Design and Development of a Serverless Web application for management applications

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

Project Submitted To: Professor Subhas Desa

Submitted By: Aishwarya Janardhana Rajur

Table of Contents:

Content	Page number
Executive Summary	3
Project Proposal Overview	4 - 5
Phase - I : Foundation and user management	5 - 11
Phase - II : Backend Integration and Functionality	11 - 21
Phase - III: Build a RESTful API	21 - 28
Project Demo	28 - 30

Wild Rydes Project: Executive Summary

This project outlines the development of a fictional ride-sharing service named Wild Rydes, where users request unicorns instead of cars. This playful concept is a practical introduction to building real-world car booking applications.

Phase 1: Foundation and User Management

- Established a static web application using AWS Amplify for continuous deployment.
- Implemented user management with Amazon Cognito for secure access.

Phase 2: Backend Integration and Functionality

- Created a backend process using AWS Lambda to handle user requests for unicorns.
- Developed a Lambda function named "RequestUnicorn" that:
- Selects a unicorn from a simulated fleet.
- Stores the request details in a DynamoDB table named "Rides."
- Responds with dispatched unicorn information.
- Tested the Lambda function independently.

Phase 3: Build a RESTful API

- Constructed a RESTful API using Amazon API Gateway named "WildRydes" with a public endpoint.
- Secured the API with an Amazon Cognito User Pools Authorizer for user authentication.
- Configured the API to integrate with the "RequestUnicorn" Lambda function.
- Deployed the API and updated the website configuration to interact with it.
- Enabled users to request unicorns through the website and receive confirmation.

Overall Achievement:

This project successfully built a functional prototype of Wild Rydes, demonstrating the core functionalities of a real-world ride-sharing application. Users can interact with the web interface, request unicorns, and receive confirmation - all powered by serverless backend services.

Future Considerations:

- Implement features like unicorn selection, payment processing, and ride tracking.
- Integrate map services for real-time location display.
- Enhance the user interface for a more engaging experience.

Project Proposal Overview:

This project outlines the development of a serverless web application mimicking a ride-sharing service but with a fantastical twist - Wild Rydes uses unicorns instead of cars. This fictional scenario is a practical introduction to building **real-world car booking applications.**

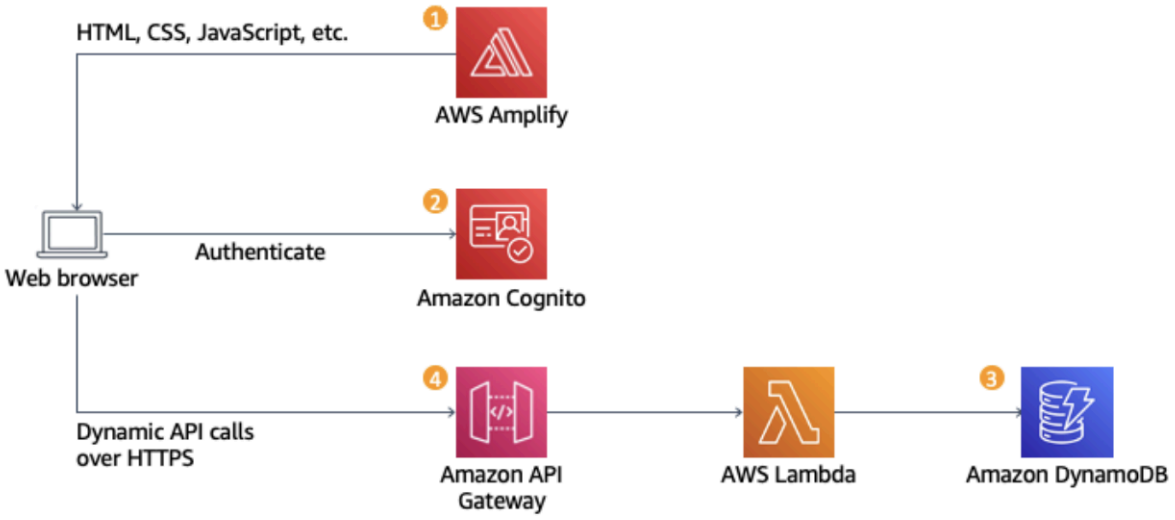
Why Unicorns? A Gateway to Real-World Concepts

While unicorns are mythical creatures, they offer a playful and engaging way to explore the core functionalities of a ride-sharing app:

User Interface: The user interface will focus on functionalities relevant to real-world car booking apps - specifying pick-up locations, logging in/registering users, and requesting rides.

Backend Services: The core logic behind rider requests, driver (unicorn) dispatch, and payment processing can be implemented on the backend, mimicking real-world interactions between users, drivers, and the ridesharing platform.

Application Architecture:
The application architecture uses AWS Lambda, Amazon API Gateway, Amazon DynamoDB, Amazon Cognito, and AWS Amplify Console. Amplify Console provides continuous deployment and hosting of static web resources, including HTML, CSS, JavaScript, and image files, which are loaded into 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 the API's Lambda function can store data.



From the above figure, the functionalities are explained below

1: Static Web Hosting - AWS Amplify hosts static web resources, including HTML, CSS, JavaScript, and image files, which are loaded into the user's browser.

2: User Management - Amazon Cognito provides user management and authentication functions to secure the backend API.

3. Serverless Backend - Amazon DynamoDB provides a persistence layer where data can be stored by the API's Lambda function.

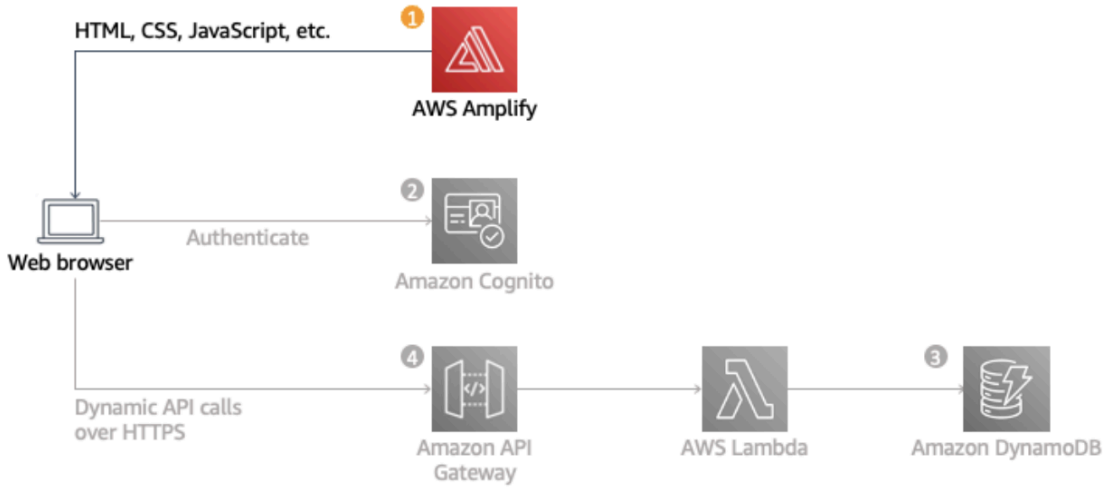
4. RESTful API - JavaScript executed in the browser sends and receives data from a public backend API built using Lambda and API Gateway.

Phase 1: Foundation and User Management

[Section A]

In this phase, I will configure AWS Amplify to host the static resources for web applications with continuous deployment built-in. The Amplify Console provides a git-based workflow for continuously deploying and hosting full-stack web apps. In subsequent phases, I will add dynamic functionality to these pages using JavaScript to call remote RESTful APIs built with AWS Lambda and Amazon API Gateway.

Architecture overview:



The architecture for this phase is straightforward. AWS Amplify Console will manage all static web content, including HTML, CSS, JavaScript, images, and other files. The end users will then access the

site using the public URL that AWS Amplify Console exposes. Here, I don't need to run web servers or use other services to make the site available.

Implementation:

Step 1: Select a region
This web application can be deployed in any AWS Region that supports all the services used, including AWS Amplify, AWS CodeCommit, Amazon Cognito, AWS Lambda, Amazon API Gateway, and Amazon DynamoDB.

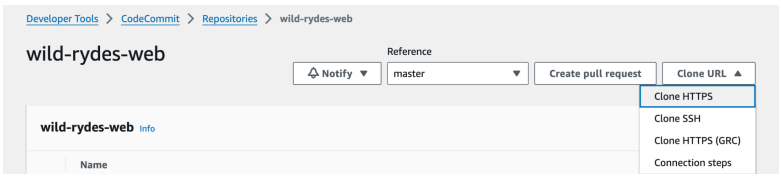
The supported regions are:

US East (N. Virginia)
US East (Ohio)
US West (Oregon)
US West (N.California)
EU (Frankfurt)
EU (Ireland)
EU (London)
Asia Pacific (Tokyo)
Asia Pacific (Seoul)
Asia Pacific (Sydney)
Asia Pacific (Mumbai)

Here, I'm going with the region **US West(N.California)**, as it is closer and supported region for my website.

Step 2: Create a repository using AWS CodeCommit
There are two ways to manage source code: AWS CodeCommit or Github. Here, I will use CodeCommit to store application code.

1)Open the AWS CodeCommit console.
2)Choose Create Repository.
3)Enter wildrydes-site for the Repository name.
4)Choose Create.
5)Select Clone HTTPS from the Clone URL dropdown to copy the HTTPS URL.



6) From your terminal window run git clone and paste the HTTPS URL of the repository.

7)By using AWS CodeCommit to create your git repository and clone it locally, I copied the website content from an existing, publicly-accessible S3 bucket associated and added the content to my repository.

Change directory into your repository and copy the static files from S3 using the following commands

cd wildrydes-site aws s3 cp s3://wildrydes-us-east-1/WebApplication/1_StaticWebHosting/website ./ --recursive

8)Add, commit, and push the git files.

Step 3: Enable web hosting using AWS Amplify

After creating repository with souce code, I will use AWS Amplify to deploy the website I've just committed to git. The Amplify Console takes care of the work of setting up a place to store static web application code and provides a number of helpful capabilities to simplify both the lifecycle of that application as well as enable best practices.

1)Launch the AWS Amplify console.

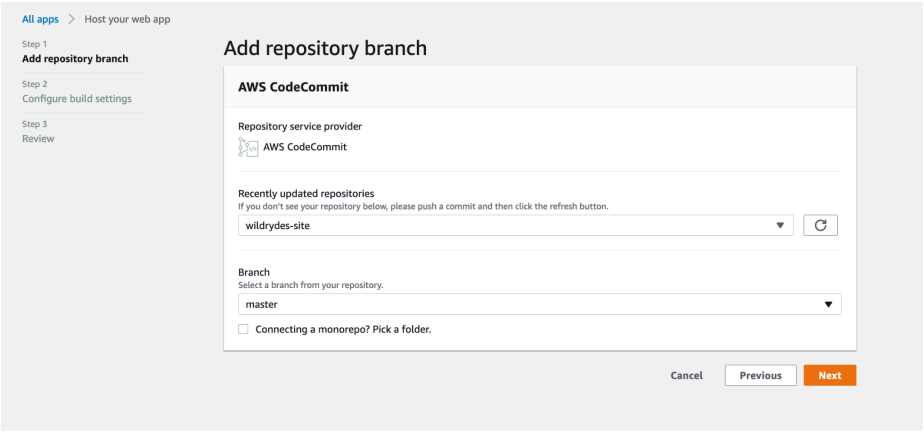
2)Choose Get Started.

3)Under the Amplify Hosting Host your web app header, choose Get Started.

4)On the Get started with Amplify Hosting page, select AWS CodeCommit and choose Continue.

5)On the Add repository branch step, select **wildrydes-site** from the Select a repository dropdown.

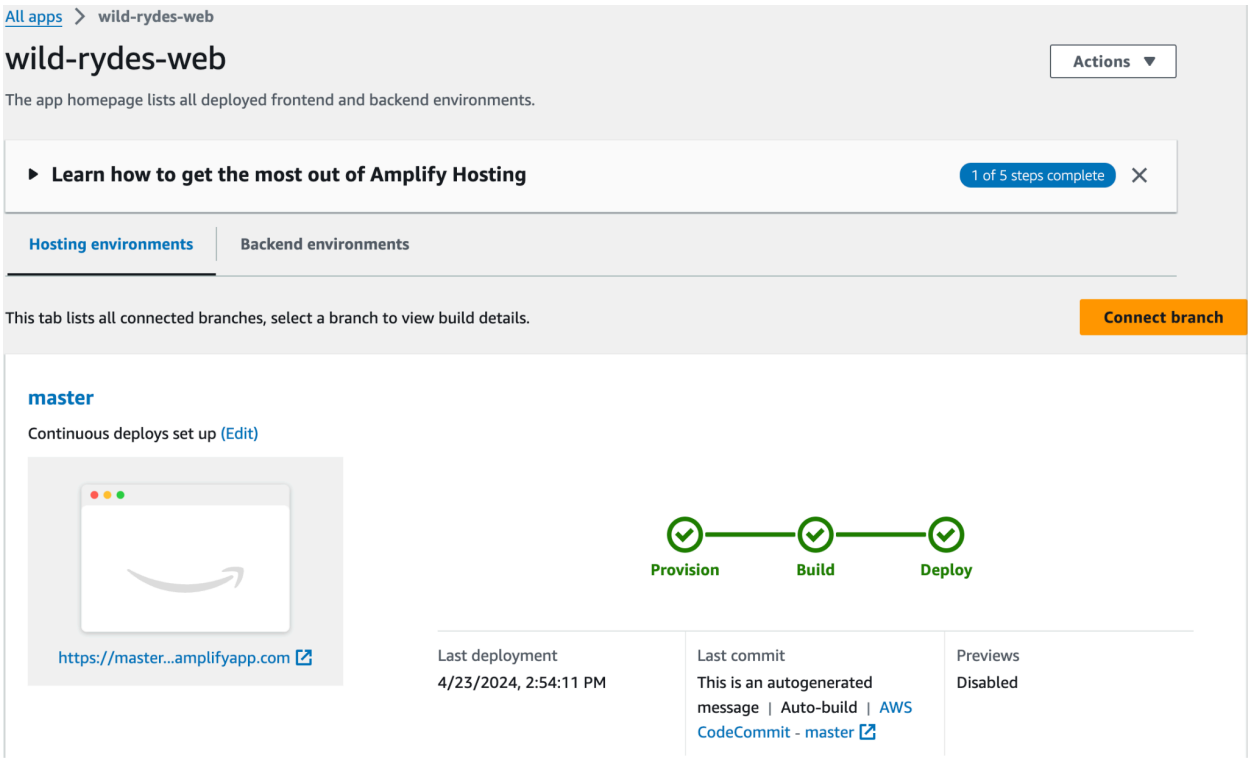
6)In the Branch dropdown select master and choose Next.



7) On the Build settings page, leave all the defaults, select Allow AWS Amplify to automatically deploy all files hosted in your project root directory and choose Next.

8)On the Review page select Save and deploy.

9)The process takes a couple of minutes for Amplify Console to create the necessary resources and to deploy your code.



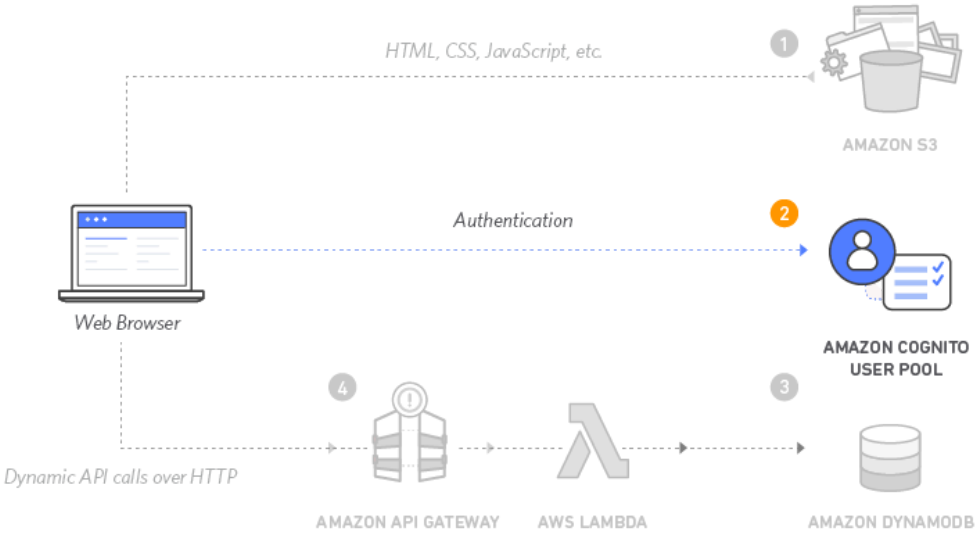
Once, the website is deployed, using the web link generated by AWS, I can access the website. Thus, website is hosted. https://master.d3sqsjnhukdzpo.amplifyapp.com/

Until now, I've created static website which will be the base for the Wild Rydes business. AWS Amplify Console can deploy static websites following a continuous integration and delivery model. It has the capability to build more complicated Javascript framework-based applications, and has features such as feature branch deployments, easy custom domain setup, instant deployments, and password protection.

[Section B]

In this section of the project phase, I'll create an Amazon Cognito user pool to manage users' accounts. I'll deploy pages that enable customers to register as a new user, verify their email address, and sign into the site.

Architecture overview:



When users visit the website they will first register a new user account, we'll only require them to provide an email address and password to register. However, we can configure Amazon Cognito to require additional attributes.

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 the site and enter their email address and the verification code they received. We can also confirm user accounts using the Amazon Cognito console with a 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 phase to authenticate against the RESTful API to build with Amazon API Gateway.

Step 1: Create an Amazon cognito user pool and integrate an app with user pool

Amazon Cognito provides two different mechanisms for authenticating users. You can use Cognito User Pools to add sign-up and sign-in functionality to your application or use Cognito Identity Pools to authenticate users through social identity providers such as Facebook, Twitter, or Amazon, with SAML identity solutions, or by using your own identity system. For this phase, I'll use a user pool as the backend for the provided registration and sign-in pages.

1)In the Amazon Cognito console, choose **Create user pool.**

2)On the Configure sign-in experience page, in the Cognito user pool sign-in options section, select User name. Keep the defaults for the other settings, such as Provider types and do not make any User name requirements selections. Choose Next.

3)On the Configure security requirements page, keep the Password policy mode as Cognito defaults. You can choose to configure multi-factor authentication (MFA) or choose No MFA and keep other configurations as default. Choose Next.

4)On the Configure sign-up experience page, keep everything as default. Choose Next.

5)On the Configure message delivery page, for Email provider, confirm that Send email with Amazon SES - Recommended is selected. In the FROM email address field, select an email address that you have verified with Amazon SES, following the instructions in Verifying an email address identity in the Amazon Simple Email Service Developer Guide.

Note: If you don't see the verified email address populating in the dropdown, ensure that you have created a verified email address in the same Region you selected at the beginning of the tutorial.

6)On the Integrate your app page, name your user pool: WildRydes. Under Initial app client, name the app client: WildRydesWebApp and keep the other settings as default.

7)On the Review and create page, choose Create user pool.

8)On the User pools page, select the User pool name to view detailed information about the user pool you created. Copy the User Pool ID in the User pool overview section and save it in a secure location on your local machine.

9)Select the App Integration tab and copy and save the Client ID in the App clients and analytics section of your newly created user pool.

Step 2: Update the website config file

The js/config.js file contains settings for the user pool ID, app client ID and Region. Update this file with the settings from the user pool and app you created in the previous steps and upload the file back to your bucket.

1) From the local machine, open the wildryde-site/js/config.js file in a text editor.

2)Update the cognito section of the file with the correct values for the User pool ID and App Client ID you saved in Steps 8 and 9 in the previous section. The userPoolID is the User pool ID from the User pool overview section, and the userPoolClientID is the App Client ID from the App Integration > App clients and analytics section of Amazon Cognito.

3)The value for region should be the AWS Region code where the user pool was created. The updated config.js file should look like the following code:

```
window._config = {
   cognito: {
      userPoolId: 'us-west-1_qnmNomWUj', // e.g. us-east-2_uXboG5pAb
      userPoolClientId: '72dug64mtu2fmfu5ljnvvfuicl', // e.g. 25ddkmj4v6hfsfvruhpfi7n4hv
      region: 'us-west-2' // e.g. us-east-2
   },
   api: {
      invokeUrl: '' // e.g. https://rc7nyt4tql.execute-api.us-west-2.amazonaws.com/prod',
   }
};
```

4)Save the modified file.

5)In the codecommit window, commit, and push the file to the repository to have it automatically deploy to Amplify Console.

Phase - I Summary:

The project began by hosting a static website on AWS Amplify and AWS CodeCommit. The backend, managed by AWS Cognito, handles user pools. In the upcoming phase, AWS Lambda and Amazon DynamoDB will process web application requests. This application enables users to request a unicorn to a chosen location. Amazon API Gateway will expose the Lambda function as a secure RESTful API using Amazon Cognito. Finally, client-side JavaScript will make AJAX calls to these APIs, transforming the static site into a dynamic web application.

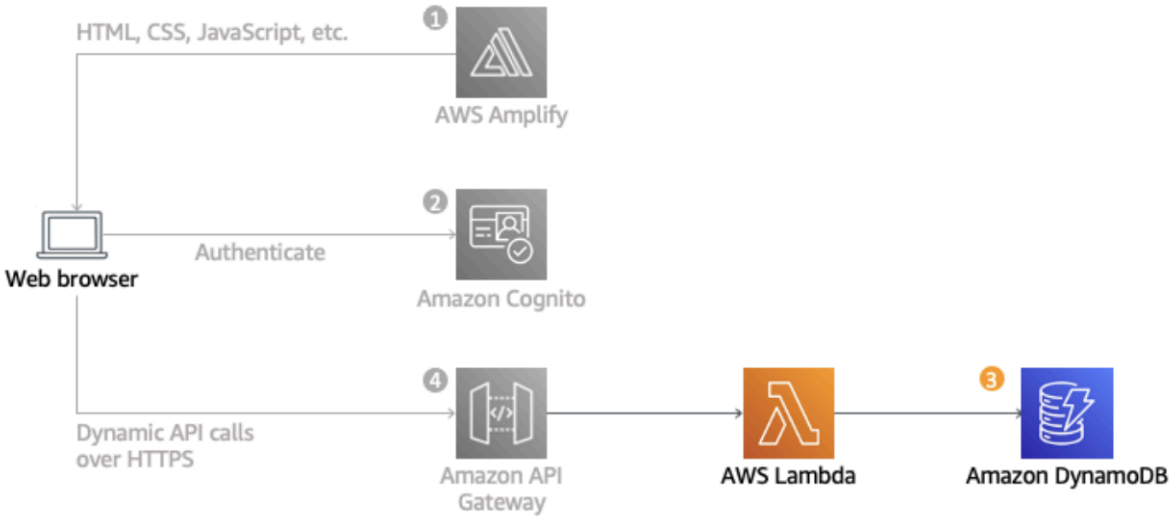
Phase 2: Backend Integration and Functionality

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

Architecture Overview:

Implemented a Lambda function that will be invoked each time a user requests a unicorn. The function will select a unicorn from the fleet, record the request in a DynamoDB table, and then respond to the frontend application with details about the unicorn being dispatched.

The function is invoked from the browser using Amazon API Gateway. I will implement that connection in the next Phase. For this module, I will just test function in isolation.



Implementation:

Step 1: Create an Amazon Dynamo DB table

→ In the Amazon DynamoDB console, choose Create table.

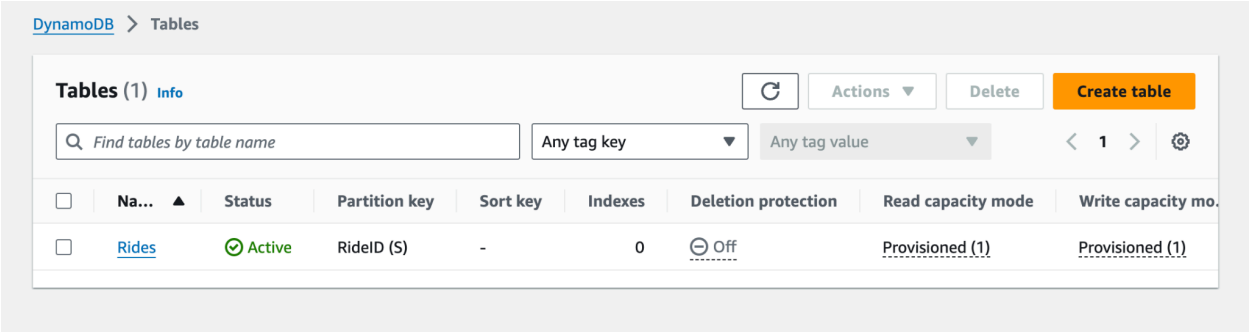
→For the Table name, enter Rides. This field is case sensitive.

→For the Partition key, enter RideId and select String for the key type. This field is case sensitive.

→In the Table settings section, ensure Default settings is selected, and choose Create table.

→On the Tables page, wait for your table creation to complete. Once it is completed, the status will say Active. Select your table name.

→In the Overview tab > General Information section of your new table and choose Additional info. Copy the ARN. You will use this in the next section.



Step 2: Create an IAM role for the 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. Here, I'll need to create an IAM role that grants Lambda function permission to write logs to Amazon CloudWatch Logs and access to write items to the DynamoDB table.

→In the IAM console, select Roles in the left navigation pane and then choose Create Role.

→In the Trusted Entity Type section, select AWS service. For Use case, select Lambda, then choose Next.
Note: Selecting a role type automatically creates a trust policy for your role that allows AWS services to assume this role on my behalf.

→Enter AWSLambdaBasicExecutionRole in the filter text box and press Enter.

→Select the checkbox next to the AWSLambdaBasicExecutionRole policy name and choose Next.

→Enter WildRydes-lambda for the Role Name. Keep the default settings for the other parameters.

→Choose Create Role.

→In the filter box on the Roles page type WildRydes-Lambda and select the name of the role you just created.

→On the Permissions tab, under Add permissions, choose Create Inline Policy.

→In the Select a service section, type DynamoDB into the search bar, and select DynamoDB when it appears.

→Choose Select actions.

→In the Actions allowed section, type PutItem into the search bar and select the checkbox next to PutItem when it appears.

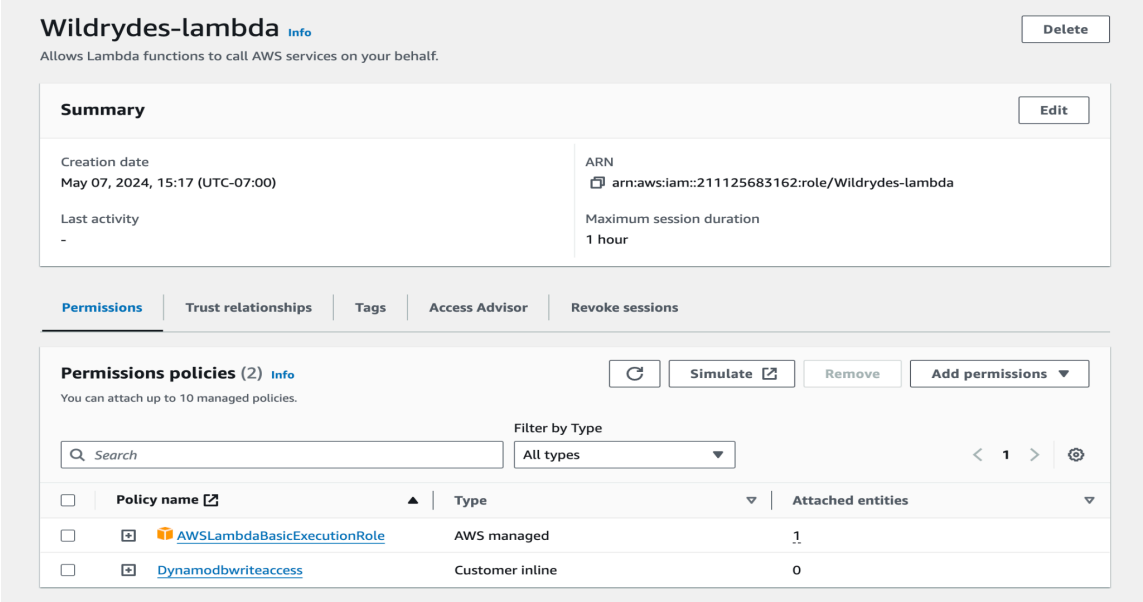
→In the Resources section, with the Specific option selected, choose the Add ARN link

.

→Select the Text tab. Paste the ARN of the table you created in DynamoDB (Step 6 in the previous section), and choose Add ARNs.

→Choose Next.

→Enter DynamoDBWriteAccess for the policy name and choose Create policy.



Step 3: Create a Lamba function for handling requests

AWS Lambda will run code in response to events such as an HTTP request. In this step I'll build the core function that will process API requests from the web application to dispatch a unicorn. In the next phase I'll use Amazon API Gateway to create a RESTful API that will expose an HTTP endpoint that can be invoked from your users' browsers. I'll then connect the Lambda function created in this step to that API in order to create a fully functional backend for web application.

Use the AWS Lambda console to create a new Lambda function called RequestUnicorn that will process the API requests. Use the following requestUnicorn.js example implementation for function code. Just copy and paste from that file into the AWS Lambda console's editor.

Make sure to configure your function to use the WildRydes-Lambda IAM role you created in the previous section.

→From the AWS Lambda console, choose Create a function.

→Keep the default Author from scratch card selected.

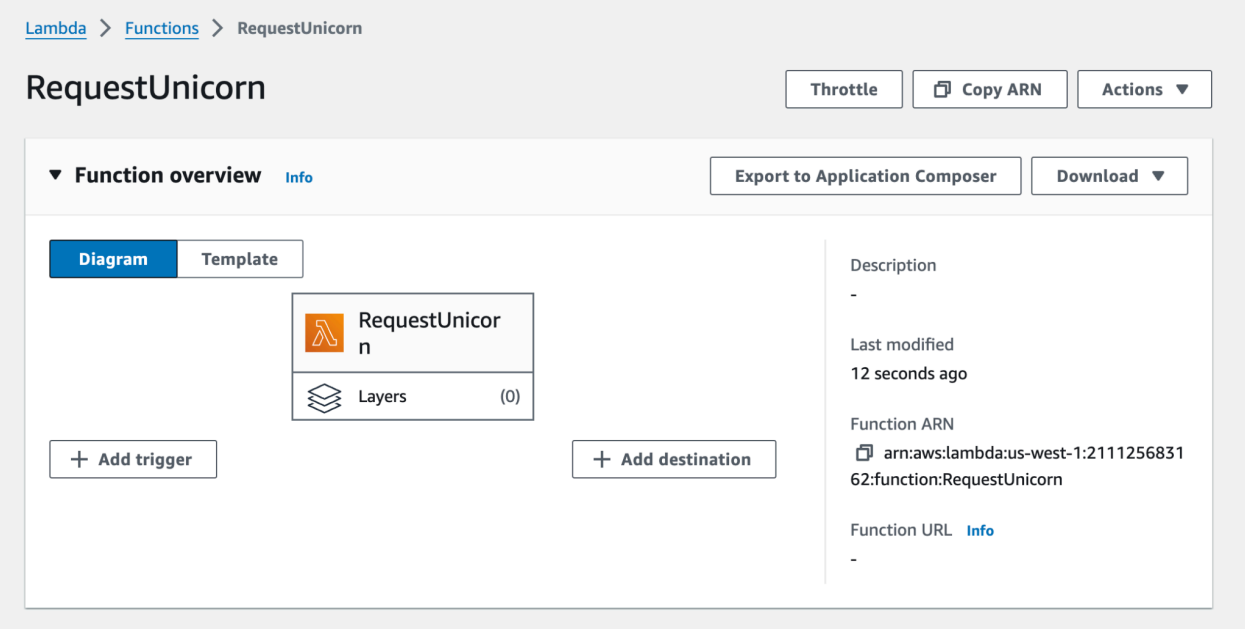
→Enter RequestUnicorn in the Function name field.

→Select Node.js 16.x for the Runtime

→Select Use an existing role from the Change default execution role dropdown.

→Select WildRydes-Lambda from the Existing Role dropdown.

→Click on Create function.



Scroll down to the Code source section and replace the existing code in the index.js code editor with the contents of requestUnicorn.js. The following code block displays the requestUnicorn.js file. Copy and paste this code into the index.js tab of the code editor.

```
const randomBytes = require('crypto').randomBytes;
const AWS = require('aws-sdk');
const ddb = new AWS.DynamoDB.DocumentClient();

const fleet = [
```

```
    {
      Name: 'Angel',
      Color: 'White',
      Gender: 'Female',
    },
    {
      Name: 'Gil',
      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
                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,
        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': '*',
   },
 });
}
```

Step 4: Validating the Implementation:

Here, I will test the function that you built using the AWS Lambda console. In the next phase I will add a REST API with API Gateway so I can invoke your function from the browser-based application that I deployed in the first phase.

→In the RequestUnicorn function built in the previous step, choose Test in the Code source section, and select Configure test event from the dropdown.

→Keep the Create new event default selection.

→Enter TestRequestEvent in the Event name field.

→Copy and paste the following test event into the Event JSON section:

Here, to test if Unicorn is getting dispatched, I write a code requesting unicorn and check if that is getting “written” in my table in DynamoDB

The test code:


```
{
  "path": "/ride",
  "httpMethod": "POST",
  "headers": {
    "Accept": "*/*",
    "Authorization": "eyJraWQiOiJLTzRVMWZs",
    "content-type": "application/json; charset=UTF-8"
  },
  "queryStringParameters": null,
  "pathParameters": null,
  "requestContext": {
    "authorizer": {
      "claims": {
        "cognito:username": "the_username"
      }
    }
  },
  "body":
"{\"PickupLocation\":{\"Latitude\":47.6174755835663,\"Longitude\":-122.2883706665018
5}}"
}
```

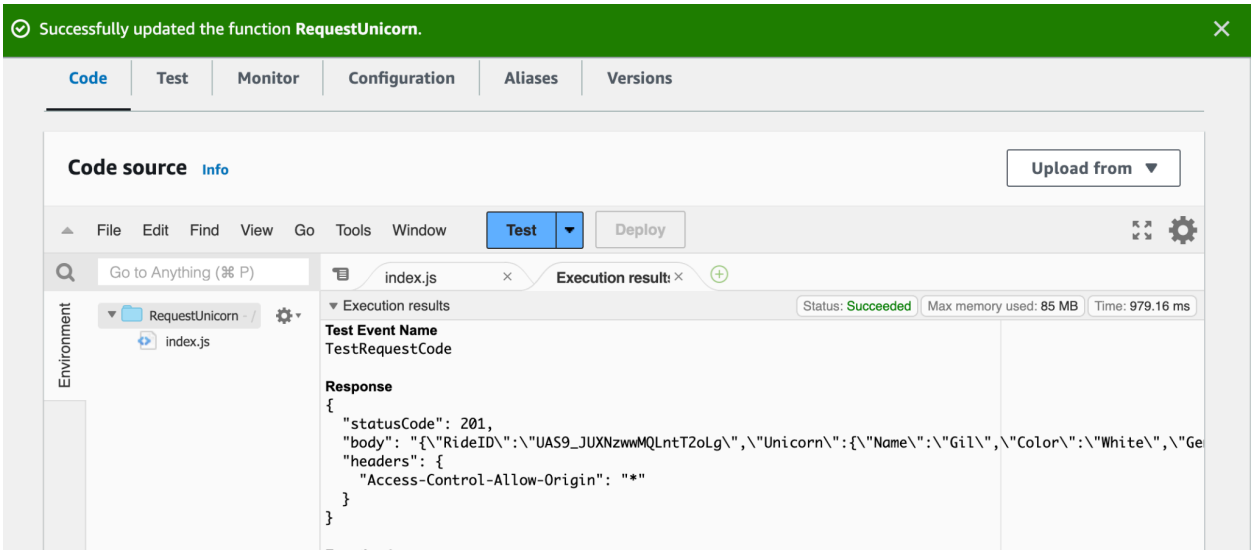

→Choose Save.

→ In the Code source section of your function, choose Test and select TestRequestEvent from the dropdown.

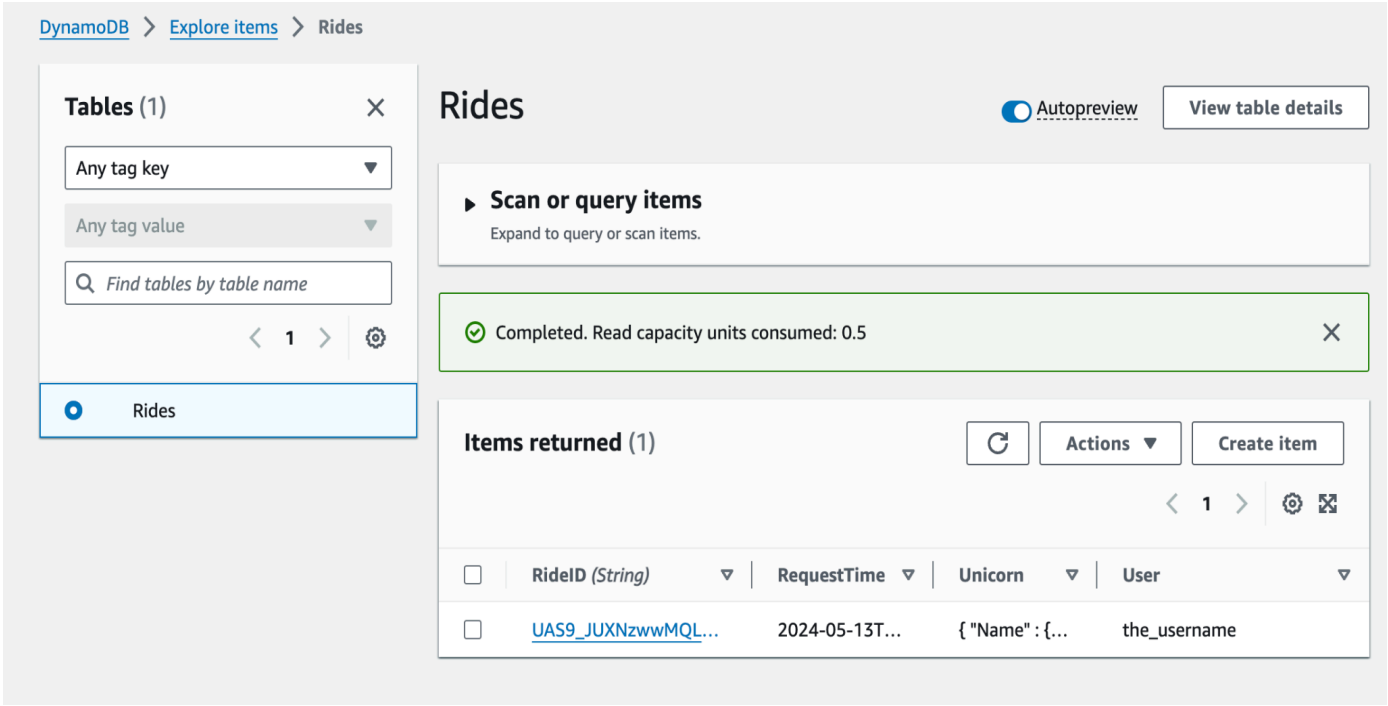
→On the Test tab, choose Test.

→ In the Executing function: succeeded message that appears, expand the Details dropdown.

→ Verify that the function result looks like the following:



We can see from the below screenshot that the request is being written into the table.



Phase - II Summary:

In Phase 2 of the Wild Rydes project, the focus shifted to backend integration and functionality, explicitly using AWS Lambda and Amazon DynamoDB to handle requests for the web application. Here's a summary of the implementation steps:

DynamoDB Table Creation: Created a table named "Rides" in Amazon DynamoDB to store ride requests. The table has a primary key, "RideId," of type String.

IAM Role Creation: Created an IAM role named "WildRydes-lambda" to permit Lambda functions to write logs to CloudWatch Logs and access DynamoDB to write items to the "Rides" table.

Lambda Function Creation: Created a Lambda function named "RequestUnicorn" that processes API requests from the web application to dispatch a unicorn. The function selects a unicorn from a fleet, records the request in the DynamoDB table, and responds with details about the dispatched unicorn.

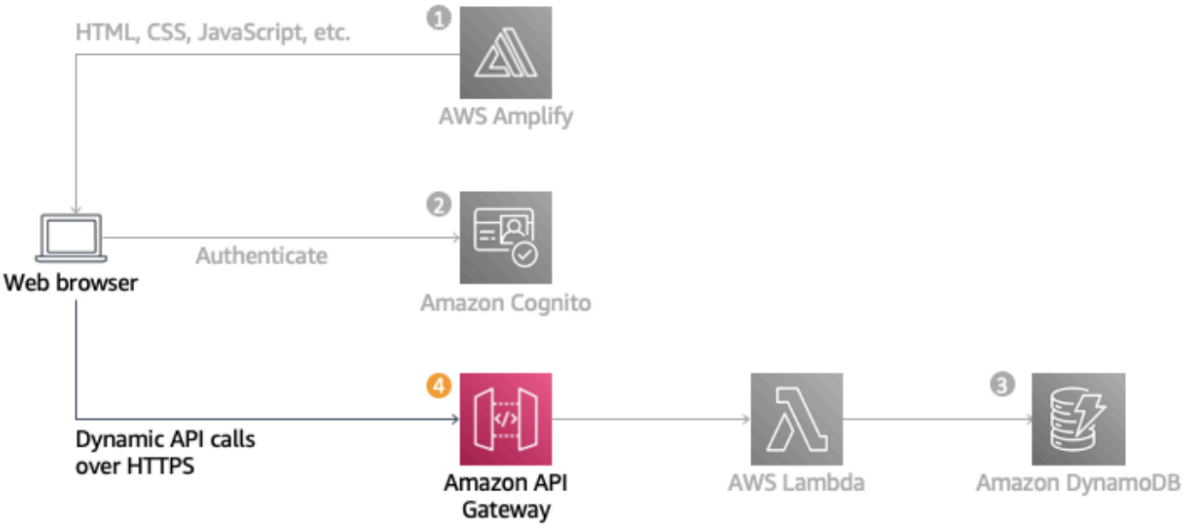
Testing the Lambda Function: Tested the Lambda function using the AWS Lambda console. A test event was created to simulate a request for a unicorn, verifying that the function successfully writes the request to the DynamoDB table.

Overall, Phase 2 laid the groundwork for the backend functionality of the Wild Rydes application, enabling users to request unicorns and dispatch them to the specified locations.

Phase - 3: Build a RESTful API

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

Architecture overview:



The diagram above shows how the API Gateway component I will build in this phase integrates with the existing components I built previously. The grayed-out items are pieces that I have already implemented in previous stages of the project.

The static website I deployed in the first phase already has a page configured to interact with the API that I will build in this phase. The page at /ride.html has a straightforward map-based interface for requesting a unicorn ride. After authenticating using the /signin.html page, your users can select their pickup location by clicking a point on the map and requesting a ride by choosing the "Request Unicorn" button in the upper right corner.

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

Implementation:

Step 1: Create a new REST API

→Select APIs in the left navigation pane in the Amazon API Gateway console.

→Choose Build under REST API.

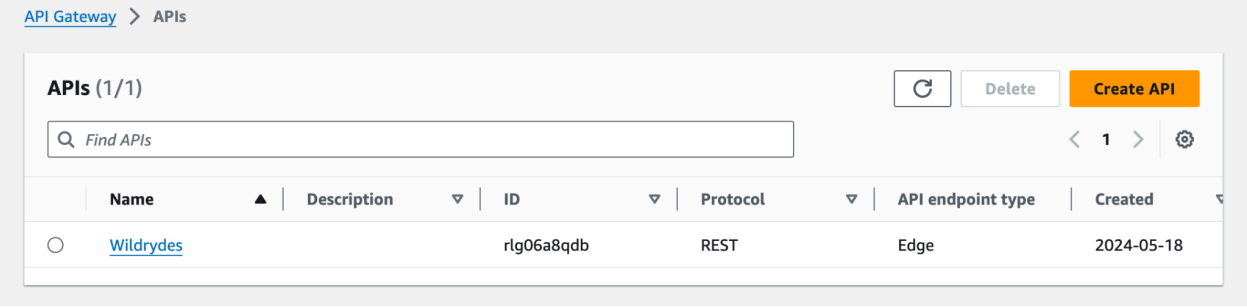
→In the Choose the protocol section, select REST.

→In the Create new API section, select New API.

→In the Settings section, enter WildRydes for the API Name and select Edge Optimized in the Endpoint Type dropdown.

Note: Use edge-optimized endpoint types for public services being accessed from the Internet. Regional endpoints are typically used for APIs accessed primarily within the same AWS Region.

→Choose Create API.



Step 2: Create an authorizer

I need to create an Amazon Cognito User Pools Authorizer. Amazon API Gateway uses JSON web tokens (JWT), which are returned by the Amazon Cognito User Pool (developed in phase 2) to authenticate the API calls. In this step, I will make an Authorizer for the API so we can use the user pool.

The following steps to configure the Authorizer in the Amazon API Gateway console:

→In the left navigation pane of the WildRydes API you just created, select Authorizers.

→Choose Create New Authorizer.

→Enter WildRydes into the Authorizer Name field.

→Select Cognito as the Type.

→Under Cognito User Pool, in the Region drop-down, select the same Region you have been using for the rest of the tutorial. Enter WildRydes in the Cognito User Pool name field.

→Enter Authorization for the Token Source.

→Choose Create.

To verify the authorizer configuration, select Test,
Paste the Authorization Token copied from the ride.html webpage in the Validate the implementation section of Phase 2 into the Authorization (header) field, and verify that the HTTP status Response code is 200.

Step 3: Create a new resource and method

In this section, I will create a new resource for the API. Then, make a POST method for that resource and configure it to use a Lambda proxy integration backed by the RequestUnicorn function created in this phase's first step.

→In the left navigation pane of your WildRydes API, select Resources.

→From the Actions dropdown, select Create Resource.

→Enter the ride as the Resource Name, automatically creating the Resource Path /ride.

→Select the checkbox to enable API Gateway CORS.

→Choose Create Resource.

→With the newly created /ride resource selected, from the Actions dropdown, select Create Method.

→Select POST from the new dropdown under OPTIONS, then select the checkmark icon.

→Select Lambda Function for the Integration type.

→Select the checkbox to use Lambda Proxy integration.

→Select the Region you have been using throughout the tutorial for the Lambda Region.

→Enter RequestUnicorn for Lambda Function.

→Choose Save.

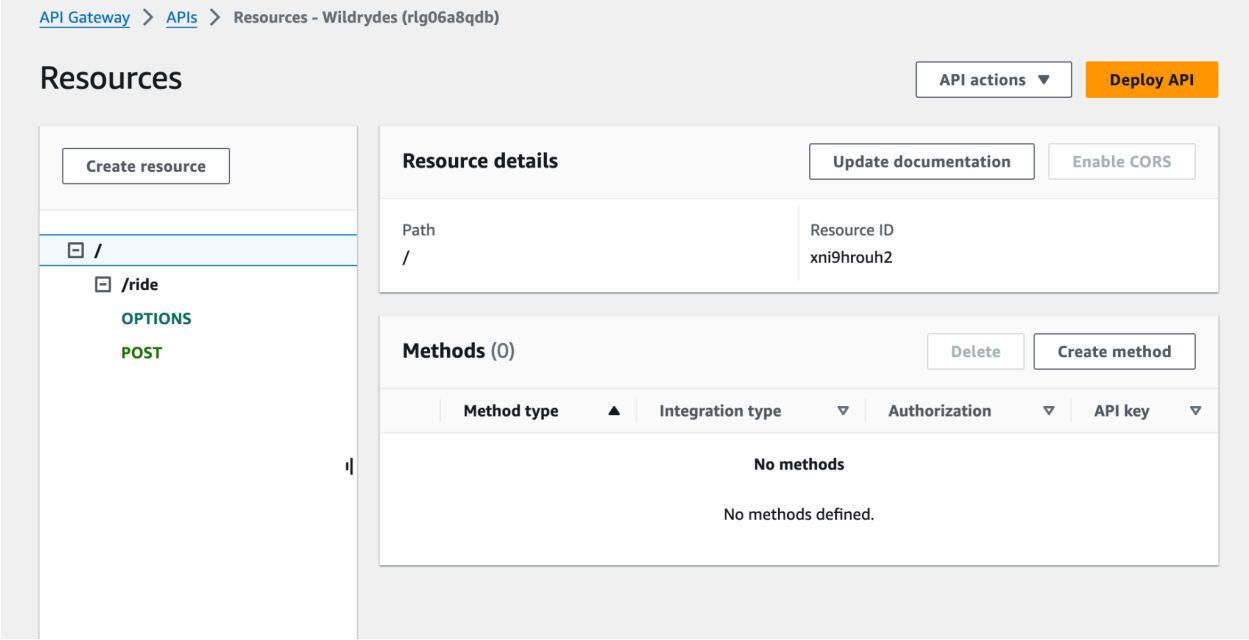
Note: If you get an error that your function does not exist, check that the selected region matches the one you used in the previous modules.

→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 select the checkmark icon.



Step : 4: Deploy the API

I will deploy the API from the **Amazon API Gateway** console in this step.

→ In the Actions drop-down list, select Deploy API.

→ Select [New Stage] in the Deployment stage drop-down list.

→ Enter prod for the Stage Name.

→ Choose Deploy.

→ Copy the Invoke URL.

Step: 5 Update the website config

In this step, I will update the /js/config.js file in the website deployment to include the Invoke URL of the stage just created. I will copy the Invoke URL directly from the top of the stage editor page on the Amazon API Gateway console and paste it into the invoke URL key of your site's config.js file. Your config file will still contain the updates you made in the previous module for your Amazon Cognito userPoolID, userPoolClientID, and region.

→ On your local machine, navigate to the js folder and open the config.js file in a text editor.

→ Paste the Invoke URL copied from the Amazon API Gateway console in the previous section into the invoke URL value of the config.js file.

→Save the file.

Step: 6 Validating the Implementation

→ Update the ArcGIS JS version from 4.3 to 4.6 (newer versions will not work) in the ride.html file as:

<script src="https://js.arcgis.com/4.6/"></script>

<link rel="stylesheet" href="https://js.arcgis.com/4.6/esri/css/main.css">

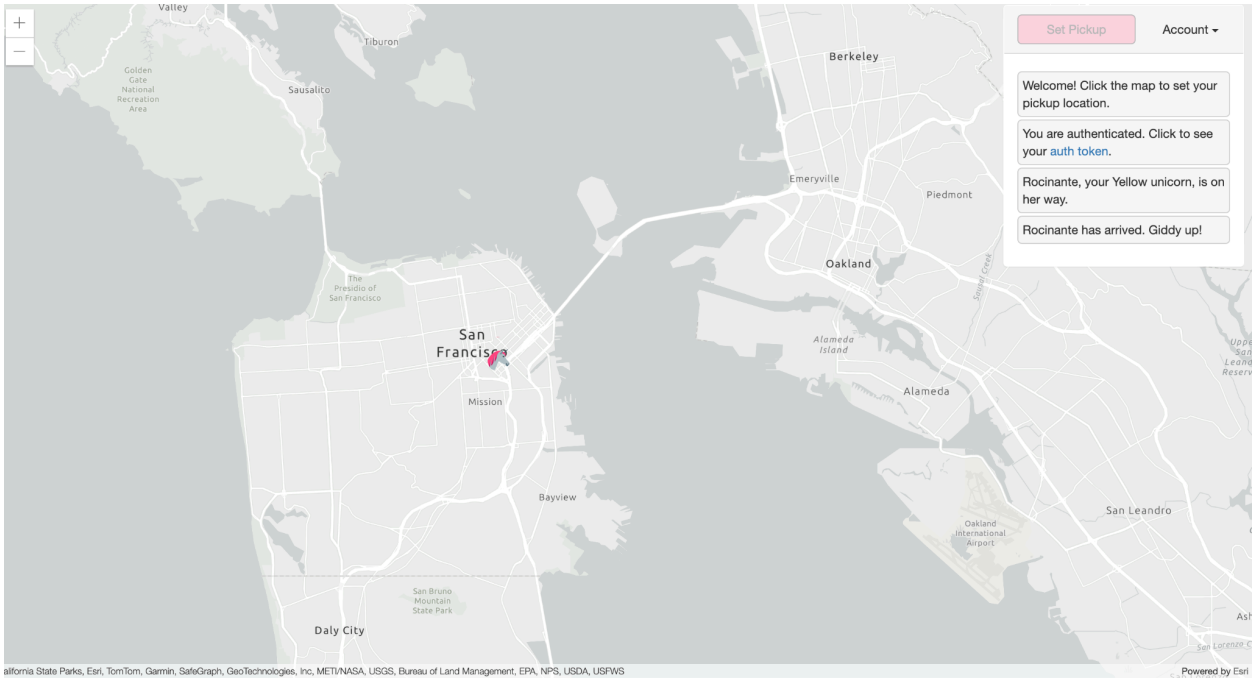
→ Save the modified file.

→ Now, open the deployed website using the domain given.

→ Create an account on the WildRydes website, and then the home page is loaded

→ After the map has loaded, click anywhere on the map to set a pickup location.

→ Choose Request Unicorn. You should see a notification in the right sidebar that a unicorn is on its way and then see a unicorn icon fly to your pickup location.



Phase - III Summary:

Phase 3 of the Wild Rydes project focuses on building a RESTful API using Amazon API Gateway. This API will allow the static website from Phase 1 to become a dynamic web application.

Here's a summary of the key steps:

API Gateway Setup: A new RESTful API named WildRydes is created in API Gateway. It uses an edge-optimized endpoint to be accessible from the public internet.

Authorizer Creation: An Amazon Cognito User Pools Authorizer named WildRydes is created to secure the API using JWTs issued by the user pool developed in Phase 2.

Resource and Method Configuration: A new resource named /ride is created under the API. A POST method is added to this resource and configured to use a Lambda proxy integration. This integration invokes the RequestUnicorn Lambda function created in Phase 2.

API Deployment: The API is deployed to a new stage named prod. The Invoke URL for this stage is copied for use in the next step.

Website Configuration Update: The website's config.js file includes the copied Invoke URL from API Gateway. This URL allows the website to make calls to the API.

Website Validation: The ArcGIS JS version in the ride.html file is updated to 4.6 (newer versions are incompatible). Users can now test the functionality by creating an account, setting a pickup location on

the map, and requesting a unicorn. A notification and unicorn icon movement should confirm a successful ride request.

Phase 3 establishes the communication channel between the user interface and the backend functionality built in Phase 2. This enables users to interact with the Wild Rydes application through the website.

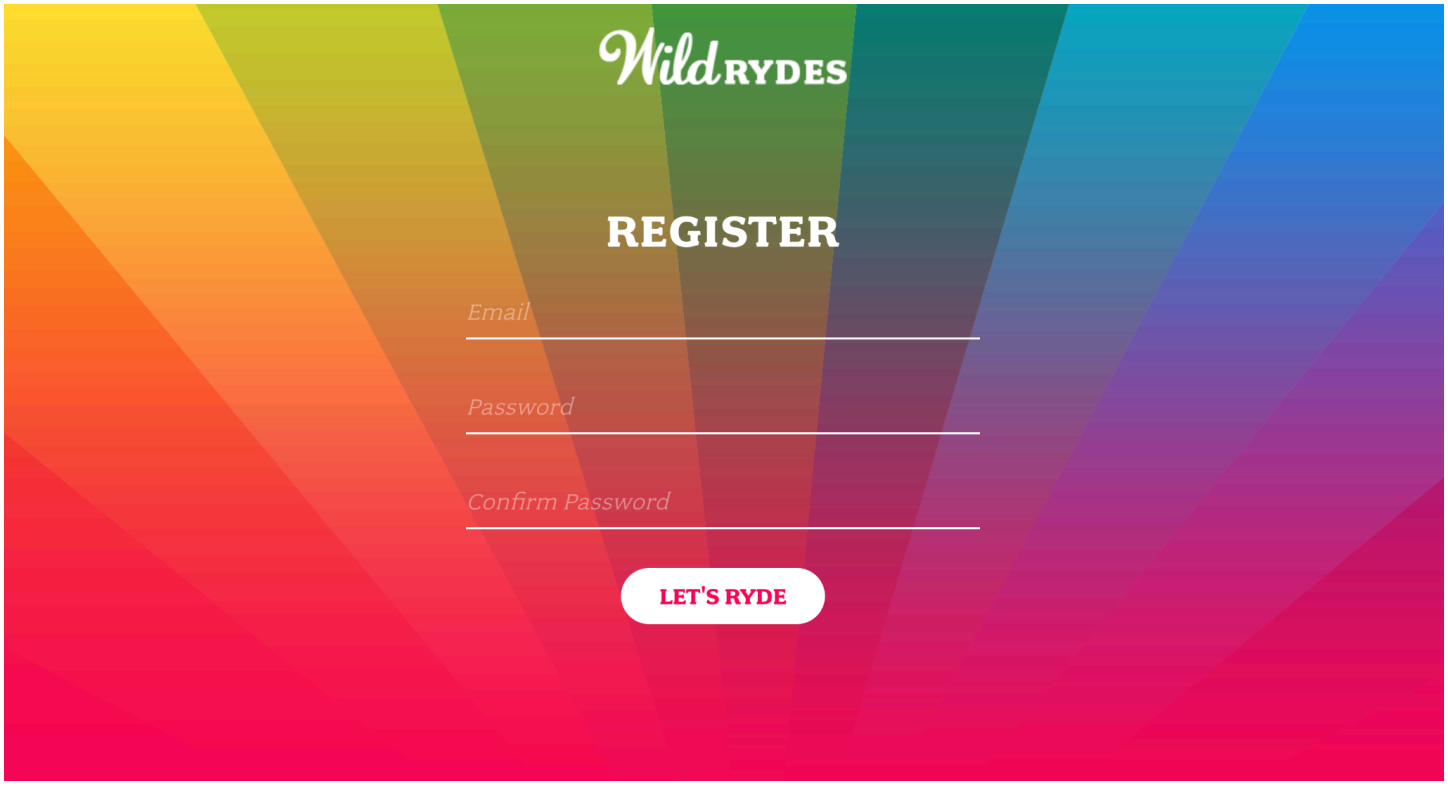
Project Demo

This section showcases the functionalities of the Wild Rydes application through screenshots captured during the demonstration.

Home Page:

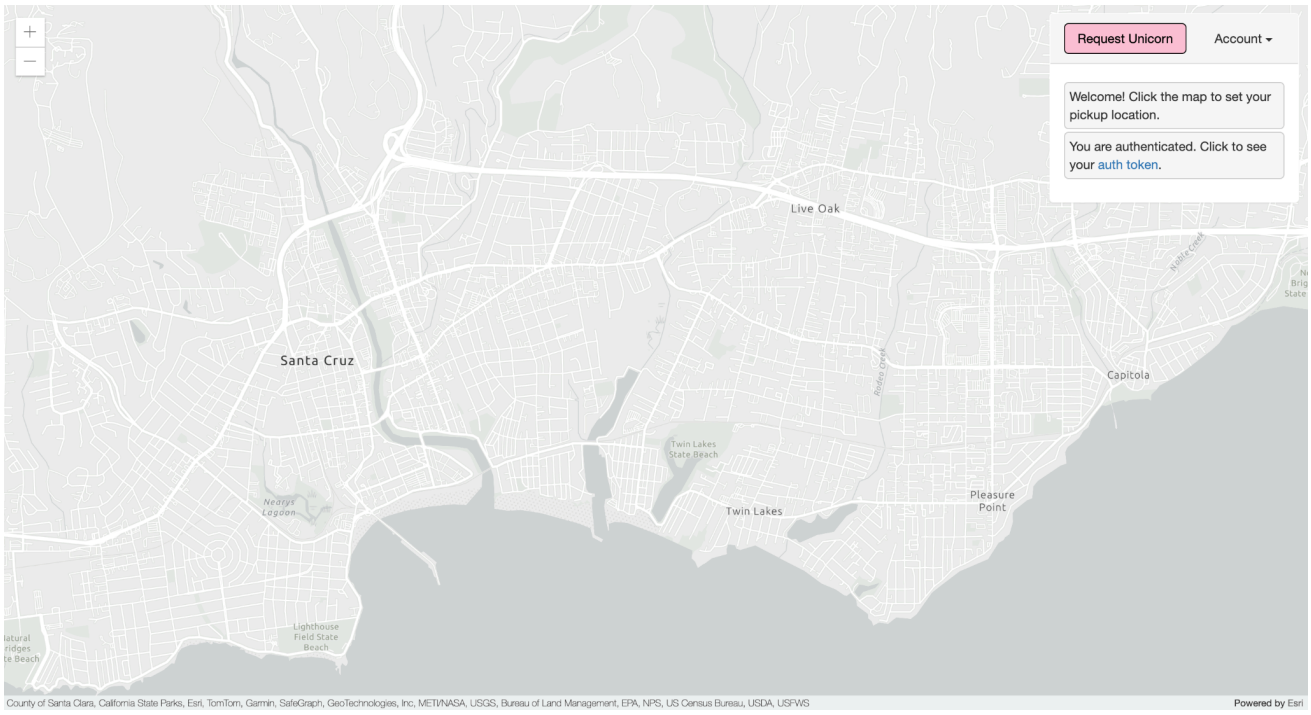
The home page serves as the initial landing point for users. It provides a brief introduction to Wild Rydes and its fantastical twist on ride-sharing.

Login Page:



The login page allows existing users to access their accounts and request unicorns. Users can also create new accounts through this page.

Map Interface:



The map interface facilitates user interaction with the application. Users can set their pick-up location by clicking on a desired point on the map.

Request Ride Confirmation:

After selecting a pick-up location and requesting a ride, users receive confirmation. This confirmation might include details about the dispatched unicorn and an estimated arrival time.

Future Developments:

This section outlines potential areas for further development of the Wild Rydes application:

- Implement a system for users to choose specific unicorns based on preferences (e.g., fastest, friendliest).
- Integrate a payment processing system to simulate real-world transactions.
-
- Enable users to specify their desired destination in addition to the pick-up location.

Conclusion

The Wild Rydes project successfully demonstrated the core functionalities of a ride-sharing application using a fantastical theme. The application utilizes serverless technologies on AWS to provide a scalable and secure platform. This project serves as a valuable foundation for further development, paving the way for a more feature-rich and engaging user experience.