Module	Assessment Type
Distributed and Cloud Systems Programming	Individual Report

Workshop 3

Student Id : 2049867 (NP03A190017)
Student Name : Roshan Parajuli
Section : L5CG3
Module Leader : Rupak Koirala
Lecturer /Tutor : Saroj Sharma
Submitted on : 2020-03-30

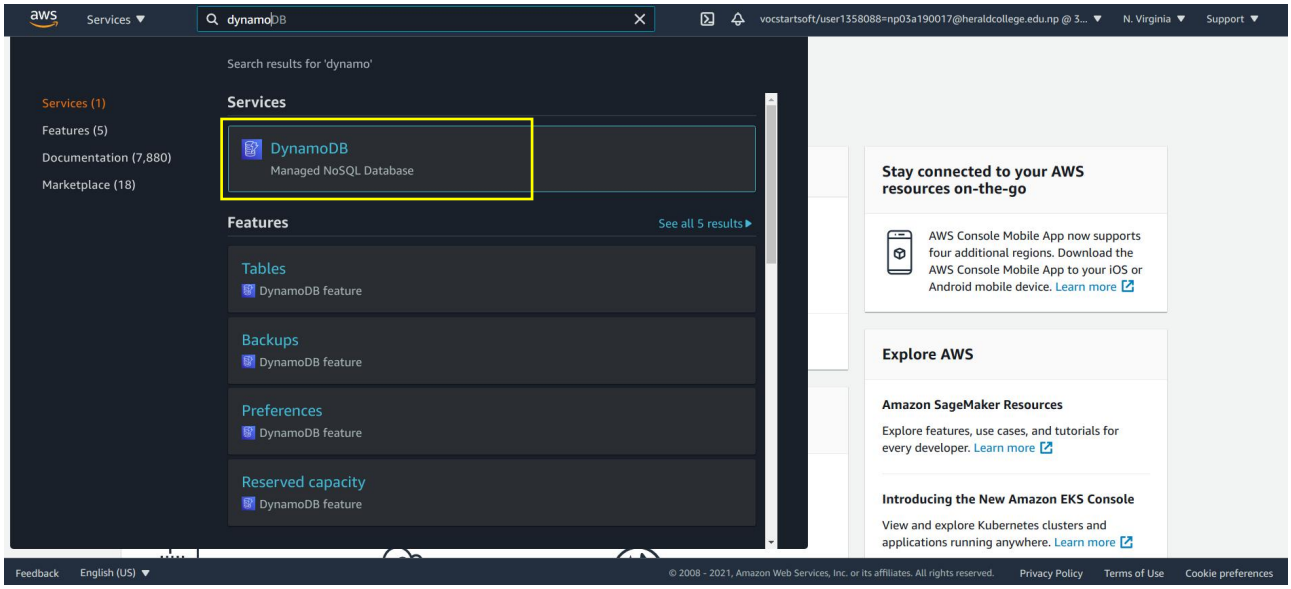
Table of contents

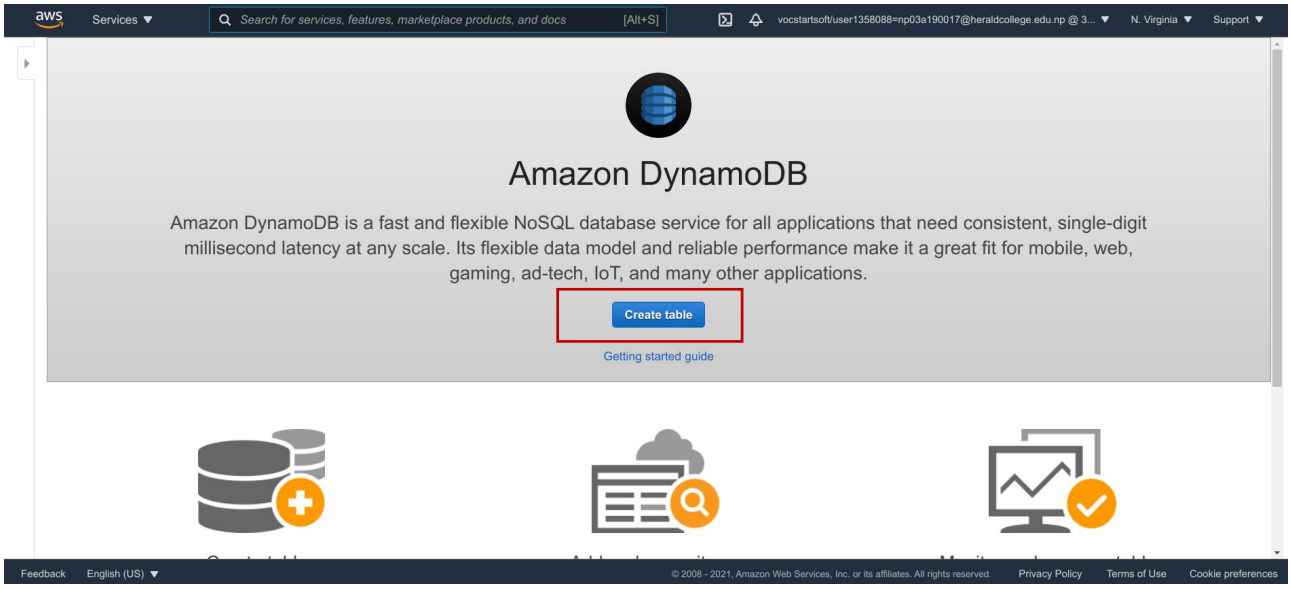
Task 1: Serverless Service Backend

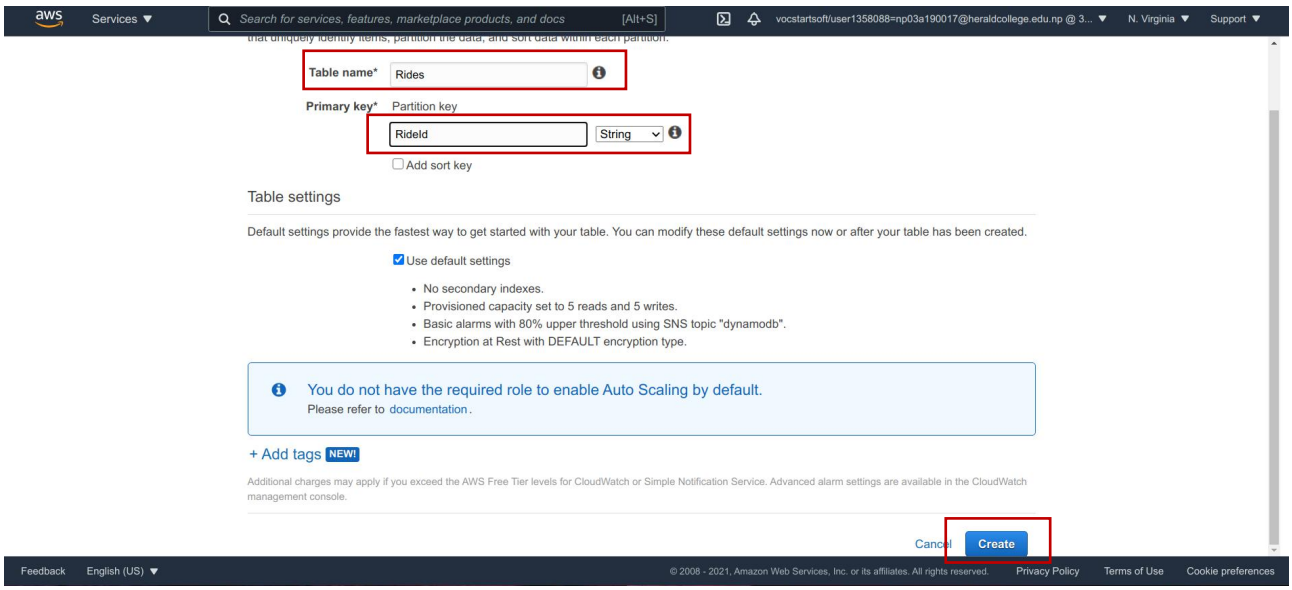
Continuing the workshop 2, a **lambda function** is to be implemented which needs to be invoked everytime a user requests a ride. As a first part of the task, Amazon DynamoDB table is created.

Creating a DynamoDB table

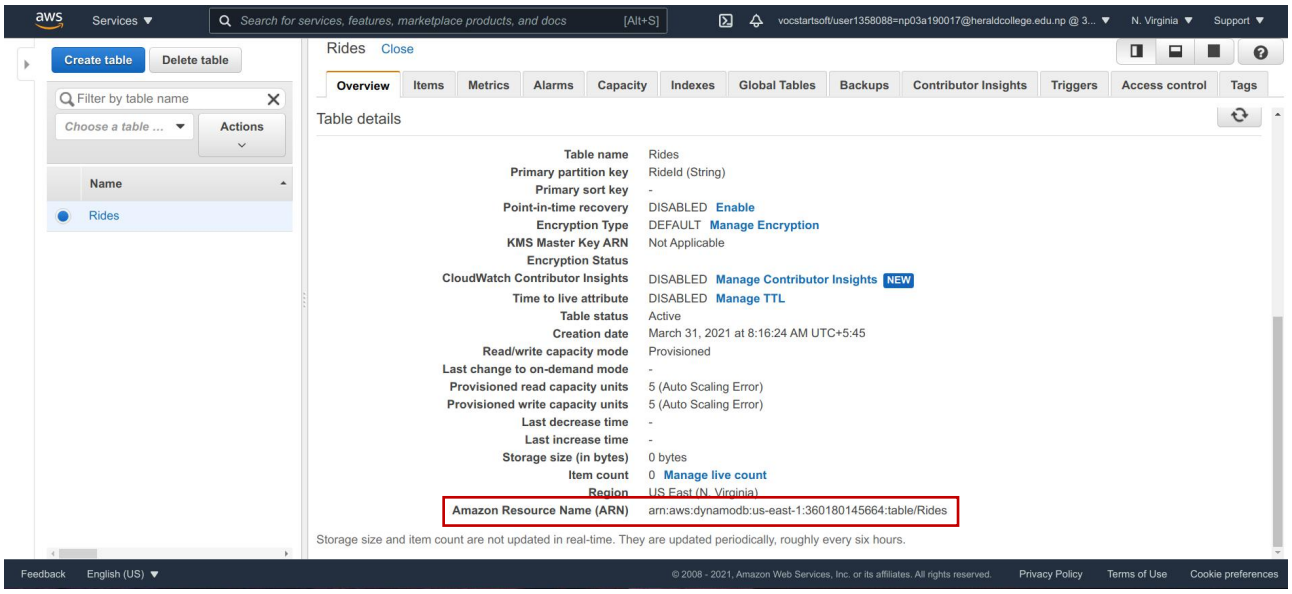
First, DynamoDB service is selected from the search box. **Amazon DynamoDB** is a fully managed **NoSQL database** service that provides fast and predictable performance with seamless scalability.







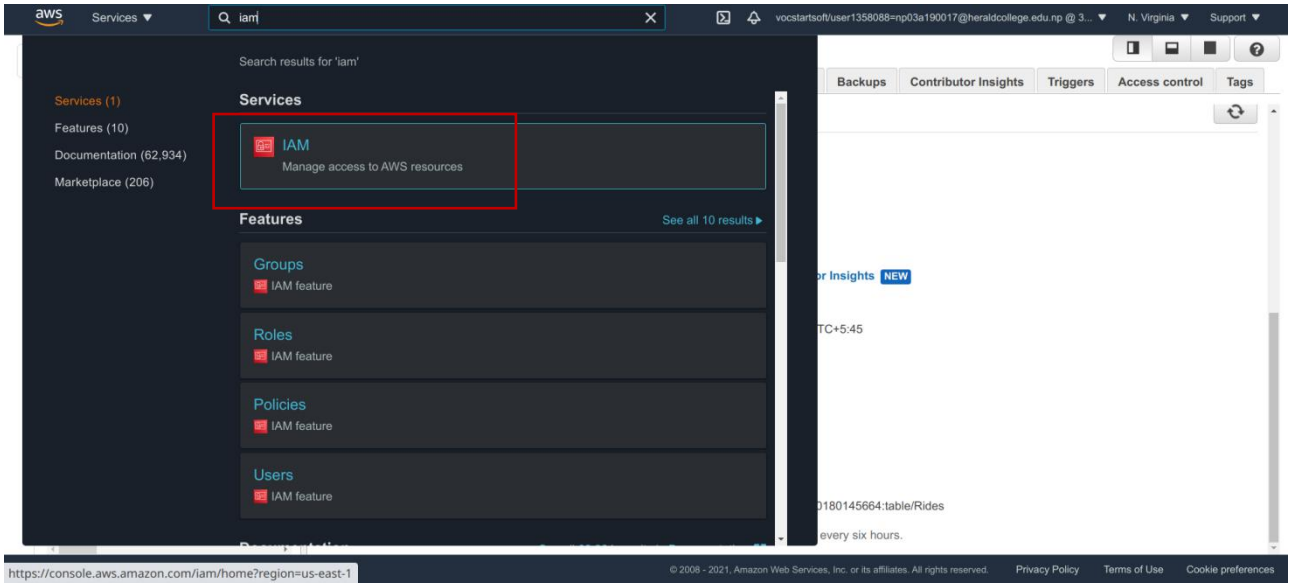
The table is named “Rides” and a partition key called “RideId” is provided to the table. Partition key is a simple primary key which helps in identify each record uniquely in the database.

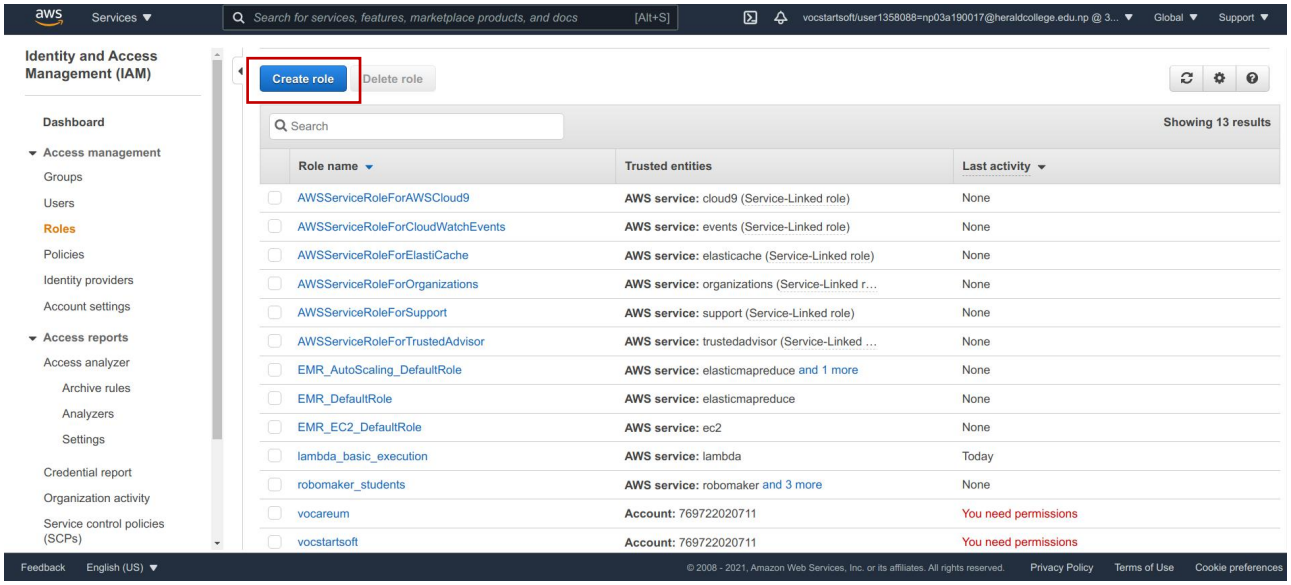


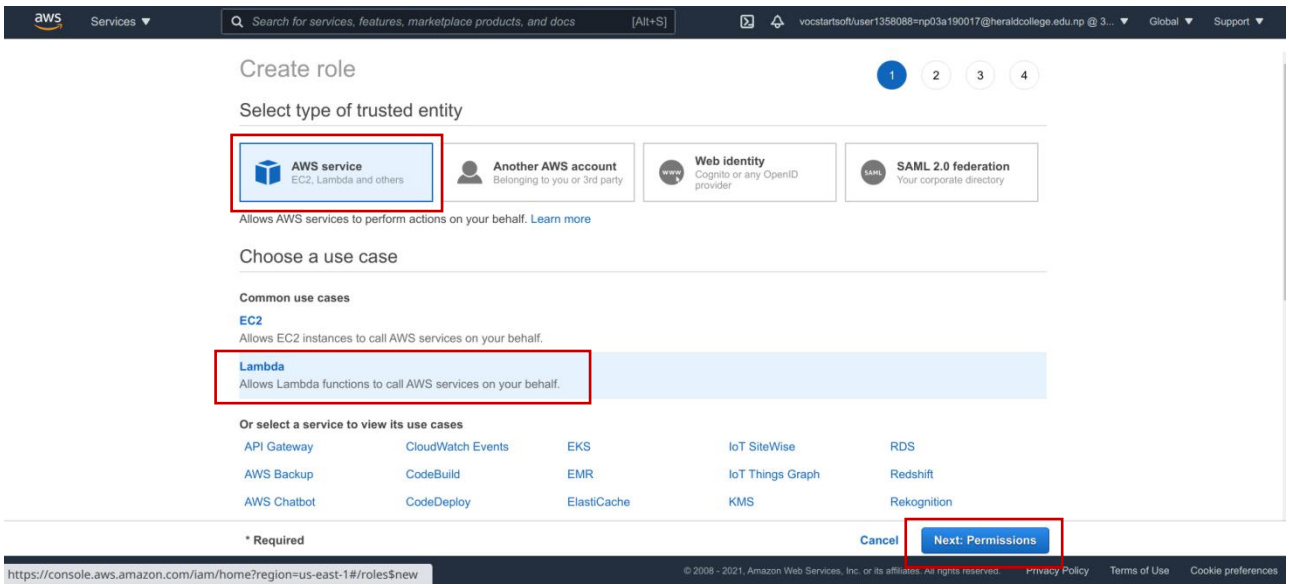
The **Amazon Resource Name (ARN)** was noted for further use. ARN is a file naming convention used to identify a particular resource in the Amazon Web Services (AWS) public cloud.

Creating an IAM Role for the Lambda function

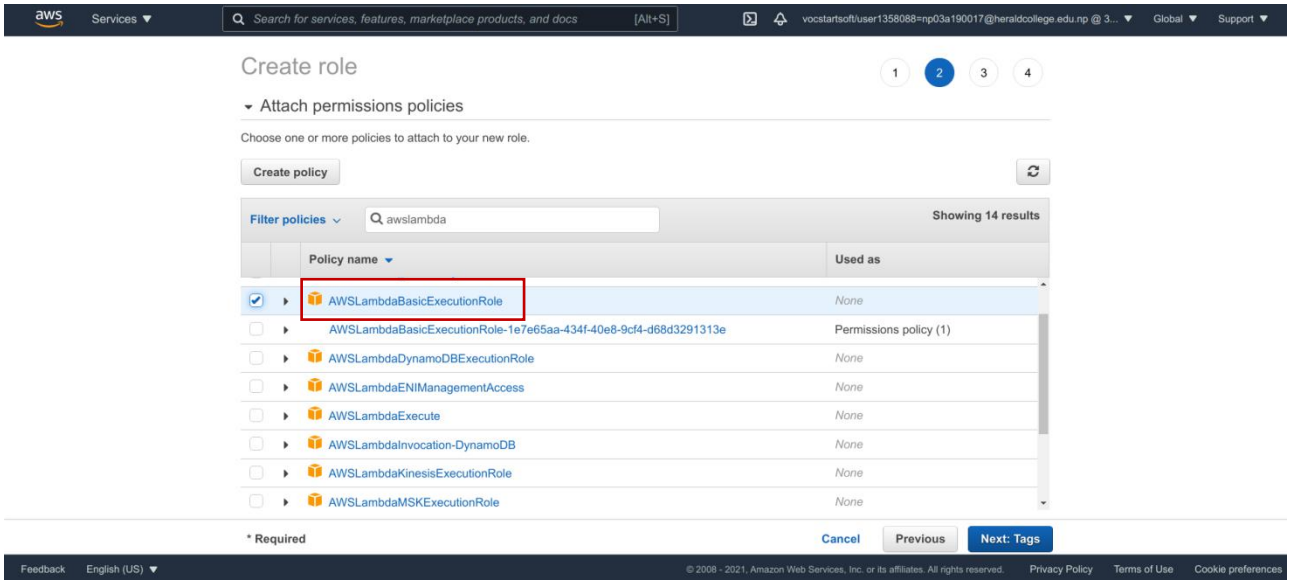
Before a lambda function is created for handling requests, an Identity and Access Management (IAM) role is to be created for giving the appropriate permissions to the lambda function. It defines what other AWS services the function is allowed to interact with.



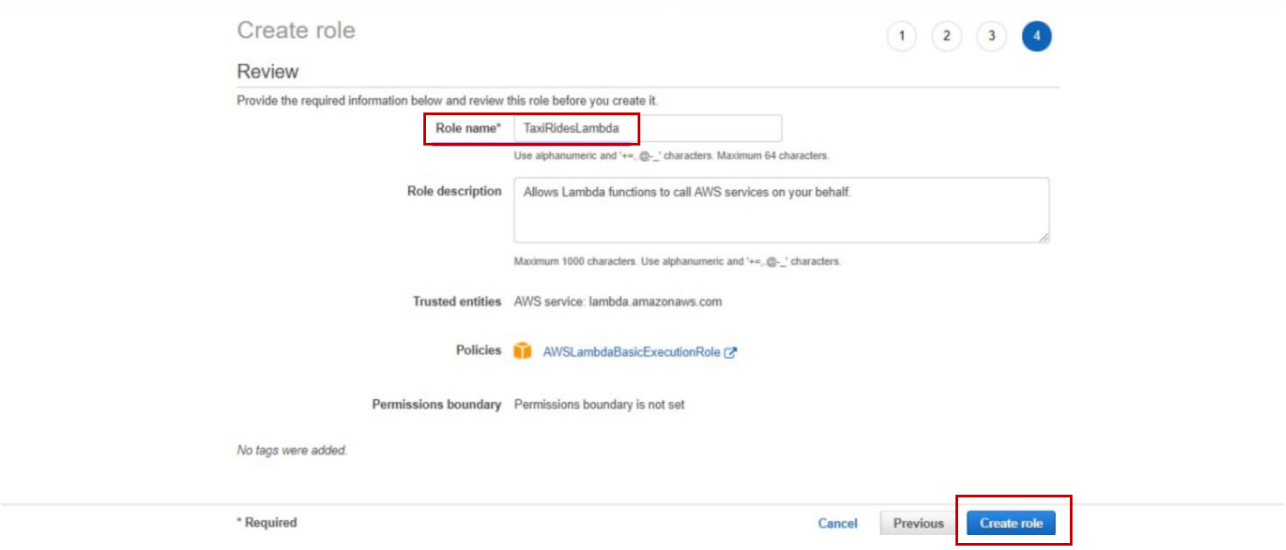




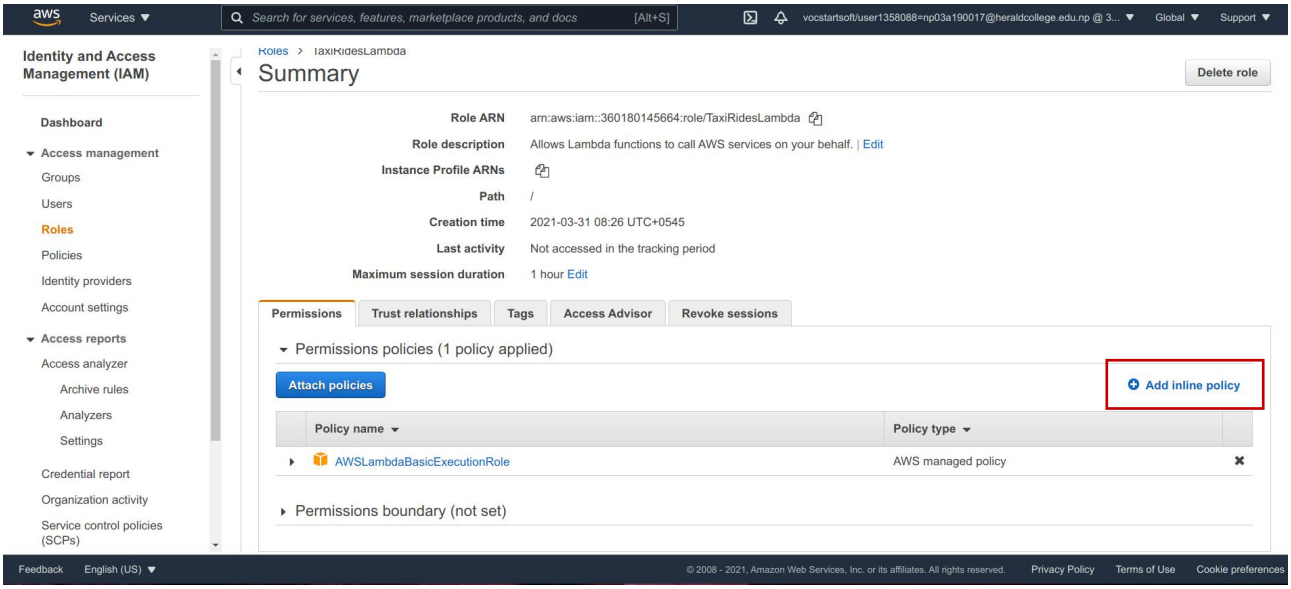
We now have to select what permissions are to be given for the role.



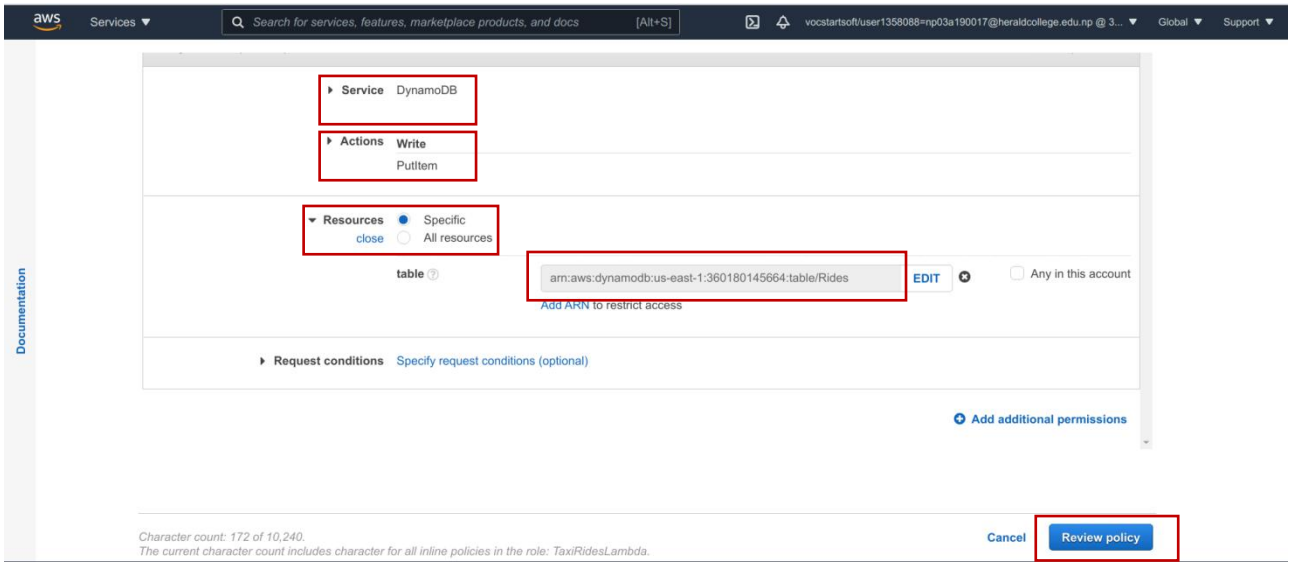
As the name suggests, AWSLambdaBasicExecutionRole provides very basic permissions for the lambda function to work. Using this would be an easier option instead of manually adding CloudWatch Logs permissions as it handles the logs on its own.



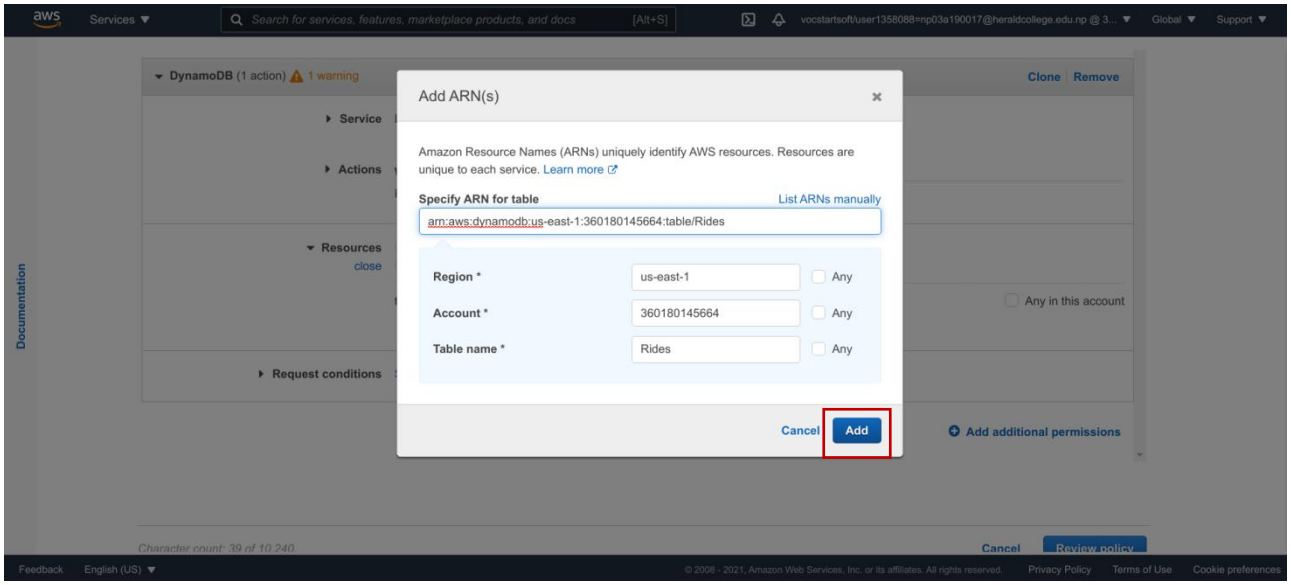
The role is created successfully. Now, we need to give the role appropriate permissions for adding items to the DynamoDB.



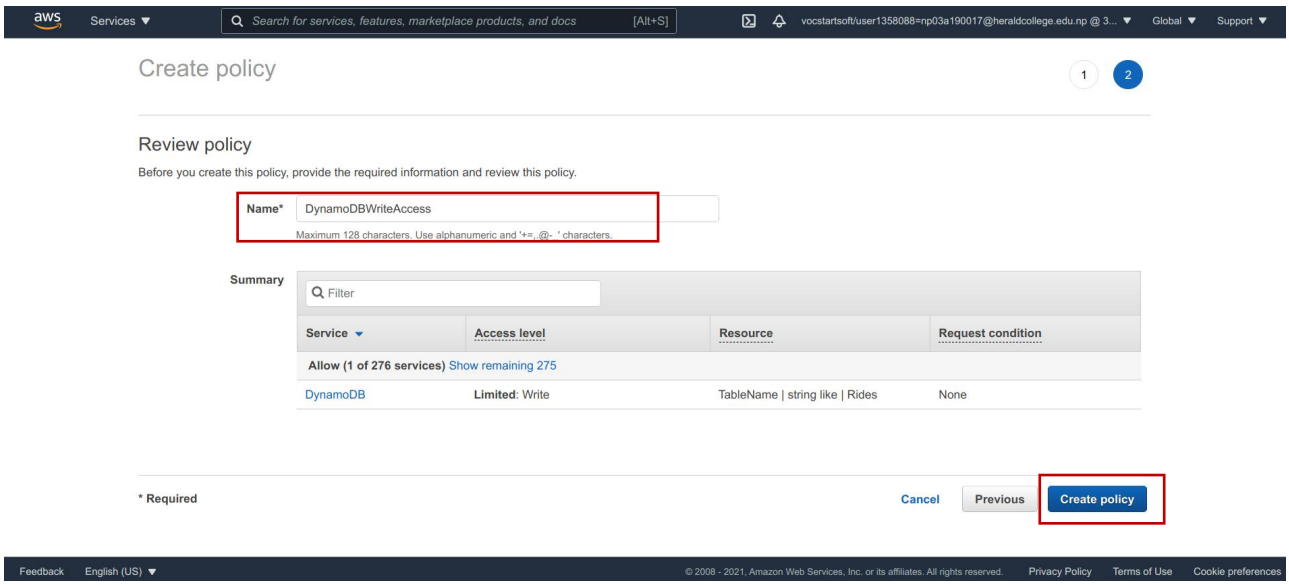
This is a summary page for the role just created. For adding the permissions for the DynamoDB, we need to add an inline policy. **Inline policies** are policies that the user creates and manages and is embeded directly into a single user, group, or role.



First of all, DynamoDB service is selected and the action that the role needs is the write access. The role is allocated a specific role and to identify the DynamoDB table, its ARN which was noted before is pasted.



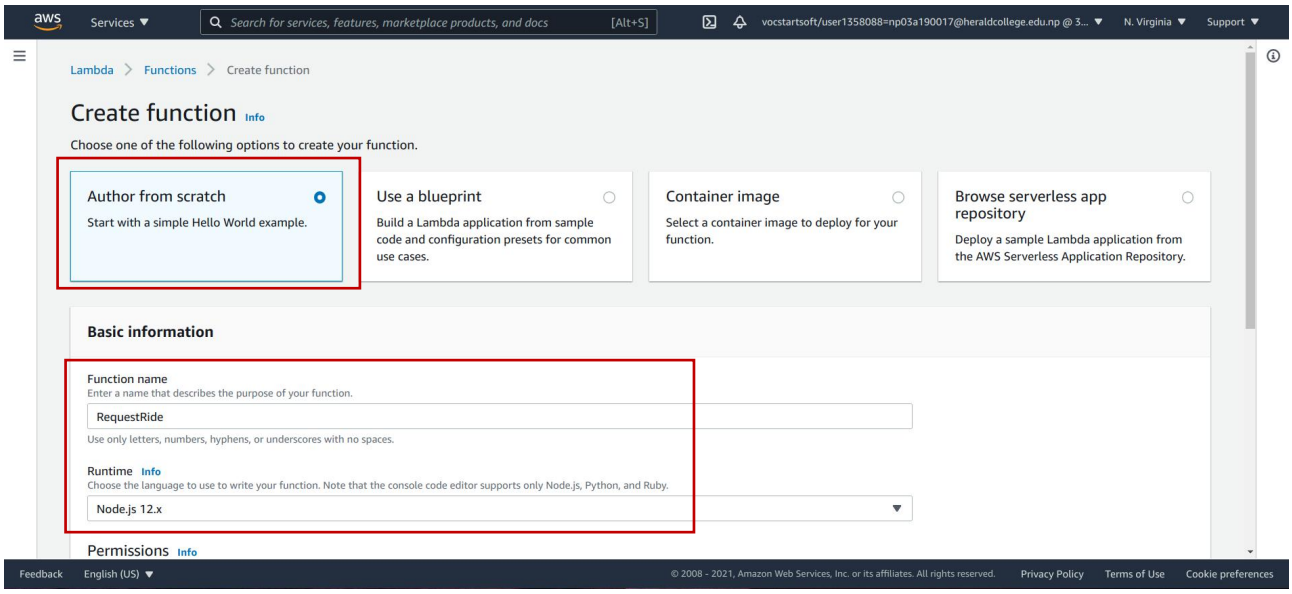
The ARN for table helps to identify the AWS Region, Account and table name. It is added into the policy.



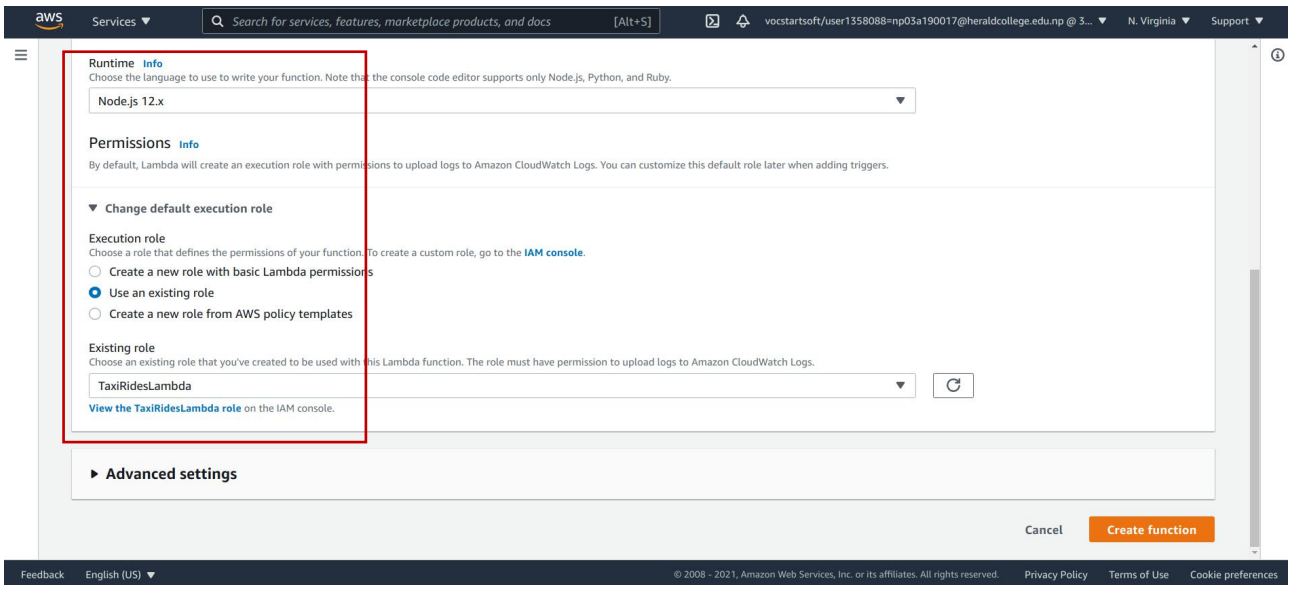
Policy is created with the name DynamoDBWriteAccess and is now attached to the lambda function.

Creating a Lambda Function for Handling requests

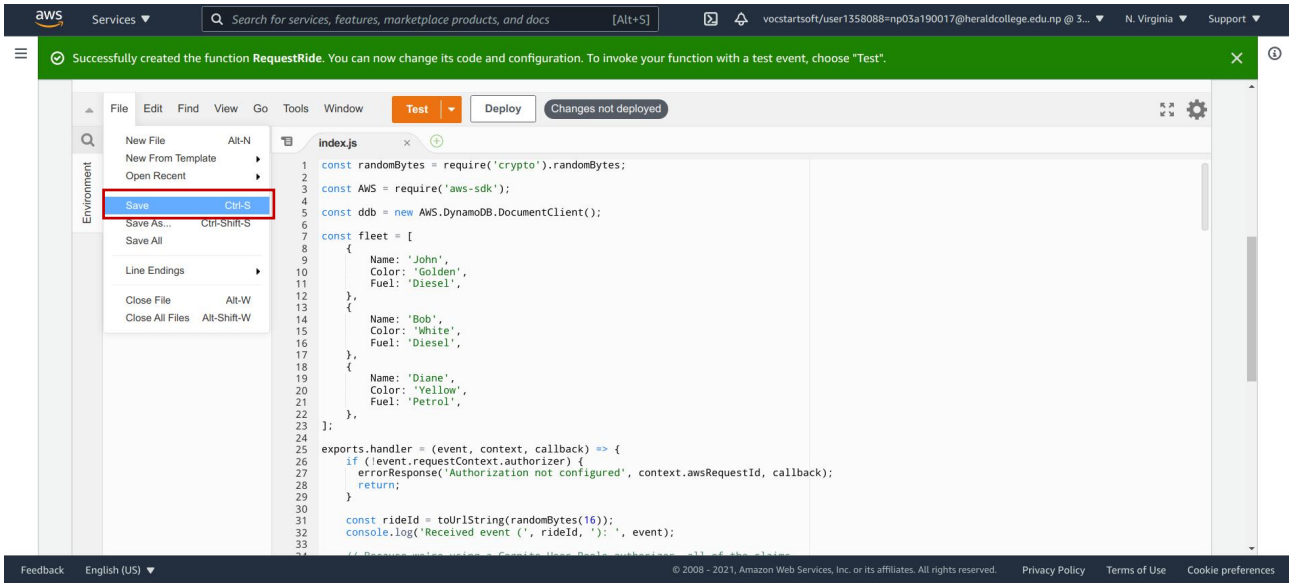
Lambda function is created so that it would respond to the HTTP request by executing the code in it.



A new lambda function named RequestRide is created and the runtime is the Node.js 12.x which is one version behind from the latest release.

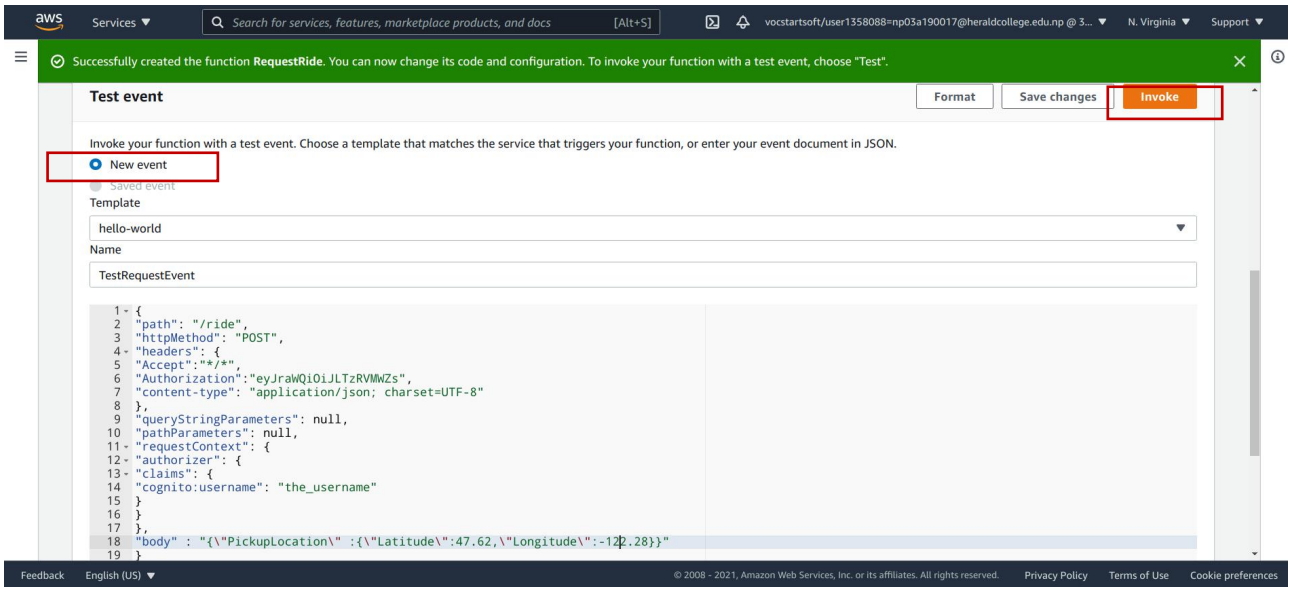


The role which is just created few steps ago i.e. “TaxiRidesLambda” is assigned to the newly created lambda function.

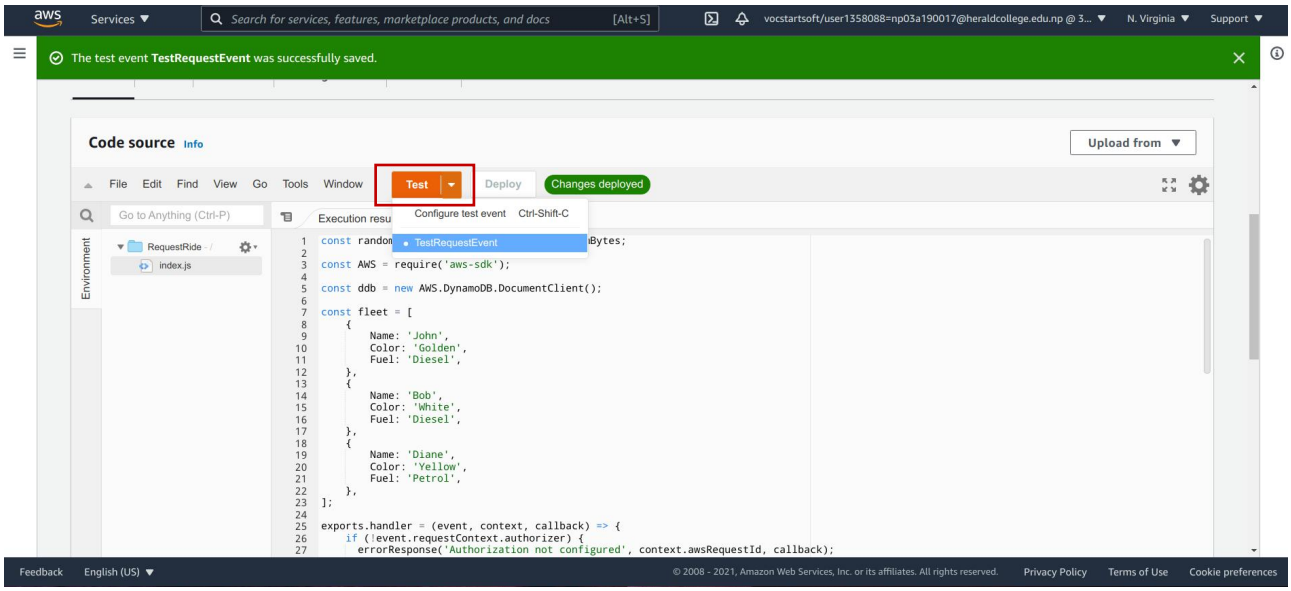


The default index.js code is replaced with the code from the provided file RequestRide.js which is pre-populated with the necessary code. The change is now saved.

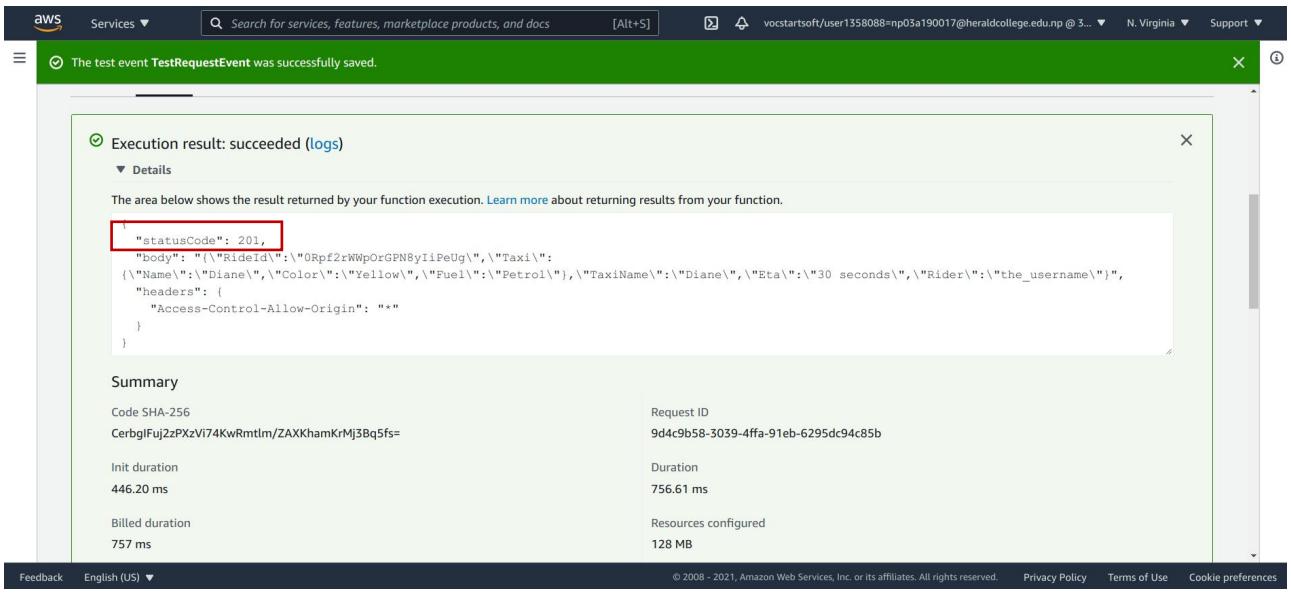
Validating the implementation



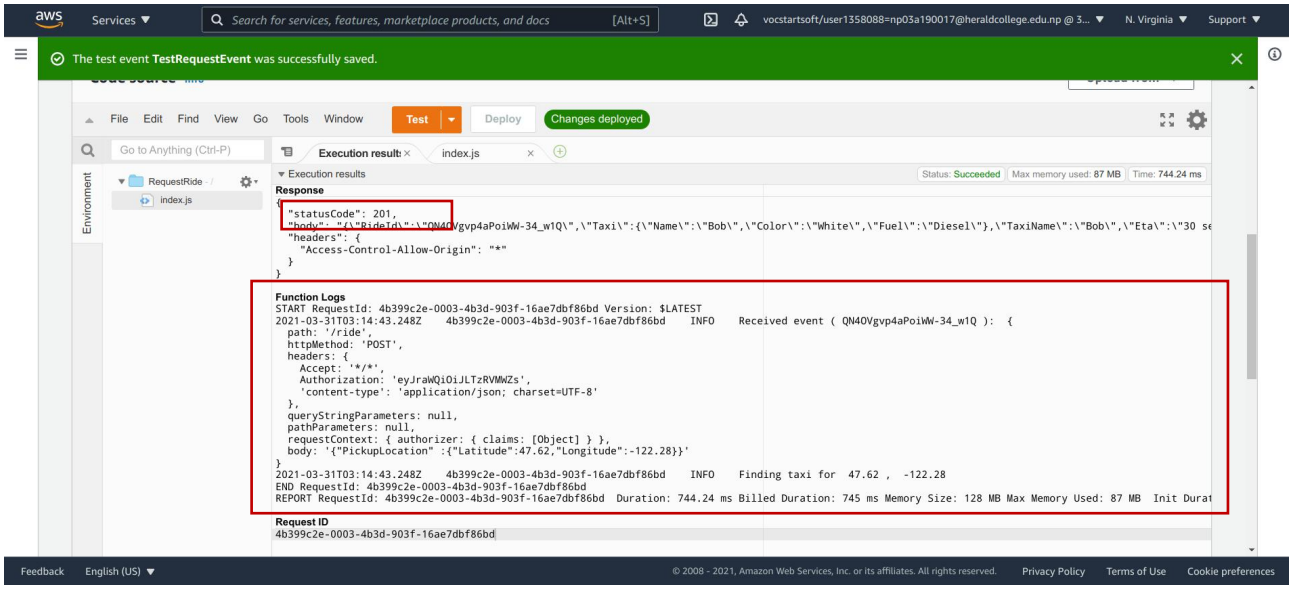
A new test event is created with the provided template for invoking the lambda function.



After invoking the function will the created test event, the following result is seen:



The status code returned is 201. The HTTP 201 Created success status response code indicates that the request has succeeded and has led to the creation of a resource.

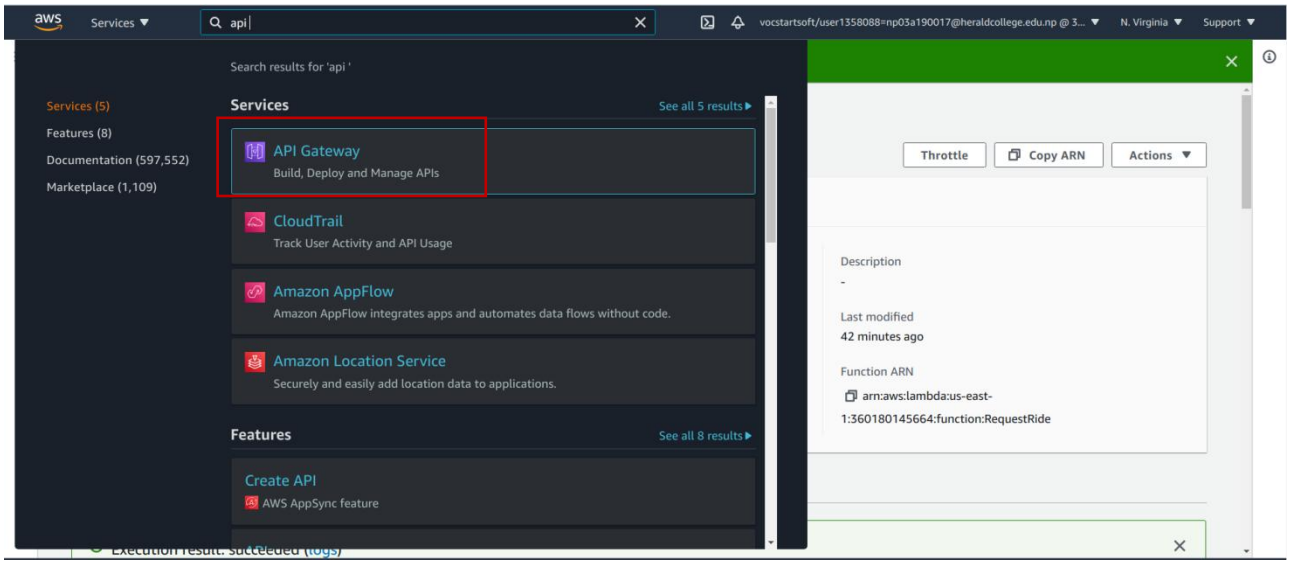


It is the same response with the actions that the function performed i.e. the function logs.

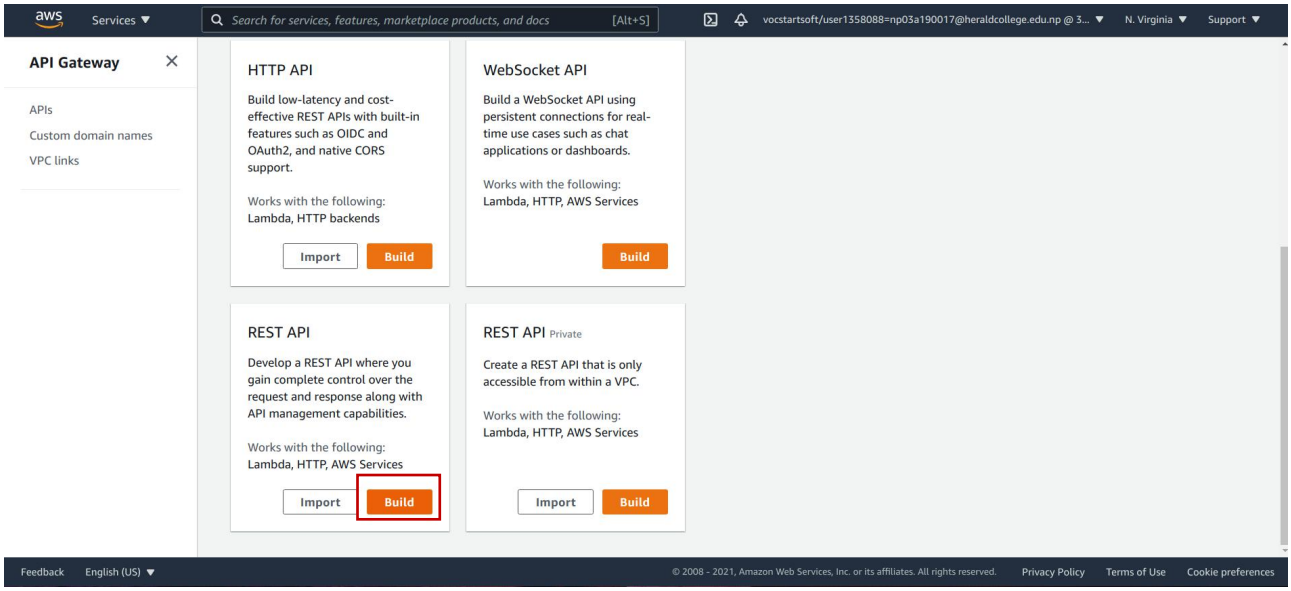
Task 2: Serverless Service Backend

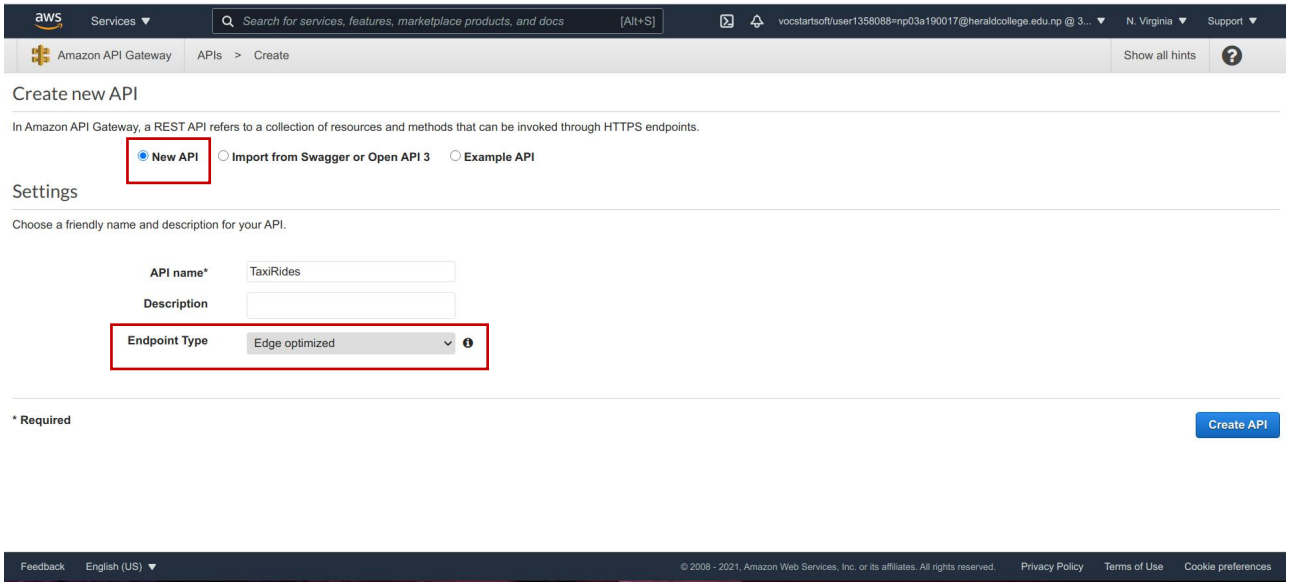
Amazon API Gateway is used here to expose the Lambda function that is built in the previous part as a RESTful API.

Creating a new REST API



For creating a new API Gateway, the required service is selected and a new REST API is created through the click of a “Build” button.

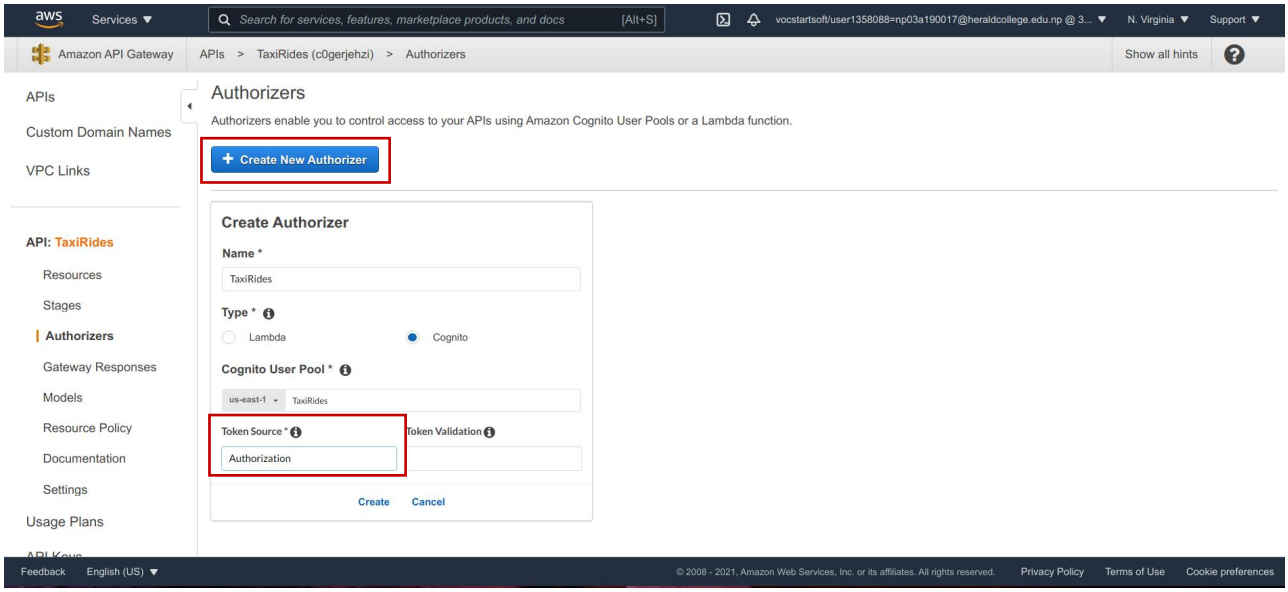




A new API is created here where the endpoint type is selected to be edge optimized. Edge-optimized APIs are endpoints that are accessed through a CloudFront distribution that is created and managed by API Gateway.

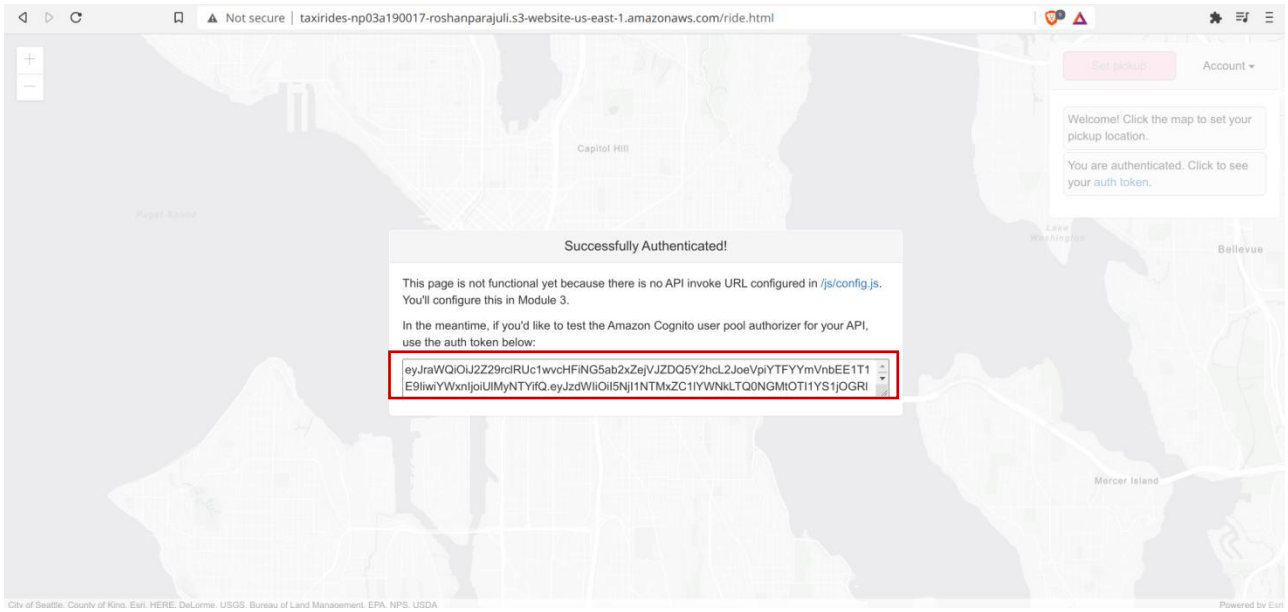
Creating a Cognito User Pools Authorizer

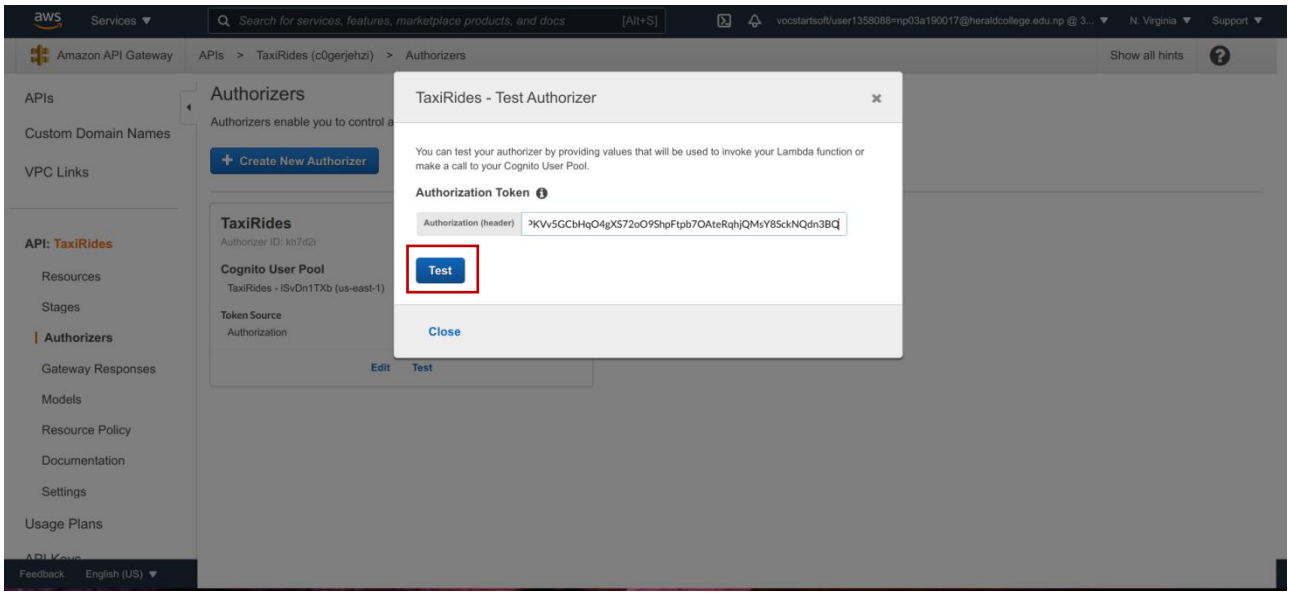
An authorizer needs to be configured so that the Amazon API Gateway can use the tokens returned by Cognito User Pools to authenticate API calls. The authorizer is named TaxiRides and the token source is “Authorization” as it is made for authorization purposes.



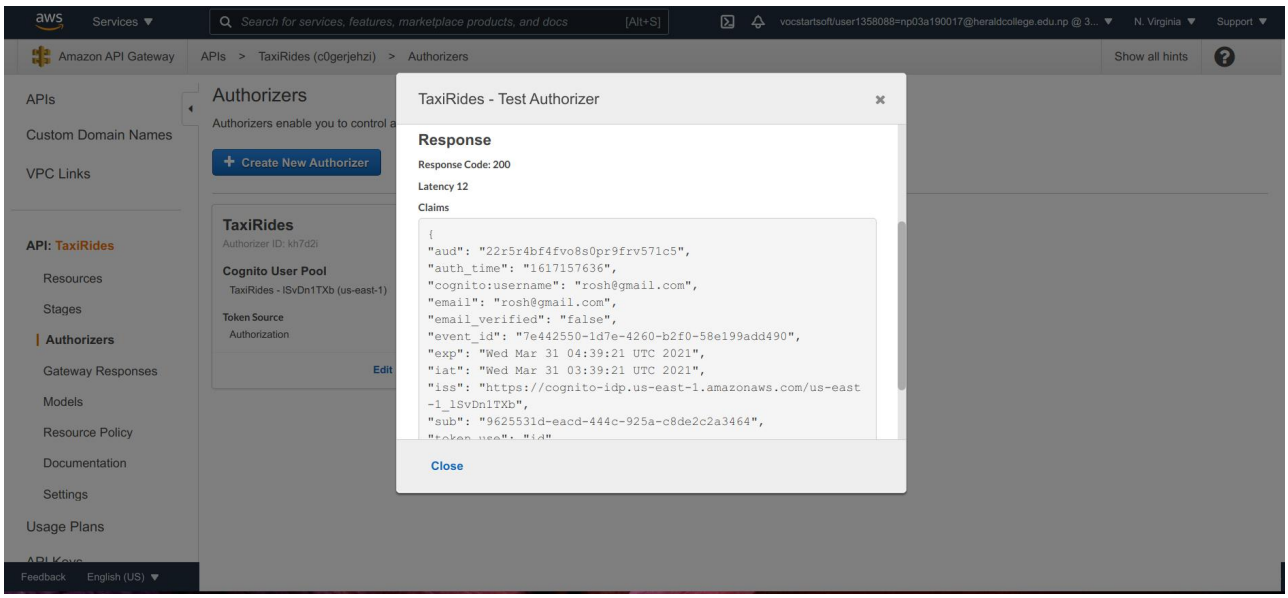
Verifying the authorizer configuration

The auth token is copied from the /ride.html webpage. The token includes the details of the request including the cognito details and many other details.





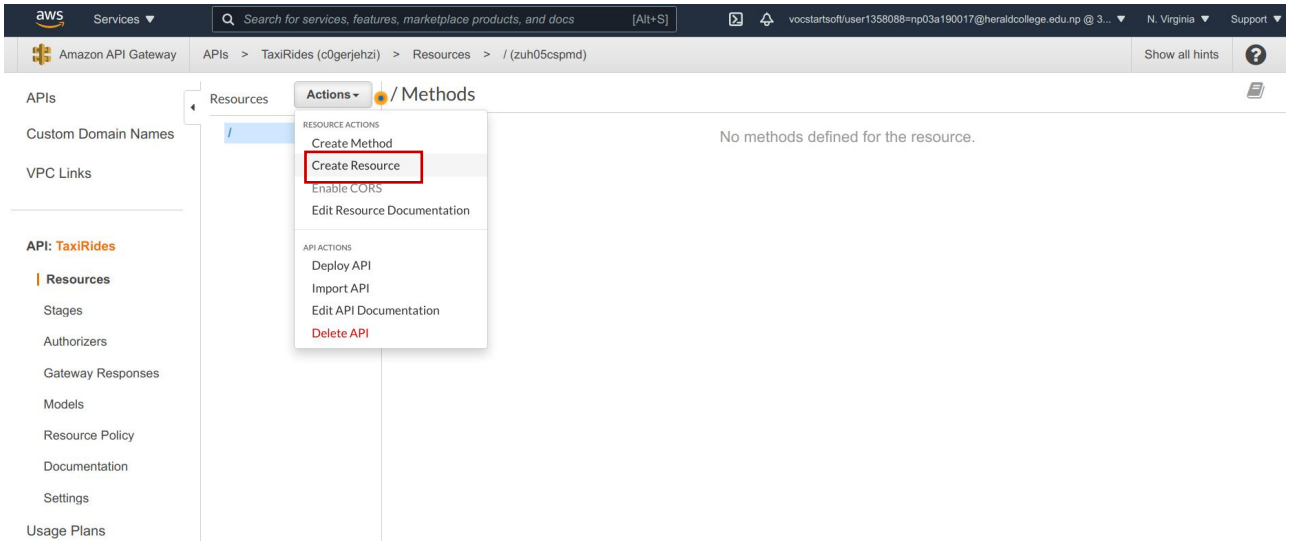
After the token is pasted in the authorizer created above, we can observe all the deails about the request including response code 200 which means that everything was successful.

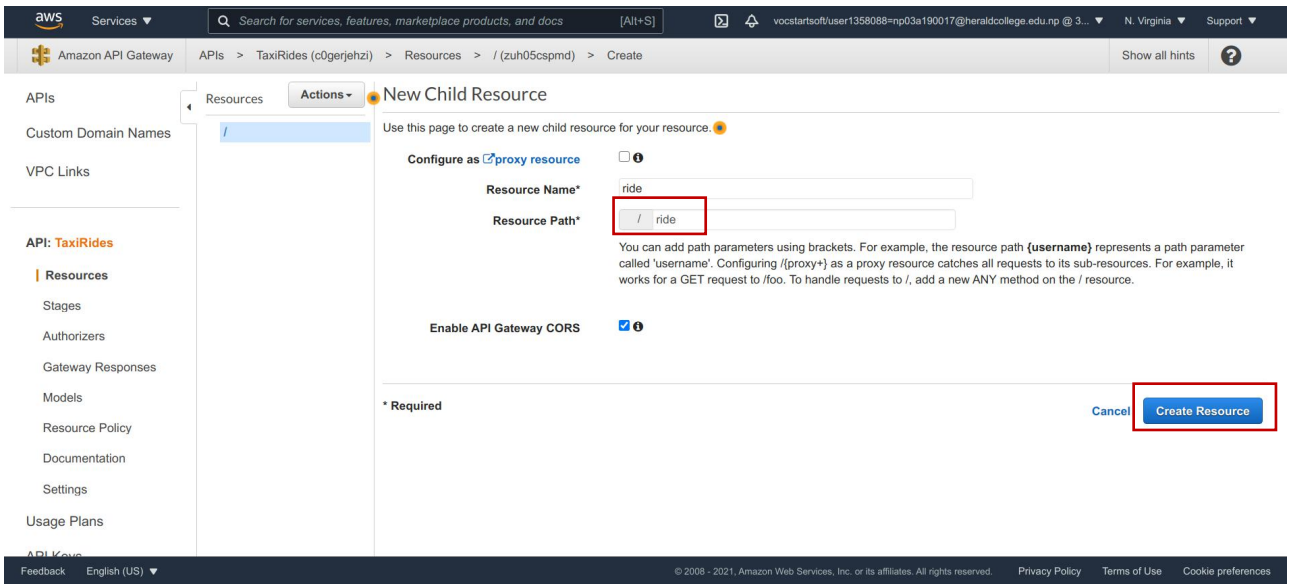


The details from the token is displayed in the screenshot above.

Creating a new resource and method

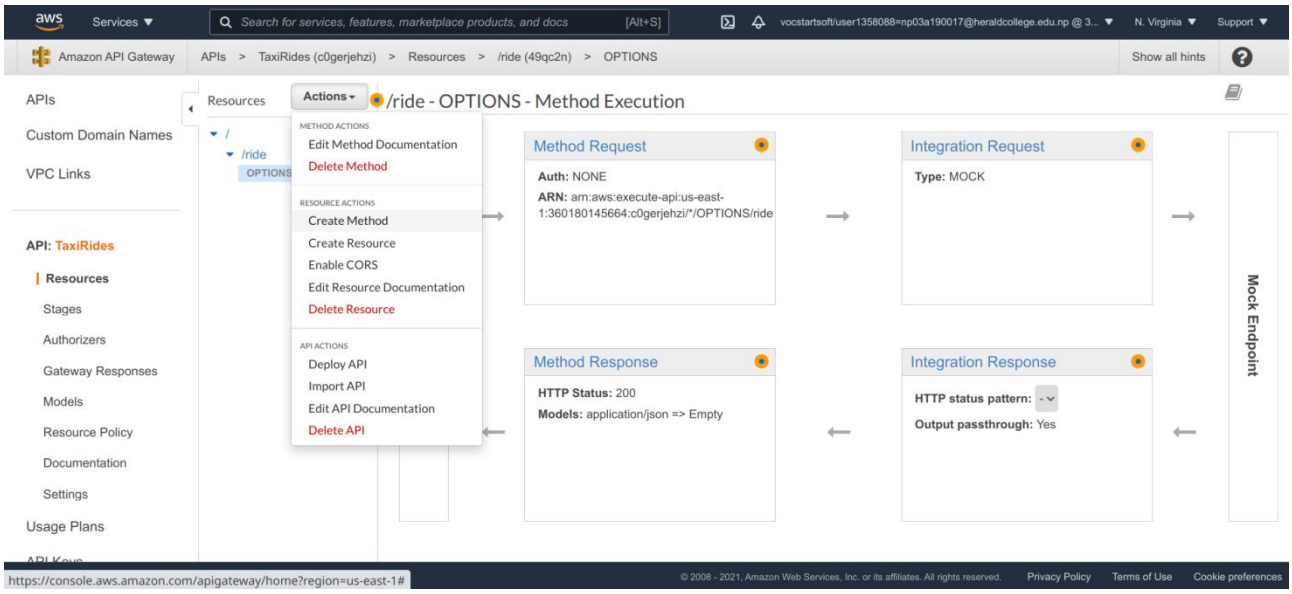
A new resource called /ride is created within the api and then create a POST method with the configuration to use a Lambda proxy integration backend by the RequestRide function that was created.

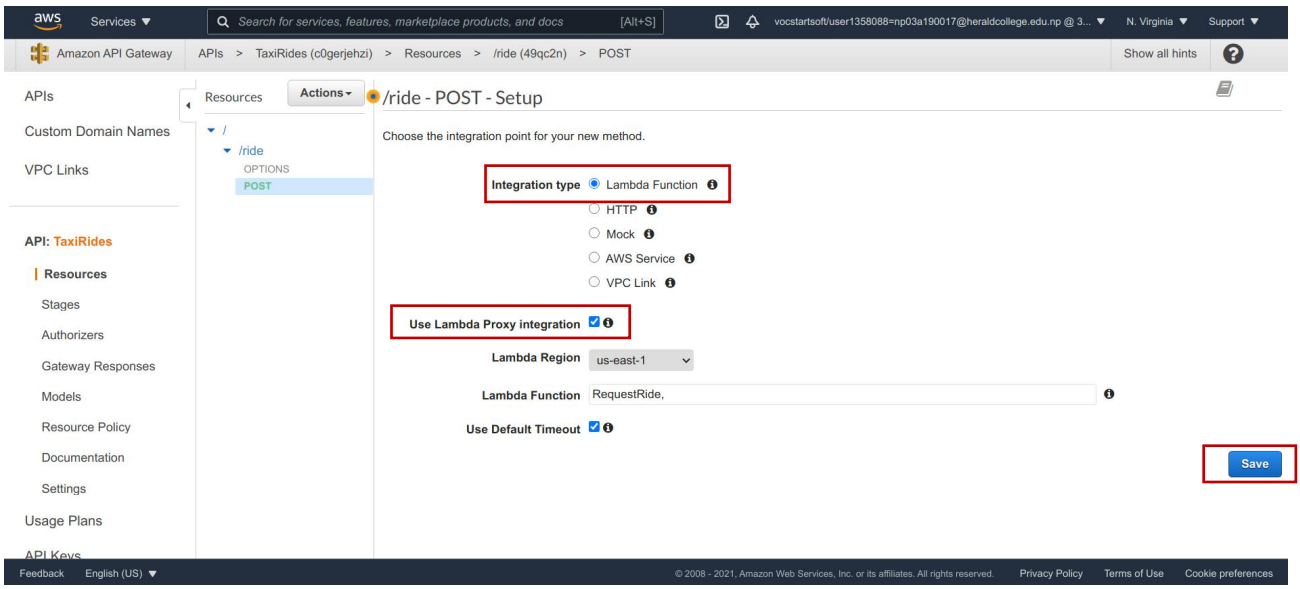




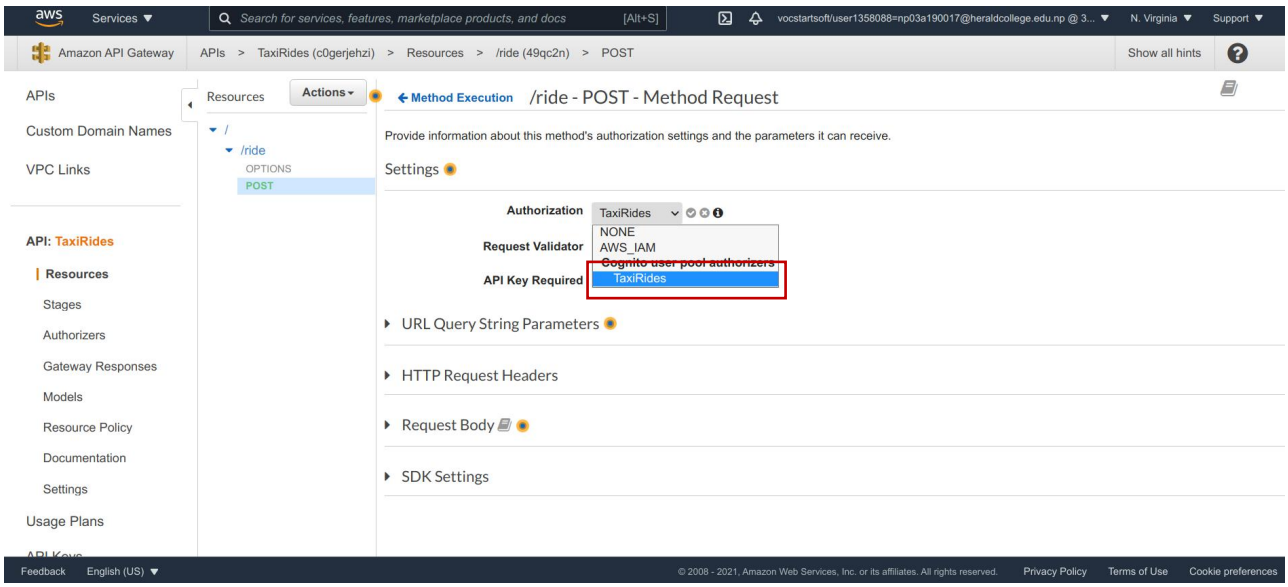
After a new resource path /ride is created, the resource is created.

Now, A new method is created for the POST action which is going to be integrated to the Lambda Function and the Lambda Proxy integration is enabled.



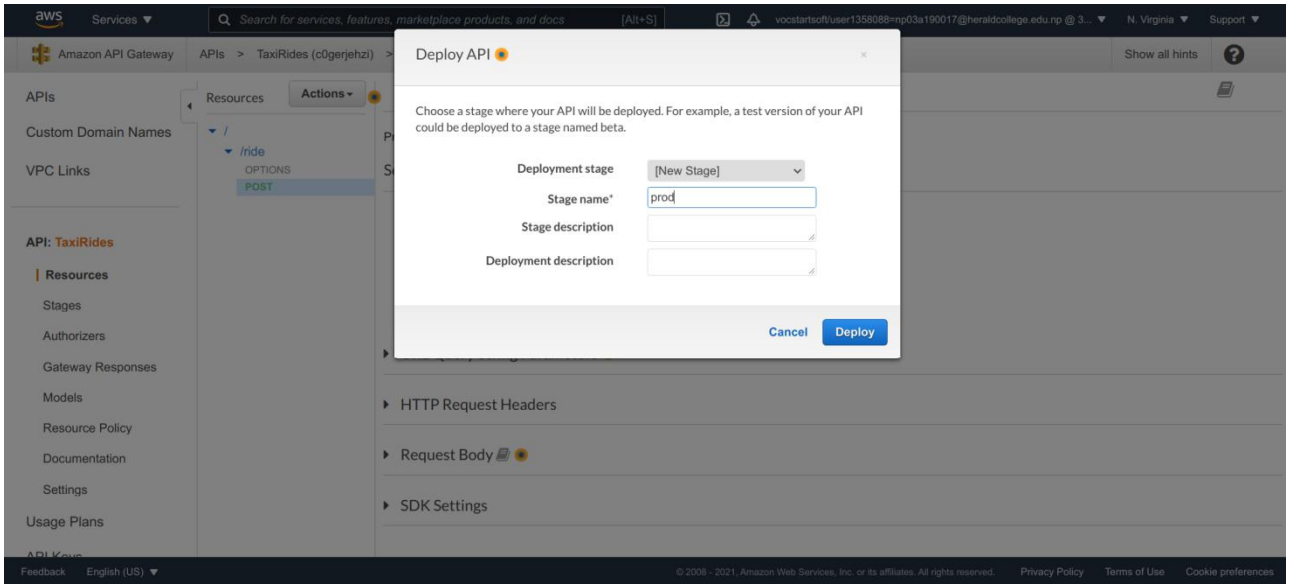


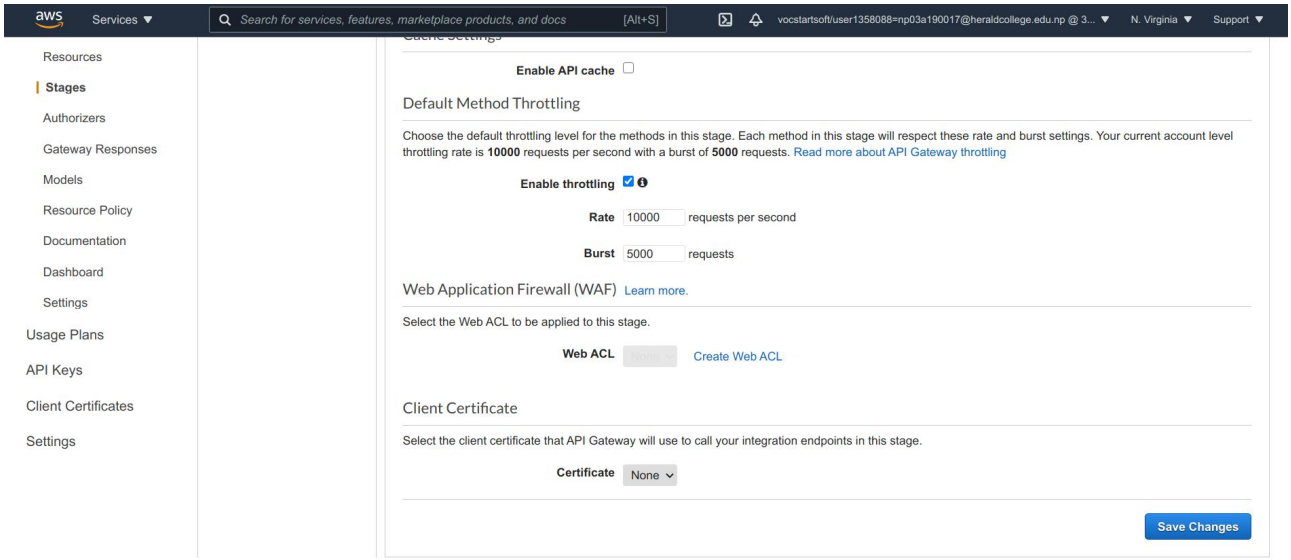
Here the method is integrated to the Lambda Function and the Lambda Proxy integration is enabled.

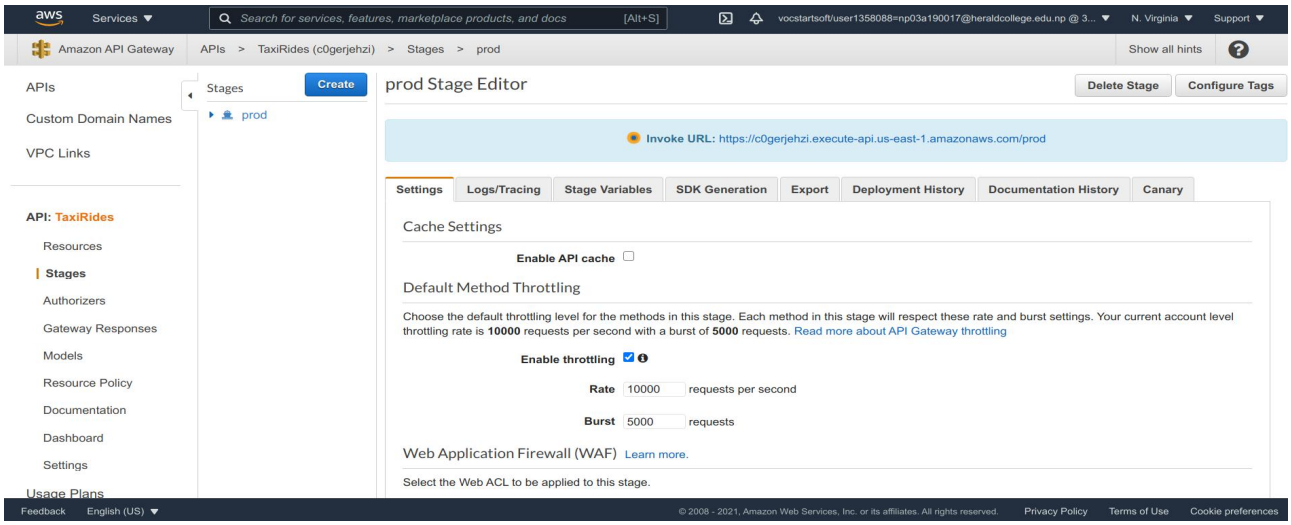


The POST method is authorized with the help of the authorizer created above.

Deploying the API



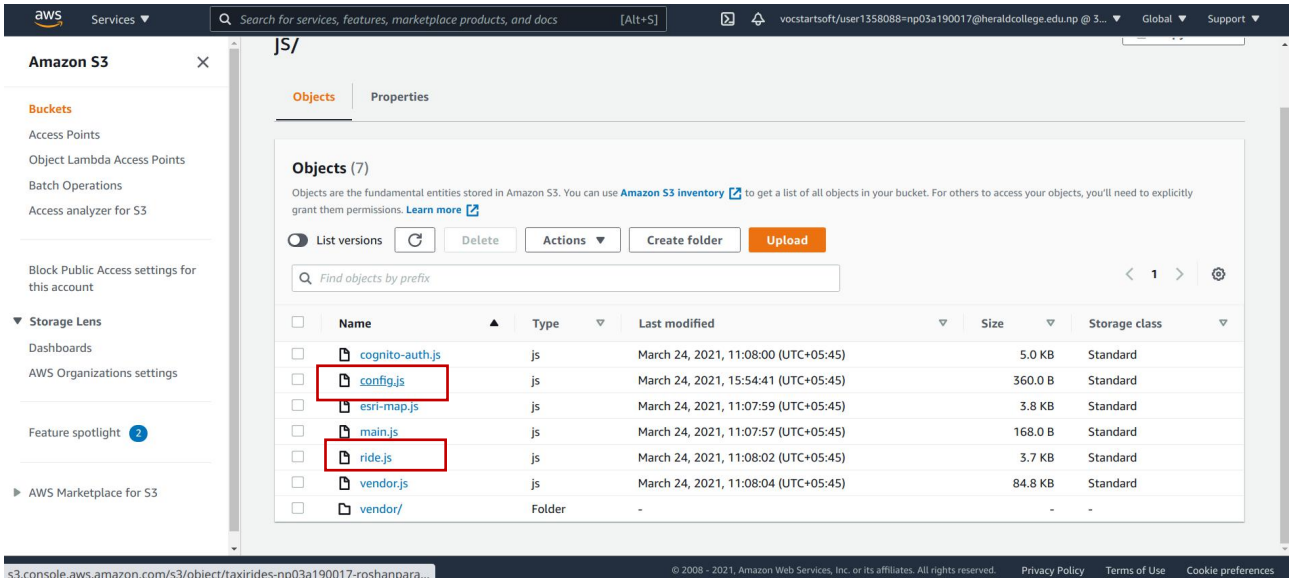


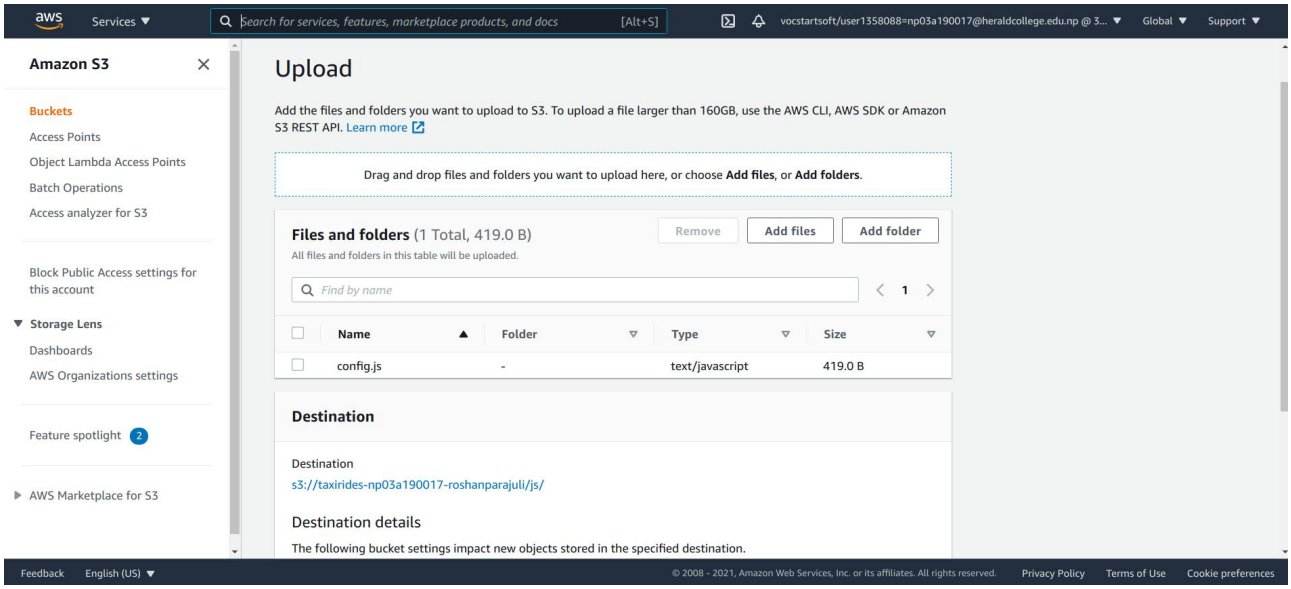


The API is deployed for production for the website to work with the API correctly.

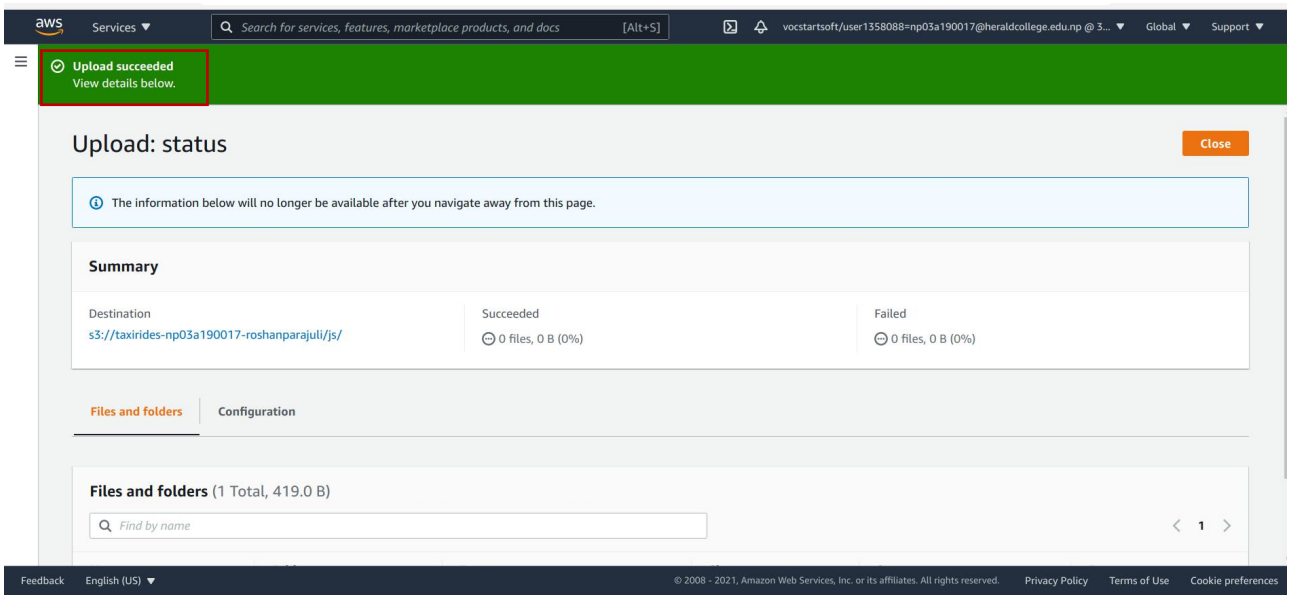
Updating the Website Config

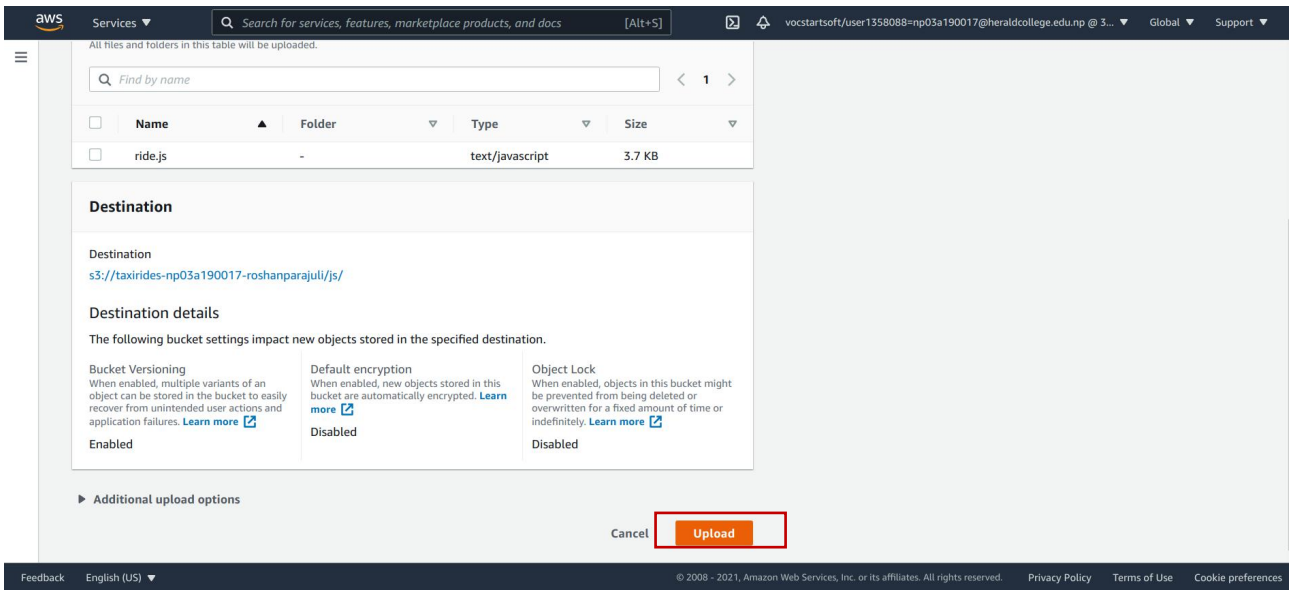
On the S3 bucket, the config file is updated with the new API invocation URL provided in the previous stage. Also, ride.js file is updated in this stage.





Here, the new config file with updation is uploaded.





The config file and ride file is updated successfully.

Validating the implementation

The following is the ride.html configured properly.