## ****Level 1: Beginner****

### ****Step 1: Create Your First REST API****

1. **Navigate to API Gateway**:
   * Go to the **AWS Management Console**.
   * Search for **API Gateway** and open the service.
2. **Create a New REST API**:
   * Click **Create API** > Select **REST API** > Click **Build**.
   * Provide an API name (e.g., HelloWorldAPI) and description, and click **Create API**.
3. **Create a Resource**:
   * In the left menu, click **Resources** > Click **Actions** > **Create Resource**.
   * Provide a name (e.g., hello) and click **Create Resource**.
4. **Add a GET Method**:
   * Select the /hello resource > Click **Actions** > **Create Method** > Choose **GET** and click the checkmark.
   * Select **Mock Integration** as the integration type.
   * Configure the response to return a static message (e.g., {"message": "Hello, World!"}).
   * Save and deploy the method.
5. **Deploy the API**:
   * Click **Actions** > **Deploy API**.
   * Create a new stage (e.g., dev) and deploy.
   * Copy the **Invoke URL** and test it in your browser or Postman.

### ****Step 2: Enable CORS****

1. **Why CORS?**
   * Cross-Origin Resource Sharing (CORS) allows browsers to access resources from a different domain. It's crucial for APIs accessed from web applications.
2. **Enable CORS**:
   * Go to the /hello resource > Select the **GET** method.
   * Click **Actions** > **Enable CORS**.
   * Select the default headers and click **Enable CORS and Replace Existing CORS Headers**.
3. **Test CORS**:
   * Test the API using a browser-based tool to ensure the CORS headers are applied.

## ****Level 2: Intermediate****

### ****Step 3: Integrate with AWS Lambda****

1. **Create a Lambda Function**:
   * Go to the **AWS Lambda Console**.
   * Click **Create Function** > Select **Author from Scratch**.
   * Name the function (e.g., HelloWorldFunction) and use the default settings.
   * Add the following code:

def lambda\_handler(event, context):

name = event['queryStringParameters'].get('name', 'World')

return {

'statusCode': 200,

'body': f"Hello, {name}!"

}

* + Deploy the function.

1. **Integrate API Gateway with Lambda**:
   * In API Gateway, select the /hello resource > **GET** method.
   * Change the **Integration Type** to **Lambda Function**.
   * Select your Lambda function and save.
2. **Test Lambda Integration**:
   * Pass a query string (e.g., ?name=Abhishek) and test the response.

### ****Step 4: Configure API Keys****

1. **Create an API Key**:
   * In API Gateway, go to **API Keys** > Click **Create API Key**.
   * Name the key and save.
2. **Enable API Key for the API**:
   * Select the /hello resource > Click the **GET** method.
   * In the **Method Request**, enable **API Key Required**.
3. **Create a Usage Plan**:
   * Go to **Usage Plans** > Click **Create Usage Plan**.
   * Link the API and enable throttling or quota limits.
   * Associate the API key with the usage plan.
4. **Test API Key Security**:
   * Use Postman to pass the API key in the x-api-key header and test the API.

## ****Level 3: Advanced****

### ****Step 5: Enable Caching****

1. **Why Caching?**
   * Caching reduces backend load by storing frequently requested responses.
2. **Enable Caching**:
   * In API Gateway, go to the /hello resource > **GET** method.
   * Enable **Caching** in the **Integration Request** and set a TTL value (e.g., 300 seconds).
3. **Test Caching**:
   * Make repeated requests and observe reduced response times.

### ****Step 6: Configure Throttling****

1. **Set Up Throttling**:
   * In the **Usage Plan**, configure the throttling rate (e.g., 10 requests per second) and burst limits.
2. **Test Throttling**:
   * Use a script or tool to send multiple requests in a short time and observe the throttling behavior.

### ****Step 7: Create a WebSocket API****

1. **Create a WebSocket API**:
   * In API Gateway, choose **Create API** > **WebSocket API**.
   * Define the routes ($connect, $disconnect, $default).
2. **Write Lambda Handlers**:
   * Create Lambda functions for each route.
   * For $default, echo back messages:

def lambda\_handler(event, context):

return {"statusCode": 200, "body": f"You said: {event['body']}"}

1. **Test WebSocket API**:
   * Use a WebSocket client (e.g., wscat) to connect and test the API.

## ****Level 4: Expert****

### ****Step 8: Automate with CloudFormation****

1. **Write a CloudFormation Template**:
   * Define your API, resources, methods, and Lambda integrations in YAML:

Resources:

MyApi:

Type: AWS::ApiGateway::RestApi

Properties:

Name: MyAPI

1. **Deploy the Template**:
   * Use the AWS Management Console or CLI to deploy the stack.

### ****Step 9: Multi-Region Deployment****

1. **Deploy API in Multiple Regions**:
   * Repeat deployment steps for different AWS regions.
2. **Set Up Route 53**:
   * Create a Route 53 latency-based routing policy to direct traffic to the nearest region.

**Step 10: Advanced Integrations**

## ****1. DynamoDB Integration: Perform CRUD Operations****

### ****Goal****: Use API Gateway to interact with a DynamoDB table for Create, Read, Update, and Delete (CRUD) operations.

#### ****Steps****:

1. **Create a DynamoDB Table**:
   * Open the **DynamoDB Console**.
   * Click **Create Table** and provide:
     + Table name: ItemsTable
     + Primary key: ItemId (String)
   * Leave other settings as default and click **Create Table**.
2. **Create a Lambda Function for CRUD Operations**:
   * Open the **Lambda Console** and create a new function.
   * Name the function DynamoDB\_CRUD.
   * Add the following code for basic CRUD operations:

import boto3

import json

from botocore.exceptions import ClientError

dynamodb = boto3.resource('dynamodb')

table = dynamodb.Table('ItemsTable')

def lambda\_handler(event, context):

http\_method = event['httpMethod']

body = json.loads(event['body']) if event.get('body') else {}

try:

if http\_method == 'POST': # Create

table.put\_item(Item=body)

return {"statusCode": 201, "body": json.dumps({"message": "Item created"})}

elif http\_method == 'GET': # Read

item\_id = event['queryStringParameters']['ItemId']

response = table.get\_item(Key={'ItemId': item\_id})

return {"statusCode": 200, "body": json.dumps(response.get('Item', {}))}

elif http\_method == 'PUT': # Update

table.update\_item(

Key={'ItemId': body['ItemId']},

UpdateExpression="set #attr = :val",

ExpressionAttributeNames={"#attr": body['AttributeName']},

ExpressionAttributeValues={":val": body['AttributeValue']})

return {"statusCode": 200, "body": json.dumps({"message": "Item updated"})}

elif http\_method == 'DELETE': # Delete

item\_id = body['ItemId']

table.delete\_item(Key={'ItemId': item\_id})

return {"statusCode": 200, "body": json.dumps({"message": "Item deleted"})}

else:

return {"statusCode": 400, "body": json.dumps({"message": "Unsupported method"})}

except ClientError as e:

return {"statusCode": 500, "body": json.dumps({"message": str(e)})}

1. **Grant Lambda Permissions**:
   * Attach the AmazonDynamoDBFullAccess policy to your Lambda function's IAM role.
2. **Create an API in API Gateway**:
   * Open the **API Gateway Console** and create a new REST API.
   * Add resources (/items) and methods (POST, GET, PUT, DELETE).
   * Set the integration type to **Lambda Function** and select the DynamoDB\_CRUD Lambda function.
3. **Test CRUD Operations**:
   * Deploy the API to a stage (e.g., dev).
   * Use Postman or cURL to test POST, GET, PUT, and DELETE methods with appropriate request payloads and query parameters.

## ****2. SQS Integration: Send Messages to an SQS Queue****

### ****Goal****: Use API Gateway to send messages to an SQS queue for asynchronous processing.

#### ****Steps****:

1. **Create an SQS Queue**:
   * Open the **SQS Console** and create a new queue.
   * Name the queue MyQueue.
   * Use default settings and click **Create Queue**.
2. **Create an IAM Role for API Gateway**:
   * Go to the **IAM Console** and create a new role.
   * Choose **API Gateway** as the trusted service.
   * Attach the AmazonSQSFullAccess policy to the role.
3. **Create an API in API Gateway**:
   * Open **API Gateway Console** and create a new REST API.
   * Add a resource (/send-message) and a POST method.
   * Set the integration type to **AWS Service**:
     + Choose **SQS** as the service.
     + Set the action to SendMessage.
     + Provide the queue URL of MyQueue.
4. **Configure Request Templates**:
   * In the **Integration Request**, add a mapping template for the POST method:
     + Content-Type: application/json
     + Template:

{

"QueueUrl": "https://sqs.<region>.amazonaws.com/<account-id>/MyQueue",

"MessageBody": "$input.body"

}

1. **Test the API**:
   * Deploy the API and use Postman to send a POST request with a JSON body (e.g., {"message": "Hello SQS"}).
   * Check the SQS console to confirm the message was received.

## ****3. SNS Notifications: Trigger SNS Topics****

### ****Goal****: Use API Gateway to trigger an SNS topic and send notifications.

#### ****Steps****:

1. **Create an SNS Topic**:
   * Open the **SNS Console** and create a new topic.
   * Name the topic MyTopic.
   * Copy the **Topic ARN**.
2. **Create an IAM Role for API Gateway**:
   * Go to the **IAM Console** and create a new role.
   * Choose **API Gateway** as the trusted service.
   * Attach the AmazonSNSFullAccess policy to the role.
3. **Create an API in API Gateway**:
   * Open **API Gateway Console** and create a new REST API.
   * Add a resource (/publish-message) and a POST method.
   * Set the integration type to **AWS Service**:
     + Choose **SNS** as the service.
     + Set the action to Publish.
     + Provide the ARN of MyTopic.
4. **Configure Request Templates**:
   * In the **Integration Request**, add a mapping template for the POST method:
     + Content-Type: application/json
     + Template:

{

"TopicArn": "arn:aws:sns:<region>:<account-id>:MyTopic",

"Message": "$input.body"

}

1. **Subscribe an Endpoint to the Topic**:
   * Add a subscription to the topic (e.g., email or Lambda).
   * Confirm the subscription via the provided endpoint (e.g., email verification).
2. **Test the API**:
   * Deploy the API and use Postman to send a POST request with a JSON body (e.g., {"message": "Hello SNS"}).
   * Verify the notification was sent to the subscribed endpoint.