

BUILD A SERVERLESS WEB APPLICATION USING AWS

Submitted in partial fulfillment of the requirements
for the award of
Bachelor of Engineering degree in Computer Science and Engineering

by

SAI RAM DISOWJA MEKA (REG.NO. 37110662)
RAVI TEJA RUTTALA (REG.NO. 37110651)



**DEPARTMENT OF COMPUTER SCIENCE
AND ENGINEERING
SCHOOL OF COMPUTING**

SATHYABAMA

**INSTITUTE OF SCIENCE AND TECHNOLOGY
(DEEMED TO BE UNIVERSITY)**

Accredited with Grade "A" by NAAC

**JEPPIAAR NAGAR, RAJIV GANDHI SALAI,
CHENNAI - 600 119**

MARCH - 2021



SATHYABAMA

INSTITUTE OF SCIENCE AND TECHNOLOGY

(DEEMED TO BE UNIVERSITY)

Accredited with "A" grade by NAAC

Jeppiaar Nagar, Rajiv Gandhi Salai, Chennai – 600119

www.sathyabama.ac.in



DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

BONAFIDE CERTIFICATE

This is to certify that this Project Report is the bonafide work of **SAI RAM DISOWJA MEKA (REG.NO. 37110662)**, **RAVI TEJA RUTTALA (REG.NO. 37110651)** who have done the Project work as a team who carried out the project entitled "**BUILD A SERVERLESS WEB APPLICATION USING AWS**" under my supervision from **NOVEMBER 2020** to **MARCH 2021**.

Internal Guide

Dr. P. SARDAR MARAN, M.E., Ph.D.

Head of the Department

Submitted for Viva voce Examination held on

Internal Examiner

External Examiner

DECLARATION

I, **SAI RAM DISOWJA MEKA (REG.NO. 37110231)** and **RAVI TEJA RUTTALA (REG.NO. 37110651)** hereby declare that the Project Report entitled “**BUILD A SERVERLESS WEB APPLICATION USING AWS**” done by us under the guidance of **Dr. P. SARDAR MARAN, M.E., Ph.D.**, (Internal) is submitted in partial fulfillment of the requirements for the award of Bachelor of Engineering degree in Computer Science and Engineering.

DATE:

PLACE:

SIGNATURE OF THE CANDIDATE

ACKNOWLEDGEMENT

We are pleased to acknowledge our sincere thanks to **Board of Management of SATHYABAMA INSTITUTE OF SCIENCE AND TECHNOLOGY** for their kind encouragement in doing this project and for completing it successfully. We are grateful to them.

We convey our thanks to **Dr. T. Sasikala M.E., Ph.D., Dean**, School of Computing, **Dr. S. Vigneshwari M.E., Ph.D., and Dr. L. Lakshmanan M.E., Ph.D., Heads** of the Department of Computer Science & Engineering for providing us necessary support and details at the right time during the progressive reviews.

We would like to express our sincere and deep sense of gratitude to our Project Guide **Dr. P. SARDAR MARAN, M.E., Ph.D.**, for his valuable guidance, suggestions and constant encouragement paved way for the successful completion of our project work.

we wish to express our thanks to all Teaching and Non-teaching staff members of the **Department of Computer Science & Engineering** who were helpful in many ways for the completion of the project.

ABSTRACT

In this project, you'll create a serverless web application that enables users to request unicorn rides from the wild Rydes fleet. The application will present users with an HTML based user interface for indicating the location where they would like to be picked up and will interface on the backend with a RESTful web service to submit the request and dispatch a nearby unicorn. The application will also provide facilities for users to register with the service and log in before requesting rides. The application architecture uses AWS Lambda, Amazon API Gateway, Amazon Dynamo DB, Amazon Cognito, and AWS Amplify Console. 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. Amazon Cognito provides user management and authentication functions to secure the backend API. Finally, DynamoDB provides a persistence layer where data can be stored by the API's Lambda function.

CHAPTER NO.	TABLE OF CONTENTS	PAGE NO.
	ABSTRACT	
	LIST OF FIGURES	
	LIST OF TABLES	
	LIST OF ABBREVIATIONS	
1	INTRODUCTION	1
	1.1 MODULES	1
	1.2 APPLICATION ARCHITECTURE	2
2	HOST A STATIC WEBSITE	3
	2.1 ARCHITECTURE OVERVIEW	3
	2.2 CODECOMMIT REPOSITORY	4
3	USER MANAGEMENT	6
	3.1 ARCHITECTURE DIAGRAM	6
	3.2 COGNITO USERPOOL USING AWS AMPLIFY CLIENT	8
	3.3 USER POOL'S APP CLIENT	8
4	BUILD A SERVERLESS BACKEND	9
	4.1 STRUCTURE OF SERVERLESS BACKEND	9
	4.2 CREATE AN AMAZON DYNAMODB TABLE	10
	4.3 CREATE AN IAM ROLE FOR LAMBDA FUNCTION	11
	4.4 CREATION OF LAMBDA FUNCTION	13
5	DEPLOY RESTFUL API	15
	5.1 API GATEWAY	15
	5.2 CREATE A NEW REST API	16
	5.3 CREATION OF COGNITO USERPOOL AUTHORIZER	17
	5.4 CREATE A NEW RESOURCE AND METHOD	18
	5.5 UPDATE THE WEBSITE CONFIGURATION	21
6	CONCLUSION	22
	APPENDIX	23
	A. SCREEN SHOTS	24

	B. SOURCE CODE	26
7	REFERENCES	29

FIGURE NO.	LIST OF FIGURES	PAGE NO.
	TITLE	
1.2.1	SERVERLESS ARCHITECTURE	2
2.1.1	STATIC WEB HOSTING	3
2.1.2	AWS AMPLIFY	4
2.2.1	WEBSITE URL CONSOLE	5
3.1.1	COGNITO USER POOL	6
3.1.2	COGNITO USERPOOL AUTHENT	7
4.1.1	STRUCTURE TOOLS OF BACKEND	9
4.2.1	DYNAMO DB TABLE	11
4.3.1	IAM POLICIES	12
4.4.1	LAMBDA FUNCTION	14
5.1.1	API GATEWAY	15
5.3.1	AUTHORIZER	17
5.4.1	API RESOURCE	18
5.4.2	CREATE OF RIDE METHODS	19
5.4.3	POST SET UP	20

LIST OF ABBREVIATIONS

AWS- AMAZON WEB SERVICES

HTML- HYPERTEXT MARKUP LANGUAGE

REST OR RESTful- REPRESENTATIONAL STATE TRANSFER

API- APPLICATION PROGRAM INTERFACE

CSS- CASCADING STYLE SHEETS

HTTP- HYPERTEXT TRANSFER PROTOCOL

IAM- IDENTITY AND ACCESS MANAGEMENT

CLI- COMMAND LINE INTERFACE

URL- UNIFORM RESOURCE LOCATOR

JWT- JSON WEB TOKEN

JSON- JAVASCRIPT OBJECT NOTATION

SRP- SECURE REMOTE PASSWORD

S3- SIMPLE STORAGE SERVICE

ARN- AMAZON RESOURCE NAME

AJAX- ASYNCHRONOUS JAVASCRIPT AND XML

XML- EXTENSIBLE MARKUP LANGUAGE

POST- POWER-ON-SELF-TEST

CORS- CROSS ORIGIN RESOURCE SHARING

VPC- VIRTUAL PRIVATE CLOUD

CHAPTER 1

INTRODUCTION

Serverless computing allows you to build and run applications and services without thinking about servers. With serverless computing, your application still runs on servers, but all the server management is done by AWS. Using AWS and its Serverless Platform, you can build and deploy applications on cost-effective services that provide built-in application availability and flexible scaling capabilities. This lets you focus on your application code instead of worrying about provisioning, configuring, and managing servers. In this project, you'll create a serverless web application that enables users to request unicorn rides from the wild Rydes fleet. The application will present users with an HTML based user interface for indicating the location where they would like to be picked up and will interface on the backend with a RESTful web service to submit the request and dispatch a nearby unicorn. The application will also provide facilities for users to register with the service and log in before requesting rides. The application architecture

1.1 MODULES

This project is divided into five modules. Each module describes a scenario of what we're going to build and step-by-step directions to help you implement the architecture and verify your work.

- Static Web Hosting
- User Management
- Serverless Backend
- RESTful API

1.2 APPLICATION ARCHITECTURE

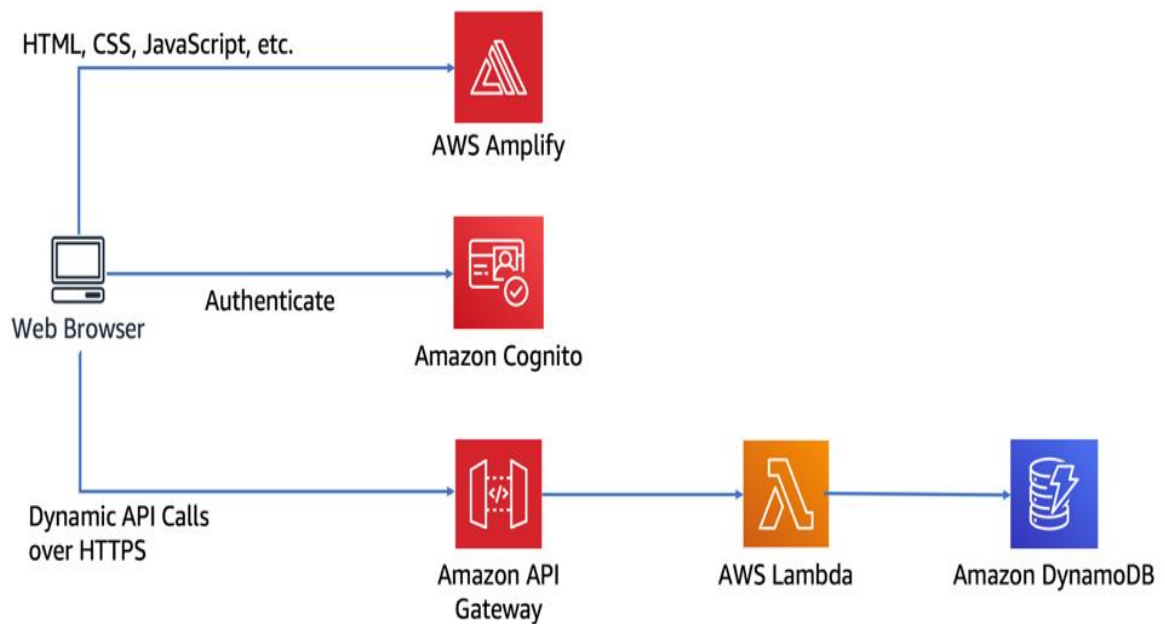


FIG 1.2.1 SERVERLESS ARCHITECTURE

The application architecture uses AWS Lambda, Amazon API Gateway, Amazon Dynamo DB, Amazon Cognito, and AWS Amplify Console. 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. Amazon Cognito provides user management and authentication functions to secure the backend API. Finally, DynamoDB provides a persistence layer where data can be stored by the API's Lambda function.

CHAPTER 2

HOST A STATIC WEBSITE

A web hosting service (often shortened to web host) is a type of Internet hosting service that allows individuals and organizations to make their website accessible via the World Wide Web. Web hosts are companies that provide space on a server owned or leased for use by clients, as well as providing Internet connectivity, typically in a data center. Now you'll configure AWS Amplify to host the static resources for your web application with continuous deployment built-in. The Amplify Console provides a git-based workflow for continuous deployment & hosting of fullstack web apps. In subsequent modules you'll add dynamic functionality to these pages using JavaScript to call remote RESTful APIs built with AWS Lambda and Amazon API Gateway.

2.1 ARCHITECTURE OVERVIEW

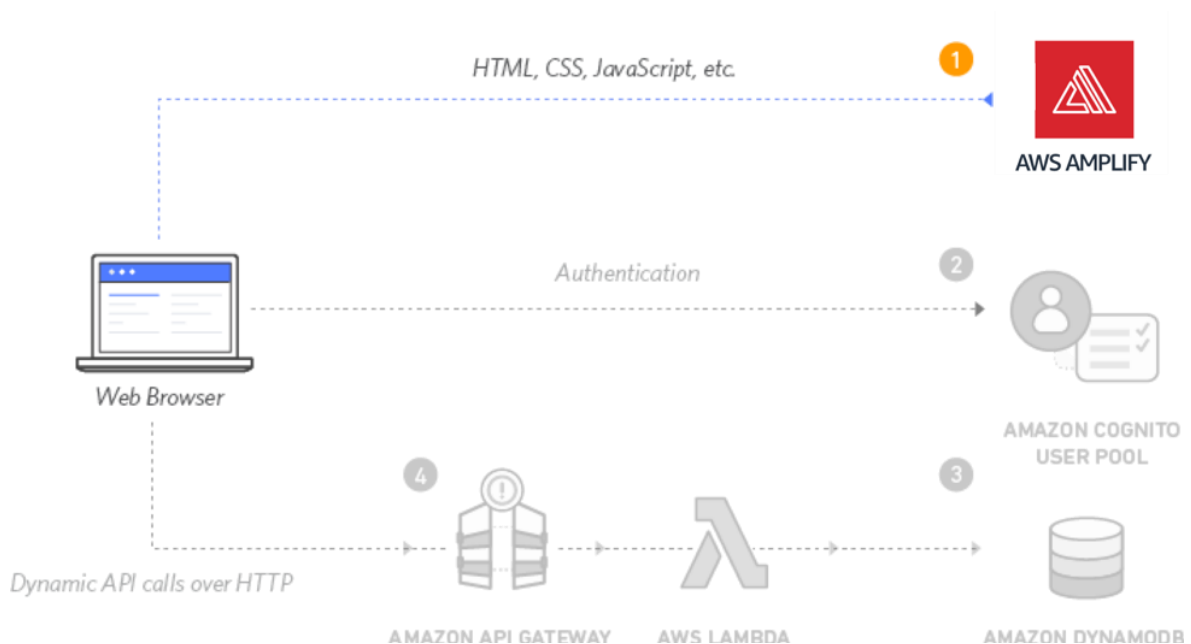


FIG 2.1.1 Static Web Hosting

The architecture for this module is very straightforward. All of your static web content including HTML, CSS, JavaScript, images and other files will be managed by AWS

Amplify Console. Your end users will then access your site using the public website URL exposed by AWS Amplify Console. You don't need to run any web servers or use other services in order to make your site available. The architecture for this module is very straightforward. All of your static web content including HTML, CSS, JavaScript, images and other files will be managed by AWS Amplify Console. Your end users will then access your site using the public website URL exposed by AWS Amplify Console. You don't need to run any web servers or use other services in order to make your site available.

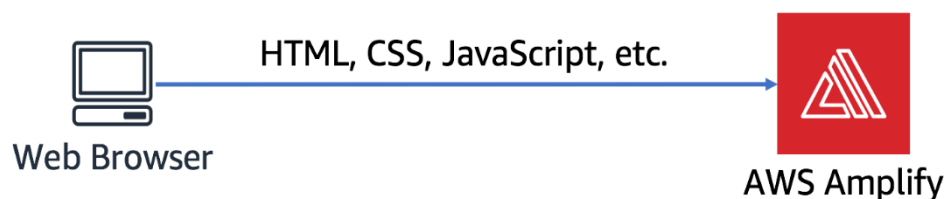


FIG 2.1.2 AWS Amplify

2.2 CODECOMMIT REPOSITORY

CodeCommit is a secure, highly scalable, managed source control service that hosts private Git repositories. CodeCommit eliminates the need for you to manage your own source control system or worry about scaling its infrastructure. You can use CodeCommit to store anything from code to binaries. It supports the standard functionality of Git, so it works seamlessly with your existing Git-based tools.

With CodeCommit, you can:

- Benefit from a fully managed service hosted by AWS. CodeCommit provides high service availability and durability and eliminates the administrative overhead of managing your own hardware and software. There is no hardware to provision and scale and no server software to install, configure, and update.
- Store your code securely. CodeCommit repositories are encrypted at rest as well as in transit.
- Work collaboratively on code. CodeCommit repositories support pull requests, where users can review and comment on each other's code

- changes before merging them to branches; notifications that automatically send emails to users about pull requests and comments; and more.
- Easily scale your version control projects. CodeCommit repositories can scale up to meet your development needs. The service can handle repositories with large numbers of files or branches, large file sizes, and lengthy revision histories.
 - Store anything, anytime. CodeCommit has no limit on the size of your repositories or on the file types you can store.
 - Integrate with other AWS and third-party services. CodeCommit keeps your repositories close to your other production resources in the AWS Cloud, which helps increase the speed and frequency of your development lifecycle. It is integrated with IAM and can be used with other AWS services and in parallel with other repositories. For more information, see Product and service integrations with AWS CodeCommit.
 - Easily migrate files from other remote repositories. You can migrate to CodeCommit from any Git-based repository.
 - Use the Git tools you already know. CodeCommit supports Git commands as well as its own AWS CLI commands and APIs.

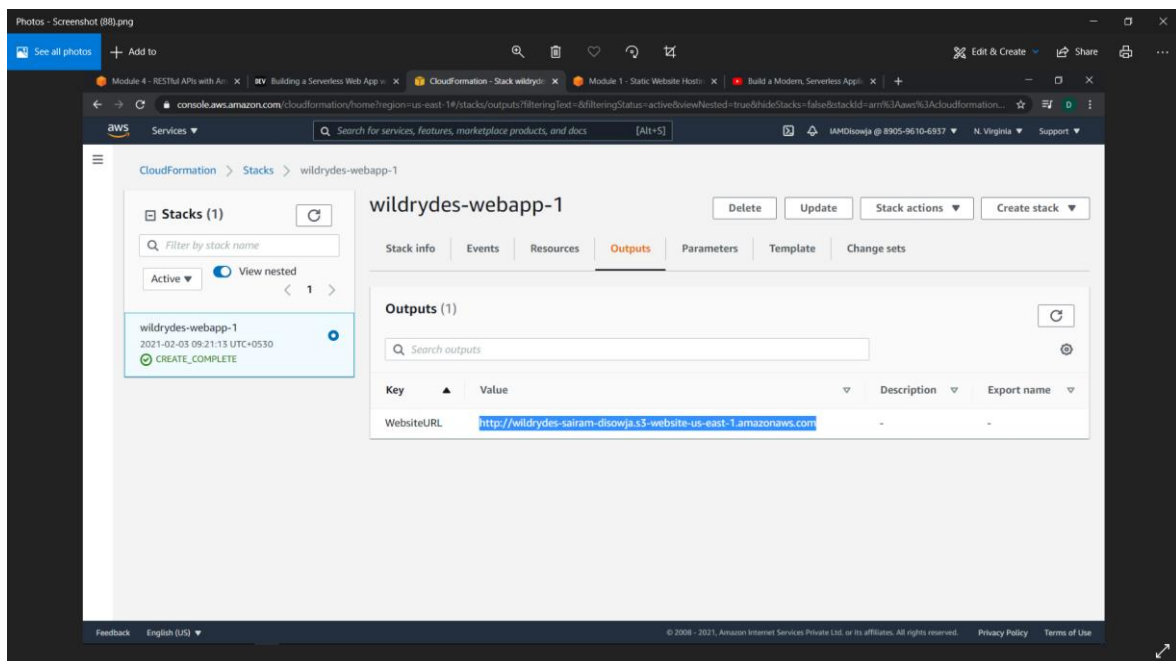


FIG 2.2.1 WEBSITE URL CONSOLE

CHAPTER 3

USER MANAGEMENT

INTRODUCTION

Amazon Cognito user pool used to manage your users accounts. You'll deploy pages that enable customers to register as a new user, verify their email address, and sign into the site. When users visit your website they will first register a new user account. For the purposes of this workshop we'll only require them to provide an email address and password to register. However, you can configure Amazon Cognito to require additional attributes in your own applications.

3.1 ARCHITECTURE DIAGRAM

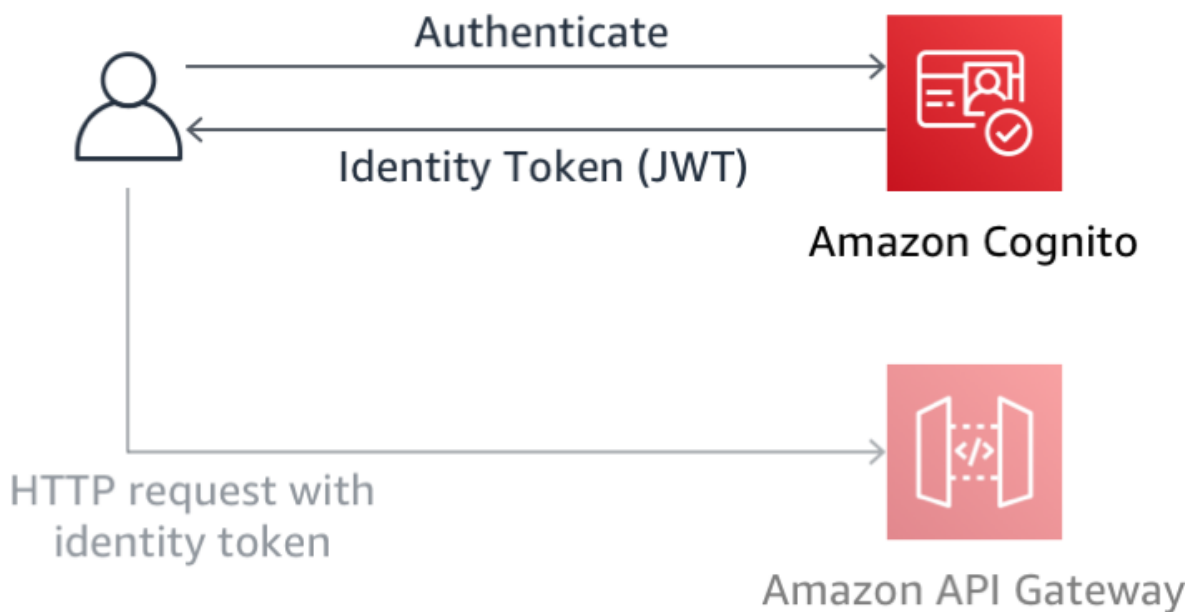


FIG 3.1.1 COGNITO USER POOL

When users visit your website they will first register a new user account. For the purposes of this workshop we'll only require them to provide an email address and password to register. However, you can configure Amazon Cognito to require additional attributes in your own applications.

After users submit their registration, Amazon Cognito will send a confirmation email with a verification code to the address they provided. To confirm their account, users will return to your site and enter their email address and the verification code they received. You can also confirm user accounts using the Amazon Cognito console if you want to use fake email addresses for testing.

After users have a confirmed account (either using the email verification process or a manual confirmation through the console), they will be able to sign in. When users sign in, they enter their username (or email) and password. A JavaScript function then communicates with Amazon Cognito, authenticates using the Secure Remote Password protocol (SRP), and receives back a set of JSON Web Tokens (JWT). The JWTs contain claims about the identity of the user and will be used in the next module to authenticate against the RESTful API you build with Amazon FI

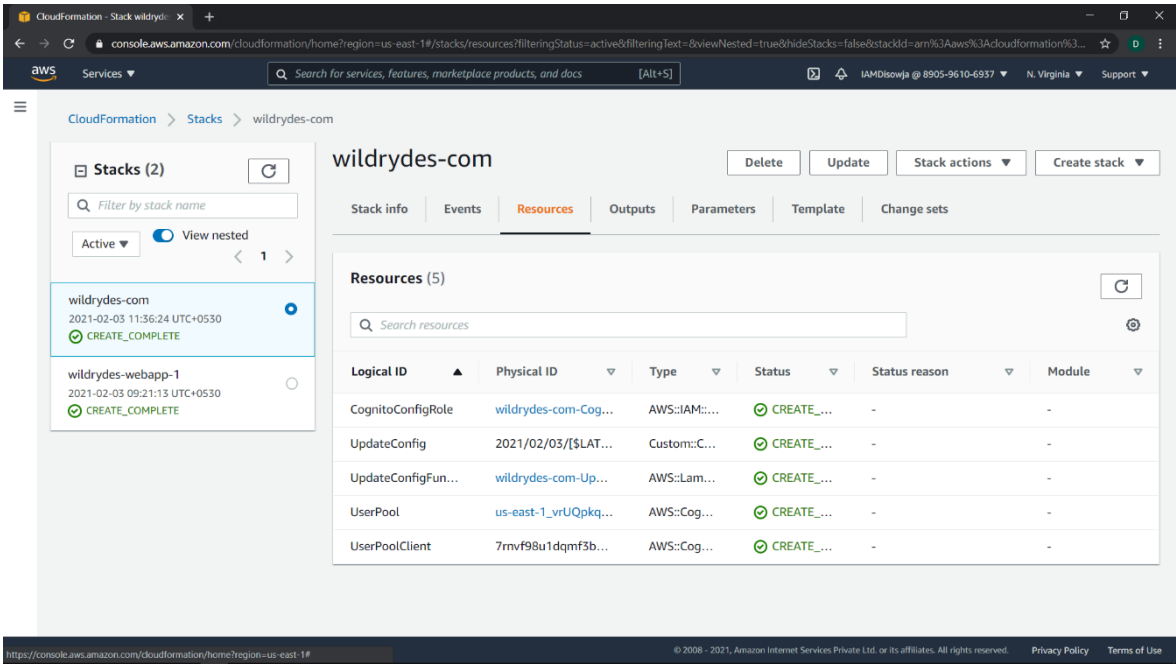


FIG 3.1.2 COGNITO USERPOOL AUTHENTICATION

3.2 COGNITO USERPOOL USING AWS AMPLIFY CLIENT

The AWS Amplify Authentication module provides Authentication APIs and building blocks for developers who want to create user authentication experiences. Amazon Cognito User Pools is a full-featured user directory service to handle user registration, authentication, and account recovery. Amazon Cognito Federated Identities on the other hand, is a way to authorize your users to use AWS services. Amplify interfaces with User Pools to store your user information. This includes federation with other OpenID providers like Facebook & Google. Amplify also uses Federated Identities to manage user access to AWS Resources, like allowing a user to upload a file to an S3 bucket. The Amplify CLI automates the access control policies for these AWS resources and provides fine grained access controls via GraphQL for protecting data in your APIs.

3.3 USER POOL'S APP CLIENT

- Go to the Amazon Cognito Console
- Choose Manage User Pools. Here you will see a new userpool generated by the AWS Amplify CLI

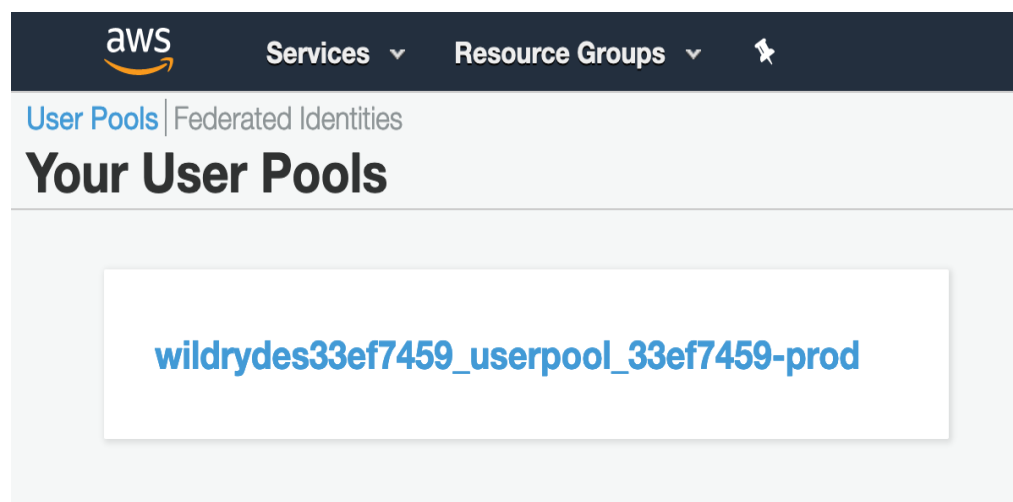


FIG 3.3.1 COGNITO USER POOL

- Click the new user pool to open the Pool Details page
- From the Pool Details page, select App clients from the General settings section in the left navigation bar.

CHAPTER 4

BUILD A SERVERLESS BACKEND

In this module you'll use AWS Lambda and Amazon DynamoDB to build a backend process for handling requests for your web application. The browser application that you deployed in the first module allows users to request that a unicorn be sent to a location of their choice. In order to fulfill those requests, the JavaScript running in the browser will need to invoke a service running in the cloud.

4.1 ARCHITECTURE OVERVIEW

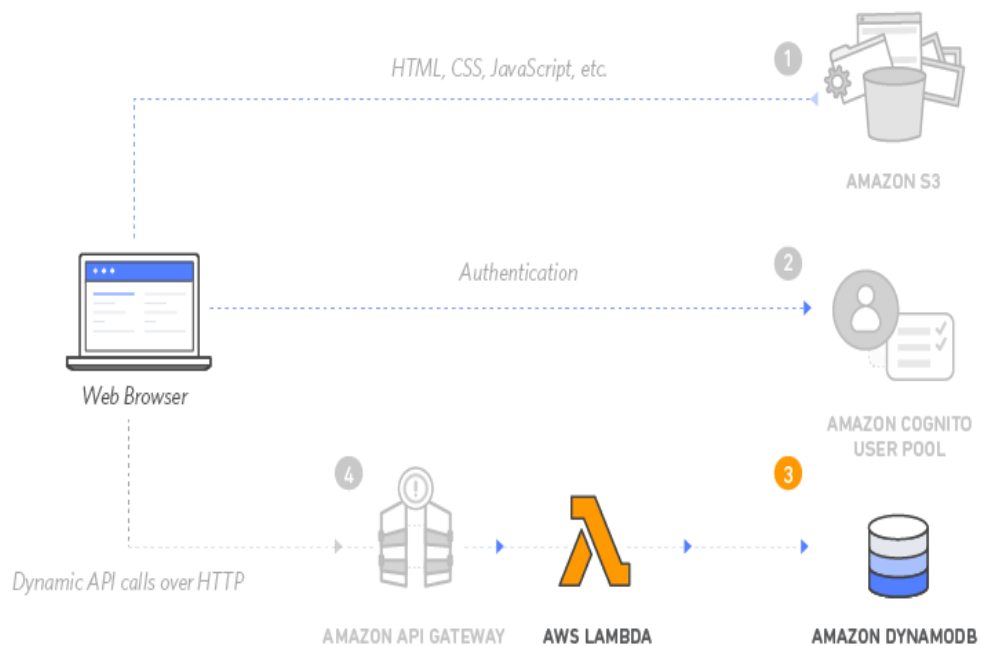


FIG 4.1.1 STRUCTURE TOOLS OF BACKEND

In this section you'll use AWS Lambda and Amazon DynamoDB to build a backend process for handling requests from your web application. The browser application that you deployed in the first module allows users to request that a unicorn be sent to a location of their choice. In order to fulfill those requests, the JavaScript running in the browser invokes a service running in the cloud.

You'll implement a Lambda function that will be invoked each time a user requests a unicorn. The function selects a unicorn from the fleet, records the request in a DynamoDB table, and responds to the front-end application with details about the dispatched unicorn.

4.2 CREATE AN AMAZON DYNAMODB TABLE

We Use the Amazon DynamoDB Console to create a new DynamoDB table. Call your table Rides and give it a partition key called RideId with type String. The table name and partition key are case sensitive. Make sure you use the exact IDs provided. Use the defaults for all other settings.

After you've created the table, record the ARN for use in the next step.

Step-by-step directions

- Go to the Amazon DynamoDB Console
- Choose Create table.
- Enter Rides for the Table name. This field is case sensitive.
- Enter RideId for the Partition key and select String for the key type. This field is case sensitive.

Create DynamoDB table

[Tutorial](#)

DynamoDB is a schema-less database that only requires a table name and primary key. The table's primary key is made up of one or two attributes that uniquely identify items, partition the data, and sort data within each partition.

Table name* ⓘ

Primary key* Partition key

ⓘ

☐ Add sort key

Table settings

Default settings provide the fastest way to get started with your table. You can modify these default settings now or after your table has been created.

☒ Use default settings

- No secondary indexes.
- Provisioned capacity set to 5 reads and 5 writes.
- Basic alarms with 80% upper threshold using SNS topic "dynamodb".
- On-Demand Backup and Restore Enabled **NEW!**

ⓘ You do not have the required role to enable Auto Scaling by default.
Please refer to [documentation](#).

Additional charges may apply if you exceed the AWS Free Tier levels for CloudWatch or Simple Notification Service. Advanced alarm settings are available in the CloudWatch management console.

[Cancel](#)[Create](#)

FIG 4.2.1 DYNAMODB TABLE

4.3 CREATE AN IAM ROLE FOR LAMBDA FUNCTION

Every Lambda function has an IAM role associated with it. This role defines what other AWS services the function is allowed to interact with. In this workshop, you create an IAM role that grants your Lambda function permission to write logs to Amazon CloudWatch Logs and access to write items to your DynamoDB table.

Use the IAM console to create a new role. Name it WildRydesLambda and select AWS Lambda for the role type. You'll need to attach policies that grant your function permissions to write to Amazon CloudWatch Logs and put items to your DynamoDB table. Attach the managed policy called AWS Lambda Basic Execution Role to this role to grant the necessary CloudWatch Logs permissions. Also, create a custom

inline policy for your role that allows the dynamodb:putItem action for the table you created in the previous section.

- While in the IAM Console on the roles page type WildRydesLambda into the filter box on the Roles page and choose the role you just created.
- On the Permissions tab, choose the Add inline policy link in the upper right corner to create a new inline policy.

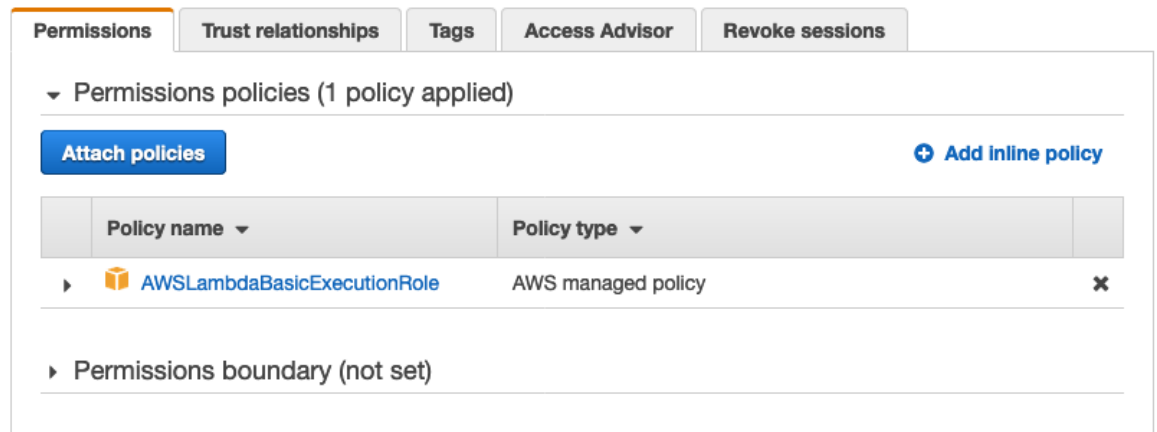


FIG 4.3.1 IAM POLICIES

- Select Choose a service Begin typing DynamoDB into the search box labeled Find a service and select DynamoDB when it appears.
- Choose Select actions.
- Begin typing PutItem into the search box labeled Filter actions and check the box next to PutItem when it appears.
- Select the Resources section.
- With the Specific option selected, choose the Add ARN link in the table section.
- Paste the ARN of the table you created in the previous section in the Specify ARN for table field, and choose Add.
- Choose Review Policy Enter DynamoDBWriteAccess for the policy name and choose Create policy.

4.4 CREATION OF LAMBDA FUNCTION

AWS Lambda runs your code in response to events such as an HTTP request. In this step you build a function that processes API requests from the web application to dispatch a unicorn. In the next module you use Amazon API Gateway to create a RESTful API that exposes an HTTP endpoint that can be invoked from your users' browsers. Then you connect the Lambda function you create in this step to that API to create a fully functional backend for your web application.

Step-by-step directions

- Go to the AWS Lambda console
- Click Create function
- Keep the default Author from scratch card selected
- Enter RequestUnicorn in the Name field
- Select Node.js 12.x for the Runtime
- Expand *Change default execution role* under Permissions
- Ensure Use an Existing Role is selected from the Role dropdown
- Select WildRydesLambda from the Existing Role Dropdown
- Choose Create Function

Create function Info

Choose one of the following options to create your function.

Author from scratch ☒

Start with a simple Hello World example.



Use a blueprint ☐

Build a Lambda application from sample code and configuration presets for common use cases.



Browse serverless app repository ☐

Deploy a sample Lambda application from the AWS Serverless Application Repository.



Basic information

Function name
Enter a name that describes the purpose of your function.

RequestUnicorn

Use only letters, numbers, hyphens, or underscores with no spaces.

Runtime Info
Choose the language to use to write your function.

Node.js 12.x

Permissions Info
By default, Lambda will create an execution role with permissions to upload logs to Amazon CloudWatch Logs. You can customize this default role later when adding triggers.

▼ Change default execution role

Execution role
Choose a role that defines the permissions of your function. To create a custom role, go to the [IAM console](#).

☐ Create a new role with basic Lambda permissions

☒ Use an existing role

☐ Create a new role from AWS policy templates

Existing role
Choose an existing role that you've created to be used with this Lambda function. The role must have permission to upload logs to Amazon CloudWatch Logs.

WildRydesLambda

↺

[View the WildRydesLambda role](#) on the IAM console.

► Advanced settings

Cancel>Create function

FIG 4.4.1 LAMBDA FUNCTION

19

CHAPTER 5

DEPLOY RESTFUL API

INTRODUCTION

In this module we use API Gateway to expose the Lambda function you built in the previous module as a RESTful API. This API will be accessible on the public Internet. It will be secured using the Amazon Cognito user pool you created in the User Management module. Using this configuration you will then turn your statically hosted website into a dynamic web application by adding client-side JavaScript that makes AJAX calls to the exposed APIs.

5.1 ARCHITECTURE OVERVIEW

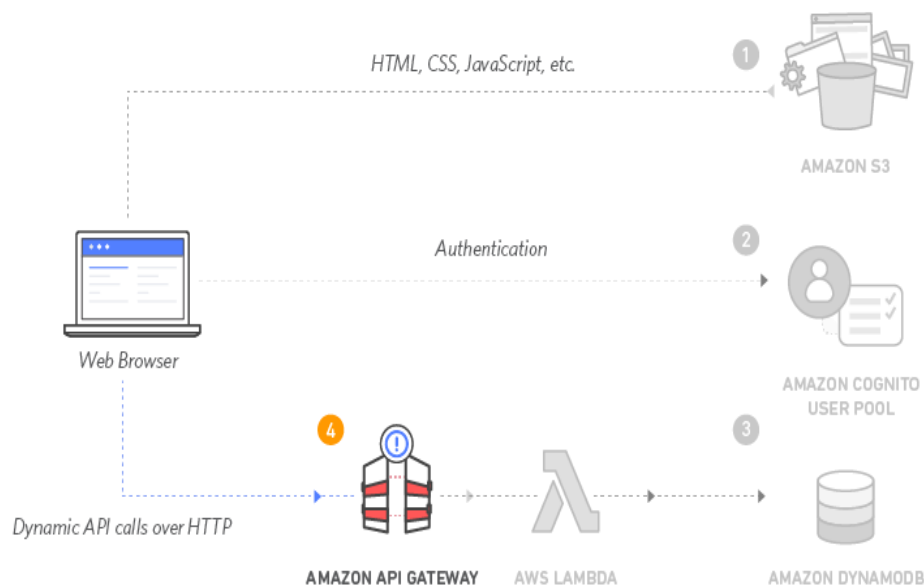


FIG 5.1.1 API GATEWAY

The diagram above shows how the API Gateway component you will build in this module integrates with the existing components you built previously. The grayed out items are pieces you have already implemented in previous steps.

The static website you deployed in the first module already has a page configured to interact with the API you'll build in this module. The page at `/ride.html` has a simple map-based interface for requesting a unicorn ride. After authenticating using the `/signin.html` page, your users will be able to select their pickup location by clicking a point on the map and then requesting a ride by choosing the "Request Unicorn" button in the upper right corner.

This module will focus on the steps required to build the cloud components of the API, but if you're interested in how the browser code works that calls this API, you can inspect the `ride.js` file of the website. In this case the application uses jQuery's `ajax()` method to make the remote request.

5.2 CREATE A NEW REST API

Step-by-step directions

- Go to the Amazon API Gateway Console
- On the REST API card, choose **Build**.
- In the section *Create new API* select *New API* to clear the example API definition.
- Enter WildRydes for the API Name.
- Select Regional from the Endpoint Type dropdown.
- Choose Create API

Create new API

In Amazon API Gateway, a REST API refers to a collection of resources and methods that can be invoked through HTTPS endpoints.

☒ New API ☐ Clone from existing API ☐ Import from Swagger or Open API 3 ☐ Example API

Settings

Choose a friendly name and description for your API.

API name*	WildRydes
Description	
Endpoint Type	Regional ⓘ

* Required

Create API

FIG 5.2.1 API GATEWAY

5.3 CREATION OF COGNITO USERPOOL AUTHORIZER

Amazon API Gateway can use the JWT tokens returned by Cognito User Pools to authenticate API calls.

Step-by-step directions

- Under your newly created API, choose Authorizers.
- Choose Create New Authorizer.
- Enter WildRydes for the Authorizer name.
- Select Cognito for the type.
- In the Region drop-down under Cognito User Pool, select the Region where you created your Cognito user pool in the User Management module (by default the current region should be selected).
- Enter wildrydes in the Cognito User Pool input, the name will auto-complete and allow you to select the name of the user pool that was generated when the user pool was created.
- Enter Authorization for the Token Source.
- Choose Create.

Create Authorizer

Name *

WildRydes

Type * ⓘ

☐ Lambda ☒ Cognito

Cognito User Pool * ⓘ

us-east-2 WildRydes

Token Source * ⓘ **Token Validation ⓘ**

Authorization

Create Cancel

FIG 5.3.1 AUTHORIZER

5.4 CREATE A NEW RESOURCE AND METHOD

Create a new resource called `/ride` within your API. Then create a POST method for that resource and configure it to use a Lambda proxy integration backed by the RequestUnicorn function you created in the first step of this module.

Step-by-step directions

- In the left nav, click on Resources under your WildRydes API.
- From the Actions dropdown select Create Resource.
- Enter ride as the Resource Name.
- Ensure the Resource Path is set to ride.
- Select Enable API Gateway CORS for the resource.
- Choose Create Resource.

New Child Resource

Use this page to create a new child resource for your resource.

Configure as [proxy resource](#)

☐ ⓘ

Resource Name*

ride

Resource Path*

/ ride

You can add path parameters using brackets. For example, the resource path {username} represents a path parameter called 'username'. Configuring /{proxy+} as a proxy resource catches all requests to its sub-resources. For example, it works for a GET request to /foo. To handle requests to /, add a new ANY method on the / resource.

Enable API Gateway CORS

☒ ⓘ

* Required

Cancel

Create Resource

FIG 5.4.1 API RESOURCE

- With the newly created /ride resource selected, from the Action dropdown select Create Method.
- Select POST from the new dropdown that appears, then click the checkmark.

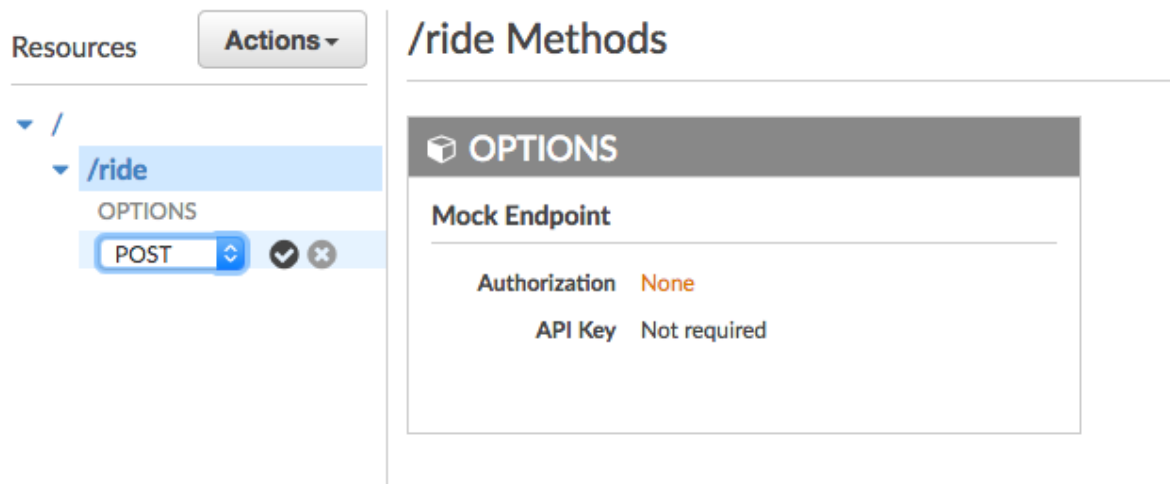


Fig 5.4.2 CREATE OF RIDE METHODS

- Select Lambda Function for the integration type.
- Check the box for Use Lambda Proxy integration.
- Select the Region you are using for Lambda Region.
- Enter the name of the function you created in the previous module, Request unicorn, for Lambda Function.
- Choose Save. Please note, if you get an error that you function does not exist, check that the region you selected matches the one you used in the previous module.

/ride - POST - Setup



Choose the integration point for your new method.

Integration type ☒ Lambda Function ⓘ
☐ HTTP ⓘ
☐ Mock ⓘ
☐ AWS Service ⓘ
☐ VPC Link ⓘ

Use Lambda Proxy integration ☒ ⓘ

Lambda Region

Lambda Function ⓘ

Use Default Timeout ☒ ⓘ

Save

FIG 5.4.3 POST SETUP

- When prompted to give Amazon API Gateway permission to invoke your function, choose OK.
- Select the Method Request card.
- Choose the pencil icon next to Authorization.
- Select the WildRydes Cognito user pool authorizer from the drop-down list, and click the checkmark icon.

DEPLOYING API

From the Amazon API Gateway console, choose Actions, Deploy API. You'll be prompted to create a new stage. You can use prod for the stage name.

5.5 UPDATE THE WEBSITE CONFIGURATION

Update the `/src/config.js` file in your website deployment to include the invoke URL of the stage you just created. You should copy the invoke URL directly from the top of the stage editor page on the Amazon API Gateway console and paste it into the `config.api.invokeUrl` key of your site `/src/config.js` file. Make sure when you update the config file it still contains the updates you made in the previous module for your Cognito user pool.

- On your Cloud9 development environment open `src/config.js`
- Update the invoke Url setting under the api key in the `config.js` file. Set the value to the Invoke URL for the deployment stage you created in the previous section.
- Save the modified file making sure the filename is still `config.js`.
- Commit the changes to your git repository:

CHAPTER 5

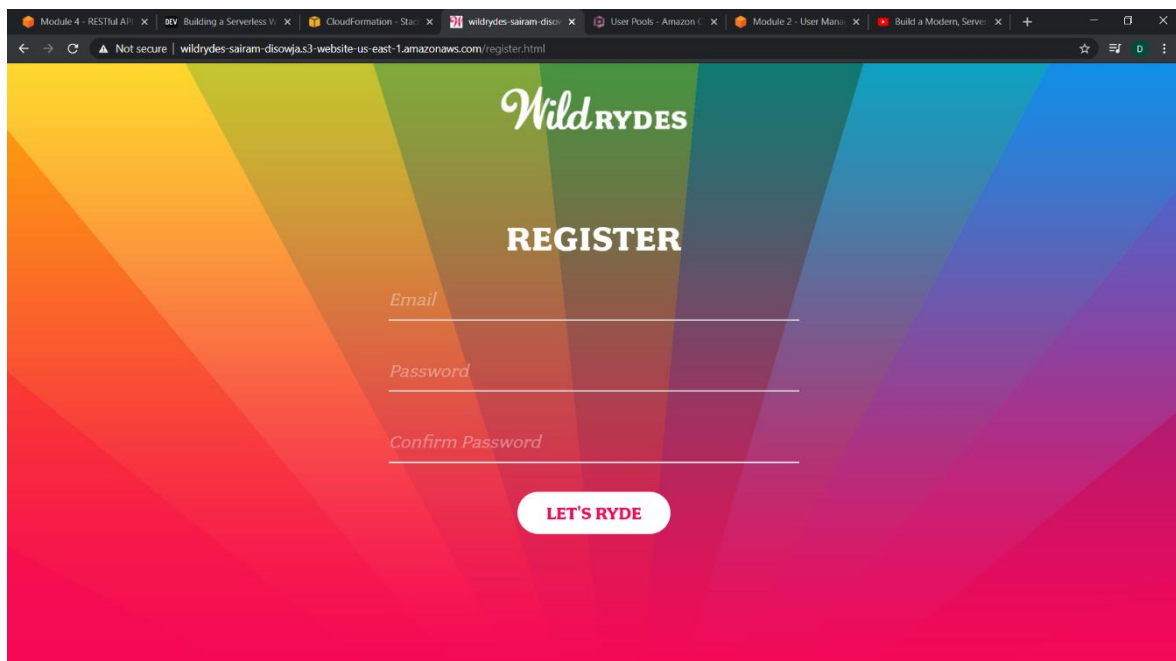
CONCLUSION

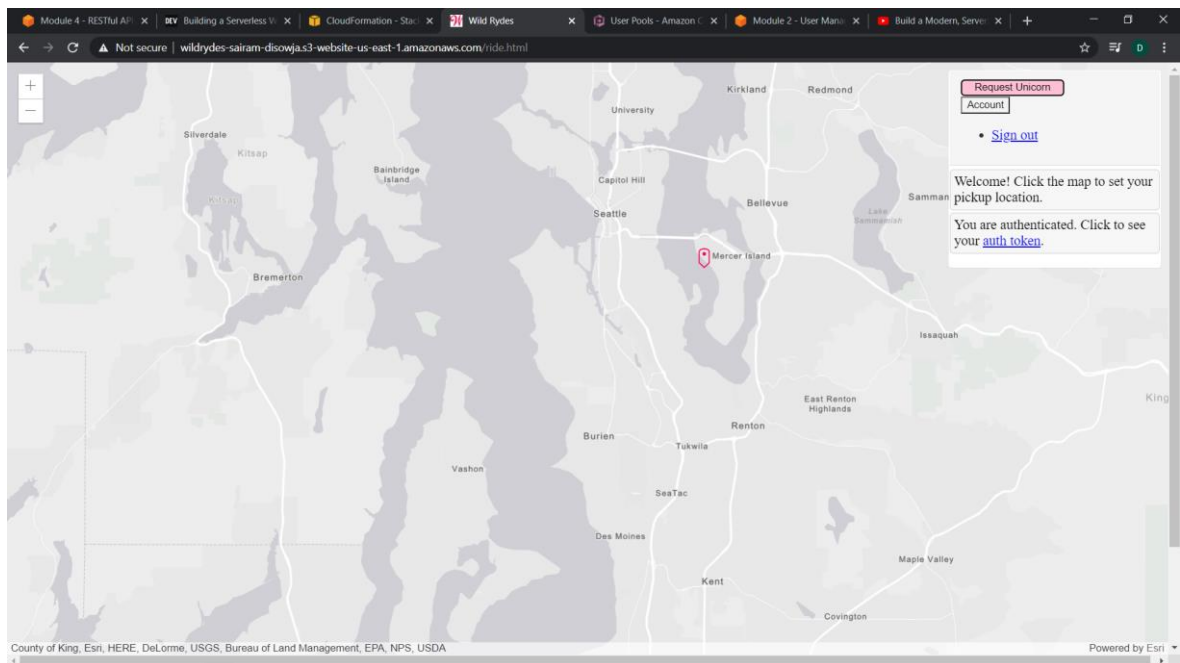
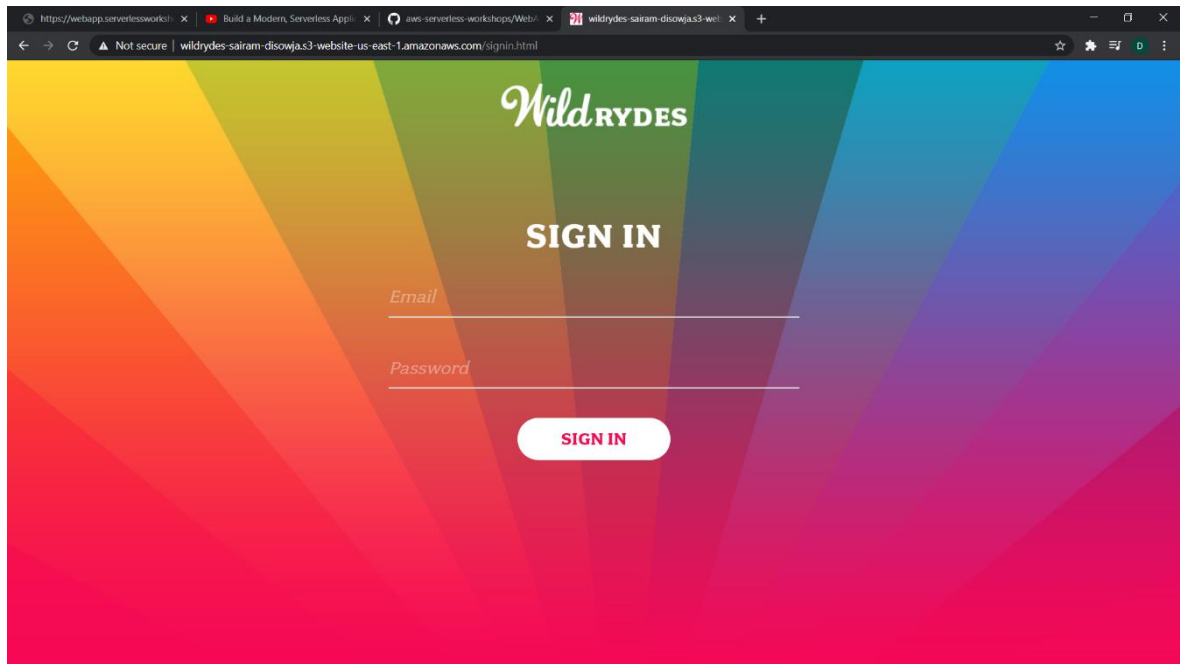
In this Project, you will go through the steps to all the resources you created throughout this project. You will use your AWS Amplify app, an Amazon Cognito User Pool, an AWS Lambda function, an IAM role, a DynamoDB table, a REST API, and a CloudWatch Log. Building serverless applications on AWS relieves you of the responsibilities and constraints that servers introduce. Using AWS Lambda as your serverless logic layer enables you to build faster and focus your development efforts on what differentiates your application. Alongside Lambda, AWS provides additional serverless capabilities so that you can build robust, performant, event-driven, reliable, secure, and cost-effective applications.

Finally using these AWS Resources tools we will build a website of wild rides. Which we can request a unicorn for a ride to desired place. The unicorn will be different types of colours and according to customer need they can book horse chariot or single horse for suitable and comfortable rides. Using AWS resources its time efficient we will deploy code in Lambda and run in that so that we can request a different types of unicorn. This shows a map of geographical location where the customer request a unicorn.

APPENDIX

A. SCREEN SHOTS





B.SOURCE CODE

```
const randomBytes = require('crypto').randomBytes;

const AWS = require('aws-sdk');

const ddb = new AWS.DynamoDB.DocumentClient();

const fleet = [
  {
    Name: 'Bucephalus',
    Color: 'Golden',
    Gender: 'Male',
  },
  {
    Name: 'Shadowfax',
    Color: 'White',
    Gender: 'Male',
  },
  {
    Name: 'Rocinante',
    Color: 'Yellow',
    Gender: 'Female',
  },
];

exports.handler = (event, context, callback) => {
  if (!event.requestContext.authorizer) {
    errorResponse('Authorization not configured', context.awsRequestId,
callback);
    return;
  }

  const rideId = toUrlString(randomBytes(16));
```

```

console.log('Received event (', rideId, '): ', event);

// Because we're using a Cognito User Pools authorizer, all of the claims
// included in the authentication token are provided in the request context.
// This includes the username as well as other attributes.
const username = event.requestContext.authorizer.claims['cognito:username'];

// The body field of the event in a proxy integration is a raw string.
// In order to extract meaningful values, we need to first parse this string
// into an object. A more robust implementation might inspect the Content-Type
// header first and use a different parsing strategy based on that value.
const requestBody = JSON.parse(event.body);

const pickupLocation = requestBody.PickupLocation;

const unicorn = findUnicorn(pickupLocation);

recordRide(rideId, username, unicorn).then(() => {
  // You can use the callback function to provide a return value from your
Node.js
  // Lambda functions. The first parameter is used for failed invocations. The
  // second parameter specifies the result data of the invocation.

  // Because this Lambda function is called by an API Gateway proxy
integration
  // the result object must use the following structure.
  callback(null, {
    statusCode: 201,
    body: JSON.stringify({
      RideId: rideId,
      Unicorn: unicorn,
      UnicornName: unicorn.Name,
      Eta: '30 seconds',
      Rider: username,

```

```

    }),
    headers: {
        'Access-Control-Allow-Origin': '*',
    },
});
}).catch((err) => {
    console.error(err);

    // If there is an error during processing, catch it and return
    // from the Lambda function successfully. Specify a 500 HTTP status
    // code and provide an error message in the body. This will provide a
    // more meaningful error response to the end client.
    errorResponse(err.message, context.awsRequestId, callback)
});
};

// This is where you would implement logic to find the optimal unicorn for
// this ride (possibly invoking another Lambda function as a microservice.)
// For simplicity, we'll just pick a unicorn at random.
function findUnicorn(pickupLocation) {
    console.log('Finding unicorn for ', pickupLocation.Latitude, ', ',
pickupLocation.Longitude);
    return fleet[Math.floor(Math.random() * fleet.length)];
}

function recordRide(rideId, username, unicorn) {
    return ddb.put({
        TableName: 'Rides',
        Item: {
            RideId: rideId,
            User: username,
            Unicorn: unicorn,
            UnicornName: unicorn.Name,
            RequestTime: new Date().toISOString(),

```

```

    },
  }).promise();
}

```

```

function toUrlString(buffer) {
  return buffer.toString('base64')
    .replace(/\+/g, '-')
    .replace(/\//g, '_')
    .replace(/=/g, '');
}

```

```

function ErrorResponse(errorMessage, awsRequestId, callback) {
  callback(null, {
    statusCode: 500,
    body: JSON.stringify({
      Error: errorMessage,
      Reference: awsRequestId,
    }),
    headers: {
      'Access-Control-Allow-Origin': '*',
    },
  });
}

```

REFERENCES

- Sbarski, P.kroonenburg: Serverless Architectures on AWS With examples using AWS Lambda. in preparation (2019). URL <https://www.manning.com/books/serverless-architectures-on-AWS>
- American Journal of Software Engineering. 2019, 6(1), 1-10. DOI: 10.12691/ajse-6-1-1
- <https://blog.bitsrc.io/building-a-serverless-webapp-why-you-should-consider-the-monolith-2020-4f0105935589>
- https://www.infoq.com/news/2021/02/serverless-react-apps-aws-lambda/?itm_source=presentations_about_serverless&itm_medium=link&itm_campaign=serverless
- <https://www.serverless.com/blog/how-create-dynamic-website-with-serverless-components-2020/19>
- <https://docs.aws.amazon.com/serverless-application-model/latest/2018/developerguide/what-is-sam.html>
- <https://github.com/aws/serverless-application-model>