Activity3 — Building a Serverless Application (Hand-held) 🛠️

Objective 🎯

In this activity, you will **build a complete serverless application step by step** using the Serverless Framework v4 on AWS.

By the end of this activity, you should be able to:

- Initialize a Serverless project with Python runtime.
- Create and deploy Lambda functions triggered by API Gateway.
- Define and use DynamoDB, SQS, SNS, and CloudWatch with Lambda.
- Attach IAM roles and permissions properly.
- Inspect the CloudFormation stack generated by Serverless.
- Clean up AWS resources safely.

Prerequisites 📝

- Must have completed **Activity1 (Environment Setup)** and **Activity2 (Core Concepts)**.
- AWS CLI configured:

Shell
```
aws configure --profile serverless-lab
```

- Logged in to Serverless Dashboard:

Shell
```
serverless login
```

⚠️ **Note:** AWS CLI and Serverless Dashboard configuration are a **one-time job per activity**. You must do it once at the start of each new activity.

Expected Folder Structure

Your project should look like this as we build it step by step:

None

```
activity3-app/
├── handlers/
|   ├── hello.py
|   ├── create_todo.py
|   ├── publish_message.py
|   └── consume_message.py
├── requirements.txt   # (optional, if using dependencies like boto3, requests
etc.)
├── serverless.yml
└── .gitignore
```

Lab Setup ⚙️

1. Initialize a new service:

Shell

```
sls
```

- Choose template: **AWS - Python - Simple Function**
- Name project: **activity3-app**
- Choose Create new app: **serverless-app**
- **Skip & Set Later (AWS SSO, ENV Vars)**

2. Move into the project folder:

Shell

```
cd activity3-app
ls
```

You should see `serverless.yml` and a sample `handler.py`.

Step 1 — Initialize project and configure provider

Edit `serverless.yml` and set up:

None

```
service: activity3-app         # Name of the service (used in stack/resource naming)

provider:
  name: aws                   # Cloud provider (AWS in this case)
  runtime: python3.10         # Lambda runtime environment
  region: ap-south-1          # Default AWS region for deployment
  stage: ${opt:stage, 'dev'}  # Deployment stage (defaults to 'dev' if not passed via --stage)
  profile: serverless-lab     # AWS CLI profile to use for credentials
```

This ensures we are always deploying to `ap-south-1` using our lab IAM profile.

Here's a more descriptive write-up you can use for your **Step 2 — IAM Role setup** section. I've kept the same YAML, but added explanation so it's clear what each block is doing and why:

Step 2 — IAM Role setup

In AWS, every Lambda function needs an **execution role** (an IAM role that the function assumes at runtime). This role defines what actions the function is allowed to perform on AWS resources. By customizing the IAM role in `serverless.yml`, we can enforce **least-privilege access** — each function only gets the exact permissions it needs, nothing more.

Add the following under your `provider:` block:

None

```
provider:
  iam:
    role:
      statements:
        # ✅ CloudWatch Logs permissions
        - Effect: Allow
          Action:
            - logs:CreateLogGroup
            - logs:CreateLogStream
            - logs:PutLogEvents
          Resource:
"arn:aws:logs:${self:provider.region}:*:log-group:/aws/lambda/${self:service}-$
{self:provider.stage}-*:*"

        # ✅ DynamoDB access (for TodoTable operations)
        - Effect: Allow
          Action:
            - dynamodb:PutItem
            - dynamodb:GetItem
            - dynamodb:Scan
          Resource:
            - arn:aws:dynamodb:ap-south-1:*:table/TodoTable

        # ✅ SQS access (send & receive messages)
        - Effect: Allow
          Action:
            - sqs:SendMessage
            - sqs:ReceiveMessage
          Resource: "*"

        # ✅ SNS publish access (so Lambda can push to our topic)
        - Effect: Allow
          Action:
            - sns:Publish
          Resource:
            - Ref: MyTopic
```

🔎 Breakdown

- **CloudWatch Logs** Every Lambda writes logs by default. Without these permissions, your functions won't be able to create log groups/streams or send log events. This block scopes access tightly to only the Lambda log groups of your service and stage.

- **DynamoDB** Only allows basic read/write operations (`PutItem`, `GetItem`, `Scan`) on the specific `TodoTable`. This prevents accidental access to other tables.
- **SQS** Allows sending and receiving messages. The resource is set to `"*"` here, but ideally, you'd restrict this to the ARN of your own queue (e.g., `!GetAtt MyQueue.Arn`).
- **SNS** Grants `sns:Publish` specifically to the topic you created (`MyTopic`). Using `Ref: MyTopic` ensures CloudFormation injects the correct ARN at deploy time.

✅ Why this matters

- Without these, you'll see runtime errors like `AccessDenied` when your function tries to log, insert into DynamoDB, or publish to SNS.
- By scoping each permission to just what's required (least privilege), you reduce risk if your function is compromised.
- Defining IAM in `serverless.yml` keeps everything **as code** — easy to review and audit.

Step 3 — First Lambda function (Hello World)

Create `handlers/hello.py`:

```Python
def handler(event, context):
    return {
        "statusCode": 200,
        "body": "Hello from Activity3 app!"
    }
```

Add it in `serverless.yml`:

```
None
functions:
  hello:                                # Logical name of the function
    handler: handlers/hello.handler     # File + function to execute
(handlers/hello.py → handler())
    events:                             # Triggers for this Lambda
      - httpApi:                        # HTTP API Gateway event
          path: /hello                  # Endpoint path
          method: get                   # HTTP method
```

Deploy:

```
Shell
sls deploy --stage dev
```

Test endpoint:

```
Shell
curl https://<api-id>.execute-api.ap-south-1.amazonaws.com/hello
```

After Step 3 — Verify Deployment & Check Resources

Before moving to DynamoDB integration, let’s pause and **verify that everything created so far (till Step 3)** is working correctly.

1) If Deployment Fails — How to Debug

If `sls deploy` did not finish successfully or curl fails:

1. Run the logs command for your function:

```
Shell
sls logs -f hello --stage dev --tail
```

- This will stream the CloudWatch logs for your `hello` function.

  - Check for Python errors, missing handler issues, or permission problems.

2. If resources failed to create, go to **AWS Console** → **CloudFormation** → **Stacks** → **activity3-app-dev** → **Events**.
   - Look for **CREATE_FAILED** messages.
   - The error message will indicate what went wrong (e.g., IAM permission denied, bucket already exists).

Only proceed once your stack shows status `CREATE_COMPLETE`.

2) Open AWS Console & Set Region

1. Go to [AWS Console](https://console.aws.amazon.com).
2. In the top-right region selector, choose **Asia Pacific (Mumbai) — ap-south-1**.

3) Inspect CloudFormation Stack

1. Services → **CloudFormation** → **Stacks**.
2. Find the stack **`activity3-app-dev`**.
3. Confirm status is **CREATE_COMPLETE**.
4. Click on it → **Resources** tab. You should see:
   - `AWS::Lambda::Function` → `hello` function created.
   - `AWS::ApiGatewayV2::Api` → the HTTP API created.
   - `AWS::ApiGatewayV2::Stage` → stage (`dev`) created.
   - `AWS::IAM::Role` → execution role for the function.
   - `AWS::LogGroup` → CloudWatch log group for the function.

4) Inspect Lambda Function

1. Services → **Lambda** → **Functions**.
2. Verify a function named like `activity3-app-dev-hello` exists.
3. Check:
   - Runtime = `python3.10`
   - Handler = `handlers/hello.handler`
   - Permissions tab → linked IAM role

5) Inspect API Gateway (HTTP API)

1. Services → **API Gateway** → **HTTP APIs**.

2. Locate the API with name like `activity3-app-dev`.
3. Verify route:
   - `GET /hello` is present.
4. Copy **Invoke URL** and test with curl:

Shell
```
curl https://<api-id>.execute-api.ap-south-1.amazonaws.com/hello
```

6) Inspect CloudWatch Logs

1. Services → **CloudWatch** → **Logs** → **Log groups**.
2. Find log group `/aws/lambda/activity3-app-dev-hello`.
3. Check the latest log stream for request logs (after your curl test).

Step 4 — Add DynamoDB integration

1. Add DynamoDB table under `resources`:

None
```
resources:
  Resources:
    TodoTable:                          # Logical resource name
      Type: AWS::DynamoDB::Table        # Resource type (DynamoDB table)
      Properties:
        TableName: TodoTable            # Actual table name in AWS
        BillingMode: PAY_PER_REQUEST    # On-demand billing (no capacity planning)
        AttributeDefinitions:           # Define attributes and their types
          - AttributeName: id
            AttributeType: S            # 'S' = String
        KeySchema:                      # Define primary key
          - AttributeName: id
            KeyType: HASH               # Partition key (no sort key here)
```

2. Create function `handlers/create_todo.py`:

Python

```python
import json, uuid, boto3

dynamodb = boto3.resource("dynamodb")
table = dynamodb.Table("TodoTable")

def handler(event, context):
    body = json.loads(event["body"])
    item = {"id": str(uuid.uuid4()), "task": body["task"]}
    table.put_item(Item=item)
    return {"statusCode": 200, "body": json.dumps(item)}
```

3. Add function in `serverless.yml`:

None

```
functions:
  createTodo:
    handler: handlers/create_todo.handler
    events:
      - httpApi:
          path: /todos
          method: post
```

Deploy and test:

Shell

```shell
curl -X POST https://<api-id>.execute-api.ap-south-1.amazonaws.com/todos \
  -H "Content-Type: application/json" \
  -d '{"task": "Finish Activity3"}'
```

Check **DynamoDB** table in AWS Console → **Explore Items**.

Step 5 — SQS & SNS:

Before wiring services, first understand what they are and how they work.

Conceptual overview

- **SNS (Simple Notification Service)**
  - Pub/Sub (publish–subscribe) messaging service.
  - A *publisher* sends a message to an **SNS topic**.
  - The topic *fans out* the message to all **subscribers** (HTTP endpoint, Lambda, SQS queue, email, SMS).
  - Use case: broadcast an event to multiple consumers.

- **SQS (Simple Queue Service)**
  - Durable message queue.
  - Producers send messages to a **queue**. Consumers poll the queue and process messages.
  - Guarantees at-least-once delivery (so make consumers idempotent).
  - Use case: decouple processing, buffer bursts, retry/backoff.

- **Common pattern (SNS → SQS → Lambda)**
  1. API or producer publishes to **SNS topic**.
  2. SNS forwards message to an **SQS queue** (subscription).
  3. A Lambda function is triggered by messages arriving in the SQS queue and processes them (scales independently).

  - This pattern provides fanout + durable, retryable processing.

Integration — what we will create

- An **SNS topic** (`MyTopic`).
- An **SQS queue** (`MyQueue`).
- A CloudFormation **Subscription** that subscribes `MyQueue` to `MyTopic` (**SQS <- SNS**).
- A **publishMessage** HTTP endpoint which publishes to the SNS topic.
- A **consumeMessage** Lambda that is triggered by messages arriving in the SQS queue.

1) Add resources (serverless `resources:`)

Add these to `serverless.yml` under `resources: Resources:` (we include the subscription so SNS forwards to SQS and we expose the TopicArn via Outputs):

None

```
Resources:
  MyQueue:
    Type: AWS::SQS::Queue
    Properties:
      QueueName: my-queue

  MyTopic:
    Type: AWS::SNS::Topic
    Properties:
      TopicName: my-topic

  MyTopicSubscription:
    Type: AWS::SNS::Subscription
    Properties:
      Protocol: sqs
      TopicArn:
        Ref: MyTopic
      Endpoint:
        Fn::GetAtt:
          - MyQueue
          - Arn
      # Allow SNS to send messages to the queue (policy attached to the queue)
  MyQueuePolicy:
    Type: AWS::SQS::QueuePolicy
    Properties:
      Queues:
        - Ref: MyQueue
      PolicyDocument:
        Version: "2012-10-17"
        Statement:
          - Sid: Allow-SNS-SendMessage
            Effect: Allow
            Principal: "*"
            Action: "sqs:SendMessage"
            Resource:
              Fn::GetAtt:
                - MyQueue
                - Arn
            Condition:
              ArnEquals:
                "aws:SourceArn":
                  Ref: MyTopic
```

Why include `MyQueuePolicy`? SNS must be allowed to send messages to the SQS queue; the queue policy by default restricts senders to send messages to its queue.

Broken into sub-parts and explained in detail:

A — `MyQueue` (SQS queue)

- **Purpose:** durable queue that buffers messages for asynchronous processing.
- **CFN Type:** `AWS::SQS::Queue`.
- **Properties explained:**
  - `QueueName`: friendly name. If omitted, CloudFormation generates a unique name (often recommended to avoid collisions).
- **Best-practices / enhancements to add in real projects:**
  - `VisibilityTimeout` — how long a message is hidden while being processed. Tune based on expected processing time.
  - `MessageRetentionPeriod` — how long messages persist (default 4 days). Lower to save cost if appropriate.
  - `ReceiveMessageWaitTimeSeconds` — long polling (recommended >0 to reduce empty receive calls).
  - `RedrivePolicy` — configure a Dead-Letter Queue (DLQ) to handle poison messages. Example:

None

```
RedrivePolicy:
  deadLetterTargetArn: !GetAtt MyDeadLetterQueue.Arn
  maxReceiveCount: 5
```

  - `KmsMasterKeyId` or `SqsManagedSseEnabled` — enable encryption at rest if needed.

- **Verification:** after deploy, check SQS console → queue attributes, note Queue URL and ARN.
- **Pitfalls:** hardcoding `QueueName` may cause create failures if name already exists in account/region. Consider using `${self:service}-${self:provider.stage}-my-queue` pattern.

B — `MyTopic` (**SNS topic**)

- **Purpose:** pub/sub topic for broadcasting messages to one or many subscribers (HTTP, Lambda, SQS, email, SMS).
- **CFN Type:** `AWS::SNS::Topic`.
- **Properties explained:**
  - `TopicName`: friendly name for the topic. Like SQS, leaving it out lets CloudFormation generate a safe unique name.
- **Best-practices / enhancements:**
  - `DisplayName` for SMS-friendly name if sending SMS.
  - `Subscription` block can be declared inline here instead of separate `AWS::SNS::Subscription`.
  - Enable server-side encryption (use KMS) if messages contain sensitive data.
- **Verification:** Console → SNS → Topics → confirm Topic ARN.
- **Pitfalls:** topic names may collide across account/region if hardcoded — prefer stage-scoped names.

C — `MyTopicSubscription` (**SNS → SQS subscription wiring**)

- **Purpose:** connects the SNS topic to the SQS queue so that messages published to the topic are delivered to the queue. This enables durable fanout.
- **CFN Type:** `AWS::SNS::Subscription`.
- **Properties explained:**
  - `Protocol: sqs` — subscription protocol type (other types: `lambda`, `http`, `https`, `email`, etc.).
  - `TopicArn: Ref: MyTopic` — `Ref` returns the topic's logical reference (Topic ARN for `AWS::SNS::Topic`).
  - `Endpoint: Fn::GetAtt: [MyQueue, Arn]` — `Fn::GetAtt` gets the queue ARN (SQS requires ARN as endpoint for `sqs` protocol).

- **Behavior:** when subscription is created, SNS will attempt to deliver messages to the SQS endpoint. For SQS endpoints, no confirmation handshake is required (unlike HTTP/email).
- **Verification:** CloudFormation Resources should show `MyTopicSubscription`. In SNS console → Subscriptions you should see a subscription with protocol `sqs` and endpoint equal to SQS queue ARN.
- **Pitfalls:** subscription alone is not enough — SQS must allow SNS to send messages (see `MyQueuePolicy`); otherwise deliveries will be blocked.

D — `MyQueuePolicy` (SQS resource policy allowing SNS to send)

- **Purpose:** allow the SNS topic principal to call `sqs:SendMessage` on the SQS queue. Without this, SNS cannot deliver messages to SQS even if subscription exists.
- **CFN Type:** `AWS::SQS::QueuePolicy`.
- **Properties explained:**
  - `Queues: - Ref: MyQueue` — targets the physical queue(s) this policy applies to (you can attach same policy to multiple queues).
  - `PolicyDocument` — standard IAM policy document applied to the queue resource:
    - `Version`: policy version.
    - `Statement`: list of permission statements.
      - `Sid`: statement id (optional label).
      - `Effect: Allow` — allow the action.
      - `Principal: "*"` — allows any principal, but the `Condition` below restricts who can actually send.
      - `Action: sqs:SendMessage` — only allow sending messages.
      - `Resource: Fn::GetAtt: [MyQueue, Arn]` — the ARN of the queue.
      - `Condition: ArnEquals: "aws:SourceArn": Ref: MyTopic` — restrict allowed sender to messages whose `SourceArn` equals this specific SNS Topic ARN.

- **Why this pattern:** SNS publishes to SQS on behalf of SNS's service principal; the queue policy must explicitly permit that service principal when `aws:SourceArn` equals the topic ARN — this prevents other SNS topics or principals from sending messages to your queue.

- **Verification:** After deploy, in SQS console → Queue → Access Policy you should see a JSON policy that includes the `Allow-SNS-SendMessage` statement with the correct `aws:SourceArn`.
- **Pitfalls & security notes:**
  - Do **not** leave overly-broad policies (e.g., allow all `sqs:SendMessage` from `Principal: "*"` without `Condition`), as this can let other accounts or services inject messages.
  - Ensure `Condition` matches the exact `Ref: MyTopic` (it resolves to the topic ARN).
  - If you have cross-account SNS topics, you may need a different policy allowing specific AWS account principals.

E — How these pieces interact (flow)

1. **Producer** publishes to `MyTopic` (SNS).
2. **SNS** looks at its subscriptions: it has a subscription where `Protocol=sqs` and `Endpoint=MyQueue.Arn`.
3. **SNS** delivers the message to the SQS queue by calling `sqs:SendMessage`.
4. **SQS** receives the message and stores it until a consumer (Lambda triggered by SQS) retrieves it.
5. **Queue policy** ensures only the configured SNS Topic (and no other source) can send messages.

F — Additional recommended production considerations

- **Dead-Letter Queue (DLQ):** define a separate SQS queue as DLQ and attach via `RedrivePolicy` to handle poison messages.
- **Encryption:** enable KMS for both SNS and SQS if sensitive data is involved.
- **FIFO queues:** if ordering is important use `FifoQueue: true` and topic-to-FIFO subscription rules (requires `.fifo` naming and `MessageGroupId`).
- **Monitoring:** enable CloudWatch alarms for SQS `ApproximateAgeOfOldestMessage` and SNS delivery failures.
- **Name collisions:** use stage/service prefix to avoid collisions: `!Sub "${AWS::StackName}-my-queue"`.

G — Quick troubleshooting checklist if messages don't arrive

- Confirm `MyTopicSubscription` exists in CloudFormation Resources.
- Check SQS queue policy for `Allow-SNS-SendMessage` with `aws:SourceArn` equal to topic ARN.
- In SNS console, view topic metrics for DeliveryFailures.
- Check CloudWatch logs for Lambda consumer errors.
- Use `aws sns publish` CLI to send test message and `aws sqs receive-message` to poll queue manually.

2) Publish function — `handlers/publish_message.py`

Use the Topic ARN from CloudFormation outputs or resolve via environment variable. Prefer injecting the Topic ARN as an environment variable in `serverless.yml` (shown below).

`handlers/publish_message.py`:

```python
import json, boto3, os

sns = boto3.client("sns")
TOPIC_ARN = os.environ.get("MY_TOPIC_ARN")  # injected by serverless.yml

def handler(event, context):
    body = json.loads(event.get("body", "{}"))
    message = body.get("message", "hello from publishMessage")
    resp = sns.publish(TopicArn=TOPIC_ARN, Message=message)
    return {"statusCode": 200, "body": json.dumps({"MessageId":
resp.get("MessageId")})}
```

3) Consumer function — `handlers/consume_message.py`

This function receives SQS event records (Lambda event format for SQS):

```python
import json
```

```
def handler(event, context):
    for record in event["Records"]:
        # record['body'] contains the message published to SNS (string)
        print("Received SQS message:", record["body"])
        # TODO: parse/process the message and apply idempotency if needed
    return {"statusCode": 200, "body": "Processed messages"}
```

Note: Processing must be idempotent because SQS + Lambda can deliver messages more than once.

4) Update `functions:` in `serverless.yml` (wiring + env var)

Add both functions, inject the Topic ARN into the publisher, and wire the SQS event for the consumer:

```
None
functions:
  publishMessage:
    handler: handlers/publish_message.handler
    environment:
      MY_TOPIC_ARN:
        Ref: MyTopic
    events:
      - httpApi:
          path: /publish
          method: post

  consumeMessage:
    handler: handlers/consume_message.handler
    events:
      - sqs:
          arn:
            Fn::GetAtt: [MyQueue, Arn]
          batchSize: 5       # process up to 5 messages per Lambda invocation
          maximumBatchingWindow: 30   # seconds; optional
          enabled: true
```

5) Outputs (optional — expose TopicArn and QueueArn)

Add to `outputs:` under `resources:` so you can easily see ARNs with `sls info --stage dev --verbose`:

None

```
Outputs:
    MyTopicArn:
      Description: "SNS Topic ARN"
      Value: !Ref MyTopic
      # Optional: export it for cross-stack usage
      Export:
        Name: ${self:service}-${self:provider.stage}-MyTopicArn

    MyQueueArn:
      Description: "SQS Queue ARN"
      Value: !GetAtt MyQueue.Arn
      Export:
        Name: ${self:service}-${self:provider.stage}-MyQueueArn
```

6) Deploy

Deploy the updated service (packs, uploads, CloudFormation):

Shell

```
sls deploy --stage dev
```

Watch the output for created resources — confirm `MyTopic`, `MyQueue`, the subscription, and functions.

7) Test the flow (end-to-end)

1. **Publish via HTTP endpoint**:

Shell

```
curl -X POST https://<api-id>.execute-api.ap-south-1.amazonaws.com/publish \
  -H "Content-Type: application/json" \
  -d '{"message": "hello world"}'
```

Response should contain `MessageId`.

2. **Verify SQS received message**:

    - Console: SQS → my-queue → Monitoring / Messages available (may be zero if Lambda immediately consumed).
    - Or poll the queue manually (CLI) to peek:

```Shell
aws sqs receive-message --queue-url $(aws sqs get-queue-url --queue-name
my-queue --query QueueUrl --output text) --max-number-of-messages 1
--visibility-timeout 0
```

    - If Lambda consumes, check **CloudWatch Logs** for `consumeMessage` to see **printed messages**:

```Shell
sls logs -f consumeMessage --stage dev --tail
```

3. **If consumer didn't process** (messages remain), ensure:

    - `consumeMessage` Lambda exists and its event mapping to SQS is present.
    - IAM role has necessary `sqs:ReceiveMessage`/`sqs:DeleteMessage` permissions (we set Send/Receive earlier; refine as needed).

8) Troubleshooting tips

- **Subscription not visible**: check `MyTopicSubscription` in CloudFormation Resources and SQS queue policy.

- **Messages not delivered**: confirm `MyQueuePolicy` allows `sqs:SendMessage` from the SNS Topic ARN.
- **Lambda errors**: stream logs with `sls logs -f consumeMessage --tail` and inspect stack Events in CloudFormation for deployment failures.
- **Idempotency**: design `consumeMessage` to detect and ignore duplicate processing (store processed IDs in DynamoDB if needed).

9) Clean up (if you want to remove the entire stack)

Shell

```
sls remove --stage dev
```

Final serverless.yml (Full Stack)

None

```
# "org" ensures this Service is used with the correct Serverless Framework Access Key.
org: sammagnet7
# "app" enables Serverless Framework Dashboard features and sharing them with other Services.
app: serverless-app
# "service" is the name of this project. This will also be added to your AWS resource names.
service: activity3-app

provider:
  name: aws
  runtime: python3.10
  region: ap-south-1
  stage: ${opt:stage, 'dev'}
  profile: serverless-lab
  iam:
    role:
      statements:
        - Effect: Allow
          Action:
            - logs:CreateLogGroup
            - logs:CreateLogStream
            - logs:PutLogEvents
          Resource:
"arn:aws:logs:${self:provider.region}:*:log-group:/aws/lambda/${self:service}-$
{self:provider.stage}-*:*"
        - Effect: Allow
          Action:
            - dynamodb:PutItem
            - dynamodb:GetItem
            - dynamodb:Scan
          Resource:
            - arn:aws:dynamodb:ap-south-1:*:table/TodoTable
        - Effect: Allow
          Action:
            - sqs:SendMessage
            - sqs:ReceiveMessage
          Resource: "*"
        # <-- ADD THIS BLOCK FOR SNS-->
        - Effect: Allow
          Action:
            - sns:Publish
          Resource:
            - Ref: MyTopic

functions:
  hello:
    handler: handlers/hello.handler
    events:
      - httpApi:
          path: /hello
          method: get
  createTodo:
    handler: handlers/create_todo.handler
    events:
      - httpApi:
          path: /todos
          method: post
  publishMessage:
    handler: handlers/publish_message.handler
    environment:
      MY_TOPIC_ARN:
        Ref: MyTopic
    events:
      - httpApi:
          path: /publish
          method: post
  consumeMessage:
    handler: handlers/consume_message.handler
    events:
      - sqs:
          arn:
            Fn::GetAtt: [MyQueue, Arn]
          batchSize: 5       # process up to 5 messages per Lambda invocation
          maximumBatchingWindow: 30   # seconds; optional
          enabled: true

resources:
  Resources:
    TodoTable:
      Type: AWS::DynamoDB::Table
      Properties:
        TableName: TodoTable
        BillingMode: PAY_PER_REQUEST
        AttributeDefinitions:
          - AttributeName: id
            AttributeType: S
        KeySchema:
          - AttributeName: id
            KeyType: HASH
    MyQueue:
      Type: AWS::SQS::Queue
      Properties:
        QueueName: my-queue

    MyTopic:
      Type: AWS::SNS::Topic
      Properties:
        TopicName: my-topic

    MyTopicSubscription:
      Type: AWS::SNS::Subscription
      Properties:
        Protocol: sqs
        TopicArn:
          Ref: MyTopic
        Endpoint:
          Fn::GetAtt:
            - MyQueue
            - Arn
        # Allow SNS to send messages to the queue (policy attached to the
queue)
    MyQueuePolicy:
      Type: AWS::SQS::QueuePolicy
      Properties:
        Queues:
          - Ref: MyQueue
        PolicyDocument:
          Version: "2012-10-17"
          Statement:
            - Sid: Allow-SNS-SendMessage
              Effect: Allow
              Principal: "*"
              Action: "sqs:SendMessage"
              Resource:
                Fn::GetAtt:
                  - MyQueue
                  - Arn
              Condition:
                ArnEquals:
                  "aws:SourceArn":
                    Ref: MyTopic
  Outputs:
    MyTopicArn:
      Description: "SNS Topic ARN"
      Value: !Ref MyTopic
      # Optional: export it for cross-stack usage
      Export:
        Name: ${self:service}-${self:provider.stage}-MyTopicArn

    MyQueueArn:
      Description: "SQS Queue ARN"
      Value: !GetAtt MyQueue.Arn
      Export:
        Name: ${self:service}-${self:provider.stage}-MyQueueArn
```

Verification Checklist ✅

- Hello endpoint working via API Gateway.
- DynamoDB table created, items inserted via POST.

- SNS → SQS → Lambda flow works.
- Logs visible in CloudWatch.
- CloudFormation stack shows correct resources.

Notes & Cautions ⚠️

- Always clean up (`sls remove`) to avoid charges.
- IAM roles must follow least privilege.
- Keep region consistent (`ap-south-1`).
- Ensure you update `TopicArn` correctly in `publish_message.py`.

Final Note 📌

This activity is not evaluated.

Students must perform all hands-on steps for self-learning. Do attempt the quiz.

In the next activity, you will be asked to create a similar stack from scratch — and that will be evaluated.

═══ END OF DOCUMENT ═══