assumed by services like Lambda or EC2.

**Example**: Create an IAM role for an EC2 instance.

**lib/cdk-stack.ts**:

typescript

Copy code

import \* as cdk from 'aws-cdk-lib';

import \* as iam from 'aws-cdk-lib/aws-iam';

import { Construct } from 'constructs';

export class IamRoleStack extends cdk.Stack {

constructor(scope: Construct, id: string, props?: cdk.StackProps) {

super(scope, id, props);

// IAM Role

const ec2Role = new iam.Role(this, 'EC2Role', {

assumedBy: new iam.ServicePrincipal('ec2.amazonaws.com'),

description: 'Role for EC2 instance to access S3 and CloudWatch',

});

// Attach policies

ec2Role.addManagedPolicy(iam.ManagedPolicy.fromAwsManagedPolicyName('AmazonS3ReadOnlyAccess'));

ec2Role.addManagedPolicy(iam.ManagedPolicy.fromAwsManagedPolicyName('CloudWatchAgentServerPolicy'));

}

}

### 2. ****IAM Role with Inline Policies****

Sometimes you might need more granular control over permissions. You can define inline policies directly within the IAM role.

**Example**: Create an IAM role with an inline policy for a Lambda function to access a specific S3 bucket.

**lib/cdk-stack.ts**:

typescript

Copy code

import \* as cdk from 'aws-cdk-lib';

import \* as iam from 'aws-cdk-lib/aws-iam';

import \* as lambda from 'aws-cdk-lib/aws-lambda';

import { Construct } from 'constructs';

import \* as path from 'path';

export class IamRoleStack extends cdk.Stack {

constructor(scope: Construct, id: string, props?: cdk.StackProps) {

super(scope, id, props);

// IAM Role

const lambdaRole = new iam.Role(this, 'LambdaRole', {

assumedBy: new iam.ServicePrincipal('lambda.amazonaws.com'),

description: 'Role for Lambda function to access specific S3 bucket',

});

// Inline policy for specific S3 bucket

const s3BucketPolicy = new iam.Policy(this, 'S3BucketPolicy', {

statements: [

new iam.PolicyStatement({

actions: ['s3:GetObject', 's3:PutObject'],

resources: ['arn:aws:s3:::my-bucket/\*'],

}),

],

});

// Attach inline policy to the role

lambdaRole.attachInlinePolicy(s3BucketPolicy);

// Create Lambda function

new lambda.Function(this, 'MyLambdaFunction', {

runtime: lambda.Runtime.NODEJS\_18\_X,

handler: 'index.handler',

code: lambda.Code.fromAsset(path.join(\_\_dirname, '../lambda')),

role: lambdaRole,

});

}

}

### 3. ****IAM Role for Different AWS Services****

IAM roles can be created for different services like Lambda, EC2, ECS, etc., each with specific permissions required by the service.

#### **Lambda Function with IAM Role**

**Example**: Create a role for a Lambda function to read from DynamoDB and write logs to CloudWatch.

**lib/cdk-stack.ts**:

typescript

Copy code

import \* as cdk from 'aws-cdk-lib';

import \* as lambda from 'aws-cdk-lib/aws-lambda';

import \* as iam from 'aws-cdk-lib/aws-iam';

import { Construct } from 'constructs';

import \* as path from 'path';

export class LambdaRoleStack extends cdk.Stack {

constructor(scope: Construct, id: string, props?: cdk.StackProps) {

super(scope, id, props);

// IAM Role for Lambda

const lambdaRole = new iam.Role(this, 'LambdaExecutionRole', {

assumedBy: new iam.ServicePrincipal('lambda.amazonaws.com'),

managedPolicies: [

iam.ManagedPolicy.fromAwsManagedPolicyName('service-role/AWSLambdaBasicExecutionRole'),

iam.ManagedPolicy.fromAwsManagedPolicyName('AmazonDynamoDBReadOnlyAccess'),

],

});

// Lambda function

new lambda.Function(this, 'MyLambda', {

runtime: lambda.Runtime.NODEJS\_18\_X,

handler: 'index.handler',

code: lambda.Code.fromAsset(path.join(\_\_dirname, '../lambda')),

role: lambdaRole,

});

}

}

#### **EC2 Instance with IAM Role**

**Example**: Create a role for an EC2 instance to access S3 and CloudWatch.

**lib/cdk-stack.ts**:

typescript

Copy code

import \* as cdk from 'aws-cdk-lib';

import \* as ec2 from 'aws-cdk-lib/aws-ec2';

import \* as iam from 'aws-cdk-lib/aws-iam';

import { Construct } from 'constructs';

export class Ec2RoleStack extends cdk.Stack {

constructor(scope: Construct, id: string, props?: cdk.StackProps) {

super(scope, id, props);

// IAM Role for EC2

const ec2Role = new iam.Role(this, 'EC2InstanceRole', {

assumedBy: new iam.ServicePrincipal('ec2.amazonaws.com'),

managedPolicies: [

iam.ManagedPolicy.fromAwsManagedPolicyName('AmazonS3ReadOnlyAccess'),

iam.ManagedPolicy.fromAwsManagedPolicyName('CloudWatchAgentServerPolicy'),

],

});

// VPC

const vpc = new ec2.Vpc(this, 'VPC');

// EC2 Instance

new ec2.Instance(this, 'Instance', {

vpc,

instanceType: new ec2.InstanceType('t3.micro'),

machineImage: new ec2.AmazonLinuxImage(),

role: ec2Role,

});

}

}

#### **ECS Task with IAM Role**

**Example**: Create a role for an ECS task to access S3.

**lib/cdk-stack.ts**:

typescript

Copy code

import \* as cdk from 'aws-cdk-lib';

import \* as ecs from 'aws-cdk-lib/aws-ecs';

import \* as iam from 'aws-cdk-lib/aws-iam';

import { Construct } from 'constructs';

export class EcsTaskRoleStack extends cdk.Stack {

constructor(scope: Construct, id: string, props?: cdk.StackProps) {

super(scope, id, props);

// IAM Role for ECS Task

const taskRole = new iam.Role(this, 'ECSTaskRole', {

assumedBy: new iam.ServicePrincipal('ecs-tasks.amazonaws.com'),

managedPolicies: [

iam.ManagedPolicy.fromAwsManagedPolicyName('AmazonS3ReadOnlyAccess'),

],

});

// ECS Cluster

const cluster = new ecs.Cluster(this, 'Cluster', {

clusterName: 'MyCluster',

});

// ECS Task Definition

const taskDefinition = new ecs.FargateTaskDefinition(this, 'TaskDef', {

taskRole,

});

// Add container to the task definition

taskDefinition.addContainer('MyContainer', {

image: ecs.ContainerImage.fromRegistry('amazon/amazon-ecs-sample'),

memoryLimitMiB: 512,

});

// ECS Service

new ecs.FargateService(this, 'Service', {

cluster,

taskDefinition,

});

}

}

### 4. ****Best Practices for IAM Roles****

* **Principle of Least Privilege**: Always grant the minimum permissions necessary.
* **Use Managed Policies**: Use AWS managed policies where possible to simplify permission management.
* **Rotate Credentials**: Regularly rotate access keys and other credentials.
* **Monitor IAM Roles**: Use AWS CloudTrail to monitor IAM role activities.

### 5. ****Advanced IAM Role Configurations****

#### **Cross-Account Access**

To grant access to a role in a different AWS account, you can define trust relationships.

**Example**: Role that allows another AWS account to assume it.

**lib/cdk-stack.ts**:

typescript

Copy code

import \* as cdk from 'aws-cdk-lib';

import \* as iam from 'aws-cdk-lib/aws-iam';

import { Construct } from 'constructs';

export class CrossAccountRoleStack extends cdk.Stack {

constructor(scope: Construct, id: string, props?: cdk.StackProps) {

super(scope, id, props);

// IAM Role for cross-account access

new iam.Role(this, 'CrossAccountRole', {

assumedBy: new iam.AccountPrincipal('123456789012'), // Replace with the actual AWS account ID

inlinePolicies: {

allowS3Read: new iam.PolicyDocument({

statements: [

new iam.PolicyStatement({

actions: ['s3:ListBucket', 's3:GetObject'],

resources: ['arn:aws:s3:::my-bucket', 'arn:aws:s3:::my-bucket/\*'],

}),

],

}),

},

});

}

}

### Summary

By using IAM roles in AWS CDK with Node.js, you can manage and secure access to your AWS resources efficiently. This guide provided examples for creating IAM roles for Lambda, EC2, ECS, and cross-account access, along with best practices for managing these roles. These configurations help ensure that your AWS applications follow the principle of