SERVERLESS WEBAPP HOSTING ON AWS

MAJOR PROJECT

Prepared by:

Viransh Bhardwaj

Yukta Rajesh Sharma

Gaddala Greeshma Devi

Sarvesh Lath

Vikas Reddy

Cloud Computing Batch July 2022

MENTOR: SHRAVANI SHETTY.

***Web Application Hosting on AWS***

Web Apps have always been required to be hosted on servers and are assigned an IP that allows the users anywhere in the world to access the webapps using that IP. PuTTY allows the integration of the instance IP with the S3 Bucket Objects and launches an Apache Server to host the webapp.

Traditional on-premises web architectures require complex solutions and accurate reserved capacity forecast in order to ensure reliability. Dense peak traffic periods and wild swings in traffic patterns result in low utilization rates of expensive hardware. This yields high operating costs to maintain idle hardware, and an inefficient use of capital for underused hardware.

Amazon Web Services (AWS) provides a reliable, scalable, secure, and highly performing infrastructure for the most demanding web applications. This infrastructure matches IT costs with customer traffic patterns in near-real time.

This whitepaper is meant for IT Managers and System Architects who want to understand how to run traditional web architectures in the cloud to achieve elasticity, scalability, and reliability.

***Static website:*** A static website contains [Web pages](https://techterms.com/definition/webpage) with fixed content. Each page is coded in [HTML](https://techterms.com/definition/html) and displays the same information to every visitor. Static sites are the most basic type of website and are the easiest to create. Unlike [dynamic websites](https://techterms.com/definition/dynamicwebsite), they do not require any Web programming or [database](https://techterms.com/definition/database) design. A static site can be built by simply creating a few HTML pages and publishing them to a Web server.

Since static Web pages contain fixed code, the content of each page does not change unless it is manually updated by the [webmaster](https://techterms.com/definition/webmaster). This works well for small websites, but it can make large sites with hundreds or thousands of pages difficult to maintain. Therefore, larger websites typically use dynamic pages, which can be updated by simply modifying a database record. Static sites that contain a lot of pages are often designed using [templates](https://techterms.com/definition/template). This makes it possible to update several pages at once, and also helps provide a consistent layout throughout the site.

AWS LAMBDA SERVICE

Lambda falls under “Compute” service in AWS (Amazon Web Services). Using Lambda, we can code without provisioning or managing servers. Lambda automatically runs our code without requiring us to provision or manage servers. We just need to write the code and upload it to the Lambda Function. Lambda executes the code only when needed. It grows automatically supporting from a few requests to thousands of requests. We are charged for every 100ms our code executes and the number of times it is triggered. We are charged only for the compute time our code consumes and not charged when the code is not being executed.

Everything to know about Amazon Dynamo DB

Amazon DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability. DynamoDB lets you offload the administrative burdens of operating and scaling a distributed database so that you don't have to worry about hardware provisioning, setup and configuration, replication, software patching, or cluster scaling. DynamoDB also offers encryption at rest, which eliminates the operational burden and complexity involved in protecting sensitive data.

With DynamoDB, you can create database tables that can store and retrieve any amount of data and serve any level of request traffic. You can scale up or scale down your tables' throughput capacity without downtime or performance degradation. You can use the AWS Management Console to monitor resource utilization and performance metrics. DynamoDB provides on-demand backup capability. It allows you to create full backups of your tables for long-term retention and archival for regulatory compliance needs.

High availability and Durability:-

DynamoDB automatically spreads the data and traffic for your tables over a sufficient number of servers to handle your throughput and storage requirements, while maintaining consistent and fast performance. All of your data is stored on solid-state disks (SSDs) and is automatically replicated across multiple Availability Zones in an AWS Region, providing built-in high availability and data durability. You can use global tables to keep DynamoDB tables in sync across AWS Regions.

Core components of Amazon Dynamo DB:-

In DynamoDB, tables, items, and attributes are the core components that you work with. A *table* is a collection of *items*, and each item is a collection of *attributes*. DynamoDB uses primary keys to uniquely identify each item in a table and secondary indexes to provide more querying flexibility. You can use DynamoDB Streams to capture data modification events in DynamoDB tables.

Basic components of Dynamo DB:-

**Tables**–Similar to other data base systems, Dynamo DB stores data in tables. A *table* is a collection of data. For example, see the example table called *People* that you could use to store personal contact information about friends, family, or anyone else of interest. You could also have a *Cars* table to store information about vehicles that people drive.

**Items**–Eachtablecontainszeroormoreitems.An*item*isagroupofattributesthatisuniquely identifiable among all of the other items. In a *People* table, each item represents a person. For a *Cars* table, each item represents one vehicle. Items in DynamoDB are similar in many ways to rows, records, or tuples in other database systems. In DynamoDB, there is no limit to the number of items you can store in a table.

**Attributes** – Each item is composed of one or more attributes. An *attribute* is a fundamental data element, something that does not need to be broken down any further. For example, an item in a *People* table contains attributes called *PersonID*, *LastName*, *FirstName*, and so on. For a *Department* table, an item might have attributes such as *DepartmentID*, *Name*, *Manager*, and so on. Attributes in DynamoDB are similar in many ways to fields or columns in other database systems.

Dynamo DB streams: -

DynamoDB Streams is an optional feature that captures data modification events in DynamoDB tables. The data about these events appear in the stream in near-real time, and in the order that the events occurred.

Each event is represented by a *stream record*. If you enable a stream on a table, DynamoDB Streams writes a stream record whenever one of the following events occurs:

* A new item is added to the table: The stream captures an image of the entire item, including all of its attributes.
* An item is updated: The stream captures the "before" and" after " image of any attributes that were modified in the item.
* An item is deleted from the table: The stream captures an image of the entire item before it was deleted.

Each stream record also contains the name of the table, the event timestamp, and other metadata. Stream records have a lifetime of 24 hours; after that, they are automatically removed from the stream.

You can use DynamoDB Streams together with AWS Lambda to create a *trigger*—code that runs automatically whenever an event of interest appears in a stream. For example, consider a *Customers* table that contains customer information for a company. Suppose that you want to send a "welcome" email to each new customer. You could enable a stream on that table, and then associate the stream with a Lambda function. The Lambda function would run whenever a new stream record appears, but only process new items added to the *Customers* table. For any item that has an Email Address attribute, the Lambda function would invoke Amazon Simple Email Service (Amazon SES) to send an email to that address.

Diagram

Description automatically generated

Dynamo DB API:-

To work with Amazon DynamoDB, your application must use a few simple API operations. The following is a summary of these operations, organized by category.

**Topics**

* **Controlplane**
* **Data plane**
* **DynamoDBStreams**
* **Transactions**

**What is Amazon API Gateway?**

Amazon API Gateway is an AWS service for creating, publishing, maintaining, monitoring, and securing REST, HTTP, and WebSocket APIs at any scale. API developers can create APIs that access AWS or other web services, as well as data stored in the AWS Cloud. As an API Gateway API developer, you can create APIs for use in your own client applications. Or you can make your APIs available to third-party app developers. For more information, see the section called “Who uses API Gateway?”

API Gateway creates RESTful APIs that:

• Are HTTP-based.

• Enable stateless client-server communication.

• Implement standard HTTP methods such as GET, POST, PUT, PATCH, and DELETE.

For more information about API Gateway REST APIs and HTTP APIs, see the section called “Choosing between REST APIs and HTTP APIs ” , Working with HTTP APIs , the section called “Use API Gateway to create REST APIs” , and the section called “Create and configure”

API Gateway creates WebSocket APIs that:

• Adhere to the WebSocket protocol, which enables stateful, full-duplex communication between client and server.

• Route incoming messages based on message content.

For more information about API Gateway WebSocket APIs, see the section called “Use API Gateway to create WebSocket APIs” and the section called “About WebSocket APIs” .

**Topics**

• Architecture of API Gateway

• Features of API Gateway

• API Gateway use cases

• Accessing API Gateway

• Part of AWS serverless infrastructure

• How to get started with Amazon API Gateway

• Amazon API Gateway concepts

**Architecture of API Gateway**

The following diagram shows API Gateway architecture.

****

This diagram illustrates how the APIs you build in Amazon API Gateway provide you or your developer customers with an integrated and consistent developer experience for building AWS serverless applications. API Gateway handles all the tasks involved in accepting and processing up to hundreds of thousands of concurrent API calls. These tasks include traffic management, authorization and access control, monitoring, and API version management.

API Gateway acts as a "front door" for applications to access data, business logic, or functionality from your backend services, such as workloads running on Amazon Elastic Compute Cloud (Amazon EC2), code running on AWS Lambda, any web application, or real-time communication applications.

**Features of API Gateway:**

• Powerful, flexible authentication mechanisms, such as AWS Identity and Access Management policies, Lambda authorizer functions, and Amazon Cognito user pools.

CloudWatch access logging and execution logging, including the ability to set alarms. For more information, see the section called “CloudWatch metrics” (p. 618) and the section called “Metrics”.

• Ability to use AWS CloudFormation templates to enable API creation. For more information, see Amazon API Gateway Resource Types Reference and Amazon API Gateway V2 Resource Types Reference.

**API Gateway use cases:**

Topics

• Use API Gateway to create REST APIs

• Use API Gateway to create HTTP APIs

• Use API Gateway to create WebSocket APIs

• Who uses API Gateway?

An API Gateway REST API is made up of resources and methods. A resource is a logical entity that an app can access through a resource path. A method corresponds to a REST API request that is submitted by the user of your API and the response returned to the user. For example, /incomes could be the path of a resource representing the income of the app user. A resource can have one or more operations that are defined by appropriate HTTP verbs such as GET, POST, PUT, PATCH, and DELETE. A combination of a resource path and an operation identifies a method of the API. For example, a POST /incomes method could add an income earned by the caller, and a GET / expenses method could query the reported expenses incurred by the caller. The app doesn't need to know where the requested data is stored and fetched from on the backend. In API Gateway REST APIs, the frontend is encapsulated by method requests and method responses. The API interfaces with the backend by means of integration requests and integration responses. For example, with DynamoDB as the backend, the API developer sets up the integration request to forward the incoming method request to the chosen backend. The setup includes specifications of an appropriate DynamoDB action, required IAM role and policies, and required input data transformation. The backend returns the result to API Gateway as an integration response.

HTTP APIs enable you to create RESTful APIs with lower latency and lower cost than REST APIs. You can use HTTP APIs to send requests to AWS Lambda functions or to any publicly routable HTTP endpoint. For example, you can create an HTTP API that integrates with a Lambda function on the backend. When a client calls your API, API Gateway sends the request to the Lambda function and returns the function's response to the client. HTTP APIs support OpenID Connect and OAuth 2.0 authorization. They come with built-in support for cross-origin resource sharing (CORS) and automatic deployments.

In a WebSocket API, the client and the server can both send messages to each other at any time. Backend servers can easily push data to connected users and devices, avoiding the need to implement complex polling mechanisms. For example, you could build a serverless application using an API Gateway WebSocket API and AWS Lambda to send and receive messages to and from individual users or groups of users in a chat room. Or you could invoke backend services such as AWS Lambda, Amazon Kinesis, or an HTTP endpoint based on message content. You can use API Gateway WebSocket APIs to build secure, real-time communication applications without having to provision or manage any servers to manage connections or large-scale data exchanges.

Targeted use cases include real-time applications such as the following:

• Chat applications • Real-time dashboards such as stock tickers • Real-time alerts and notifications

**Who uses API Gateway?**

There are two kinds of developers who use API Gateway: API developers and app developers. An API developer creates and deploys an API to enable the required functionality in API Gateway. The API developer must be an IAM user in the AWS account that owns the API. An app developer builds a functioning application to call AWS services by invoking a WebSocket or REST API created by an API developer in API Gateway. The app developer is the customer of the API developer. The app developer doesn't need to have an AWS account, provided that the API either doesn't require IAM permissions or supports authorization of users through third-party federated identity providers supported by Amazon Cognito user pool identity federation. Such identity providers include Amazon, Amazon Cognito user pools, Facebook, and Google.

**Accessing API Gateway**

• AWS Management Console – The AWS Management Console provides a web interface for creating and managing APIs. After you complete the steps in Prerequisites), you can access the API Gateway console at https://console.aws.amazon.com/apigateway.

• AWS SDKs – If you're using a programming language that AWS provides an SDK for, you can use an SDK to access API Gateway. SDKs simplify authentication, integrate easily with your development environment, and provide access to API Gateway commands. For more information, see Tools for Amazon Web Services.

• API Gateway V1 and V2 APIs – If you're using a programming language that an SDK isn't available for, see the Amazon API Gateway Version 1 API Reference and Amazon API Gateway Version 2 API Reference.

• AWS Command Line Interface – For more information, see Getting Set Up with the AWS Command Line Interface in the AWS Command Line Interface User Guide.

• AWS Tools for Windows PowerShell – For more information, see Setting Up the AWS Tools for Windows PowerShell in the AWS Tools for Windows PowerShell User Guide.

**Part of AWS serverless infrastructure**

Together with AWS Lambda, API Gateway forms the app-facing part of the AWS serverless infrastructure.

For an app to call publicly available AWS services, you can use Lambda to interact with required services and expose Lambda functions through API methods in API Gateway. AWS Lambda runs your code on a highly available computing infrastructure. It performs the necessary execution and administration of computing resources. To enable serverless applications, API Gateway supports streamlined proxy integrations () with AWS Lambda and HTTP endpoints.

**What is Amazon Cognito?**

Amazon Cognito provides authentication, authorization, and user management for your web and mobile apps. Your users can sign in directly with a user name and password, or through a third party such as Facebook, Amazon, Google or Apple. The two main components of Amazon Cognito are user pools and identity pools. User pools are user directories that provide sign-up and sign-in options for your app users. Identity pools enable you to grant your users access to other AWS services. You can use identity pools and user pools separately or together.

**An Amazon Cognito user pool and identity pool used together**

See the diagram for a common Amazon Cognito scenario. Here the goal is to authenticate your user, and then grant your user access to another AWS service.

1. In the first step your app user signs in through a user pool and receives user pool tokens after a successful authentication.

2. Next, your app exchanges the user pool tokens for AWS credentials through an identity pool.

3. Finally, your app user can then use those AWS credentials to access other AWS services such as Amazon S3 or DynamoDB.

Diagram

Description automatically generated

**Features of Amazon Cognito**

**User pools**

A user pool is a user directory in Amazon Cognito. With a user pool, your users can sign into your web or mobile app through Amazon Cognito or federate through a third-party identity provider (IdP). Whether your users sign in directly or through a third party, all members of the user pool have a directory profile that you can access through an SDK.

**User pools provide:**

• Sign-up and sign-in services.

• A built-in, customizable web UI to sign in users.

• Social sign-in with Facebook, Google, Login with Amazon, and Sign in with Apple, and through SAML and OIDC identity providers from your user pool.

• User directory management and user profiles.

• Security features such as multi-factor authentication (MFA), checks for compromised credentials, account takeover protection, and phone and email verification.

• Customized workflows and user migration through AWS Lambda triggers.

**Common Amazon Cognito scenarios**

This topic describes six common scenarios for using Amazon Cognito.

The two main components of Amazon Cognito are user pools and identity pools. User pools are user directories that provide sign-up and sign-in options for your web and mobile app users. Identity pools provide AWS credentials to grant your users access to other AWS services.

A user pool is a user directory in Amazon Cognito. Your app users can sign in either directly through a user pool or federate through a third-party identity provider (IdP). The user pool manages the overhead of handling the tokens that are returned from social sign-in through Facebook, Google, Amazon, and Apple, and from OpenID Connect (OIDC) and SAML IdPs. Whether your users sign in directly or through a third party, all members of the user pool have a directory profile that you can access through an SDK.

With an identity pool, your users can obtain temporary AWS credentials to access AWS services, such as Amazon S3 and DynamoDB. Identity pools support anonymous guest users, as well as federation through third-party IdPs.

**Integrating Amazon Cognito with web and mobile apps**

When new users discover your app, or when existing users return to it, their first task is to sign up or sign in. When you integrate Amazon Cognito with your client code, you connect your app to AWS resources that aid authentication and authorization workflows. For example, your app uses the Amazon Cognito API to create new users in your user pool, retrieve user pool tokens, and obtain temporary credentials from your identity pool. To integrate Amazon Cognito with your web or mobile app, use AWS SDKs and libraries.

Although Amazon Cognito offers visual tools such as AWS Management Console integration and the hosted UI, AWS has designed the service to work with your app code. You can only configure certain components of Amazon Cognito with the API or the AWS Command Line Interface. For example, you can only register a user for time-based one-time password (TOTP) multi-factor authentication (MFA) with a process that starts with Associate Software Token. Before you use Amazon Cognito authentication and authorization, choose an app platform, and prepare your code to integrate with the service.

**Conclusion**

The Amazon Cognito service is useful when an app developer doesn’t have the time or resources to invest in building a login page UI and maintain user credentials in a database. It also provides several features such as MFA (Multi-factor authentication), OTPs, prompts fingerprints or security questions. Phone numbers can be validated too.

You are provided with an SDK – Amazon Cognito SDK where with only a few lines of code, you can set up a working user log-in page for your app. If your app already has a solution for user authentication, you can also migrate to Amazon Cognito with only a few steps. You can also prompt the users to sign up through Google, Facebook, Amazon, and other identity providers also.

With all these benefits, the cost of the first 50000 MAF (Monthly active users is free). And it can scale itself as the user pool expands for a minimal amount per user after the free tier limit. So, this AWS service is very ideal for app developers with budget constraints and if they want their app to get going as soon as possible.

***STEP 1***

1. Create a Cloud9 instance

Graphical user interface, website

Description automatically generated

1. Graphical user interface, text, application, email

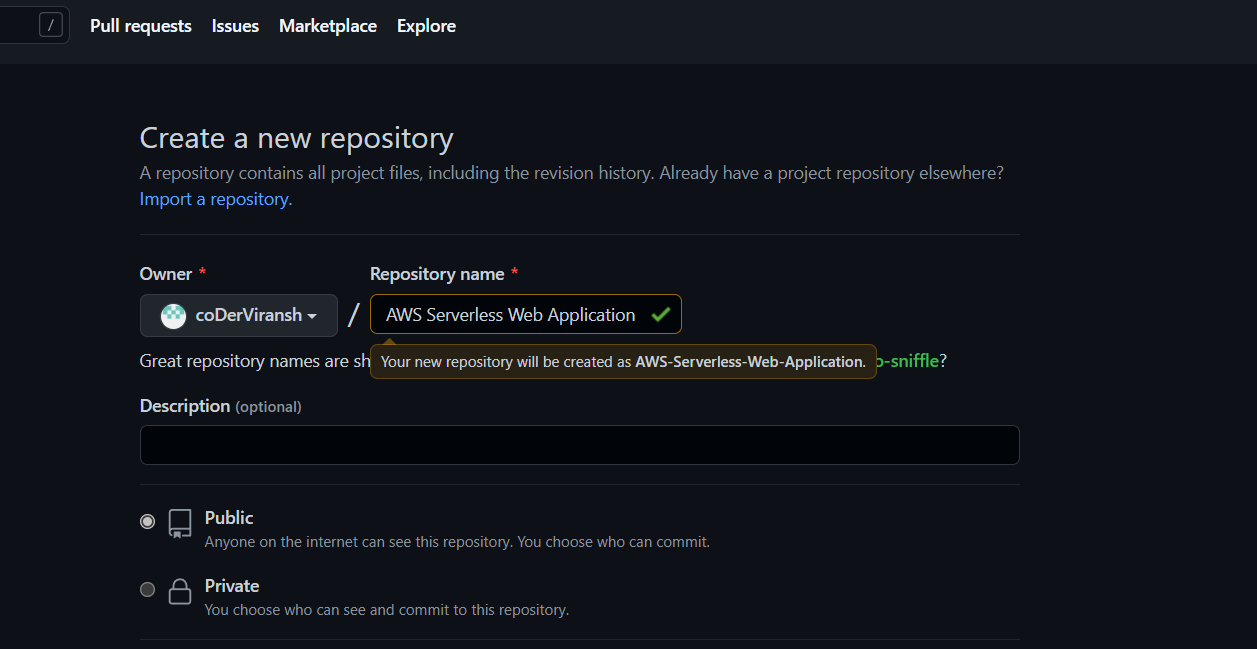
   Description automatically generated
2. Graphical user interface, text, application, email, Teams

   Description automatically generated
3. Create the environment and open github.

Graphical user interface, text, application

Description automatically generated

1. Create a repository in github and include a README File in the repository.

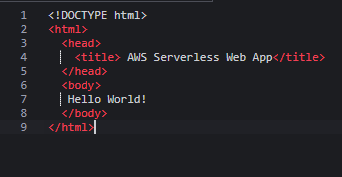


1. Click create the repository.
2. Copy the URL to the repository created to connect a cloud9 instance.
3. Open cloud9 environment and delete the README file.
4. Create a new terminal and perform the following commands.

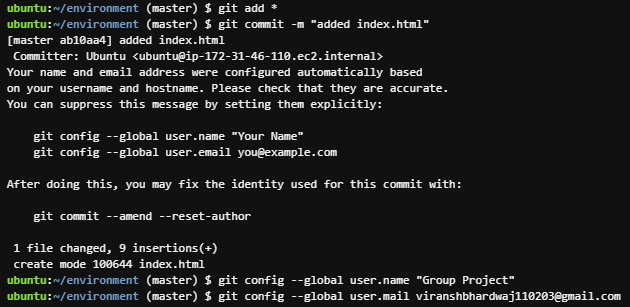
Text

Description automatically generated

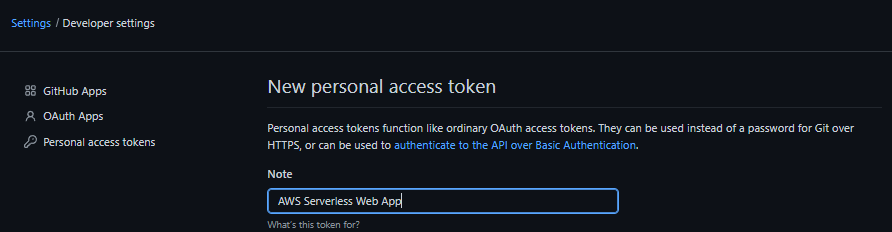
1. Create a new HTML file as follows and save the file as “index.html’.



1. Perform the following commands in the ubuntu terminal in cloud9



1. Generate a token from github setting>developer options> personal access tokens.



1. A screenshot of a computer screen

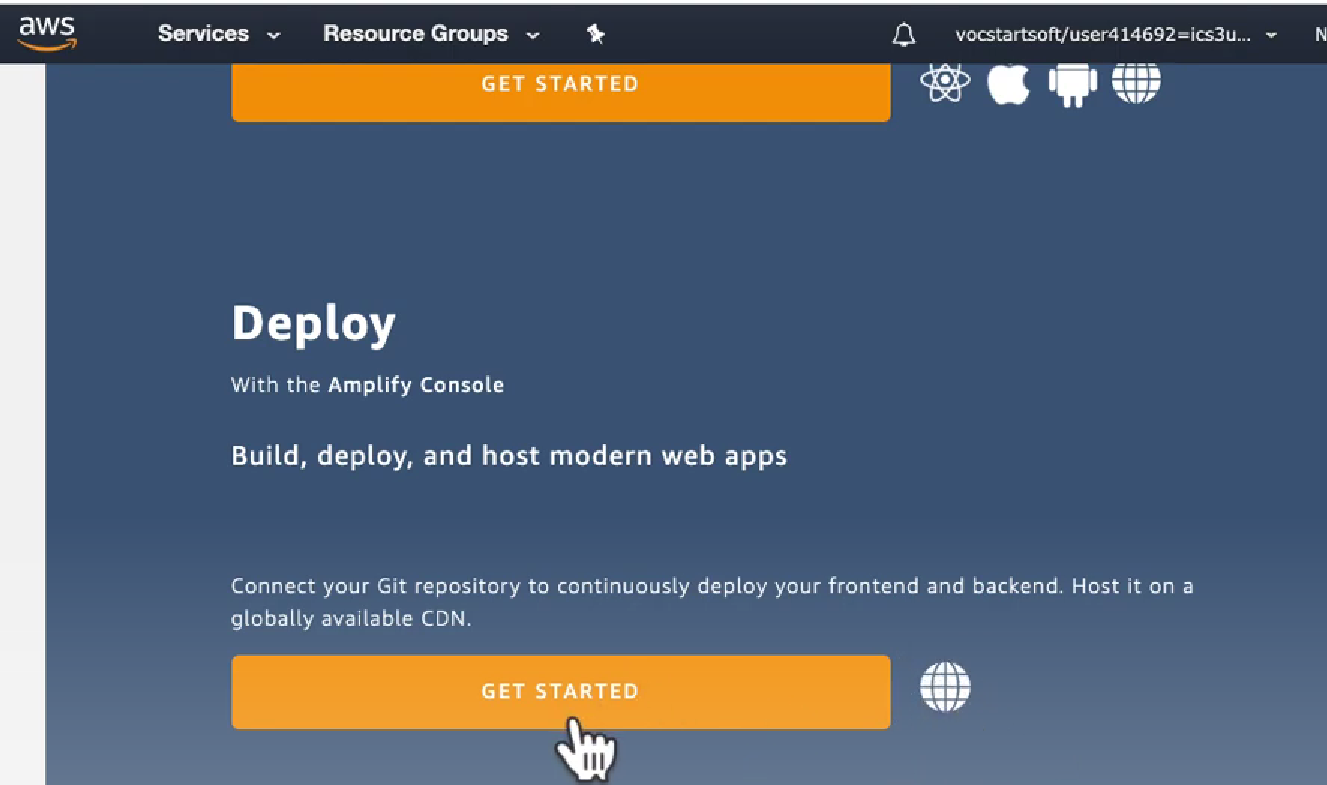
   Description automatically generated
2. Furthermore, perform the following code.

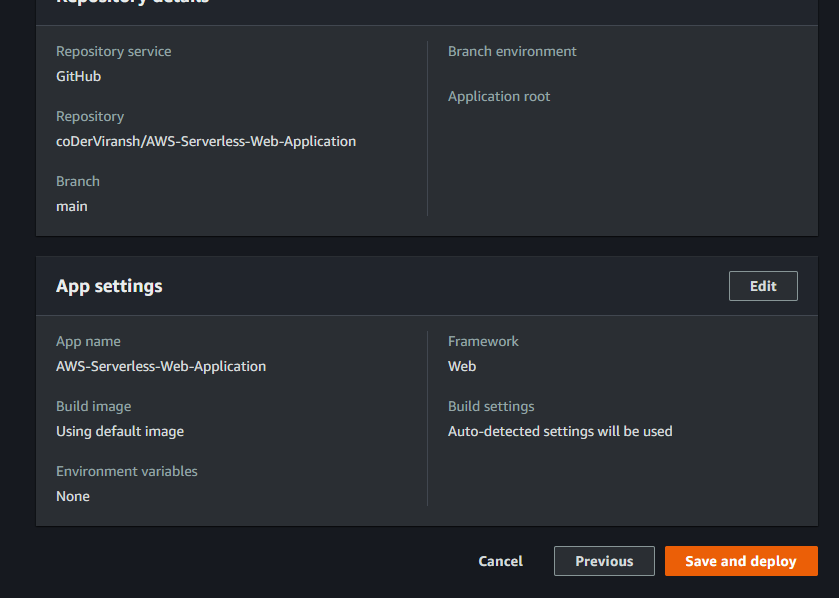


1. Refresh your github repository to have the “index.html” file

Graphical user interface, application, Teams

Description automatically generated

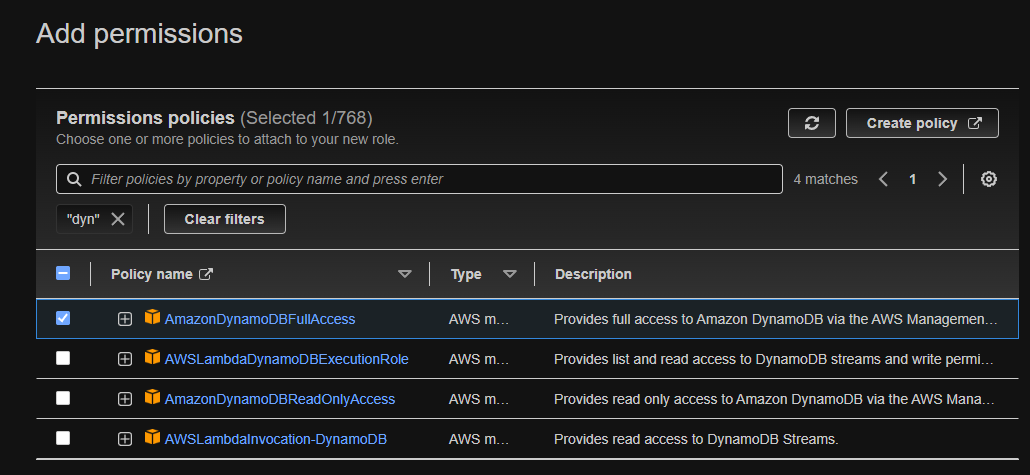
1. Open AWS Amplify console and chose the following option.
2. 
3. Connect to github repository.
4. Graphical user interface, text

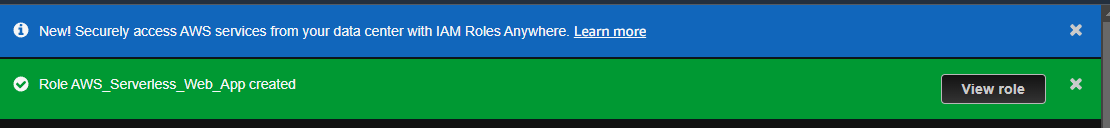
   Description automatically generated
5. Save and deploy without changing any other settings.
6. 
7. A picture containing icon

   Description automatically generated
8. Click on the URL
9. Graphical user interface, text, application, Word

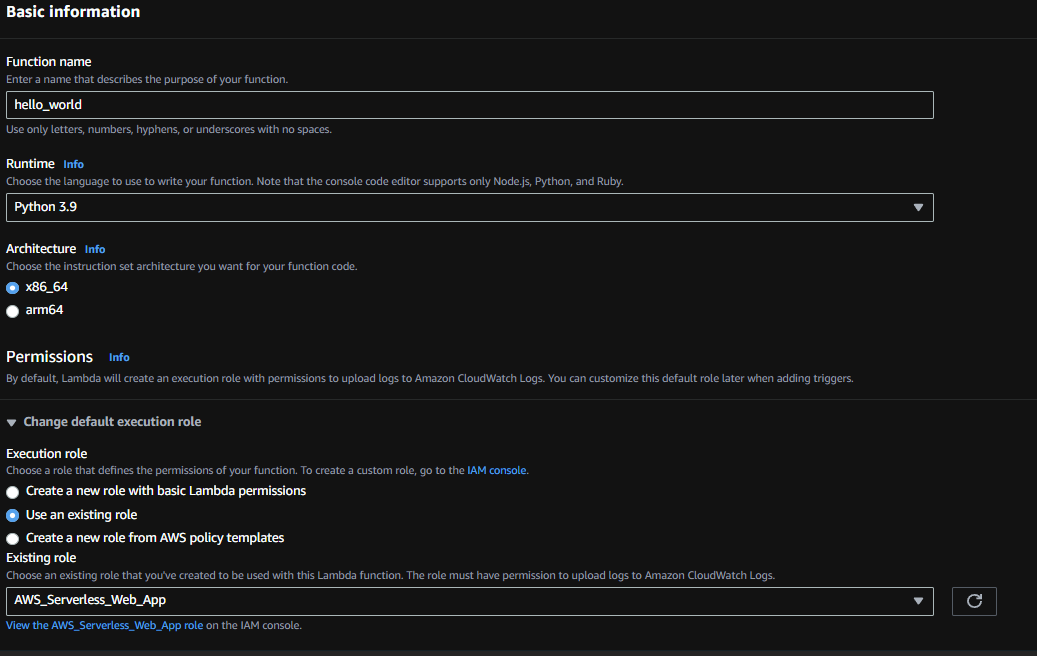
   Description automatically generated

STEP 2

1. Open IAM
2. Create a new role.
3. Lambda function role
4. Add policy to give full access of Dynamo DB.
5. 
6. Text

   Description automatically generated
7. 

Step 3

1. Open Lambda Service
2. Create a function from scratch.
3. 
4. In the lambda function type the following code.
5. A screenshot of a computer

   Description automatically generated with medium confidence
6. Test the function.
7. Graphical user interface, text

   Description automatically generated
8. Graphical user interface, application

   Description automatically generated Edit the test event.
9. A screenshot of a computer

   Description automatically generated with medium confidence Edit the code.
10. Test Event
11. Text

    Description automatically generated
12. Similarly create another function “get\_user\_info” with the sanem code as before.
13. Text

    Description automatically generated
14. Text

    Description automatically generated
15. Type this code into the function.

# this function return a row from the dynamo DB table.

import json

import boto3

import decimal

def replace\_decimals(obj):

# Helper class to Decimals in an arbitrary object

# from: https://github.com/boto/boto3/issues/369

if isinstance(obj, list):

for i in range(len(obj)):

obj[i] = replace\_decimals(obj[i])

return obj

elif isinstance(obj, dict):

for k, v in obj.items():

obj[k] = replace\_decimals(v)

return obj

elif isinstance(obj, set):

return set(replace\_decimals(i) for i in obj)

elif isinstance(obj, decimal.Decimal):

if obj % 1 == 0:

return int(obj)

else:

return float(obj)

else:

return obj

def lambda\_handler(event, context):

# TODO implement

dynamodb=boto3.resource('dynamodb')

table=dynamodb.Table('chocolate\_user')

response=table.get\_item(

Key={

'email':event['email\_address']

}

)

try:

result=response['Item']

result=replace\_decimals(result)

except:

result={}

print(result)

return\_var={

'statusCode': 200,

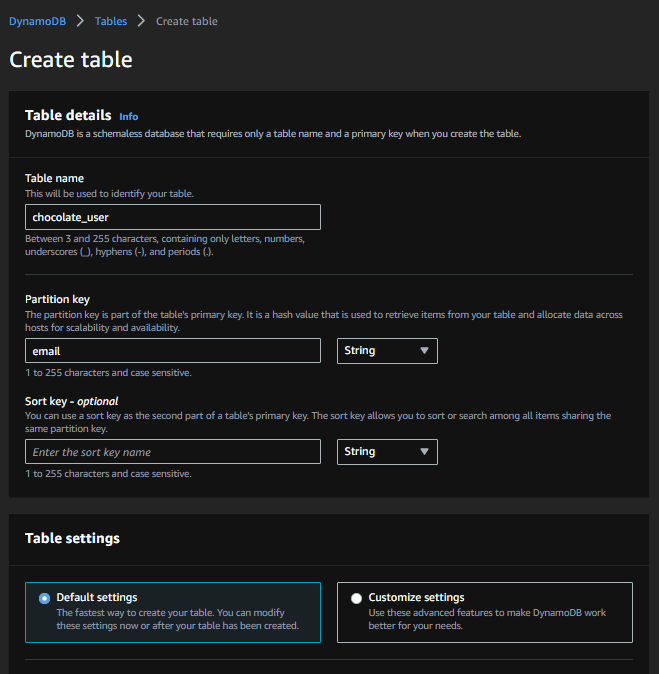
'body': json.dumps(result)

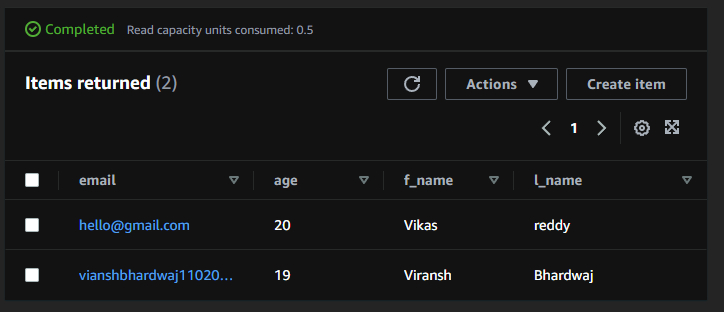
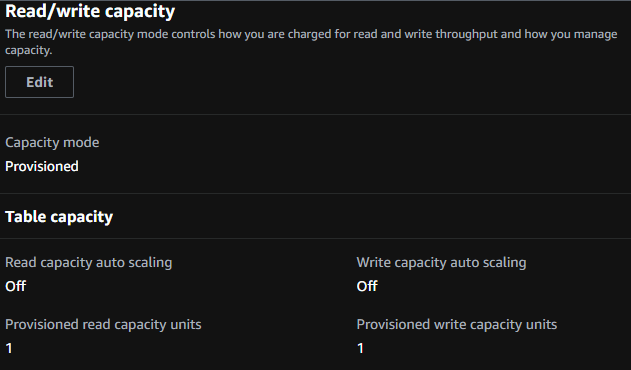
}

return return\_var

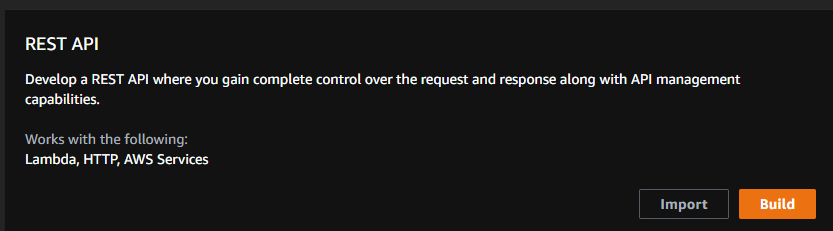
Step 4

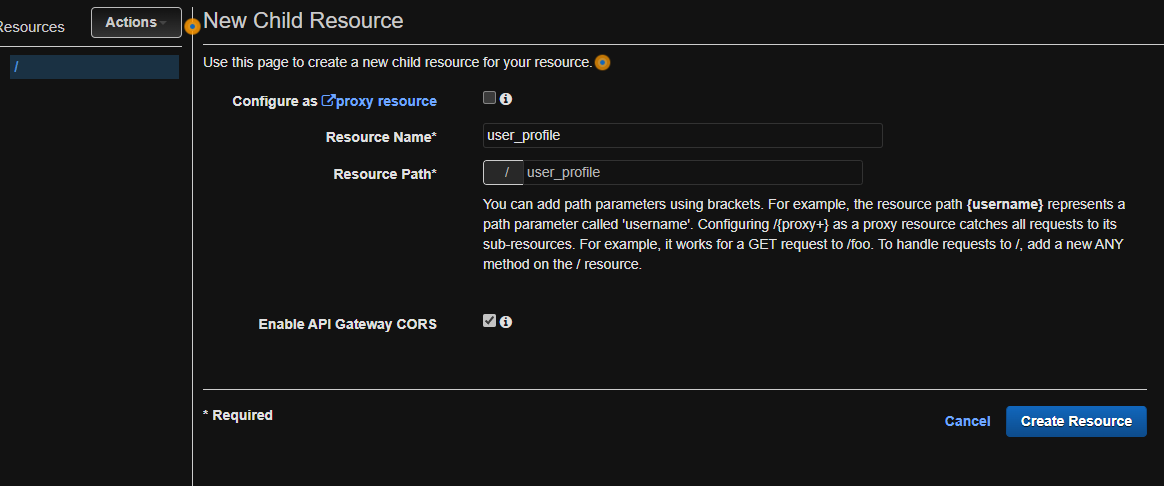
1. Create table as follows.

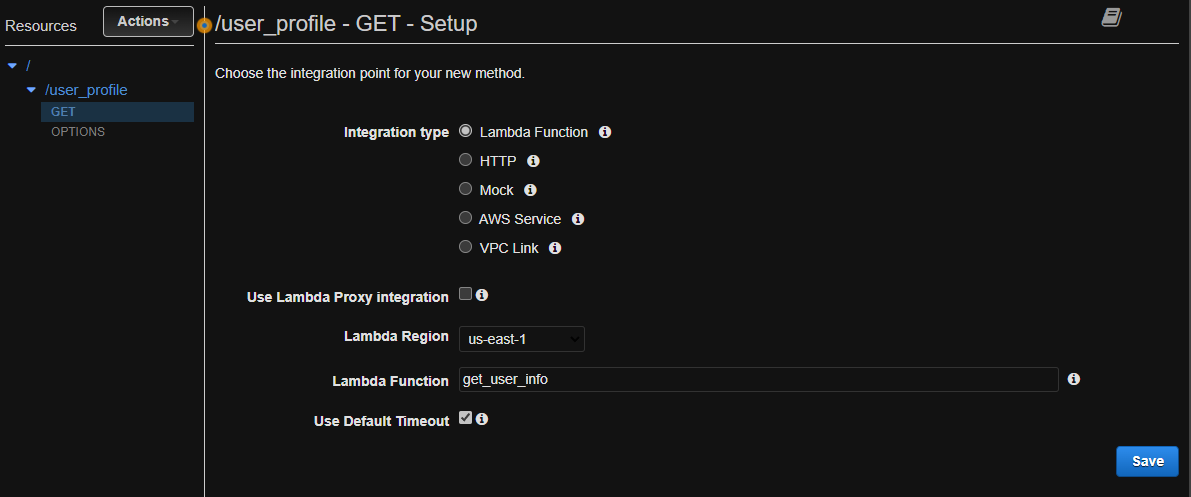


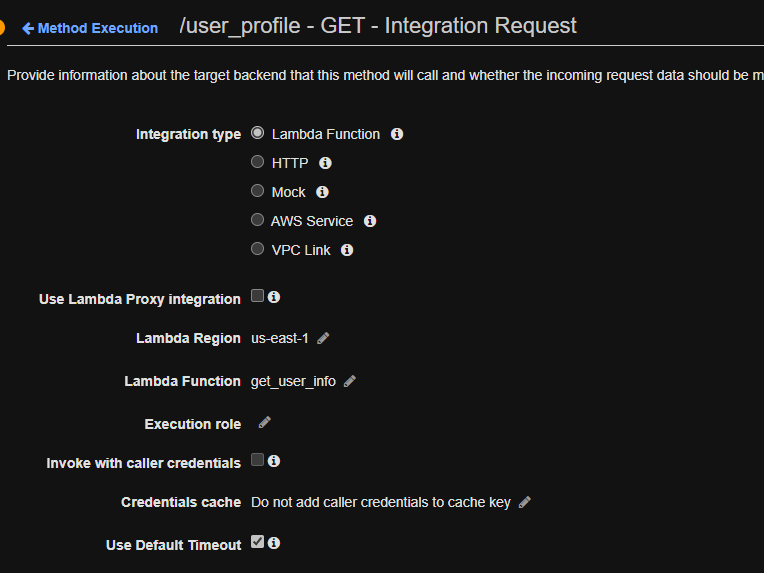
1. Create table items. As described.
2. 
3. Edit read/Write capacity of the table.
4. 

STEP 6

1. 
2. A screenshot of a computer

   Description automatically generated with medium confidence
3. 
4. Create a method GET and save.



1. 
2. A screenshot of a computer

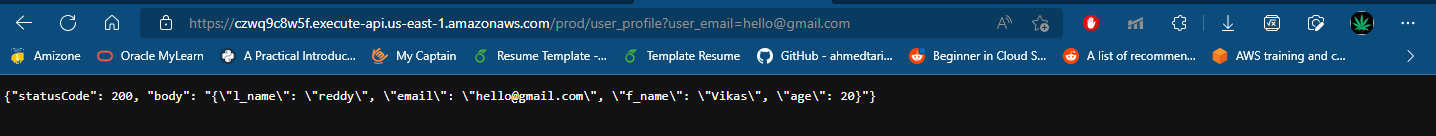
   Description automatically generated with medium confidence
3. Text

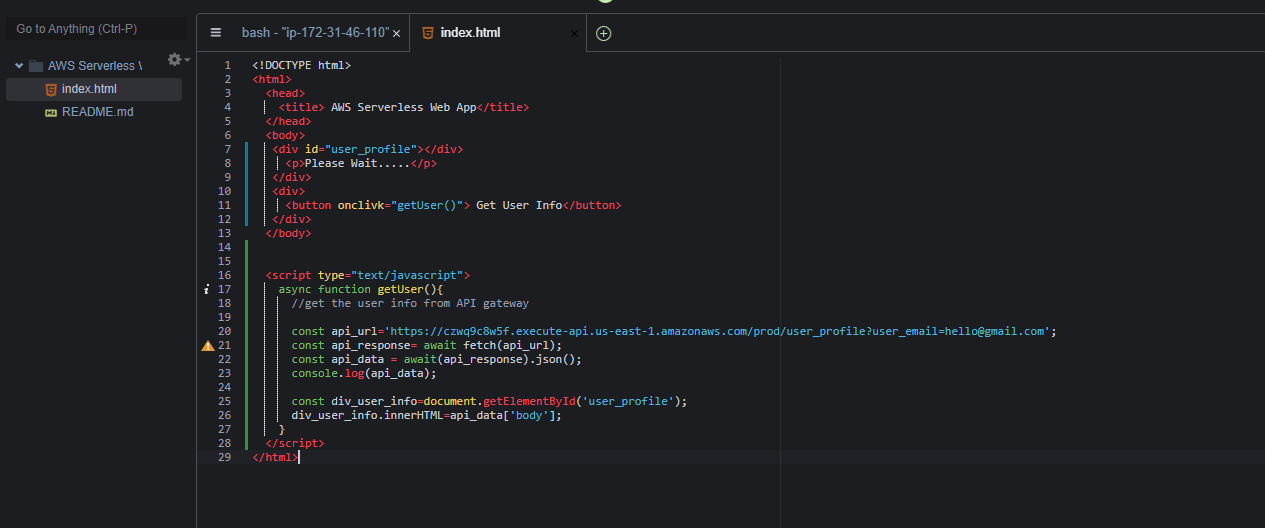
   Description automatically generated
4. Test the method.
5. Enable CORS.

A screenshot of a computer

Description automatically generated with medium confidence

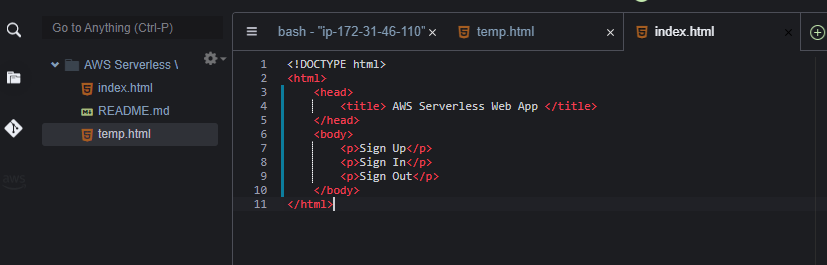
1. Deploy APIGraphical user interface, application

   Description automatically generated
2. 
3. Change the contents of the HTML file as shown.



Step 7

1. Rename the html file as *temp.html* and create a new file as *index.html*
2. Type in the contents of the new html file as shown.

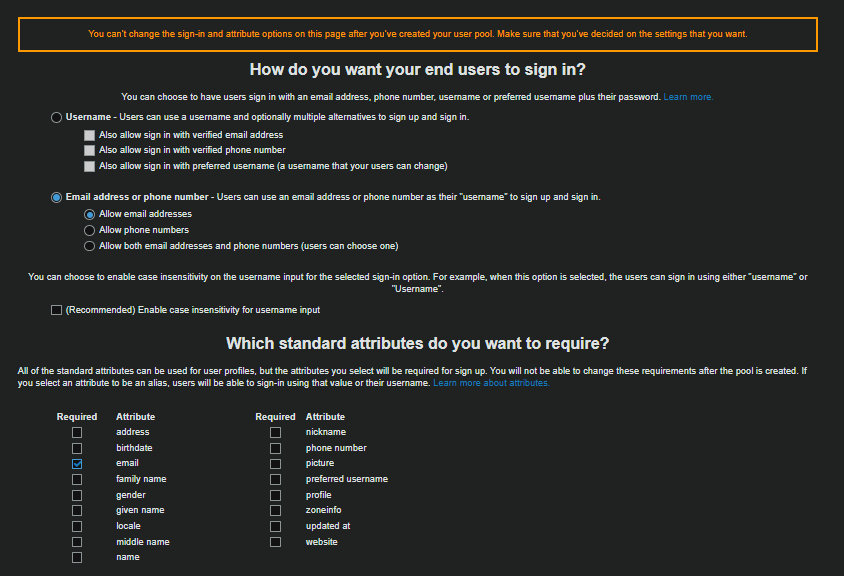


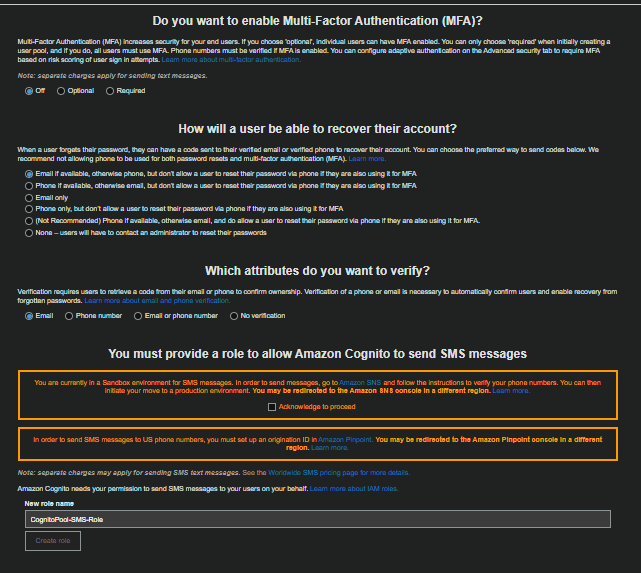
1. Open Amazon Cognito Service from the user dashboard and create a user pool.

Graphical user interface, application

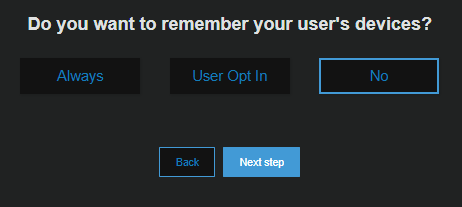
Description automatically generated

1. Graphical user interface, text, website

   Description automatically generated
2. 
3. Text

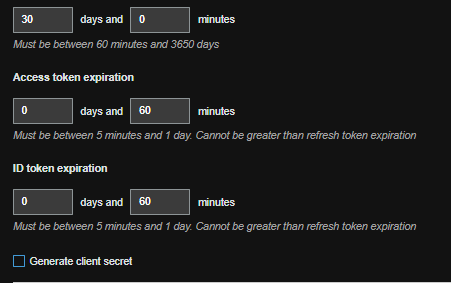
   Description automatically generated
4. 
5. Text

   Description automatically generated
6. Graphical user interface

   Description automatically generated
7. 
8. A screenshot of a computer screen

   Description automatically generated with medium confidence
9. Skip to review.
10. A screenshot of a computer

    Description automatically generated
11. A screenshot of a computer

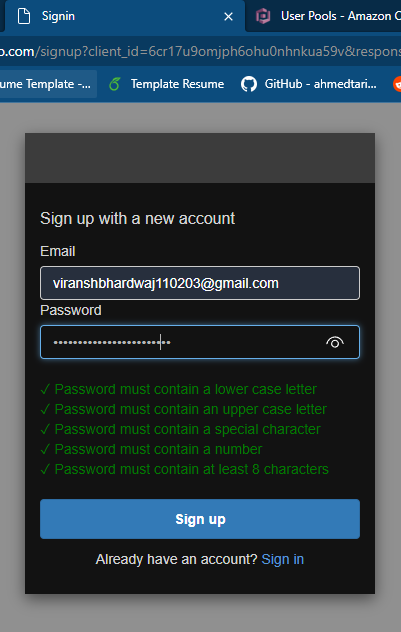
    Description automatically generated with medium confidence
12. Create app clients, enter the name of the app client, and uncheck the client secret option. 
13. A screenshot of a computer

    Description automatically generated
14. A screenshot of a computer

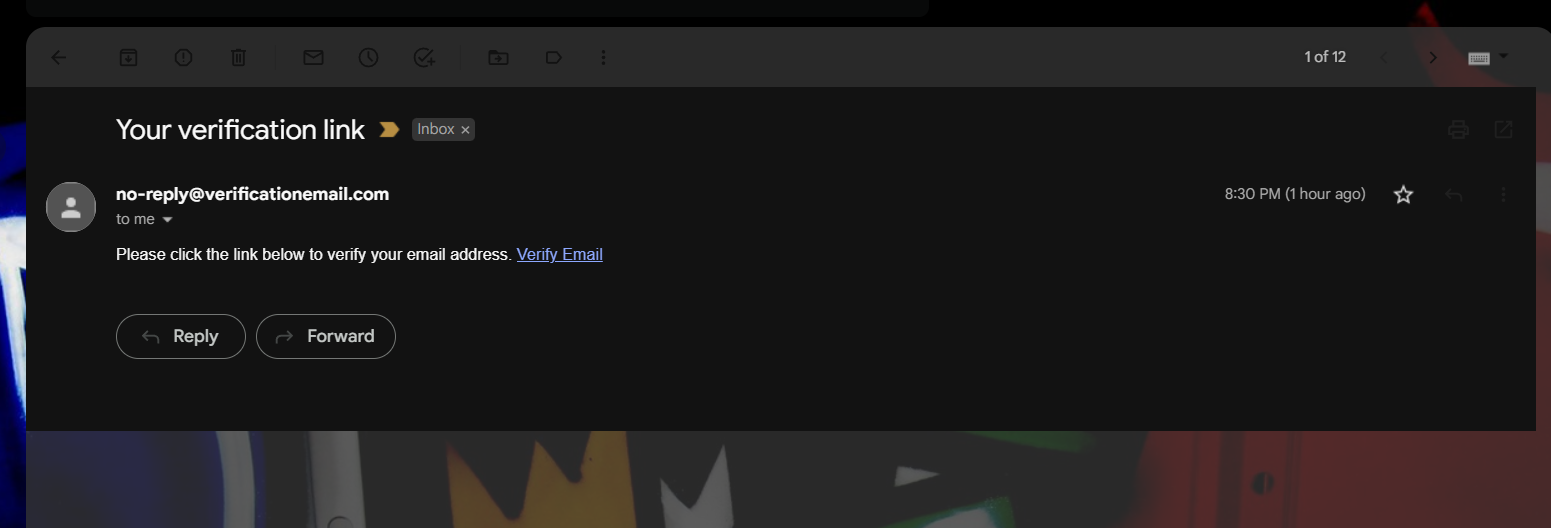
    Description automatically generated
15. Launch the Hosted UI under app client settings. Text

    Description automatically generated
16. Graphical user interface

    Description automatically generated
17. Copy the URL of the page obtained and append the url into the *“index.html”* file as shown. Graphical user interface, text

    Description automatically generated
18. Launch the file into a new tab and signup.
19. 
20. Graphical user interface, text, application, chat or text message

    Description automatically generated



1. Graphical user interface, text, application

   Description automatically generated
2. Graphical user interface, application

   Description automatically generated
3. Text

   Description automatically generated
4. A screenshot of a computer

   Description automatically generated
5. Refresh to see the active users. A screenshot of a computer

   Description automatically generated with medium confidence
6. Create a new sign-in.html file with the following contents.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="utf-8">

<!-- Javascript SDKs-->

<script src="https://code.jquery.com/jquery-3.4.1.min.js"></script>

<script src="js/amazon-cognito-auth.min.js"></script>

<script src="https://sdk.amazonaws.com/js/aws-sdk-2.596.0.min.js"></script>

<script src="js/amazon-cognito-identity.min.js"></script>

<script src="js/config.js"></script>

</head>

<body>

<form>

<h1>Please sign in</h1>

<input type="text" id="inputUsername" placeholder="Email address" name="username" required autofocus>

<input type="password" id="inputPassword" placeholder="Password" name="password" required>

<button type="button" onclick="signInButton()">Sign in</button>

</form>

<br>

<div id='logged-in'>

<p></p>

</div>

<p>

<a href="./profile.html">Profile</a>

</p>

<br>

<div id='home'>

<p>

<a href='./index.html'>Home</a>

</p>

</div>

<script>

var data = {

UserPoolId : \_config.cognito.userPoolId,

ClientId : \_config.cognito.clientId

};

var userPool = new AmazonCognitoIdentity.CognitoUserPool(data);

var cognitoUser = userPool.getCurrentUser();

function signInButton() {

// sign-in to AWS Cognito

var authenticationData = {

Username : document.getElementById("inputUsername").value,

Password : document.getElementById("inputPassword").value,

};

var authenticationDetails = new AmazonCognitoIdentity.AuthenticationDetails(authenticationData);

var poolData = {

UserPoolId : \_config.cognito.userPoolId, // Your user pool id here

ClientId : \_config.cognito.clientId, // Your client id here

};

var userPool = new AmazonCognitoIdentity.CognitoUserPool(poolData);

var userData = {

Username : document.getElementById("inputUsername").value,

Pool : userPool,

};

var cognitoUser = new AmazonCognitoIdentity.CognitoUser(userData);

cognitoUser.authenticateUser(authenticationDetails, {

onSuccess: function (result) {

var accessToken = result.getAccessToken().getJwtToken();

console.log(result);

//get user info, to show that you are logged in

cognitoUser.getUserAttributes(function(err, result) {

if (err) {

console.log(err);

return;

}

console.log(result);

document.getElementById("logged-in").innerHTML = "You are logged in as: " + result[2].getValue();

});

},

onFailure: function(err) {

alert(err.message || JSON.stringify(err));

},

});

}

</script>

</body>

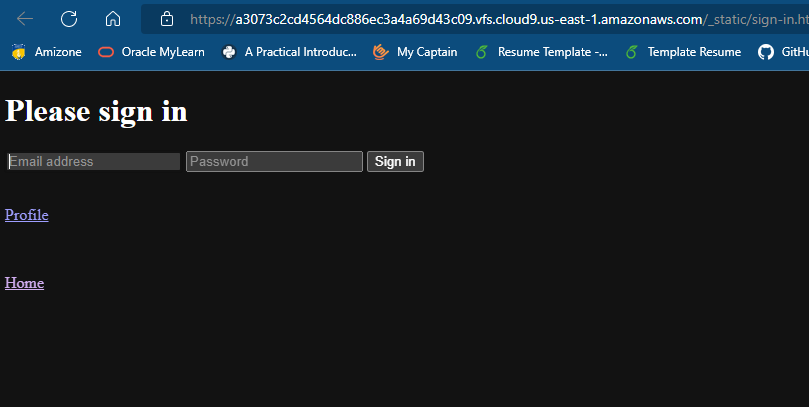
</html>

1. Add the following files into the new folder js.

Graphical user interface, text

Description automatically generated

1. The files will be attatched into the upload link.
2. Save the files and open it in the browser.

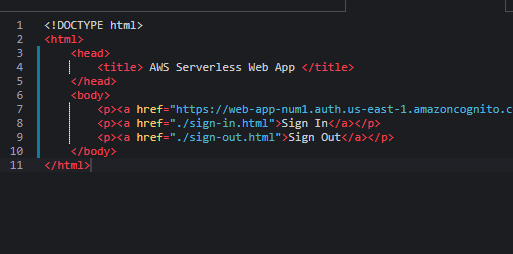


1. Enter the credentials to see the following screen.

Text

Description automatically generated

1. To create a sign out option as well, create a new html file ‘*sign-out.html’*, update the index.html file.



1. A screenshot of a computer

   Description automatically generated with medium confidence

A picture containing diagram

Description automatically generated

1. Text

   Description automatically generated
2. Create a new file ‘profile.html’ with the following contents.



<!doctype html>

<html lang="en">

<head>

<meta charset="utf-8">

<!--Cognito JavaScript-->

<script src="js/amazon-cognito-identity.min.js"></script>

<script src="js/config.js"></script>

</head>

<body>

<div class="container">

<div>

<h1>Profile</h1>

</div>

<div id='profile'>

<p></p>

</div>

<div>

<br>

<div id='home'>

<p>

<a href='./index.html'>Home</a>

</p>

</div>

<script>

async function getUser(email\_address) {

// get the user info from API Gate

const api\_url = 'https://gonvpjbyuf.execute-api.us-east-1.amazonaws.com/prod/user-profile?user\_email=' + email\_address;

const api\_response = await fetch(api\_url);

const api\_data = await(api\_response).json();

console.log(api\_data);

const div\_user\_info = document.getElementById('profile');

div\_user\_info.innerHTML = api\_data['body'];

}

var data = {

UserPoolId : \_config.cognito.userPoolId,

ClientId : \_config.cognito.clientId

};

var userPool = new AmazonCognitoIdentity.CognitoUserPool(data);

var cognitoUser = userPool.getCurrentUser();

window.onload = function(){

if (cognitoUser != null) {

cognitoUser.getSession(function(err, session) {

if (err) {

alert(err);

return;

}

//console.log('session validity: ' + session.isValid());

cognitoUser.getUserAttributes(function(err, result) {

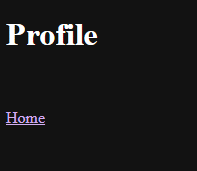
if (err) {

console.log(err);

return;

}

// user email address



console.log(result[2].getValue());

getUser(result[2].getValue())

});

});

} else {

console.log("Already signed-out")

}

}

</script>

</body>

</html>

1. Preview the page and you serverless login page is ready.

***NOTE: make sure to add the same email id into the Dynamo DB table so that it may display the data in profile page correctly.***

***OVERVIEW OF THE USER SIGN IN/OUT/UP/PROFILE***

Text

Description automatically generated with low confidence

Sign Up: Already signed up as [***viranshbhardwaj110203@gmail.com***](mailto:viranshbhardwaj110203@gmail.com)email address.

Sign IN:

Text

Description automatically generated with medium confidence



SIGN OUT:

Go back to home page and click on sign out.

Text

Description automatically generated

***Project Created By:***

***VIRANSH BHARDWAJ***

***GADDALA GREESHMA DEVI***

***YUKTA RAJESH SHARMA***

***VIKAS REDDY***

***SARVESH LATH***

***Lab files used:***

Index.html: <https://amityedu96491-my.sharepoint.com/:t:/g/personal/viransh_bhardwaj_s_amity_edu/EbpHuTXcYM1Mg-zHxYEUTjIBqtmmLeoGm-ilE5Q3BX-sXQ?e=24JWAZ>

Profile.html: <https://amityedu96491-my.sharepoint.com/:t:/g/personal/viransh_bhardwaj_s_amity_edu/EavWN297n5VGjUpMx4reavABlCkVFTf8v9k7Km_TR3XQOw?e=tWeBWG>

Sign-in.html: <https://amityedu96491-my.sharepoint.com/:t:/g/personal/viransh_bhardwaj_s_amity_edu/EYaejErCRgNKlXRBBdi5ka4BkeyAeU9jEH9hcUxd1NUojw?e=HkhXuB>

Temp.html: <https://amityedu96491-my.sharepoint.com/:t:/g/personal/viransh_bhardwaj_s_amity_edu/Eb-4gJTugadHrPdwhXTygaYBajbQc_0fI-geZGtH7-QmHg?e=cgbuBf>

Js files:

<https://amityedu96491-my.sharepoint.com/:f:/g/personal/viransh_bhardwaj_s_amity_edu/EgV0BecNShFBlWfj0mw3V8EBUSy045WxqiG5BvRZq9OD6Q?e=Fhh8lq>

PROJECT FOLDER: <https://amityedu96491-my.sharepoint.com/:f:/g/personal/viransh_bhardwaj_s_amity_edu/Ek5Dz1HelOlIh3fGjsqLSYwBFFprKpcvCTMADAm5fztoFQ?e=FQWT3m>