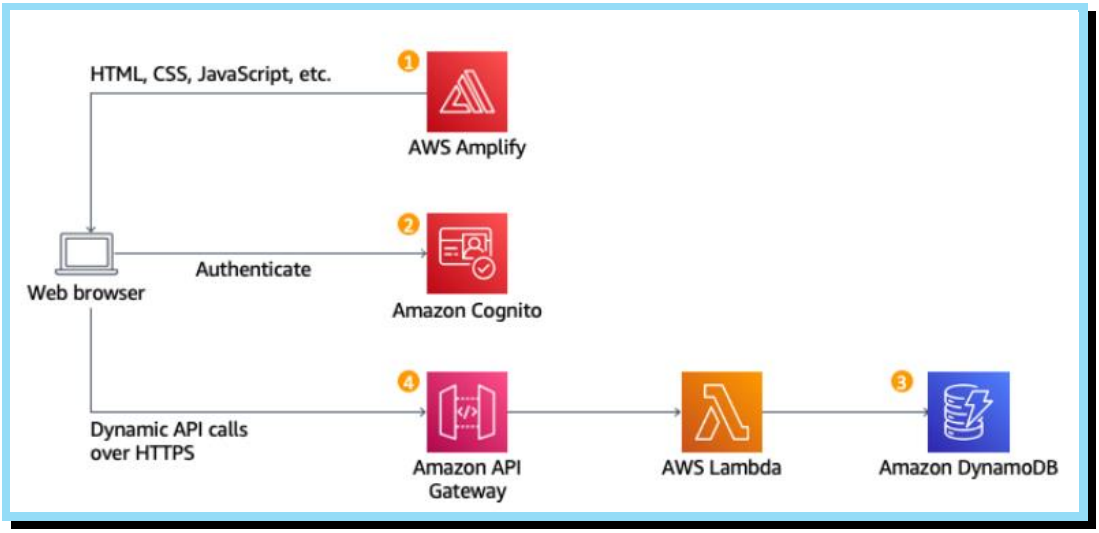
# „Build a Serverless Web Application

June. 2024

**Flow**: This application offers a Login system where users can register and log in. After the authentication, users can request a unicorn ride to the pick-up location. This Frontend sends this request to the RESTful web service to dispatch a nearby unicorn. Then, the Backend service interacts with the database and sends the information to the Frontend.




## Technologies:

AWS Amplify, Amazon Cognito, AWS Lambda, Amazon DynamoDB and Amazon API Gateway

## FIVE Steps:

1) Static Web Hosting: CodeCommit, AWS Amplify
2) User Management: Amazon Cognito
3) Serverless Backend: DynamoDB, AWS Lambda
4) RESTful API: Amazon API Gateway, AWS Lambda
5) Cleaning Up: Delete every Resource & CloudWatch Log

## 1. Process 1: Host A Static Website with Continuous Deployment

➔ AWS Amplify provides a git-based workflow that is used to deploy the web application and offers an authentication process to secure access.

### (1) Static web application codes are stored in AWS CodeCommit

- Create the Repository:
  - ⇨ Name: unicorn_webapp / Region: eu-central-1
- Create the IAM user and assign the policy to access the my Repository and external S3:
  - ⇨ Name: webapp-admin / Policies: 1 AWSCodeCommitPowerUser 2 S3ReadOnlyAccess
- Secure IAM user[webapp-admin]:
  - ⇨ Issue Access key and Secret Access key for overall AWS resources and CLI
  - ⇨ Generate HTTPS git credential for authentication of git-based system

### (2) Set up the Git environment

- $ aws configure : Store the IAM user info [Access Key & Secret Access Key, Region],
  → Which is in /.aws/credentials

➔ Configure Credentials of AWS CodeCommit to access the Repository
- $ git config --global credential.helper '!aws codecommit credential-helper $@'
- $ git config --global credential.UseHttpPath true

➔ Configure Git user Information
- $ git config --global user.email youngandtom2@gmail.com
- $ git config --global user.name "HTTPS Credential username Issued by IAM user"

➔ Populate Repository[CodeCommit] on AWS Linux
- $ git clone "my repo(unicorn_serverless_web)'s HTTP URL"

➔ Update the Repository uploading Source Code from external S3 Objects in new directory

- $ aws s3 cp s3://ttt-wildrydes/wildrydes-site ./ --recursive
  - ⇨ Using an IAM user with policies for S3 and CodeCommit access enables secure population of the repository with external resources(S3), facilitated by configuring Access Keys and HTTPS Git credentials.
- $ git remote -v : Check if the region is configured properly,
  if not, $ git remote set-url origin "my repo's HTTP URL"
- $ git add . // $ git commit -m "Set up the Repo and Upload the static Web files"
- $ git push origin master

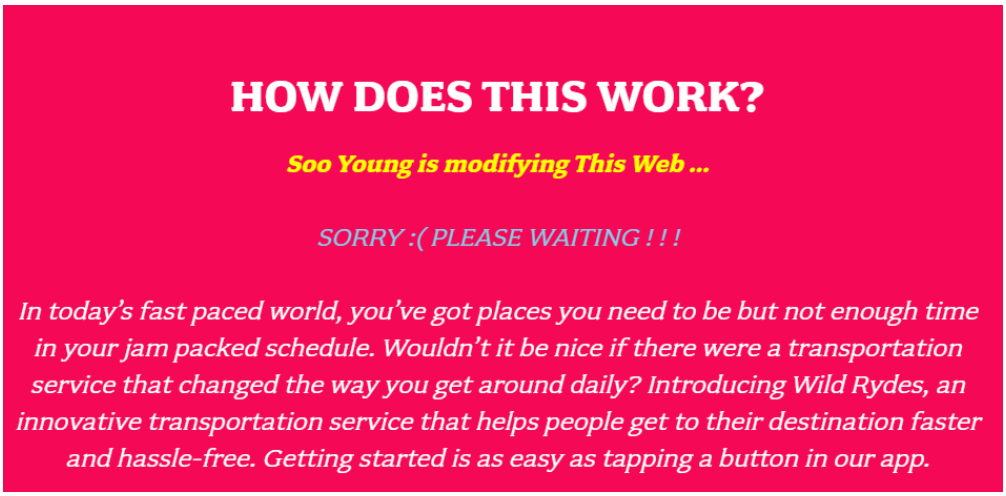
### (3) Hosting the Static Web sites with AWS Amplify

➔ Create new app with Amplify
- Code Provider: CodeCommit / Repo: unicorn_webapp / Branch: master
- App name: HEY-UNICORN-WHEREAMI

- Build settings: The final frontend-output is stored every deployment, which is used as Cache.
- Update the Service Role: Must include the policy(AdministratorAccess-Amplify)

(4) Modify the Web Content and Host the static website

- Update the code in index.html:
-  Soo Young is modifying this Web ... <br><br>
-  SORRY:( PLEASE WAITING ! ! ! <br><br>




- $ git add index.html / $ git commit -m "Update the first page: Modifying"
- $ git push origin master
- Amplify recognises the change in the registered Repo(branch), applies and deploys.

2. Process 2: Manage User Accounts by implementing a Login System

➔ The User Pool function provides an authentication process for security by registering user information, verifying the information through an email, and enabling access to the website.

(1) Create the user pool in the Region[eu-central-1] In AWS *Cognito*:

- Sign-in Options: User name / Default Passwd policy / No MFA
- Recovery: Enable / Recovery method: Email only / Email Configuration with Cognito
- Create the User Pool Name: unicorn-users/ App Client Name: web-users-group

⇨ User pool name is for this app, App client can be shared in another app or service.

(2) Update the User Pool into the Web Backend file

➔ The app includes an HTML-based frontend and a backend system written in JavaScript.

- The authentication configuration associated with Cognito is in my-repo > js > config.js
- Update the Values of UserPoolID & userPoolClientID, which are Not the Names.
- Update the Region: eu-central-1

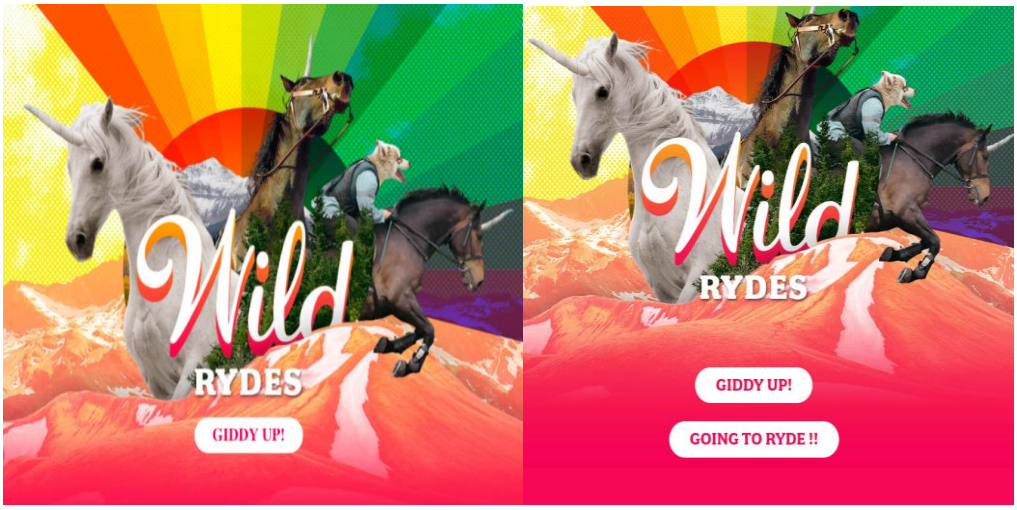
⇨ Being Updated, the config.js file enables the client to be authenticated before sending the request to the backend server.

- git add . / git commit -m “Configure authentication process: Cognito” / git push

## (3) Upgrade the homepage for usability

➔ The flow of the Login System:

- Initial Frontend: 1 index.html → register.html / 2 signin.html → ride.html:
  There were two parts: a registration part and a login part to connect to the ride service. Those pages needed to connect with each other, and the homepage added the button which heads to the sign-in page.
- Improved Frontend: index.html → register.html & signin.html → ride.html




➔ Add the button to sign in:

- Create the Button Group having the same design In index.html:

```
    <div class="home-buttons">
        <a class="home-button" href="/register.html">Giddy Up!</a>
        <a class="home-button" href="/signin.html"> Going To RYDE !!</a>
    </div>
</header>
```

- Define the .home-buttons{} and Modify the .home-button{} In main.css:

```
.home-button{
    -webkit-appearance:none; -moz-appearance:none; appearance:none;
    background:#fff;border:1px solid transparent;
    border-radius:1.25rem;color:#f50856;font-size:1.25rem;
    text-transform:uppercase;padding:.3125rem 1.5625rem;font-weight:700;
}

@media screen and (min-width:40em){
    .page-home .home-buttons{
        bottom:2%;}
}
.home-buttons{
    padding-top: 16rem;
    position: absolute;
    left: 50%;
    transform: translateX(-50%);
    display: flex;
    flex-direction: column;
    gap: 20px;
    align-items: center;
}
```

⇨ Some attributes are moved from the .home-button to the .home-buttons group

⇨ Some attributes related to deployment are added for the group, such as display, flex-direction, gap, and align-items.

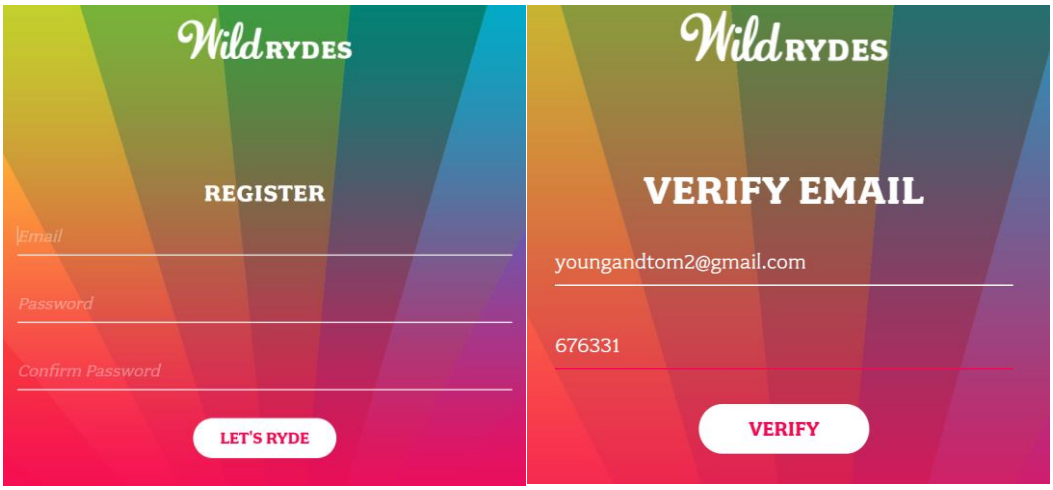
➔ Adjust the gap above the home-buttons group In main.css:

- @media screen and (min-width: 40em) { .page-home .home-buttons{ bottom: 2%} }

⇨ This adjustment positions the home-buttons group 2% above the bottom of the homepage.

## (4) Validate the implementation of Authentication

- Through the 'Giddy up' button on the homepage, users can register the information in the User Pool[store/manage] on the register.html(registration page). This process requires an email address as a username and password. Then, the user receives an email containing a verification code. The user should input the email address and the code to prove the authentication. Once verified, it redirects to the sign-in page to enable users to log in.




- How to Work with the AWS Cognito with the Serverless App?

⇨ In register.html & signin.html:

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

- These scripts enable the two pages to reference the backend configuration files, which include the Cognito configuration information to interact with the User Pool.

- Upon successful login, the auth token for the API is given to the user on the ride.html page. In other words, the Cognito authenticates the user and issues the JWT token.
- In this app, when the client sends an API request, the API gateway checks if the token in the authorisation header is validated.

## 3. Process 3: Serverless Service Backend

➔ Build a backend process with AWS Lambda & DynamoDB to handle requests for a web app.
➔ A Lambda function is invoked upon the trigger(Each time a user requests a unicorn!).

➔ Process
⇨ 1 Web Client ----- REQUEST----> API Gateway (Check if the authentication token is valid )
⇨ 2 API Gateway --- (HTTPS) Dynamic API calls ---> Lambda(Invoke the function upon trigger)
⇨ 3 AWS Lambda executes the Function (Backend) ---- RECORD (result) ----> DynamoDB
⇨ 4 AWS Lambda ----- RESPOND to the Request --- > Frontend --- PROCESS --> User

➔ This Lambda Function leads to
⇨ 1 Select the unicorn
⇨ 2 Record the Processed values of the Request the DynamoDB table
⇨ 3 Send a Response to the Frontend about the unicorn being dispatched.

(1) Create DynamoDB Table:

- Name: unicorn-request-response / Partition key(unique) -> UnicornID
- This DynamoDB Table's ARN is applied to the Role of AWS Lambda

(2) Create IAM Role[Lambda] with 2 Policies:

- Name: AWSLambdaEXEC / Policy: AWSLambdaBasicExecutionRole
- Add the new Policy about Recording the result in DynamoDB
⇨ Lambda Role > Permissions > Create Incline Policy > Service : DynamoDB
⇨ Service(DynamoDB) > Actions: PutItem(write) / > Resources : Dynamo DB's ARN

⇨ Lambda's Role has the function of 1 Lambda execution 2 Writing the result in DynamoDB

(3) Create a Lambda function to process API Requests:
- Name: UnicornDelivery / Runtime: Node.js 16.x / Default / Role: AWSLambdaEXEC
- Input & Deploy the Source Code of Lambda Function in UnicornDelivery > index.js:
⇨ Event Start > e.g. exports.handler = (event, context, callback) => { ... }

- Unicorn Kinds >>

```
const fleet = [
    {
        Name: 'Angel',
        Color: 'White',
        Gender: 'Female',
    },
    {
        Name: 'Gil',
        Color: 'White',
        Gender: 'Male',
    },
    {
        Name: 'Rocinante',
        Color: 'Yellow',
        Gender: 'Female',
    },
];
```

- < DB Change : RideId → UnicornID / Rides -> unicorn-request-response >

```
callback(null, {
    statusCode: 201,
    body: JSON.stringify({
        UnicornName: rideId,
        Unicorn: unicorn,
        Eta: '30 seconds',
        Rider: username,
    }),
```

```
function recordRide(rideId, username, unicorn) {
    return ddb.put({
        TableName: 'unicorn-request-record ',
        Item: {
            UnicornName: rideId,
            User: username,
            Unicorn: unicorn,
            RequestTime: new Date().toISOString(),
        },
    }).promise();
}
```

- Function errorResponse(errorMessage, awsRequestId, callback) {}:

```
function errorResponse(errorMessage, awsRequestId, callback) {

    // Create Structured Logging IN CloudWatch: '/aws/lambda/UnicornDelivery'
    console.error(JSON.stringify({
        level: 'ERROR',
        message: errorMessage,
        requestId: awsRequestId,
        timestamp: new Date().toISOString()
    }));

    // Create the Response when the Error Event
    callback(null, {
        statusCode: 500,
        body: JSON.stringify({
        Error: errorMessage,
        Reference: awsRequestId,
        }),
        headers: {
        'Access-Control-Allow-Origin': '*',
        },
    });
}
```

- Lambda function[UnicornDelivery] > index.js >> exports.handler = () =>
  : This handler processes the Request and Response.

⇨ It calls the recordRide().then() event, if this event implements functions successfully, it checks the callback for the statusCode:201.

⇨ If it doesn't work, it runs the '.catch(err)' part for logging an error and operating the function [errorResponse] to check the statusCode:500.

- function errorResponse(errorMessage, awsRequestId, callback) { }:

⇨ console.error(); → Logs the 'ERROR' state, errorMessage, RequestId, and timestamp to CloudWatch

⇨ callback(null, { if 500 }); → When an error occurs [HTTP 500], it notifies the client with a structured error response.

(4) Create the Test Event for Checking Lambda's Implementation:

- Lambda Function[UnicornDelivery] > Test > Test event > Name: Test-UnicornDelivery
- Test if the Lambda Function is implemented successfully using this new event.

⇨ Response >> statusCode:201 means the Executing function -> succeeded.

```
{
    "path": "/requests_unicorn",
    "httpMethod": "POST",
    "headers": {
        "Accept": "*/*",
        "content-type": "application/json; charset=UTF-8",
        "Authorization": "MY_JWT TOKEN ISSUED after signing in successfully"
    },
    "queryStringParameters": null,
    "pathParameters": null,
    "requestContext": {
        "authorizer": {
            "claims": {
                "cognito:username": "the_username"
            }
        }
    },
    "body": "{\"PickupLocation\":{\"Latitude\":47.6174755835663,\"Longitude\":-122.28837066650185}}"
}
```


< Test-UnicornDelivery in JSON >

** Test Code(JSON) and Lambda function code(JavaScript) are updated on my repository.

- This test code shows the request process from API Gateway to Lambda Function.

⇨ path: API Gateway Endpoint (Request/Response), in other words, resource path

⇨ httpMethod: Which HTTP Method?

⇨ headers: { } → configuring JWT token for authentication / Without parameters -> OK

⇨ requestContext:{ } → configuring needed information for User Pool

⇨ body: {variables : values} → Input the values for variables to test the Lambda function.

- Test the AWS Lambda function[UnicornDelivery] → Get: statusCode: 201

**Test Event Name**

Test-UnicornDelivery

**Response**


```
{
  "statusCode": 201,
  "body": "{\"UnicornID\":\"WoVV7SgGgY90xoDyWBL1Gw\",
  "headers": {
    "Access-Control-Allow-Origin": "*"
  }
}
```


## 4. Process 4: Deploy a RESTful API

➔ Amazon API Gateway uses RESTful API to expose the AWS Lambda Function.
The RESTful API provided by API Gateway is accessible on the public subnet. However, it is secured by the Amazon Cognito User Pool. By interacting with clients from the website, the static website becomes dynamic.
The added client-side JavaScript makes AJAX calls(Asynchronous HTTP) to receive the response from the backend.

(1) Start REST API

- Create REST API using Amazon API Gateway:

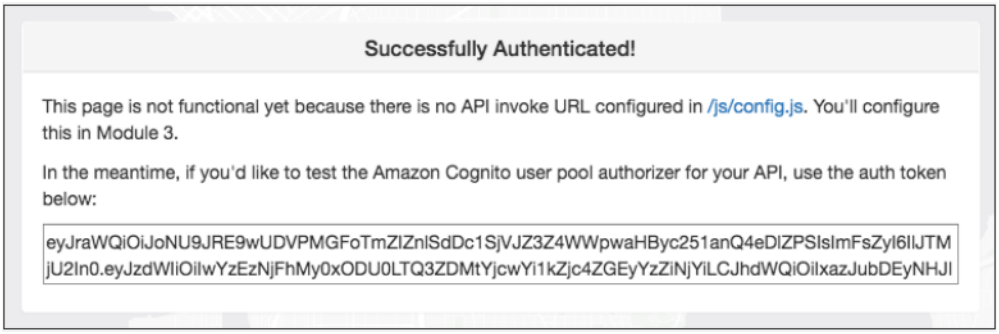
⇨ Name: Unicorn-REST-API / Endpoint: Edge-optimized (public)

- Create an Authorizer of this new REST API: Configuring User Pool

⇨ To configure[Register] the authentication of API Request from clients by secure token
⇨ Name: Unicorn-Authorizer / Type: Cognito[eu-central-1]: unicorn-users
⇨ [Token] Source: Authorization -> The place Token be included

- Verify the Token by Test the created Authorizer: To get Response Code(200)

⇨ Test Authorizer: Token Source -> Authorization / Token Value -> using 'the auth token'




## (2) Specify the API

- Create resources within the API and POST method[HTTP]:

■ Resource:

⇨ Name [Resource]: requests_unicorn (as same as the test code 'path' of Lambda Function)
⇨ Enabled: CORS -> To Protect my web application
➔ This resource path is used as a Test code in the Lambda function and
Enables the API gateway and the Lambda function to interact with each other
⇨ By configuring config.js(invoke_URL: Resource's Stage) & ride.js(for sending a response to ride.html[Frontend] ->)

■ Method (As selecting the Resource):

⇨ Method Type: POST[New Request] / Integration Type: Lambda function
⇨ Enabled: Lambda Proxy integration
› To Integrate API gateway with Lambda Function & Setting up Configuration.
⇨ Choose the Lamda function[UnicornDelivery] in eu-central-1

■ POST Method Configuration and Deploy API:

- Structure: Client 1) Method Request 2) Method Response

⇨ Method REQUEST: Authorization -> Cognity User Pool[unicorn-users]

- Deploy the API as a 'New Stage[Name: dev0625]':

⇨ In this API Gateway, the new stage is produced with an Invoke URL to Lambda.

## (3) Update the Website Configuration & Align the Version of the map for the Compatibility

- Update the js/config.js file for RESTful API configuration with authentication
: This file is composed of two parts -> 1) Cognito 2) Api: Invoke URL(API Gateway's Stage)

⇨ api: {invokeUrl: "The URL heading to Lambda function from API Gateway in POST Request"

- git add . / git commit -m “integration API gateway with Lambda ” / git push

- Modify the ArcGIS version from 4.3 to 4.6 in the ride.html file.
- git add ride.html / git commit -m “map version upgrade” / git push

### (4) Understand the flow of Ajax Call on the ride.js

```
function requestUnicorn(pickupLocation) {
    $.ajax({
        method: 'POST',
        url: _config.api.invokeUrl + '/requests_unicorn',
        headers: {
            Authorization: authToken
        },
        data: JSON.stringify({
            PickupLocation: {
                Latitude: pickupLocation.latitude,
                Longitude: pickupLocation.longitude
            }
        }),
        contentType: 'application/json',
        success: completeRequest,
        error: function ajaxError(jqXHR, textStatus, errorThrown) {
            console.error('Error requesting ride: ', textStatus, ', Details: ', errorThrown);
            console.error('Response: ', jqXHR.responseText);
            alert('An error occured when requesting your unicorn:\n' + jqXHR.responseText);
        }
    });
}
```

</js/ride.js>

- The Request from Client Process

⇨ requestUnicorn() send a request to the Server via AJAX call, with the pickupLocation data formatted in JSON as a parameter. In this AJAX structure, the endpoint URL is specified to reach the API Gateway, which includes the stage of the specific method[‘POST’] in the resource and the resource path.

⇨ It also contains headers with authentication information, data, and an error handling mechanism. It indicates that the data contents between server and client will be in JSON format.

## 5. Process 5: Final Implementation Check & Resources Termination

➔ After signing in to the app, Choose the pick-up location and Request a unicorn there.

➔ Terminate AWS resources used in the serverless application implementation.

- Delete AWS Services: Amplify -> Cognito User Pool -> Lambda function -> IAM Role
- Delete the DynamoDB table and associated CloudWatch alarms.
- Delete API Gateway(REST API) -> AWS CodeCommit
- CloudWatch Log > Delete ‘/aws/lambda/UnicornDelivery’ this group. This log group was automatically created when the AWS Lambda function was issued.