# Overview

This document provides a step by step guide to perform demos that are aligned with modules 8, 9, 10 and 11 of the **Development Intermediate 1** class in ACI.

In order to facilitate the parallel progress through the eLearning modules, the document is structure in a very specific way.

* **Level 1 headings** represent the **eLearning Skillbuilder's modules**.
  + For example, the Level 1 heading, "2 Module: Week 4 – Django Overview", matches the " Django Overview" skillbuilder module, which is offered in week 4 of the course.
* **Level 2 headings** represent a **major section** with the eLearning module
  + By major sections I mean the main divisions in an eLearning module, which are seen in the left side navigation bar.
* **Level 3 headings** represent a **heading within a section** in eLearning
  + Each eLearning section generally includes multiple topics, introduced with bold headings.
* **Level 4 headings** represent one **specific demo**
  + Every demo in this document is within a Level 4 heading, with an appropriate title, for easy referral.
  + The position of that level 4 heading, represents the closest location in the eLearning module that lines up with that specific demo. So by looking at the level 3 and level 2 headings, one can get a good approximation of which eLearning section and heading lines up with the demo provide.

## Disclaimer

This document is part of the AWS Cloud Institute, and it is provided for educational instructional purposes only.

## Where can I run these demos

**First of all, you don't have to**. These examples are primarily meant as demos, to be shown by the instructor during the class. **The module Labs are the official places created for you to practice AWS services hands on**. So you can observe the demos, and are likely to find similar steps on the Labs.

If you still want to run the examples on your own, for your own personal enjoyment, here are some of the things to consider.

### Credentials and Permissions

On order to interact with AWS services you need and AWS account, and valid AWS credentials, with permissions for the services used in the demo. Setting up IAM policies, ids and roles is beyond the scope of these demos.

### Using Jupyter Notebooks

Some of the examples are provided as Jupyter Notebooks. Jupyter Notebooks are a widely utilized format for educational purposes, as well as data analytics and AI, because they allow you to include documentation together with code. Once again, you can simply observe the demos without having to use the notebooks shared. However, if you want to view them for reference there a wide number of options:

* <https://jupyter.org/> - The official Jupyter site. You can download and run Jupyter Lab (that's what you see me using) from the site. That's free open source software. This is not an AWS product, so I don't want to get into installation steps, but you can see everything on their site.
* <https://jupyter.org/try-jupyter/lab/> - This is a free simple online version of Jupyter Notebooks, available inside the [Jupyter.org](http://jupyter.org/) site. I haven't played around with it much, but it seems to work for most notebooks I've tried.
* <https://studiolab.sagemaker.aws/> - SageMaker Studio Lab, which is an Amazon supported Jupyter Lab environment in the Cloud. This is currently free for anyone (no credit card required). Check the website for registration and all details
* Others - Multiple IDEs, including VS Code, will support Jupyter Notebooks

Any of the options above will let you open and view the notebook, so you'll be able to see the nicely formatted explanations in them. However, at the risk of being repetitive, in order to run them, you would need to have AWS permissions setup in the environment you are executing them from.

#### Jupyter Notebook Files Shared

For each Notebook example, I'm sharing two files:

* The **Jupyter Notebook version** with all the formatting. That's the file with the ***".ipynb*" extension**.
  + This can only be viewed and/or executed in an environment that supports Jupyter Notebooks, as explained earlier.
* The equivalent **Python version**. That's the file with the **"*.py*" extension**.
  + This is plain Python code that can run in any normal Python environment
  + The formatted content becomes regular Python comments in this version

### Using the ACI Lab Environment

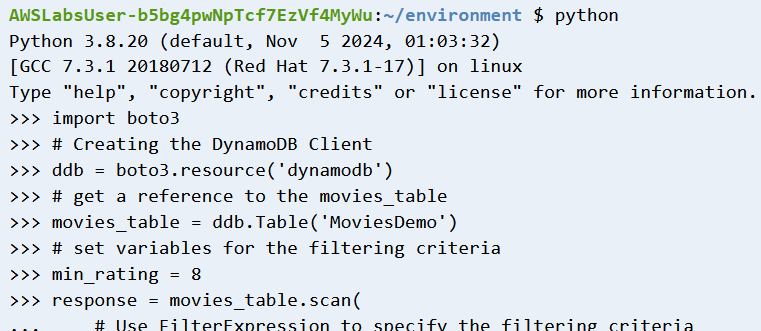
The ACI Practice Environment is a great place to practice your Python skills, but they are not meant for running AWS services. As mentioned earlier, the module Labs are the place for you to practice with the services that are being discussed in the specific modules.

The module Labs are not meant for overall practice, and they are restricted in permissions to only the steps required for the actual Lab to run. However, if a demo described happens to be performing steps that are similar in nature to the ones in one of the official module Labs, you might be able to use that Lab environment to also run the demo. There is no guarantee or support for that, but it is something you can try.

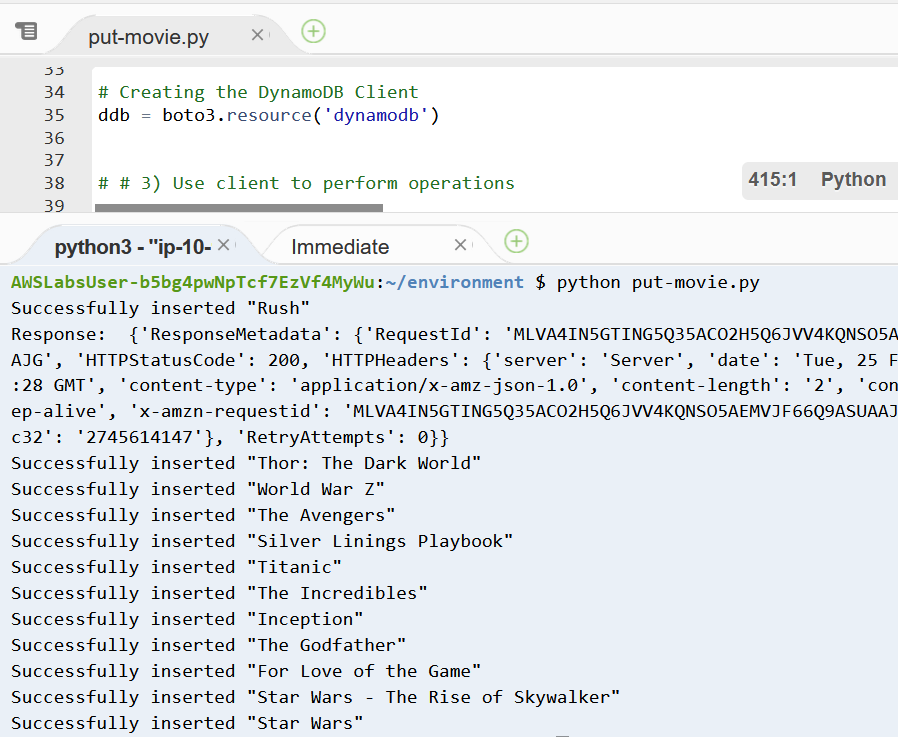
#### Jupyter Notebooks on Cloud9

Bear in mind that **Cloud9 does not support running Jupyter Notebooks directly**. You can still use the Notebooks as documentation (opened on a different viewer), and run the commands in Cloud9 in two possible ways:

* Start the Python interactive shell, then copy/paste commands from the Notebook into the Python interactive shell:



* Upload (or copy/paste) the Python exported version of a Notebook (the file with the ".py" extension). Use python to execute the file:



### Using your own account

Please remember that these demos are provided, as the name states, as **DEMOS**. **You are not required or expected to run them.** If you choose to run them on your account, you might incur costs. The only thing I will add, because it's standard AWS public knowledge, is that AWS has a number of services you can experiment with in their "Free Tier". That includes some level of support of Lambda, DynamoDB, API Gateway, S3, and Step Functions among many others. You can find information about them at <https://aws.amazon.com/free>.

# Week 8 – Microservice Data Storage Patterns

## Section: Using DynamoDB for Microservice *Data* Storage (lesson 5 of 7)

### Heading: SDK for Python

#### Demo: Create Movies Table

On AWS Console

* Navigate to DynamoDB dashboard
* Select **Tables** from left nav bar
* Click the Create table button, and enter:
  + Table name: MoviesDemo
  + Partition key: year
    - Type: Number
  + Sort key: title
    - Type: String
  + Accept all other defaults
* Click Create table button in the bottom to complete creation

#### Demo: DynamoDB SDK Operations

Execute the following Jupyter Notebooks (or Python versions) **shared in the Class Canvas**. The explanations are provided within the Notebooks.

* put-movie.ipynb
* get-movie.ipynb
* query-movies.ipynb
* scan-movies.ipynb
* update-movie.ipynb
* delete-moview.ipynb

# Week 8 – Microservice Deployment Patterns

## Section: Developing a Python Lambda Function (lesson 7 of 14)

### Heading: Lambda function handler

#### Demo: Create a Lambda Function to scan movies

Create function screen

* Navigate to Lambda dashboard
* Click **Create function** button
* Select “Author from Scratch”
* **Function name**: “list\_movies\_demo”
* **Runtime**: Python 3.12
* Expand “Change Default Execution Role”
* Check “Use an Existing Role”
* Search and select "*LambdaDynamoDBReadRole*"
* Click small “View the LambdaDynamoDBReadRole role” under to view role
  + Expand AWS managed policies to see permissions allowed
* Go back to Lambda function and Click **Create function** button

Enter function code

* On the code tab, replace the function with the code below
  + Walk through and understand code

# Import AWS Python SDK (Boto) Package and other required libraries

import boto3

import json

# Creating the DynamoDB Client

ddb = boto3.resource('dynamodb')

# set constant variables

table\_name = 'MoviesDemo'

# main handler function

def lambda\_handler(event, context):

# print event to log

print("Event received: " , event)

try:

# get a reference to the movies\_table

movies\_table = ddb.Table(table\_name)

# scan table and retrieve all movies

db\_resp = movies\_table.scan()

# create REST response

http\_response = {

'statusCode': 200,

'body': json.dumps(db\_resp, default=str),

'headers': {

'Content-Type': 'application/json',

'Access-Control-Allow-Origin': '\*'

},

}

# return reposnse

return http\_response

# catch any errors and return an response

except Exception as e:

return {

'statusCode': 400,

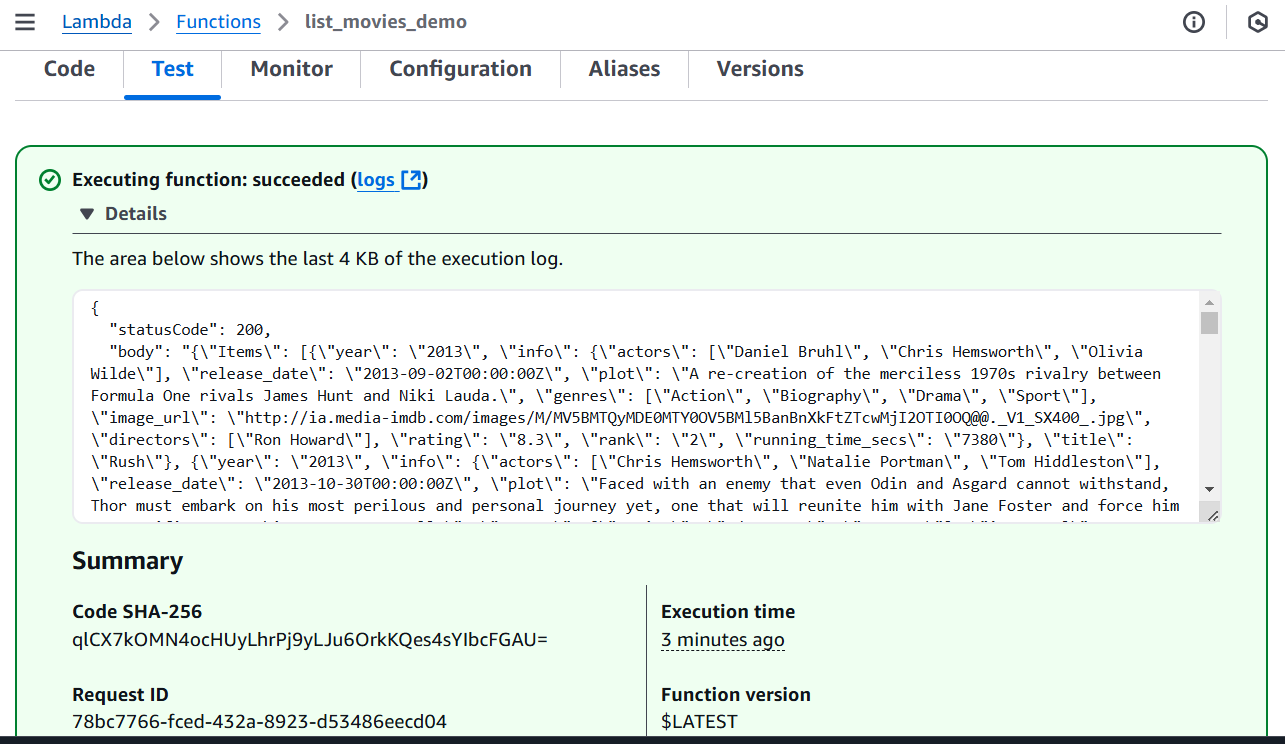
'body': json.dumps('Function Error: ' + str(e))

}

* Click **Deploy** button on the left

Test function

* Select the | Test | tab on top
* Scroll down to the "Event JSON" text box
  + Note that you can select sample events from many services
  + In this case there are no inputs necessary, since we are returning all movies
* Scroll up and click the **Test** button
* When the Execution box above shows green, with the message "Executing function: succeeded", expand the **"> Details"** section to view results
* The output should look something like this:



#### Demo: Create a Lambda Function to get a specific movie

Create function screen

* Navigate to Lambda dashboard
* Click **Create function** button
* Select “Author from Scratch”
* **Function name**: “get\_movie\_demo”
* **Runtime**: Python 3.12
* Expand “Change Default Execution Role”
* Check “Use an Existing Role”
* Search and select "*LambdaDynamoDBReadRole*"
* Click **Create function** button

Enter function code

* On the code tab, replace the function with the code below
  + Walk through and understand code

# Import AWS Python SDK (Boto) Package and other required libraries

import boto3

from boto3.dynamodb.conditions import Key

import json

# Creating the DynamoDB Client

ddb = boto3.resource('dynamodb')

# set constant variables

table\_name = 'MoviesDemo'

# main handler function

def lambda\_handler(event, context):

# print event to log

print("Event received: " , event)

# retrieve year and title from query string

title = event["queryStringParameters"]["title"]

year = int(event["queryStringParameters"]["year"])

print(f"Movie: {title} - Year: {year}")

try:

# get a reference to the movies\_table

movies\_table = ddb.Table(table\_name)

# scan table and retrieve all movies

db\_resp = movies\_table.get\_item(

Key={

'year': year,

'title': title

}

)

# create REST response

http\_response = {

'statusCode': 200,

'body': json.dumps(db\_resp, default=str),

'headers': {

'Content-Type': 'application/json',

'Access-Control-Allow-Origin': '\*'

},

}

# return reposnse

return http\_response

# catch any errors and return an response

except Exception as e:

return {

'statusCode': 400,

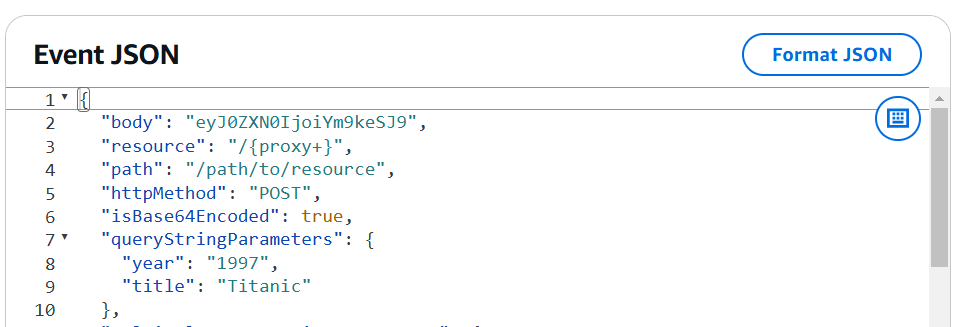
'body': json.dumps('Function Error: ' + str(e))

}

* Click **Deploy (...)** button on the left

Test function

* Select the | Test | tab on top
* Scroll down to the "Event JSON" text box
* Click on the "**Template – *optional***" drop down, and select "**API Gateway AWS Proxy**"
  + This should populate a sample request coming from an API
* In the **Event JSON**, update the "**queryStringParameters**" field to specify a movie to retrieve, by entering a movie *year* and movie *title.* For example:



* Scroll up and enter "get-movie-event" in the **Event name** field
  + Click the Save button, so this event can be reused later
* Now click the **Test** button
* When the Execution box above shows green, with the message "Executing function: succeeded", expand the **"> Details"** section to view results

### Heading: Versions

#### Demo: Create versions for a Lambda function

Open the function for editing

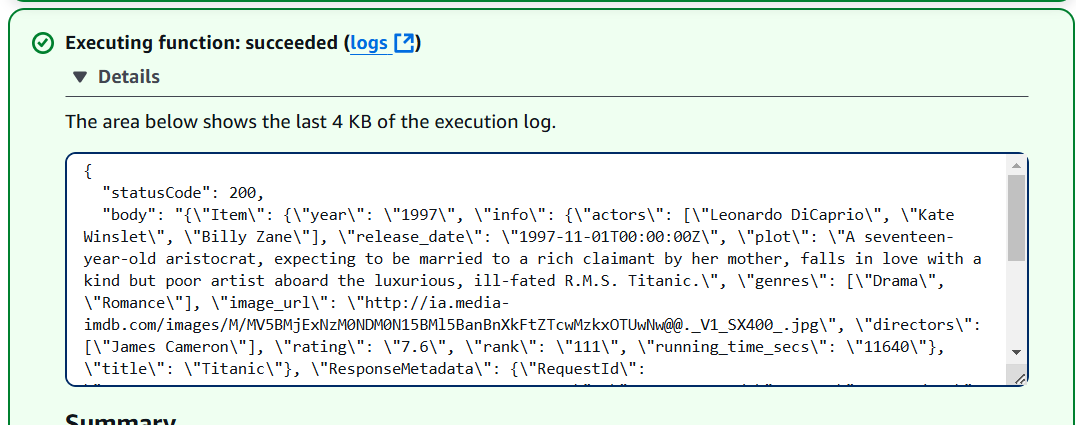
* Navigate to Lambda dashboard
* In the search box, enter "get-movie-demo" to locate the *get-movie-demo* function created earlier
* Click on the link to open the function for editing
* On the top portion of the screen, note the **Function ARN** value. It should look like:
  + arn:aws:lambda:*xxxxx-nnnnnnn*:function:get\_movie\_demo
  + The ARN uniquely identifies this particular Lambda function

Publish the initial version

* Select the | Versions | tab on top
* Click the Publish new version button
  + Enter a description, like "Initial version"
* Now look at the **Function ARN** value again. Now it should look like:
  + arn:aws:lambda:*xxxxx-nnnnnnn*:function:get\_movie\_demo**:1**
  + The extra "**:1**" at the end is enough to identify specifically this version of the function
* Select the | Code | tab on top
* You can view the code for that function, but if you try to type into the field, you will notice you cannot

Test function version

* Select the | Test | tab on top to test this version
* Test this version the same way you did earlier (you can reuse the same *get-movie-event* test event you had saved before)
* The output should look something like this:



* Note the hierarchy of the JSON structure in the "body":
  + There is an "**Item**" key, holding the movie data structure
  + Inside the item, the "**title**" and "**year**" are keys
  + The remaining fields are inside the "**info**" key on another structure.
* We will be making some changes to that output next

Return to the editable version of the function

* On the of the function page, you should see a breadcrumb like this:
  + [Lambda](https://us-east-2.console.aws.amazon.com/lambda/home?region=us-east-2#/discover) > [Functions](https://us-east-2.console.aws.amazon.com/lambda/home?region=us-east-2#/functions) > [get\_movie](https://us-east-2.console.aws.amazon.com/lambda/home?region=us-east-2#/functions/get_movie)\_demo > **Version: 1**
* Click on the [get\_movie](https://us-east-2.console.aws.amazon.com/lambda/home?region=us-east-2#/functions/get_movie)\_demo link to return to the editable version
* Select the | Code | tab on top
* You can now edit the code for the function

Change the function code

We are going to change our function such that instead of returning the movie in the multi-level structure we just saw ("Item" 🡪 "info" 🡪 key/values), every key/value will be at the first level. We'll see this better when we test.

* At the bottom of the code, add this function that will flatten the multi-level structure

# This function will take an item from a DynamoDB reponse for the movie table,

# and return a "flat" structure where every field is a key in the first level

def flatten\_movie(db\_resp):

# create an initial dictionary with just the year and title

flat\_movie = {}

flat\_movie['year'] = db\_resp['Item']['year']

flat\_movie['title'] = db\_resp['Item']['title']

# now add each key value under the "info" subsctructure

for key, value in db\_resp['Item']['info'].items():

flat\_movie[key] = value

# return the final flat movie

return flat\_movie

* Now make a small change in the handle to invoke that function, and used the result as our returned response
  + This is just a portion of the handler code, and I **highlighted in red** the places to change

# scan table and retrieve all movies

db\_resp = movies\_table.get\_item(

Key={

'year': year,

'title': title

}

)

**# flatten the movie structure**

**movie\_item = flatten\_movie(db\_resp)**

# create REST response

http\_response = {

'statusCode': 200,

'body': json.dumps(**movie\_item**, default=str),

'headers': {

'Content-Type': 'application/json',

'Access-Control-Allow-Origin': '\*'

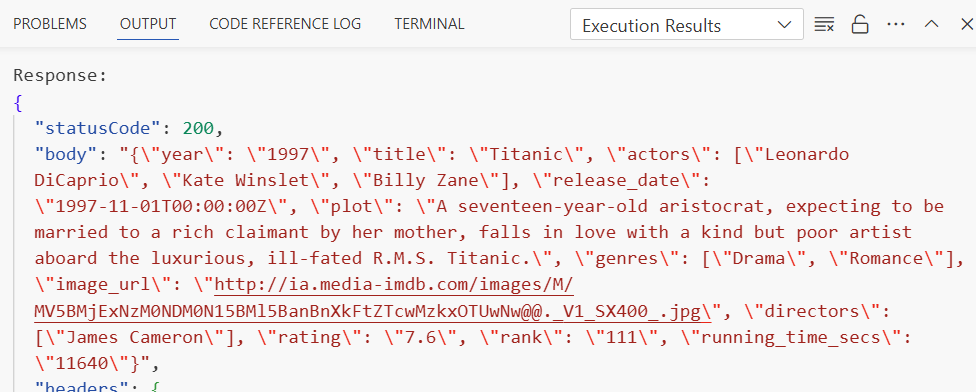
},

}

* Click **Deploy (...)** button on the left

Test function

* Now that you already have defined a tester by clicking the **Test (...)**  button on the left
* In the output, examine the response:



* Note that the movie fields now all appear in directly in the first level of the response body

Publish the update version

* Select the | Versions | tab on top
  + Note that you can now see a line for version 1 in there
* Click the Publish new version button
  + Enter a description, like "Flattened movie version"
* Now look at the **Function ARN** value again. Now it should look like:
  + arn:aws:lambda:*xxxxx-nnnnnnn*:function:get\_movie\_demo**:2**
  + The extra "**:1**" at the end is enough to identify specifically this version of the function
* You can test that version as you did with version 1, and confirm that it returns the flat results

### Heading: Aliases

#### Demo: Create aliases to different versions of a Lambda function

Open the function for editing

* If you had closed the function, open the main version as we had shown earlier. If you still had a version open, return to the editable version by clicking on the breadcrumb link as shown earlier.

Create *prod* alias

* Select the | Aliases | tab on top
* Click the Create alias button
  + Enter **name**: **prod**
  + Select **version**: **1**
  + Click the **Save** button
* Now look at the **Function ARN** value again. Now it should look like:
  + arn:aws:lambda:*xxxxx-nnnnnnn*:function:get\_movie\_demo**:prod**
  + The extra "**:prod**" at the end is enough to identify specifically this version of the function

Test *prod* alias

* Select the | Test | tab on top to test this version
* Test this version the same way you did earlier (you can reuse the same *get-movie-event* test event you had saved before)
* Verify that the output shows the same response we get from version 1 (the multi-level movie item)

Return to the editable version of the function

* Return to the main editable function clicking the [get\_movie](https://us-east-2.console.aws.amazon.com/lambda/home?region=us-east-2#/functions/get_movie)\_demo link in the breakcrumb on top as we've done earlier

Create *test* alias

* Select the | Aliases | tab on top
* Note that you can now see the prod alias in the list
* Click the Create alias button
  + Enter **name**: **test**
  + Select **version**: **2**
  + Click the **Save** button
* Now look at the **Function ARN** value again. Now it should look like:
  + arn:aws:lambda:*xxxxx-nnnnnnn*:function:get\_movie\_demo**:test**
  + The extra "**:test**" at the end is enough to identify specifically this version of the function

Test *test* alias

* Select the | Test | tab on top to test this version
* Test this version the same way you did earlier (you can reuse the same *get-movie-event* test event you had saved before)
* Verify that the output shows the same response we get from version 2 (the flattened movie item)

# Week 9 – Microservice Integration Patterns Part 1

## Section: API Gateway Integration Pattern (lesson 3 of 18)

### Heading: API gateway pattern

#### Demo: Create a simple API to retrieve movies

Recap Lambda get\_movie\_demo function to be exposed

* Go to Lambda, and open [get\_movie](https://us-east-2.console.aws.amazon.com/lambda/home?region=us-east-2#/functions/get_movie)\_demo function
* Go to | Aliases | tab, and recall the two aliases for this function
  + prod – points to the original version, with a multi-level JSON response
  + test – points to the latest version, with a flattened movie result
* Leave the tab open for reference later

Create Tucker Movies API

* Go to API Gateway and click the **Create API** button
* Scroll to **REST API** and click  **Build**  button
  + Select **New API**
  + **API name**: **Tucker Movies Demo**
  + **Description**: **Movies demo mm/dd/yy**
* Click the **Create API** button in the bottom

Create a REST resource

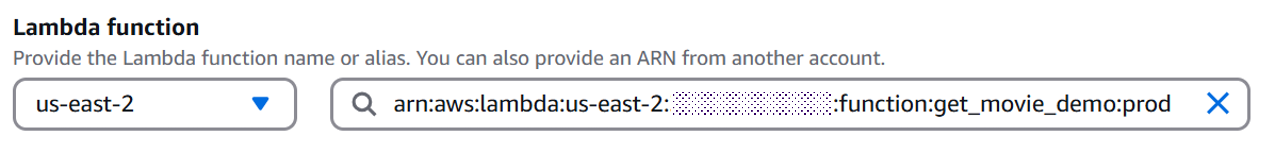
REST APIs are based on the concept of defining resources, and then operations/methods acting on those resources.

* Under **Resources**, click the Create resource button
* Under **Resource Details:**
  + **Resource Name**: **movie**
  + **CORS**: Off (unchecked)
* Click the **Create resource** button

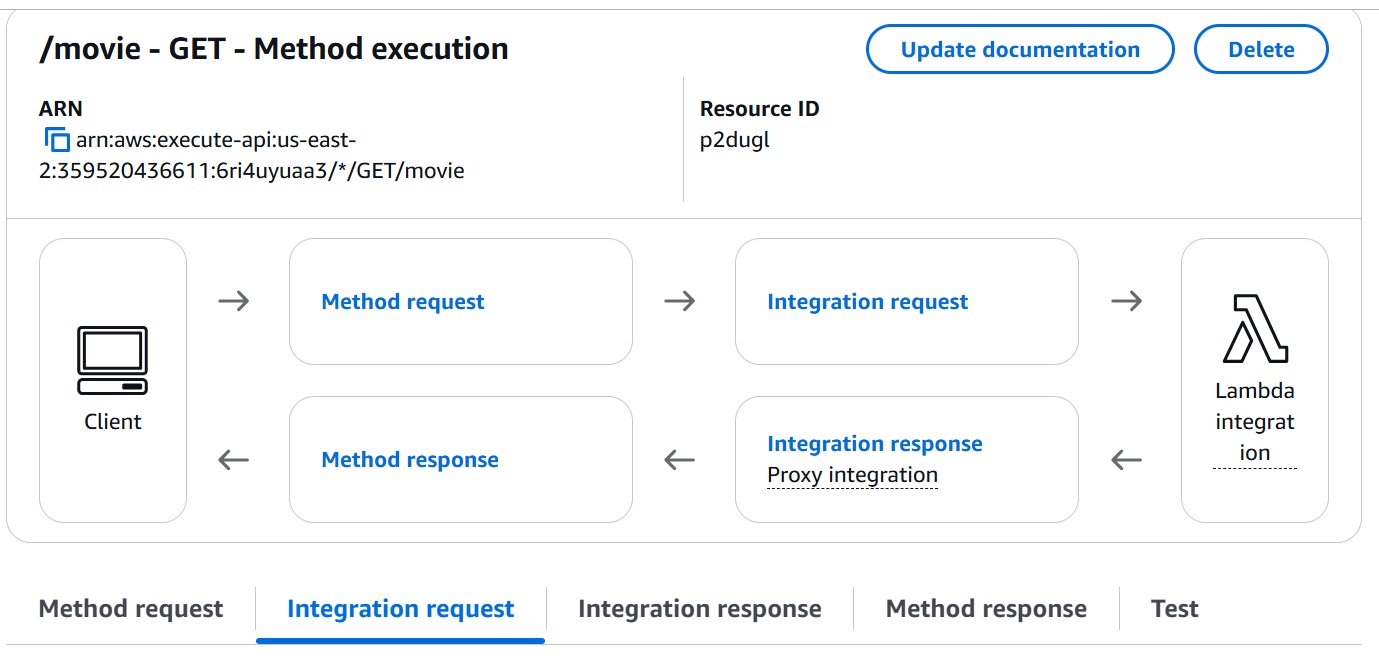
Create a REST method

Now that we have a resource, we can create REST methods on it

* Under **Resources** heading, click on the **/movie** line
  + Confirm that the **Resource details** box shows **/movies** under **Path**
* Under **Methods** box, click the Create method button
* Under **Method Details:**
  + **Method type**: Select **GET**
  + **Integration Type**: Select **Lambda function**
  + **Check** **Lambda Proxy integration**
    - This options means API will pass the inputs/outputs as is to the Lambda function
  + **Lambda Function**: Start typing "*get\_movie\_demo*", and **select** the full **get\_movie\_demo** when it shows as an option
    - Append (type in) "**:prod**" at the end of the function name, to identify the specific version we want. This should look like this:



* + **Note the message box** below, which indicates API Gateway will be given permission to invoke the Lambda function by default
  + Click the **Create method** button
* You should now see a **/movie** box like below, which represents the various points you can instrument on API Gateway for the method (we'll see that in subsequent demos)

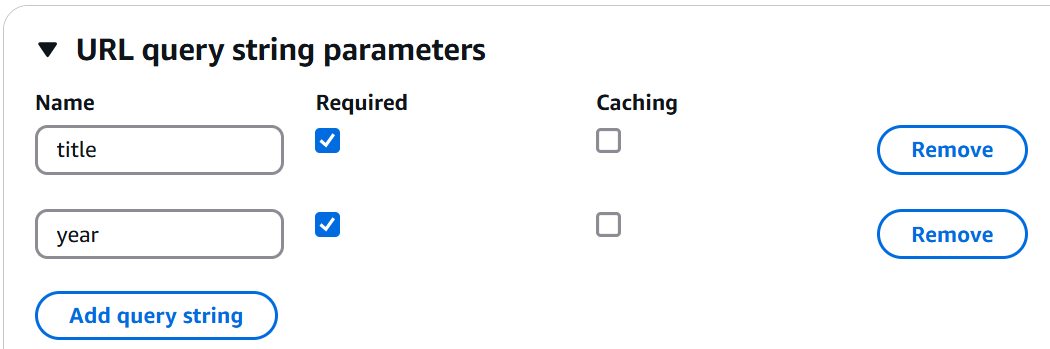


### Heading: Creating, deploying, securing, monitoring, optimizing, and testing APIs

#### Demo: Adding a request validation to the processing flow

The API Gateway processing flow supports multiple customizations and extra features through the request/response process. In this demo, we'll just add a simple validation of request inputs. Specifically, a **valid request for a movie** **must include** a movie **title** **and** a movie **year**

* This process continues from where the previous demo stopped
  + If you left the API page, just reopen the API and click on the **GET** line under the **/movie** resource to return to the four request stages you saw at the end of the previous demo
* Click on | Method request | tab, or the **Method request** link on the diagram
* Click the Edit button in the **Method request settings** box
  + **Authorization**: **None**
    - For this demo, our API will be public
  + **Request Validator**: **Validate query string parameters and headers**
  + **Expand** **> URL query string parameters**
  + Click the Add query string button in Add title and year
    - Add entries for title and year, and mark both required
    - You should end up with this:



* + Make both required, and explain validation
  + Click the **Save** button at the bottom

We should now be ready to test this functionality in the next demo.

#### Demo: Local API testing

Before deploying the API, it is important to perform validation testing.

* This process continues from where the previous demo stopped
  + If you left the API page, just reopen the API and click on the **GET** line under the **/movie** resource to return to the four request stages you saw at the end of the previous demo
* Click | Test | tab
  + **Query strings**: **title=Titanic&year=1997**
* Click the **Test** button at the bottom, and view the results:
  + Remember, I invoked the **prod** version, so I should see the multi-level JSON response
  + If you scroll down, you can also see a subset of the API processing logs, which is very useful when there are errors (the full log is available in CloudWatch)
* Now remove the year from the **Query string** so it's just: **title=Titanic**
* Click the **Test** button at the bottom, and view the results:
  + You will see the Response body

{"message": "Missing required request parameters: [year]"}

* + This response was generated directly by the API Gateway based on the request validation you added earlier
  + The validation happens before the Lambda function is called, so it will save time and cost
  + You will see the same error if you omit the title or both

#### Demo: Applying transformations using API Gateway

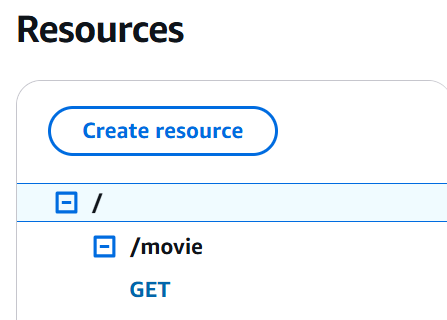
For this demo we will create a function that should return only a summary of key information for all movies. Specifically, we **only want** the movie **title, year, plot** and a poster **image URL** to use in a thumbnail.

We will reuse an existing Lambda function that returns more data than we need, then apply a transformation in API Gateway to customize the returned response.

Create a new REST resource

We will create a new REST resource for our movie summary, using the same steps we used earlier.

* Under the **Resources** tree on the left, click on the root **"/"** node, to make sure our new resource will be added on the top of the API resource tree



* Click the Create resource button
* Under **Resource Details:**
  + **Resource Name**: **moviesummary**
  + **CORS**: Off (unchecked)
* Click the **Create resource** button

Create a GET method

Now we'll create a REST method to retrieve the movies summary. In this use case, we're assuming there was already a more general ***list\_movies*** Lambda function available, which was not specifically created for this movie summary purpose.

* Under **Methods** box, click the Create method button
* Under **Method Details:**
  + **Method type**: Select **GET**
  + **Integration Type**: Select **Lambda function**
  + **Leave Lambda Proxy integration unchecked**
    - This allows for transformations of inputs/outputs to the Lambda function
  + **Lambda Function**: Start typing "*list\_movie* ", and **select** the full **list\_movies\_demo** when it shows as an option
    - Append (type in) "**:prod**" at the end of the function name, to identify the production version we want. The function name will look like:
      * arn:aws:lambda:us-east-2:xxxxxxxxx:function:**list\_movies\_demo:prod**
  + Click the **Create method** button
* You should now see a **/moviesummary** box representing the various points you can instrument on API Gateway for the method

Test initial version

* Click | Test | tab
* Since we are listing all movies, no query string is required
* Click the **Test** button at the bottom, and view the results:
  + The function works, but it returns a lot of data, represented in the more complex multi-level JSON format
  + This is not the "movie summary" we were hoping, so we'll fix this next.

Using API Gateway mapping templates

Remember that in this use case, the ***list\_movies*** function was not written specifically to support this API. One might envision that other APIs might need full details about the movies, or a different subset of fields than what we want. So changing the Lambda function is not an option. We will instead use API Gateway transformation to customize the response to our needs.

* Click on | Integration response | tab, or the **Integration response** link on the request flow diagram
* Click the Edit button in the **Default – Response** box
  + Scroll down to **Mapping Templates,** and click the arrow to **expand**
  + On **Template body**, enter the following:
    - The transformation template uses the **Apache VTL** format, which is an open source format for specifying templates with dynamic values
    - For full details on the template transformation format, and how it's used by API Gateway, please refer to <https://docs.aws.amazon.com/apigateway/latest/developerguide/models-mappings.html>

#set($inputRoot = $util.parseJson($input.path('body')))

**{**

"**movies**": **[**

#foreach($elem in $inputRoot.Items)

**{**

"**title**": "$elem.title",

"**year**": $elem.year,

"**plot**": "$elem.info.plot",

"**image\_url**": "$elem.info.image\_url"

**}**#if($foreach.hasNext),#end

#end

**]**

**}**

* + Although the details of the Apache VTL language are not in the scope of this discussion, you can generally see in the blue highlighter portions, that the format above will create a structure with a list of movies, each element containing the summary data we want
  + Click the **Save** button

Test updated version

* Click | Test | tab
* Click the **Test** button at the bottom, and view the results:
* Note the nicely formatted results with just the summary data we wanted.

#### Demo: Deploying an API to a stage

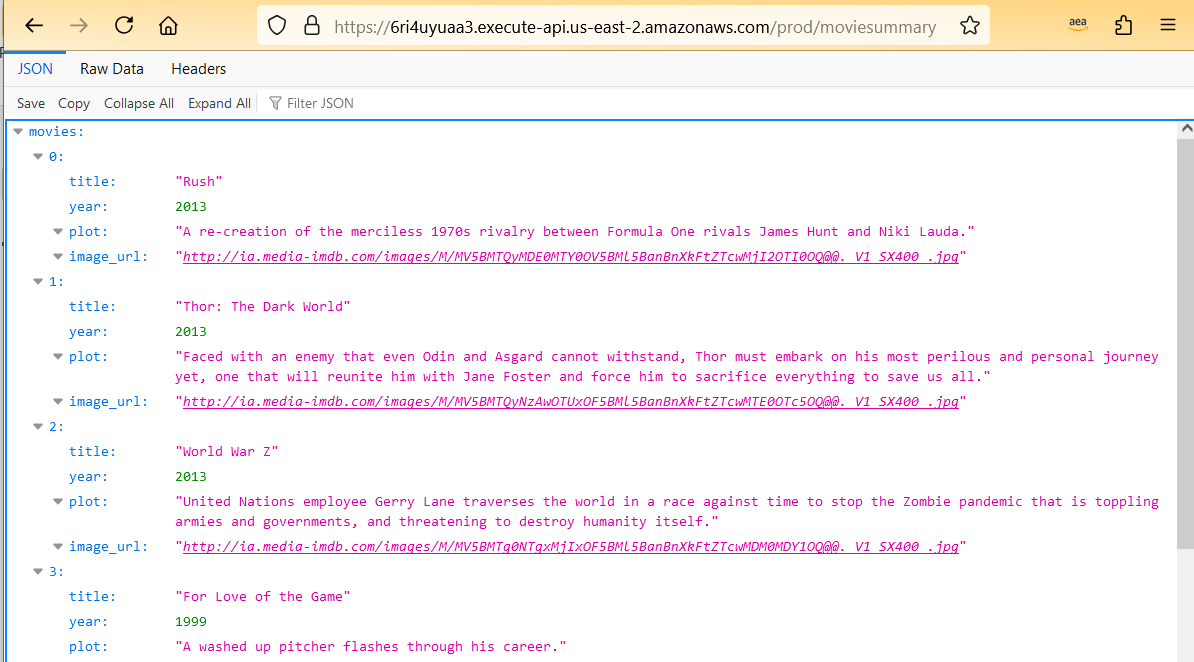
For starters, we'll deploy our prod versions of get movie and movies summaries to a production API stage.

Deploy to a stage

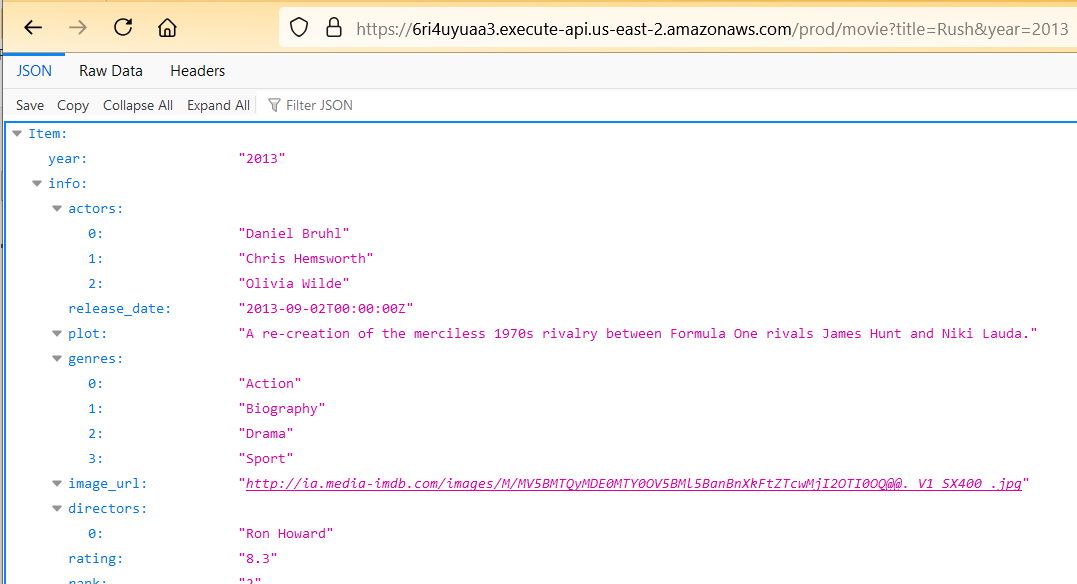
* If you had closed the Tucker Movies Demo API, open it again
* On top right, click the **Deploy API** button
  + **Stage**: Select **\*New stage\***
  + **Stage name**: **prod**
  + **Deployment description**: **Production**
* Copy the **Invoke URL** on the right side

Test the API on the browser

* Paste the invoke URL copied above on a browser, but don't enter:
* For the Movie Summary resource, add **/moviesummary** to the URL, and enter.
  + You should see a result like this:



* For the Movie resource, add **/movie?title=Rush&year=2013** to the URL, and enter.
  + You could also use any other title and year that were returned from the previous call to the **moviesummary** resource
  + You should see a result like this (note the multi-level JSON response):



#### Demo: Using stage variables to deploy an additional test stage

Let's assume now we want to have a test version of our API, which would direct requests to the test versions of our Lambda functions. We can create a Test stage of our API, but the problem is that if you recall the earlier demos, we explicitly added a ":*prod*" to the end of our Lambda ARNs to route the requests to the *prod* function. If we change it to ":*test*", then all subsequent deployments will go to test. What we need is a way to have *prod* API stage redirect to the *prod* Lambda functions, and the *test* API stage redirect to the *test* Lambda functions. We will use stage variables to accomplish that.

Update the Integration request to use a stage variable

* If you had closed the Tucker Movies Demo API, open it again
  + If you it's already opened, go to the main resource page, by clicking on **Resource** on the left navigation bar
* Click on **GET** link under **/movie** resource
* Click on the | Integration Request | tab
* Under **Integration request settings**, click the Edit button
* Under **Method Details**, scroll down to the **Lambda function** field
* On the function name, replace the “:prod” alias in the end with ":test"
* Click the **Save** button
  + We're not leaving the integration like that, but we I need to do this first so that API Gateway adds permissions to the test version, otherwise we’d have to manually add
* Click the Edit button again, to return to the **Method Details** page
* On the function name, replace the “:test” in the end with "**${stageVariables.lambdaVersion}**"
  + The function name will end up like:

arn:aws:lambda:us-east-2:xxxxxxxxx:function:**get\_movie:${stageVariables.lambdaVersion}**

* + In the format above the "**${stageVariables}**"part is always fixed, and the "**lambdaVersion**" after the "." is the name of the variable we choose.
* You can ignore the message about running a CLI command, because we accomplished the same thing by temporarily adding the ":test" alias in the previous step.
* Click the **Save** button

Test updated integration

* Click | Test | tab
* Enter:
  + **Query strings**: **title=Titanic&year=1997**
  + **lambdaVersion: test**
* Click the **Test** button at the bottom, and view the results:
  + Remember, I invoked the **test** version, so I should see simple flat JSON response

Deploy to a new stage

* On top right, click the **Deploy API** button
  + **Stage**: Select **\*New stage\***
  + **Stage name**: **test**
  + **Deployment description**: **Test**

Test the API on the browser

* Scroll down and select the | Stage variables | tab
* Click the Edit button
* Click the Add stage variable button
  + **Name**: **lambdaVersion**
  + **Value**: **test**
* Click the **Save** button
* Scroll back up, and copy the **Invoke URL** on the **Stage details**
* Paste the invoke URL copied on a browser, add **/movie?title=Rush&year=2013** to the URL, and enter.
  + You could also use any other title and year that were returned from the previous call to the **moviesummary** resource
  + You should see a result with the flattened JSON format we expect from the test Lambda function
* Verify that the prod version still works as expected, by changing "**test**" to "**prod**" in the URL, and resubmitting the URL
  + You should see a result with the multi-level JSON format we expect from the prod Lambda function

## Section: Decoupled Messaging Integration (lesson 4 of 18)

### Heading: Decouple messaging use case

#### Demo: Using Amazon SQS

This will be a simple example of SQS functionality. We will use a queue to simulate coffee orders, create a queue in the console, and use Jupyter notebooks for the send and receive examples.

Create an SQS queue

* In the AWS console, go to the SQS dashboard
* Click the **Create queue** button
  + **Type:** Select **Standard**
  + **Name: coffee-orders**
  + Leave other fields as default
* Scroll down and click the **Create queue** button

Use a Jupyter Notebook to receive messages

* Go to **SQS\_ReadMessage.ipynb**
* Review code and documentation
* Start receive message loop
  + Note that we can start receiving messages before we are even ready to send, because sending and receiving messages are completely decoupled.
  + Leave the ReadMessage notebook running, which will periodically print a status message

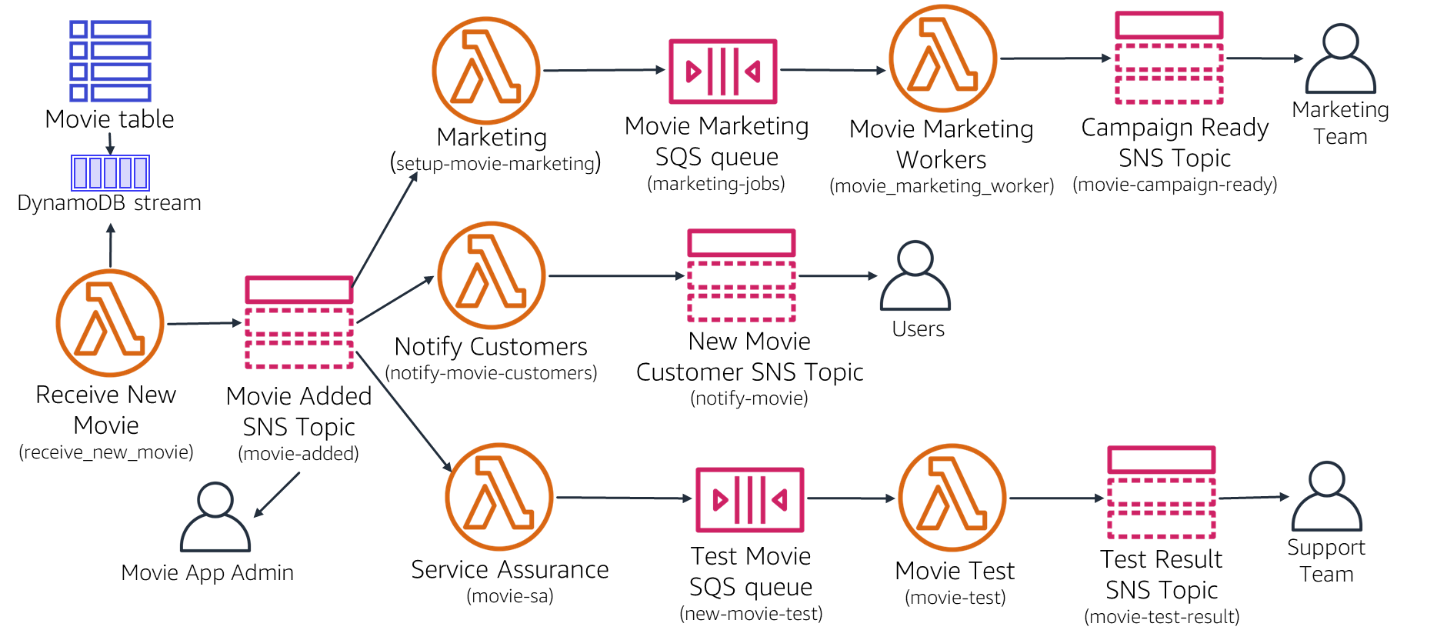
Use a Jupyter Notebook to send messages

* Go to **SQS\_SendMessage.ipynb**
* Review code and documentation
* If possible, setup your display so you can see the read message and send message notebooks running side by side
  + At this point the ReadMessage notebook will have been waiting for a while, but that's ok
* Run the code to send a message, and observe it appearing immediately on the side
  + You can repeat and send other messages, as long as you don't exit the receiving loop.

## Section: Publish-Subscribe Integration (lesson 5 of 18)

### Heading: Pub-sub use case

#### Demo: And end-to-end SQS-SNS integration use case

This end to end demo represents a fictitious use case, depicted in the diagram below. **This is not meant to be a realistic process for a movie streaming web site**, but simply a scenario contrived to show how streams, queues and topics can collaborate in an overall distributed flow. Given the large number of resources used in this example, it would be impractical to step by step set of instructions to build it. I'm providing some background here, but even as this is demoed in class, all the resources will be already created. During the demo I will how some of the integrations are setup, so please refer to the class recording for that.Receive

Business Scenario

The simple fictitious business process represented in the diagram is as follows:

* A movie is added to a movie database (such as the one created in Demo 2.1.1.1)
* A Lambda function(***receive\_new\_movie***) will be invoked when the movie is added
* The ***receive\_new\_movie*** function will publish a message to an SNS topic (***movie-added***) notifying that a move was added
* Four subscribers are listening to that topic
  + And email subscription to a movie admin who wants to be notified whenever a movie is added
  + A Lambda function (***setup-movie-marketing***) owned by the Marketing department, to start any new marketing tasks
  + A Lambda function (***notify-movie-customers*** ) to notify customer of the new movie
  + A Lambda function (***movie-sa*** ) to setup any service assurance tasks needed for the new movie.
  + Each of the three Lambda functions will start their own process flows in parallel in the subsequent steps
* The ***setup-movie-marketing*** function will identify the type of marketing required, and then send a job request to an SQS queue (***marketing-jobs***) for the movie marketing team
  + A Lambda function (***movie\_marketing\_worker***) owned by the Movie Marketing department, will perform the required tasks for marketing this movie.
  + When the marketing campaign is read, the function will publish a message to an SNS topic (***movie-campaign-ready***) announcing the campaign for this movie is ready
  + An email subscriber will be listening for on that topic
* The ***notify-movie-customers*** function will perform some basic checks, then publishes a message to an SNS topic (***notify-movie-customers***) announcing the new movie to any interested customer
  + Interested users who subscribed to that topic would subscribe to it, and receive an email
* The ***movie-sa*** function will identify the type of service assurance tasks needed (setting up monitoring, setting up alarms, performing tests, etc). In this case, it will then send a job request to an SQS queue (***new-movie-test***) for the movie testing team
  + A Lambda function (***movie\_test***) owned by the testing team, will perform the required tasks for testing this movie
  + When the test is completed, the function will publish a message to an SNS topic (***movie-test-result***) announcing the result of the test
  + An email subscriber will be listening for on that topic

Technical Details

It would be too hard to detail every step required to implement the flow above, because many of them are just variations of a similar procedure. Steps for creating Lambda functions and SQS queues have already been seen in other demos. I'll list some of the basic technical requirements here:

* The movies table needs to enable streaming, so that all the changes are captured, and can be retrieved by other services
* The ***receive\_new\_movie*** function will need to add a trigger to automatically pull any changes to the movies table
* The Lambda subscriptions on the SNS topics can be created from the SNS topic itself (using the SNS dashboard)
* To Lambda functions that read from an SQS queue can do so by creating a trigger, which will automatically poll from the specified queues
* The code for the Lambda functions is provide offline.
  + The Python code provided will have the steps required for retrieving data from the event messages for their specific data source (DynamoDB stream, SQS or SNS) , as well as the steps for publishing messages to queues or topics as shown in the process.
  + **Please note** that the code does not attempt to actually implement the business tasks, such as "marketing" or "testing". The intent of this demo is to highlight the use of the integration services.
* The initial task of adding a new movie, which will initiate this flow, can be accomplished by using the **put-movie Jupyter notebook** used in demo 2.1.1.2.

# Week 11 –Application Deployment Frameworks - Part 2

## Section: Using the AWS SAM Command Line Interface (lesson 3 of 9)

### Heading: Initializing the AWS SAM CLI environment

#### Demo: Creating a SAM application with same init

In this demo we will create a serverless application to implement the same **get\_movie** Lambda function and associated API we built in earlier demos. We start in this first demo by creating the application in SAM. This requires the SAM CLI to have been installed earlier, in environment with the AWS CLI , and appropriate AWS permissions.

Run sam init to create a SAM application

The sam init command will start a number of prompts, and we'll cover each step in the bullets below.

* Run the **sam init** command to create an application
  + Run the command in the folder where you want to create your application under
  + The steps below were executed on Windows using Powershell, but they should be similar in other supported environments.

PS C:\dev\sam> **sam init**

You can preselect a particular runtime or package type when using the `sam init` experience.

Call `sam init --help` to learn more.

* For the template source, choose "**1**" for a simple starter application template
* Select "**1**" again for the "Hello World Example"

Which template source would you like to use?

1 - AWS Quick Start Templates

2 - Custom Template Location

Choice: **1**

Choose an AWS Quick Start application template

1 - Hello World Example

2 - Data processing

3 - Hello World Example with Powertools for AWS Lambda

4 - Multi-step workflow

5 - Scheduled task

6 - Standalone function

7 - Serverless API

8 - Infrastructure event management

9 - Lambda Response Streaming

10 - GraphQLApi Hello World Example

11 - Full Stack

12 - Lambda EFS example

13 - Serverless Connector Hello World Example

14 - Multi-step workflow with Connectors

15 - DynamoDB Example

16 - Machine Learning

Template: **1**

* For runtime, enter "**N**" to view all the available languages.
  + Select "**16**" in the end, to use Python 3.12

Use the most popular runtime and package type? (python3.13 and zip) [y/N]: **N**

Which runtime would you like to use?

1 - dotnet8

2 - dotnet6

3 - go (provided.al2)

4 - go (provided.al2023)

5 - graalvm.java11 (provided.al2)

6 - graalvm.java17 (provided.al2)

7 - java21

8 - java17

9 - java11

10 - java8.al2

11 - nodejs22.x

12 - nodejs20.x

13 - nodejs18.x

14 - python3.9

15 - python3.13

16 - python3.12

17 - python3.11

18 - python3.10

19 - ruby3.3

20 - ruby3.2

21 - rust (provided.al2)

22 - rust (provided.al2023)

Runtime: **16**

* Select "**1**" to create a ZIP file for the Lambda function

What package type would you like to use?

1 - Zip

2 - Image

Package type: **1**

Based on your selections, the only dependency manager available is pip.

We will proceed copying the template using pip.

* Accept the defaults (click Enter on the prompt) for various logging options

Would you like to enable X-Ray tracing on the function(s) in your application? [y/N]:

Would you like to enable monitoring using CloudWatch Application Insights?

For more info, please view https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/cloudwatch-application-insights.html [y/N]:

Would you like to set Structured Logging in JSON format on your Lambda functions? [y/N]:

* For the project name, enter **sam-movies-aci**
  + After entering the project name, sam will create an initial project.

Project name [sam-app]: **sam-movies-aci**

Cloning from https://github.com/aws/aws-sam-cli-app-templates (process may take a moment)

-----------------------

Generating application:

-----------------------

Name: sam-movies-aci

Runtime: python3.13

Architectures: x86\_64

Dependency Manager: pip

Application Template: hello-world

Output Directory: .

Configuration file: sam-movies-aci\samconfig.toml

Next steps can be found in the README file at sam-movies-aci\README.md

Commands you can use next

=========================

[\*] Create pipeline: cd sam-movies-aci && sam pipeline init --bootstrap

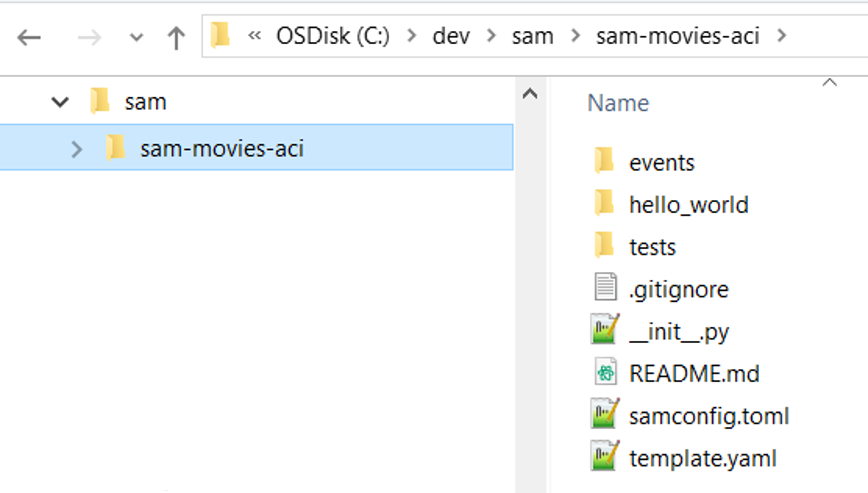
[\*] Validate SAM template: cd sam-movies-aci && sam validate

[\*] Test Function in the Cloud: cd sam-movies-aci && sam sync --stack-name {stack-name} --watch

Examine application file structure created

After sam-init completes, the following application file structure is constructed:

* The **template.yaml** file is a SAM template for a basic serverless application, which we will be customizing later
* We'll discuss some of other pre-created folders and files in subsequent demos



#### Demo: Customize the initial application to your own needs

The *sam init* command created a folder structure, and an initial app.py file. Once that's set we can use our preferred IDE to operate on the files, and add our own code.

Update the template.yaml file

* First, we update the Lambda function definition. We will keep the same overall structure for the file, and only change a few things
  + Update various references to "Hello World" to "Movies"
  + Update the Global **Timeout** to 10 seconds
  + Rename the **CodeUri** property from "**hello\_world"** to a more generic "**src**"
  + Add a name of **MoviesAciFunction** for the Lambda function, otherwise a long default one is created by SAM/CloudFormation
  + Add an IAM policy for the Lambda function to be able to read DynamoDB data
* We also need to **add permissions** for the API to invoke the Lambda function
  + Note the **MoviesAciFunctionPermission** resource added
* An updated version of the file with the changes highlighted is included below:

AWSTemplateFormatVersion: '2010-09-09'

Transform: AWS::Serverless-2016-10-31

Description: >

Sample SAM Template for sam-movies-aci

Globals:

Function:

Timeout: **10**

Resources:

**MoviesAciFunction**:

Type: AWS::Serverless::Function

Properties:

**FunctionName:** **MoviesAciFunction**

CodeUri: **src/**

Handler: app.lambda\_handler

Runtime: python3.12

**Policies:**

**- AmazonDynamoDBReadOnlyAccess**

Architectures:

- x86\_64

Events:

**MoviesAci**:

Type: Api

Properties:

Path: /**movie**

Method: get

**MoviesAciFunctionPermission:**

**Type: AWS::Lambda::Permission**

**Properties:**

**Action: lambda:InvokeFunction**

**FunctionName: !Ref MoviesAciFunction**

**Principal: apigateway.amazonaws.com**

**SourceArn: !Sub "arn:aws:execute-api:${AWS::Region}:${AWS::AccountId}:${ServerlessRestApi}/\*/\*"**

Outputs:

**MoviesAciApi**:

Description: "API Gateway endpoint URL for Prod stage for Movies Aci function"

Value: !Sub "https://${ServerlessRestApi}.execute-api.${AWS::Region}.amazonaws.com/Prod/**movie**/"

**MoviesAciFunction**:

Description: "Movies Aci Lambda Function ARN"

Value: !GetAtt **MoviesAciFunction**.Arn

**MoviesAciFunctionIamRole**:

Description: "Implicit IAM Role created for Movies Aci function"

Value: !GetAtt **MoviesAciFunctionRole**.Arn

Update folder structure to match the template

* Rename the "**hello\_world**" folder to "**src**", so that it matched the CodeUri specified in the template

#### Demo: Update the template code with an initial version of your own code

* Open the *app.py* file under the **src** folder, and replace with the contents below to retrieve a movie from the database.
  + This is the same code we used in Section 3.1.1.2

# Import AWS Python SDK (Boto) Package and other required libraries

import boto3

from boto3.dynamodb.conditions import Key

import json

# Creating the DynamoDB Client

ddb = boto3.resource('dynamodb')

# set constant variables

table\_name = 'Movies'

# main handler function

def lambda\_handler(event, context):

# print event to log

print("Event received: " , event)

# retrieve year and title from query string

title = event["queryStringParameters"]["title"]

year = int(event["queryStringParameters"]["year"])

print(f"Movie: {title} - Year: {year}")

try:

# get a reference to the movies\_table

movies\_table = ddb.Table(table\_name)

# scan table and retrieve all movies

db\_resp = movies\_table.get\_item(

Key={

'year': year,

'title': title

}

)

# create REST response

http\_response = {

'statusCode': 200,

'body': json.dumps(db\_resp, default=str),

'headers': {

'Content-Type': 'application/json',

'Access-Control-Allow-Origin': '\*'

},

}

# return reposnse

return http\_response

# catch any errors and return an response

except Exception as e:

return {

'statusCode': 400,

'body': json.dumps('Function Error: ' + str(e))

}

#### Demo: Build the application with sam build

* Change into the application folder, and execute **sam build**

PS C:\dev\sam> **cd sam-movies-aci**

PS C:\dev\sam\sam-movies-aci> **sam build**

Starting Build use cache

Manifest file is changed (new hash: 3298f13049d19cffaa37ca931dd4d421) or dependency folder

(.aws-sam\deps\3f8a6a76-64ff-4b1f-bdcc-586eeefbd505) is missing for (MoviesAciFunction), downloading dependencies and copying/building

source

Building codeuri: C:\dev\Demos\sam-movies-aci\movies runtime: python3.12 architecture: x86\_64 functions: MoviesAciFunction

Running PythonPipBuilder:CleanUp

Running PythonPipBuilder:ResolveDependencies

Running PythonPipBuilder:CopySource

Running PythonPipBuilder:CopySource

Build Succeeded

Built Artifacts : .aws-sam\build

Built Template : .aws-sam\build\template.yaml

Commands you can use next

=========================

[\*] Validate SAM template: sam validate

[\*] Invoke Function: sam local invoke

[\*] Test Function in the Cloud: sam sync --stack-name {{stack-name}} --watch

[\*] Deploy: sam deploy --guided

#### Demo: Test Lambda function locally

Using the SAM CLI we can locally test a function before deploying to AWS. This requires Docker installed, because SAM will create a Docker container with the required dependencies and execute it locally.

Create a test event

Before we run a local test, we need to configure a test event. The sam init command created an events folder with a sample event file.

* Update the query string in the **events.json** file to match the expected event for our Lambda function.
  + We can leave **events.json** as is, and just update the query string with this value.

"queryStringParameters": {

"title": "Rush",

"year": "2013"

},

Test Lambda function locally

* Enter **sam local invoke** to invoke your function locally using a Docker container
  + We specify the name of the Lambda function and the events file:
* In the first part of the output, we see SAM building a container to locally deploy the function

PS C:\dev\sam\sam-movies-aci> **sam local invoke "MoviesAciFunction" -e events/event.json**

Invoking app.lambda\_handler (python3.12)

Local image is out of date and will be updated to the latest runtime. To skip this, pass in the parameter --skip-pull-image

Building image.............................................................................................................................................................................................................................................................

Using local image: public.ecr.aws/lambda/python:3.12-rapid-x86\_64.

Mounting C:\dev\sam\sam-movies-aci\.aws-sam\build\MoviesAciFunction as /var/task:ro,delegated, inside runtime container

START RequestId: 185c43c3-5851-4604-8d89-f731f75fbe6d Version: $LATEST

* Then we see the request executed, and the ultimately we see the results returned

{"statusCode": 200, "body": "{\"Item\": {\"year\": \"2013\", \"info\": {\"actors\": [\"Daniel Bruhl\", \"Chris Hemsworth\", \"Olivia Wilde\"], \"release\_date\": \"2013-09-02T00:00:00Z\", \"plot\": \"A re-creation of the merciless 1970s rivalry between Formula One rivals James Hunt and Niki Lauda.\", \"genres\": [\"Action\", \"Biography\", \"Drama\", \"Sport\"], \"image\_url\": \"http://ia.media-imdb.com/images/M/MV5BMTQyMDE0MTY0OV5BMl5BanBnXkFtZTcwMjI2OTI0OQ@@.\_V1\_SX400\_.jpg\", \"directors\": [\"Ron Howard\"], \"rating\": \"8.3\", \"rank\": \"2\", \"running\_time\_secs\": \"7380\"}, \"title\": \"Rush\"}, \"ResponseMetadata\": {\"RequestId\": \"T5F6M3SNMPHQC81OCKBB6M1LM3VV4KQNSO5AEMVJF66Q9ASUAAJG\", \"HTTPStatusCode\": 200, \"HTTPHeaders\": {\"server\": \"Server\", \"date\": \"Sat, 15 Mar 2025 20:54:50 GMT\", \"content-type\": \"application/x-amz-json-1.0\", \"content-length\": \"613\", \"connection\": \"keep-alive\", \"x-amzn-requestid\": \"T5F6M3SNMPHQC81OCKBB6M1LM3VV4KQNSO5AEMVJF66Q9ASUAAJG\", \"x-amz-crc32\": \"3565899189\"}, \"RetryAttempts\": 0}}", "headers": {"Content-Type": "application/json", "Access-Control-Allow-Origin": "\*"}}

#### Demo: Test API locally

We can also test the API invocation locally. SAM cli will start a local HTTP server and provide a URL for the API.

Start a local API

* Enter **sam local start-api** to start a local server
  + Note the URL for the API in red

PS C:\dev\sam\sam-movies-aci> **sam local start-api**

Initializing the lambda functions containers.

Local image is up-to-date

Using local image: public.ecr.aws/lambda/python:3.12-rapid-x86\_64.

Mounting C:\dev\sam\sam-movies-aci\.aws-sam\build\MoviesAciFunction as /var/task:ro,delegated, inside runtime container

Containers Initialization is done.

Mounting MoviesAciFunction at http://127.0.0.1:3000/movie [GET]

You can now browse to the above endpoints to invoke your functions. You do not need to restart/reload SAM CLI while working on your

functions, changes will be reflected instantly/automatically. If you used sam build before running local commands, you will need to

re-run sam build for the changes to be picked up. You only need to restart SAM CLI if you update your AWS SAM template

2025-03-16 10:58:13 WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.

\* **Running on http://127.0.0.1:3000**

2025-03-16 10:58:13 Press CTRL+C to quit

Use any HTTP client tool to test the API

We can use any HTTP client tool to test our API, using the URL above. For our example, we will keep it simple, and use curl, which is generally available anywhere.

* Open a new command window, since the original one is running the API server
* On the new shell, use curl to involve the api, with the URL forma of
  + The URL will have the format: ***<server URL>/<REST API resource>/<query string>***
  + **<server URL>** is the base URL of the server starter (in red above)
  + **<REST API resource>** is "**movie**", as defined in our SAM template
  + **<query string>** are usual **year** and **title** primary key for the Movies database

PS C:\dev\sam\sam-movies-aci> **curl.exe "http://127.0.0.1:3000/movie?title=Rush&year=2013"**

{"Item": {"year": "2013", "info": {"actors": ["Daniel Bruhl", "Chris Hemsworth", "Olivia Wilde"], "release\_date": "2013-09-02T00:00:00Z", "plot": "A re-creation of the merciless 1970s rivalry between Formula One rivals James Hunt and Niki Lauda.", "genres": ["Action", "Biography", "Drama", "Sport"], "image\_url": "http://ia.media-imdb.com/images/M/MV5BMTQyMDE0MTY0OV5BMl5BanBnXkFtZTcwMjI2OTI0OQ@@.\_V1\_SX400\_.jpg", "directors": ["Ron Howard"], "rating": "8.3", "rank": "2", "running\_time\_secs": "7380"}, "title": "Rush"}, "ResponseMetadata": {"RequestId": "5FNMPKDSATVVN51EU4P54J1IF7VV4KQNSO5AEMVJF66Q9ASUAAJG", "HTTPStatusCode": 200, "HTTPHeaders": {"server": "Server", "date": "Sun, 16 Mar 2025 15:16:07 GMT", "content-type": "application/x-amz-json-1.0", "content-length": "613", "connection": "keep-alive", "x-amzn-requestid": "5FNMPKDSATVVN51EU4P54J1IF7VV4KQNSO5AEMVJF66Q9ASUAAJG", "x-amz-crc32": "3565899189"}, "RetryAttempts": 0}}

* After we've completed our local tests, we can deploy the application to AWS

#### Demo: Deploy the application with sam deploy

* The first time we deploy, we will use "sam deploy -–guide", which will prompt for some deployment options, which are saved for future deployments
  + New deployments can simply call "sam deploy"
* Walk through the deployment prompts:
  + Most parameters we leave as default, and only change the following:
    - For "*MoviesAciFunction has no authentication. Is this okay*?", enter "**y**"

PS C:\dev\sam\sam-movies-aci> **sam deploy --guided**

Configuring SAM deploy

======================

Looking for config file [samconfig.toml] : Found

Reading default arguments : Success

Setting default arguments for 'sam deploy'

=========================================

Stack Name [sam-movies-aci]: **sam-movies-aci**

AWS Region [us-east-2]:

#Shows you resources changes to be deployed and require a 'Y' to initiate deploy

Confirm changes before deploy [Y/n]:

#SAM needs permission to be able to create roles to connect to the resources in your template

Allow SAM CLI IAM role creation [Y/n]:

#Preserves the state of previously provisioned resources when an operation fails

Disable rollback [y/N]:

MoviesAciFunction has no authentication. Is this okay? [y/N]: **y**

Save arguments to configuration file [Y/n]:

SAM configuration file [samconfig.toml]:

SAM configuration environment [default]:

* SAM then performs various steps (not shown below), and ultimately creates a CloudFormation change set
* The change set is shown, and we are asked to confirm
  + Answer "**y**" to proceed to deployment

CloudFormation stack changeset

------------------------------------------------------------------------------------------------------------------------

Operation LogicalResourceId ResourceType Replacement

------------------------------------------------------------------------------------------------------------------------

+ Add MoviesAciFunctionMoviesAciPerm AWS::Lambda::Permission N/A

issionProd

+ Add MoviesAciFunctionRole AWS::IAM::Role N/A

+ Add MoviesAciFunction AWS::Lambda::Function N/A

+ Add ServerlessRestApiDeployment8a4 AWS::ApiGateway::Deployment N/A

3422167

+ Add ServerlessRestApiProdStage AWS::ApiGateway::Stage N/A

+ Add ServerlessRestApi AWS::ApiGateway::RestApi N/A

-------------------------------------------------------------------------------------------------------------------------

Changeset created successfully. arn:aws:cloudformation:us-east-2:359520436611:changeSet/samcli-deploy1742139121/06f61598-3eb1-4bcb-8175-97660ae40739

Previewing CloudFormation changeset before deployment

======================================================

Deploy this changeset? [y/N]: **y**

* We then see typical CloudFormation messages, until the stack is created

2025-03-16 11:32:17 - Waiting for stack create/update to complete

CloudFormation events from stack operations (refresh every 5.0 seconds)

-----------------------------------------------------------------------------------------------------------------------------

ResourceStatus ResourceType LogicalResourceId ResourceStatusReason

-----------------------------------------------------------------------------------------------------------------------------

CREATE\_IN\_PROGRESS AWS::CloudFormation::Stack sam-movies User Initiated

CREATE\_IN\_PROGRESS AWS::IAM::Role MoviesAciFunctionRole -

CREATE\_IN\_PROGRESS AWS::IAM::Role MoviesAciFunctionRole Resource creation Initiated

CREATE\_COMPLETE AWS::IAM::Role MoviesAciFunctionRole -

CREATE\_IN\_PROGRESS AWS::Lambda::Function MoviesAciFunction -

CREATE\_IN\_PROGRESS AWS::Lambda::Function MoviesAciFunction Resource creation Initiated

CREATE\_IN\_PROGRESS - AWS::Lambda::Function MoviesAciFunction Eventual consistency check

CONFIGURATION\_COMPLETE initiated

CREATE\_IN\_PROGRESS AWS::ApiGateway::RestApi ServerlessRestApi -

CREATE\_IN\_PROGRESS AWS::ApiGateway::RestApi ServerlessRestApi Resource creation Initiated

CREATE\_COMPLETE AWS::ApiGateway::RestApi ServerlessRestApi -

CREATE\_IN\_PROGRESS AWS::ApiGateway::Deployment ServerlessRestApiDeployment8a4 -

3422167

CREATE\_IN\_PROGRESS AWS::Lambda::Permission MoviesAciFunctionMoviesAciPerm -

issionProd

CREATE\_IN\_PROGRESS AWS::Lambda::Permission MoviesAciFunctionMoviesAciPerm Resource creation Initiated

issionProd

CREATE\_IN\_PROGRESS AWS::ApiGateway::Deployment ServerlessRestApiDeployment8a4 Resource creation Initiated

3422167

CREATE\_COMPLETE AWS::Lambda::Permission MoviesAciFunctionMoviesAciPerm -

issionProd

CREATE\_COMPLETE AWS::ApiGateway::Deployment ServerlessRestApiDeployment8a4 -

3422167

CREATE\_COMPLETE AWS::Lambda::Function MoviesAciFunction -

CREATE\_IN\_PROGRESS AWS::ApiGateway::Stage ServerlessRestApiProdStage -

CREATE\_IN\_PROGRESS AWS::ApiGateway::Stage ServerlessRestApiProdStage Resource creation Initiated

CREATE\_COMPLETE AWS::ApiGateway::Stage ServerlessRestApiProdStage -

CREATE\_COMPLETE AWS::CloudFormation::Stack sam-movies -

---------------------------------------------------------------------------

CloudFormation outputs from deployed stack

---------------------------------------------------------------------------

Outputs

---------------------------------------------------------------------------

Key MoviesAciApi

Description API Gateway endpoint URL for Prod stage for Movies Aci function

Value https://0fzgqbj5h8.execute-api.us-east-2.amazonaws.com/Prod/movie/

Key MoviesAciFunctionIamRole

Description Implicit IAM Role created for Movies Aci function

Value arn:aws:iam::359520436611:role/sam-movies-MoviesAciFunctionRole-Ca6HpDoboNkz

Key MoviesAciFunction

Description Movies Aci Lambda Function ARN

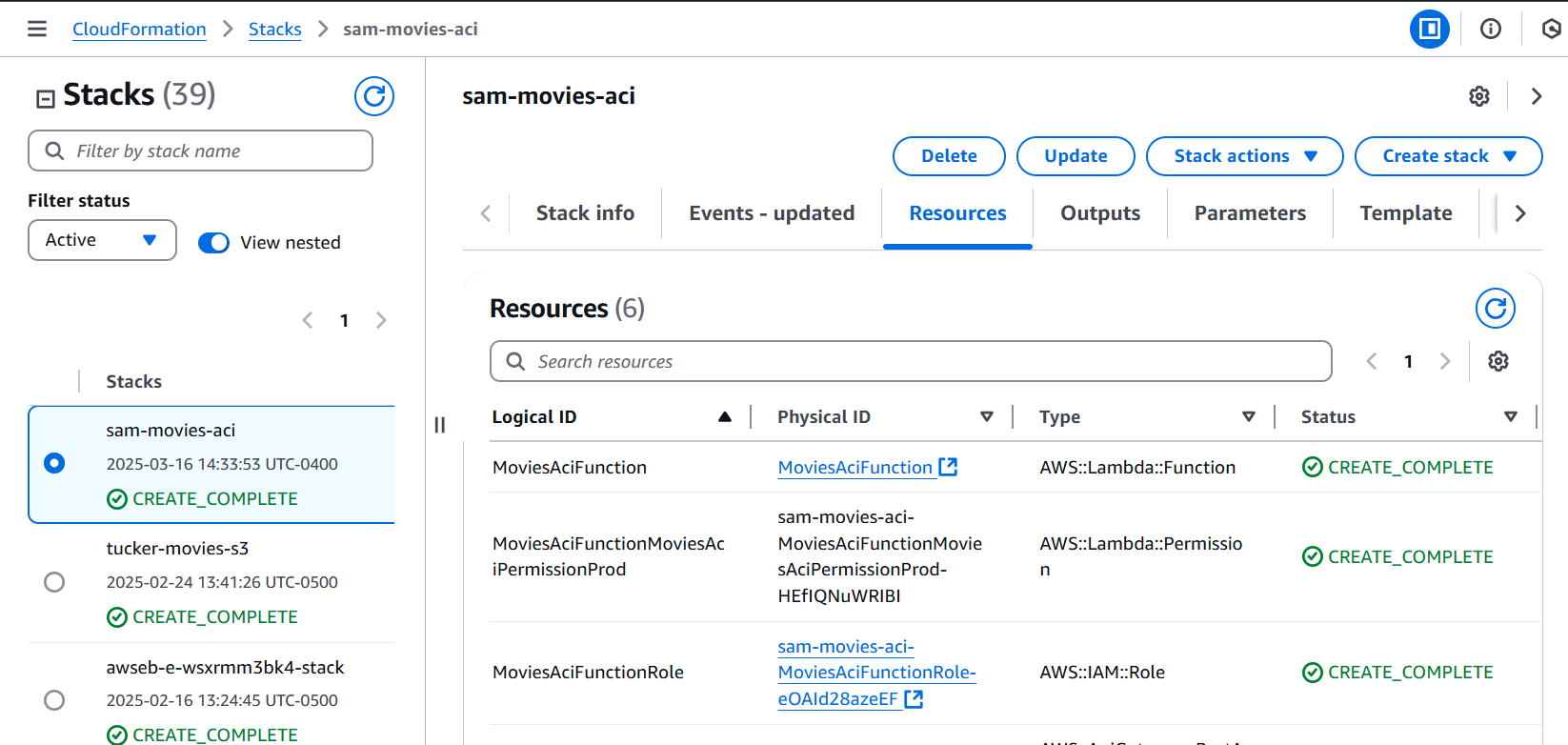
Value arn:aws:lambda:us-east-2:359520436611:function:MoviesAciFunction

---------------------------------------------------------------------------

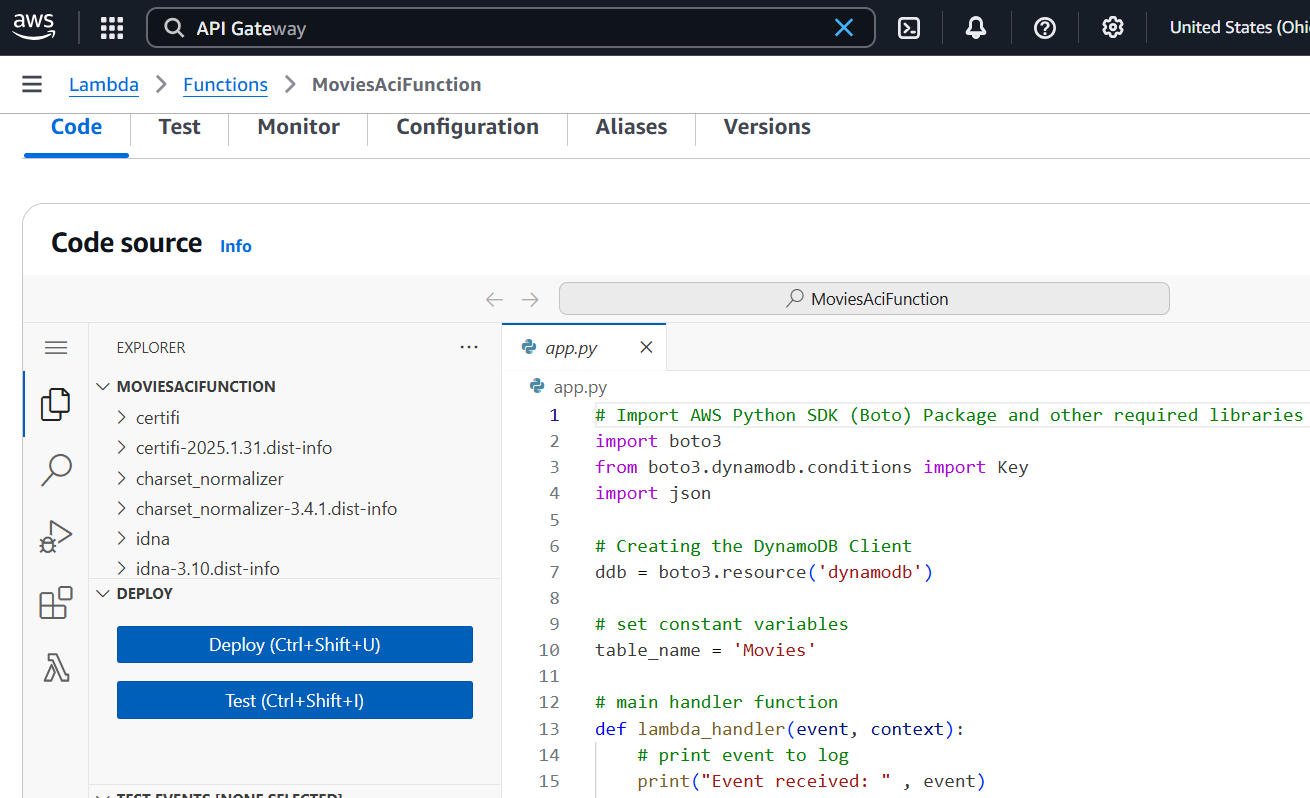
**Successfully created**/updated stack - sam-movies in us-east-2

#### Demo: Verify resources created

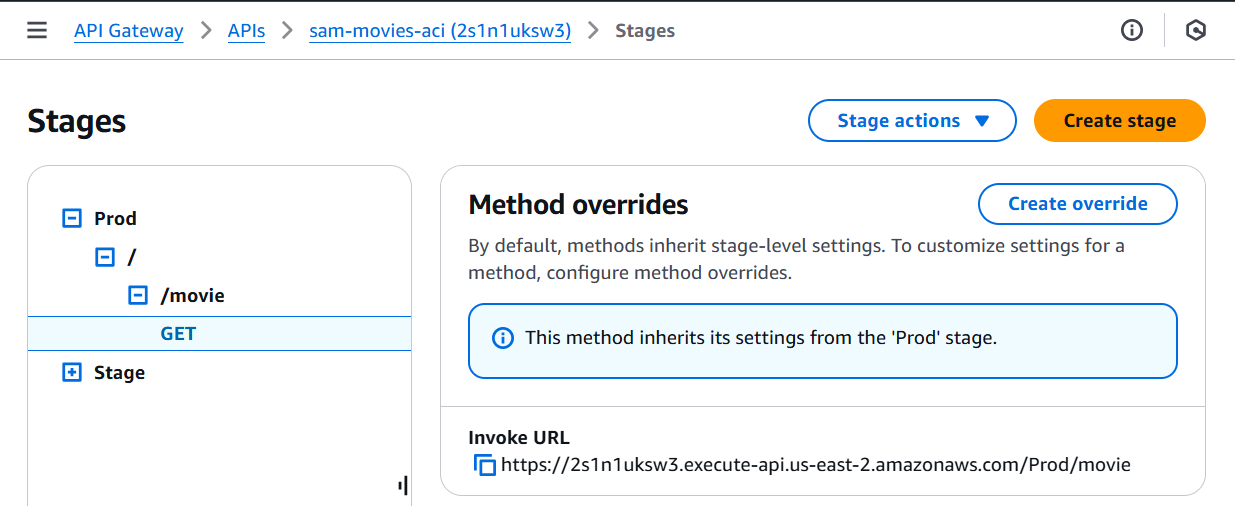
* Verify the **sam-movies-aci** CloudFormation template created



* Verify the **MoviesAciFunction** Lambda function created:



* Verify the **sam-movies-aci** API created:



* Copy the API URL from the API Gateway Stages panel, or from the output of CloudFormation
* Test the API in the browser:

