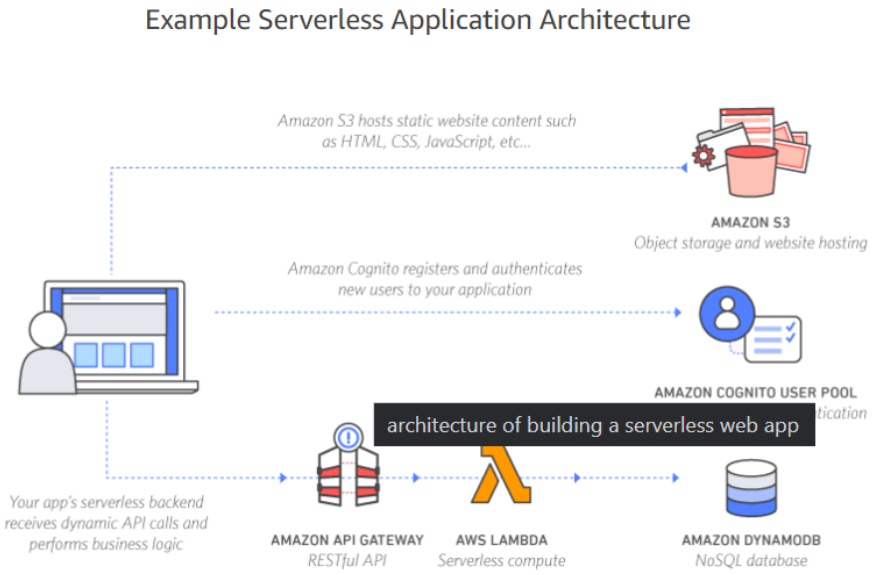
Building a serverless web application using AWS

Looking for new ways to deploy and release applications faster and more frequently, I found serverless computing is the way to go. For this purpose I used AWS which is a cloud-based service where the cloud provider manages the server. AWS dynamically allots compute storage and resources needed to execute each line of code.

In this blog I will demonstrate how I built and deployed my own dynamic, serverless web application by using several AWS services together. Each service is fully managed and does not require me to provision or manage servers. I configured them together and uploaded my application code to AWS Lambda, a serverless compute service.



Why build a serverless application?

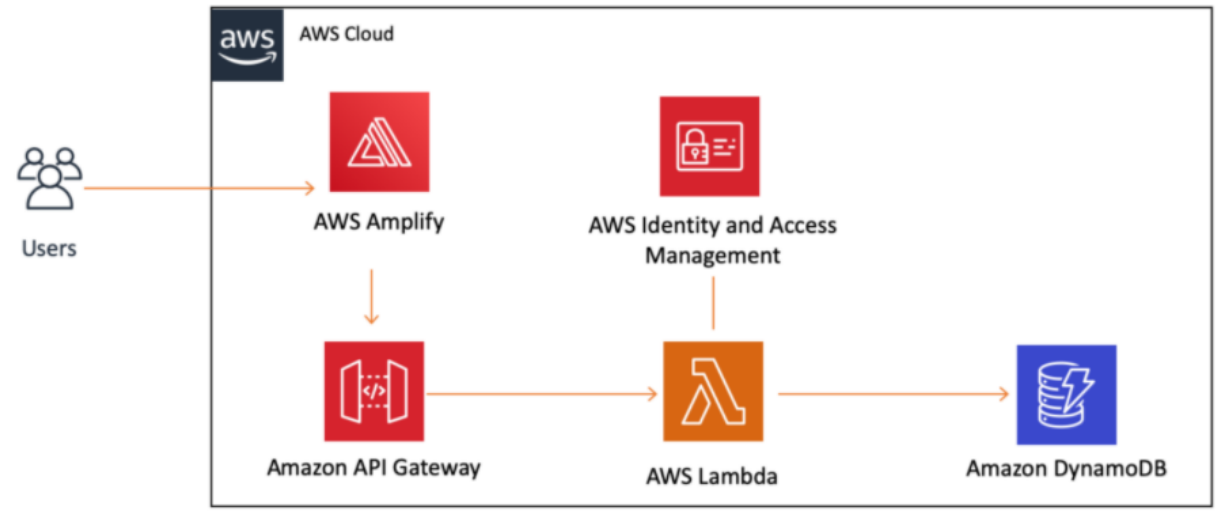
Building a serverless application allows me to focus on my application code instead of managing and operating infrastructure. I do not have to think about provisioning or configuring servers since AWS handles all of this for me.
The four main benefits are:

- No server management: There is no need to provision or maintain any servers. There is no software or runtime to install, maintain, or administer.
- Flexible scaling: My application can be scaled automatically or by adjusting its capacity through toggling the units of consumption rather than units of individual servers.
- High availability: Serverless applications have built-in availability and fault tolerance. I do not need to architect for these capabilities since the services running the application provide them by default.
- No idle capacity: I do not have to pay for idle capacity. There is no need to pre or over-provision capacity for things like compute and storage.

Application Architecture

My application uses AWS Amplify, Amazon API Gateway, AWS Lambda and Amazon DynamoDB. The diagram below provides a visual representation of the services I used and how they are connected.
Amplify Console provides continuous deployment and hosting of the static web resources including HTML, CSS, JavaScript, and image files which are loaded in the user's browser. JavaScript executed in the browser sends and receives data from a public backend API built using Lambda and API Gateway. Finally, DynamoDB provides a persistence lawyer where data can be stored by the API's Lambda function.

As we go through the steps, I will discuss the services in detail and link the resources to help you learn more in depth about them.



Create a Web APP

I used AWS Amplify Console to deploy the static resources for my web application. All of my static web content - including HTML, CSS, JavaScript, images and other files - will be hosted by AWS Amplify. I selected the Amplify service since it makes it straight forward to host and deploy static websites. The end users will access my site using the URL exposed by Amplify.

a) Creating a Web App with Amplify Console

Opened a notepad and saved it with the name of index.html. The following HTML was written in it. In a browser window, log onto Amplify Console and Deploy without Git provider was selected.

```
Markup

<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Hello World</title>
</head>

<body>
    Hello World
</body>
</html>
```

b) Testing the Web App

Under Domain Management, Copy and Paste the URL displayed in the form onto the browser

Building Serverless Function

In this segment I will be using the AWS Lambda service, a compute service that allows me to create serverless functions. These serverless functions are triggered based on specific event which will be defined in the code.

a) Create and Configure Lambda Function

Log onto the AWS Lambda Console, under “Function Code” enter the following Python code.

```python
# import the JSON utility package
import json
# define the handler function that
def lambda_handler(event, context)
# extract values from the event ob
    name = event['firstName'] +' '
# return a properly formatted JSON
    return {
    'statusCode': 200,
    'body': json.dumps('Hello from
    }
```

After configuring test events replace the JSON object with the following.

```json
{
"firstName": "Ada",
"lastName": "Lovelace"
}
```

b) Test Lambda Function

In order to check the working of the Lambda Function click the Test button at the top. Execution result: succeeded should be seen.

Test Event Name
HelloWorldTestEvent

Response

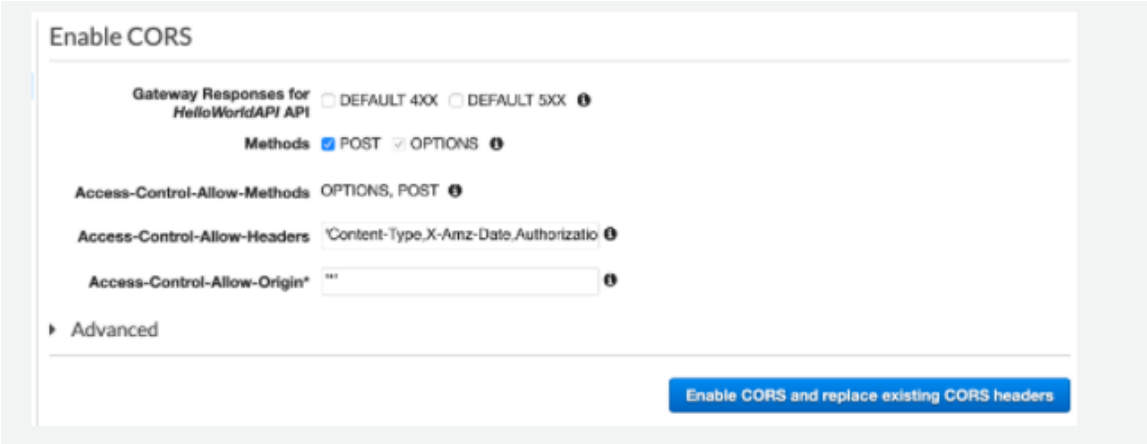
```
{
  "statusCode": 200,
  "body": "\"Hello from Lambda, Ada Lovelace\""
}
```

Linking Serverless Function to the Web App

In this section, I use Amazon API Gateway to create RESTful API that allows us to make calls to the Lambda Function from a web client. API Gateway acts as a middle layer between the HTML client and the serverless back end.

a) Create a new Resource and Method

Select the Post Method for API Gateway Console. Type in the created Lambda function in the “function” field.
Leave the Post checkbox selected and click “Enable CORS and replace existing CORS headers”.



b) Deploy and Validate API

Under deploy API, enter the dev for the Stage Name. Copy and save the URL next to “Invoke URL”.
Enter the following in the Request Body field:

```javascript
{
    "firstName":"Grace",
    "lastName":"Hopper"
}
```

You should see a response with Code 200.

Creating a Data Table

I created a table using Amazon DynamoDB. DynamoDB is a key-value database service, it has consistent performance at any scale and there are no servers to manage when using it. Additionally, I use the AWS Identity and Access Management IAM service to securely give my services the required permissions to interact with each other.

a) Adding IAM Policy to Lambda Function

After creating a table, edit our Lambda function to be able to write data to it. On the AWS Lambda Console, under Permissions click on Add inline policy.
Enter the following policy in the text area, taking care to replace the tables’s ARN in the resource field.

```json
{
"Version": "2012-10-17",
"Statement": [
    {
        "Sid": "VisualEditor0",
        "Effect": "Allow",
        "Action": [
            "dynamodb:PutItem",
            "dynamodb:DeleteItem",
            "dynamodb:GetItem",
            "dynamodb:Scan",
            "dynamodb:Query",
            "dynamodb:UpdateItem"
        ],
        "Resource": "YOUR-TABLE-ARN"
    }
    ]
}
```

b) Modifying Lambda Function to write to DynamoDB table

On the code tab, select our function from navigation panel and replace the code with the following:

```python
# import the json utility package since we will be
import json
# import the AWS SDK (for Python the package name
import boto3
# import two packages to help us with dates and da
from time import gmtime, strftime

# create a DynamoDB object using the AWS SDK
dynamodb = boto3.resource('dynamodb')
# use the DynamoDB object to select our table
table = dynamodb.Table('HelloWorldDatabase')
# store the current time in a human readable forma
now = strftime("%a, %d %b %Y %H:%M:%S +0000", gmti

# define the handler function that the Lambda serv
def lambda_handler(event, context):
# extract values from the event object we got from
    name = event['firstName'] +' '+ event['lastNam
# write name and time to the DynamoDB table using
    response = table.put_item(
        Item={
            'ID': name,
            'LatestGreetingTime':now
            })
# return a properly formatted JSON object
    return {
        'statusCode': 200,
        'body': json.dumps('Hello from Lambda, ' +
    }
```

c) Testing the Changes

Click the Test button, Execution result: succeeded should be seen.
Open DynamoDB Console, on the items tab everytime the lambda function executes, My DynamoDB table will be updated.

Adding Interactivity to the Web App

I updated the static website to invoke the REST API created in the previous section. This will add the ability to display the text based on what the user inputs.

a) Updating the Web App with Amplify Console

Replace the code in the index.html file with the following:

Markup

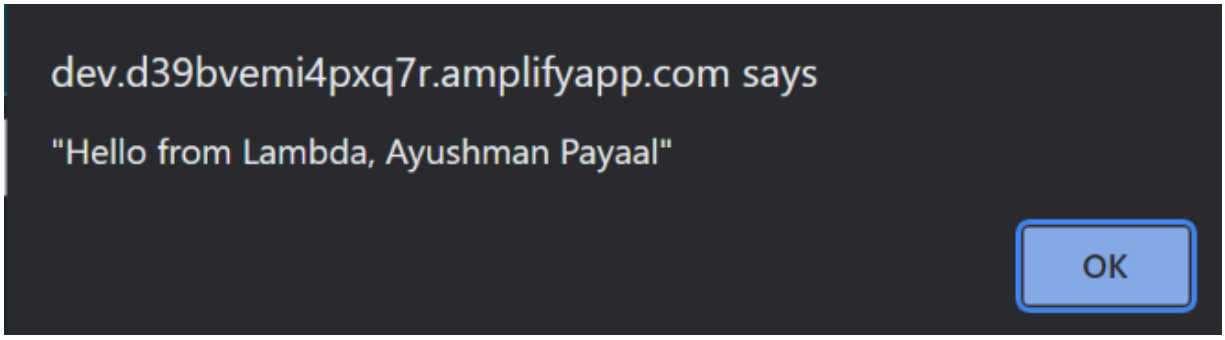
```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Hello World</title>
    <!-- Add some CSS to change client UI -->
    <style>
    body {
        background-color: #232F3E;
        }
    label, button {
        color: #FF9900;
        font-family: Arial, Helvetica, sans-serif;
        font-size: 20px;
        margin-left: 40px;
        }
     input {
        color: #232F3E;
        font-family: Arial, Helvetica, sans-serif;
        font-size: 20px;
        margin-left: 20px;
        }
    </style>
    <script>
        // define the callAPI function that takes a f
        var callAPI = (firstName,lastName)=>{
            // instantiate a headers object
            var myHeaders = new Headers();
            // add content type header to object
            myHeaders.append("Content-Type", "applica
            // using built in JSON utility package tu
            var raw = JSON.stringify({"firstName":fir
            // create a JSON object with parameters f
            var requestOptions = {
                method: 'POST',
                headers: myHeaders,
                body: raw,
                redirect: 'follow'
```

```
39              };
40              // make API call with parameters and use p
41              fetch("YOUR-API-INVOKE-URL", requestOptior
42              .then(response => response.text())
43              .then(result => alert(JSON.parse(result).l
44              .catch(error => console.log('error', erro
45          }
46      </script>
47  </head>
48  <body>
49      <form>
50          <label>First Name :</label>
51          <input type="text" id="fName">
52          <label>Last Name :</label>
53          <input type="text" id="lName">
54          <!-- set button onClick method to call functi
55          <button type="button" onclick="callAPI(docume
56      </form>
57  </body>
58  </html>
```

Making sure to add the API invoke URL on line 41. Open Amplify Console and select the ZIP file created.

b) Test Updated Web App

Under Domain click on the URL, updated web app should load on the browser, fill in name field and click the Call API button.



We now have a working web app deployed by Amplify Console that can call a Lambda function via API Gateway. When a user clicks on a button in the web app, it makes a call to our API, which triggers our Lambda function. Our Lambda function writes to a database and returns a message to our client via API Gateway. All permissions are managed by IAM.

For those looking to build event-based apps quickly and efficiently, serverless computing is the way to conserve resources, increase efficiency, and boost productivity.