***AWS Serverless Architecture***

*Step-by-Step Guide*

Made by: - **Manav Khandurie**

Source Code: -

<https://github.com/Manav-Khandurie/Attendence_Managment_System_AWS-Serverless.git>

System Design

**Project :** Attendance Management System

**Tech Stack:**

**Frontend :** React, HTML, CSS, JavaScript

**Backend** **:** Python, Boto3

**Databases :** AWS DynamoDB [NoSQL]

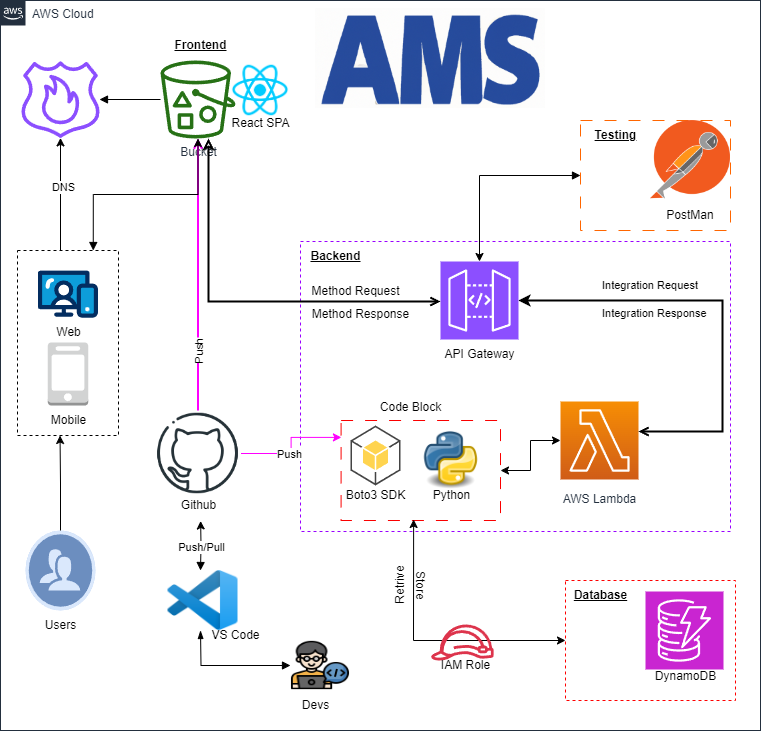
**DevOps Tools :** Git/GitHub

**IDE/Code editor :** VS Code

**Testing Tools :** Postman

**Cloud :** AWS

***Proposed Architecture***



**Explanation**

* **Frontend:** The frontend of the AMS app is a React single-page application (SPA) or HTML CSS JS. This means that the entire user interface is loaded in a single web page, which makes the app feel more like a native desktop application.
* **Backend:** The backend of the AMS app is serverless, which means that it is made up of small, independent functions that are run on demand. This makes the app more scalable and cost-effective.
* **API Gateway:** The API Gateway is the entry point for all requests to the AMS app. It routes requests to the appropriate backend function.
* **Integration Requests and Responses:** When a user interacts with the AMS app, their request is first sent to the API Gateway. The API Gateway then routes the request to the appropriate backend function. The backend function processes the request and sends a response back to the API Gateway. The API Gateway then sends the response back to the user.
* **Mobile:** The AMS app can also be accessed from mobile devices. The mobile app uses the same API Gateway as the web app.
* **Push Notifications:** The AMS app can send push notifications to users' devices. This is useful for things like keeping users up-to-date on the latest activity in their apps.
* **Version Control:** The AMS app uses Git for version control. This means that developers can track changes to the code and revert to previous versions if necessary.
* **Databases:** The AMS app uses DynamoDB for its database. DynamoDB is a NoSQL database that is highly scalable and fault-tolerant.
* **Security:** The AMS app uses a number of security features, including IAM roles, API keys, and encryption. These features help to protect the app and its data from unauthorized access.

**Procedure**

**Part 1 [Creation of DynamoDb Table]**

**Step 1 :** Install GitBash , VSCode , Nodejs on your local system

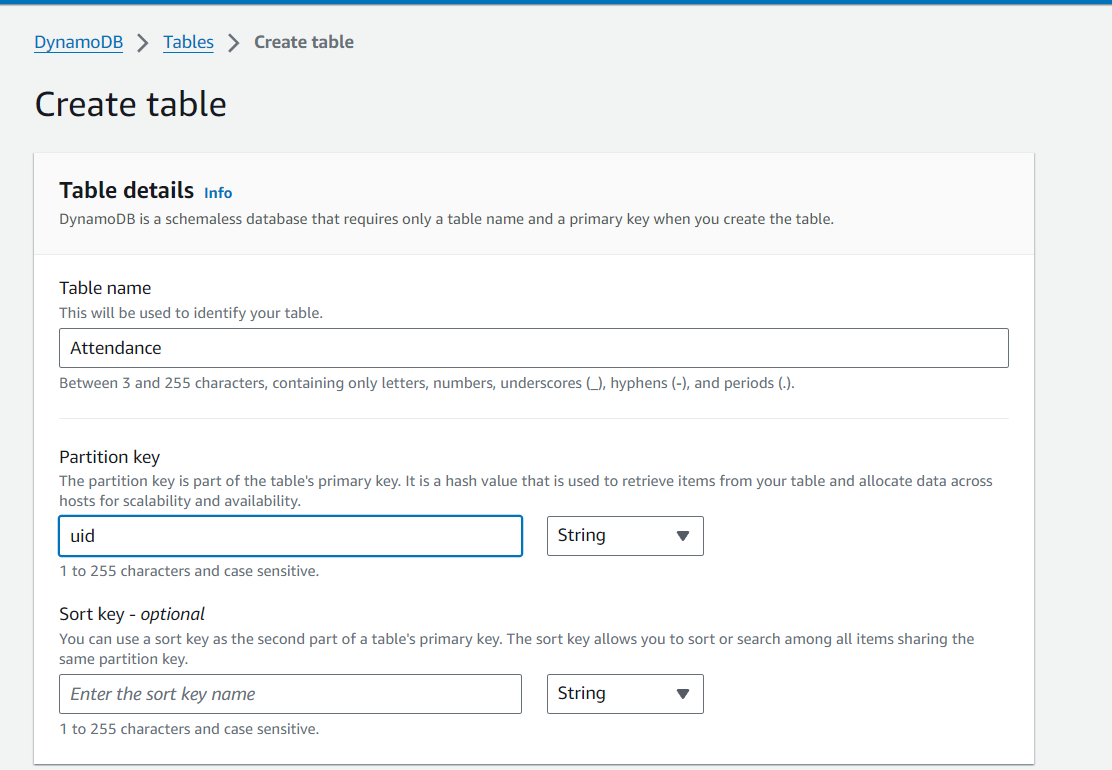
If you haven’t installed them yet follow this guide   
 GitBash : <https://youtu.be/naL0cZNQh1g?si=wPIbKcRTaayIakZK>

VSCode :  [https://youtu.be/AdzKzlp66sQ?si=\_besF5b3SW2CoEau](%20https:/youtu.be/AdzKzlp66sQ?si=_besF5b3SW2CoEau)

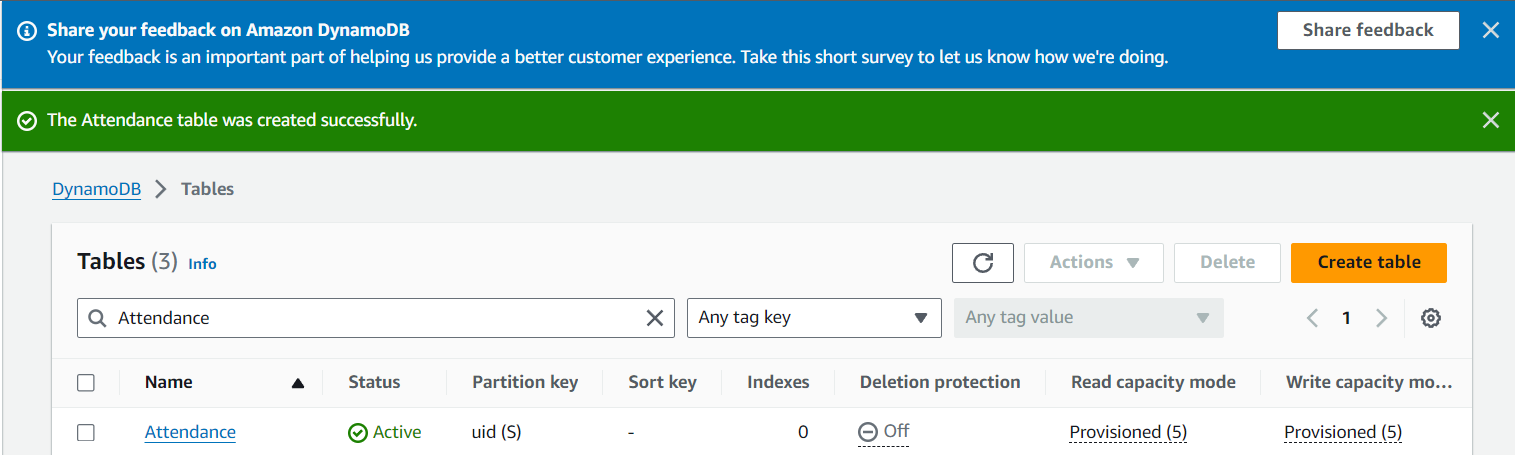
NodeJs : <https://youtu.be/JINE4D0Syqw?si=vtjgfgJ99Jh945Hc>

**Step 2 :** Login to AWS Console and go to DynamoDB

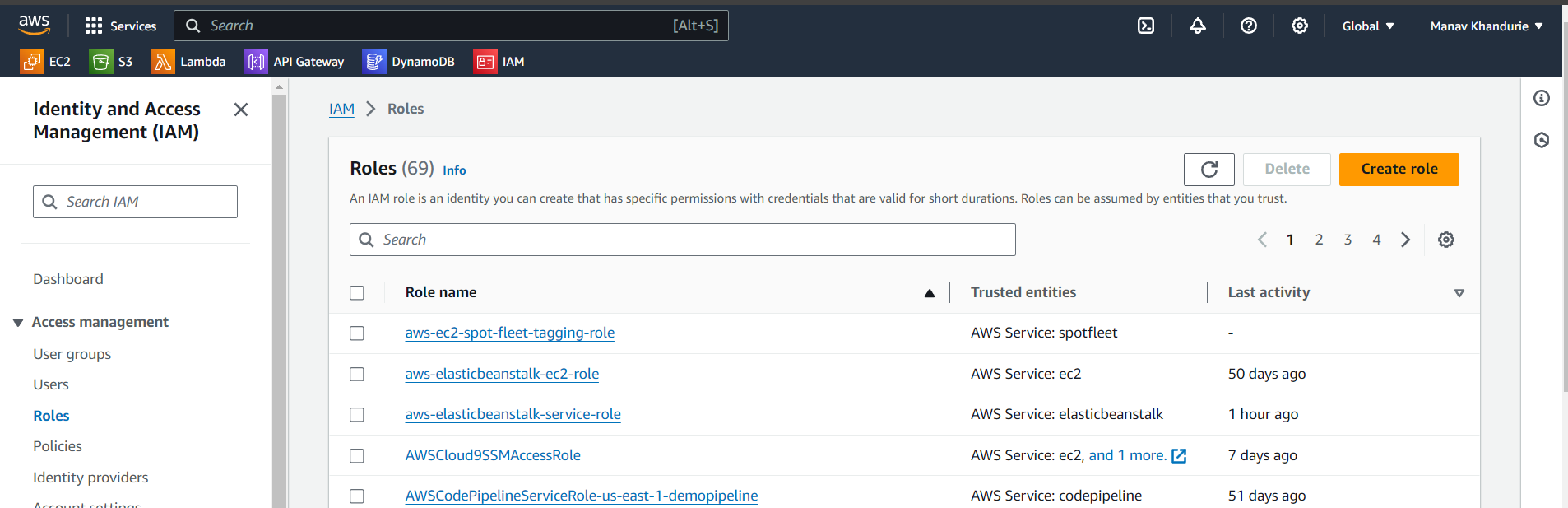
Create a new table called Attendance and set partition key as ‘uid’



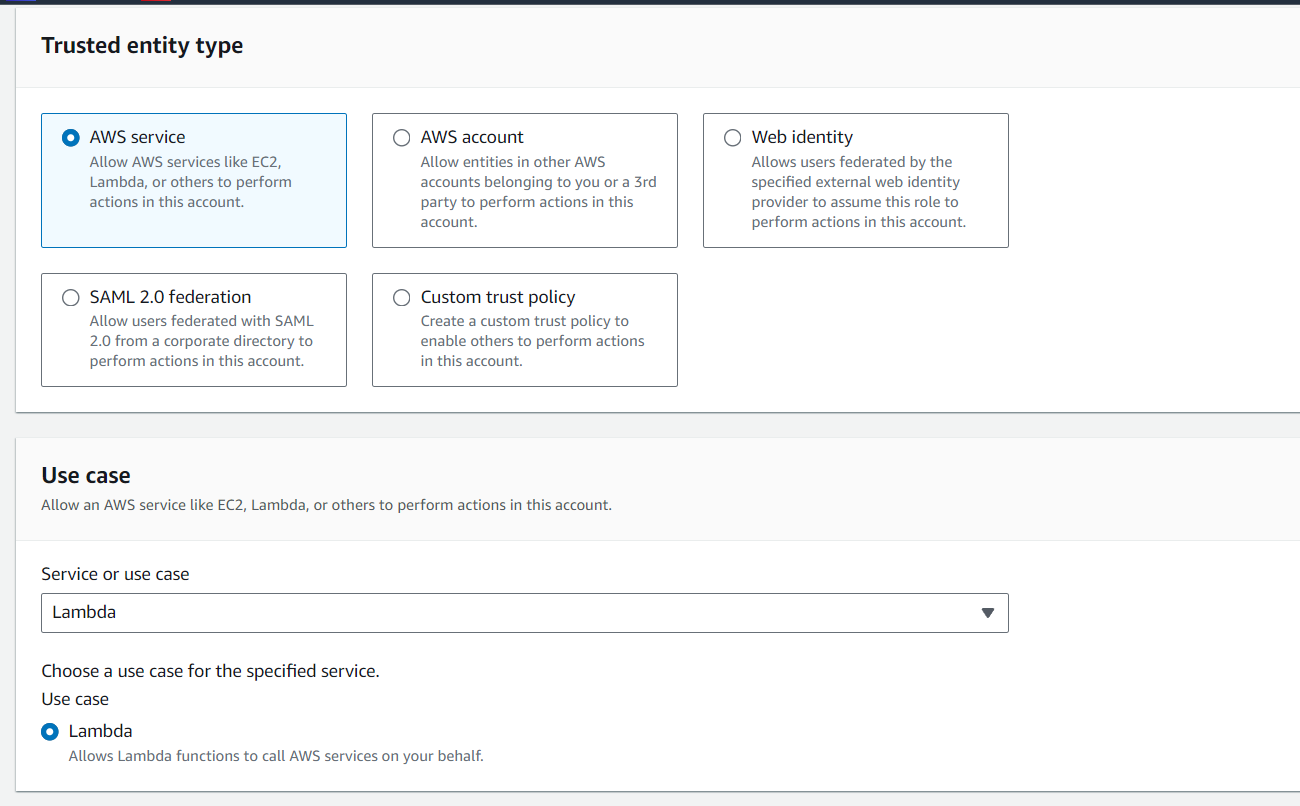
And finally create the table

**Step 3 :** Select the table and add some dummy data to it 

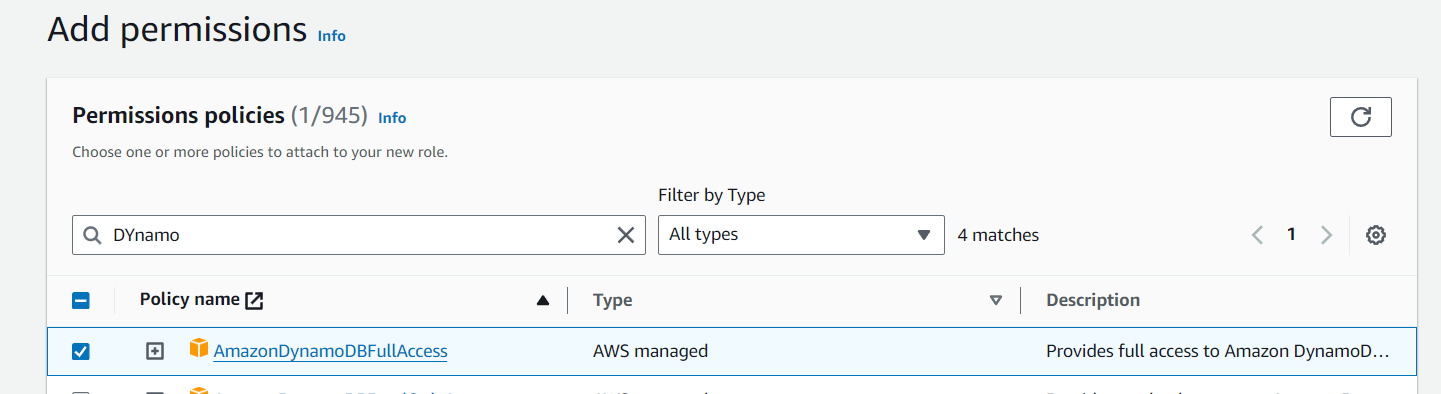
**Step 4 :** Now go to IAM service and then to IAM Role 🡪 then create role



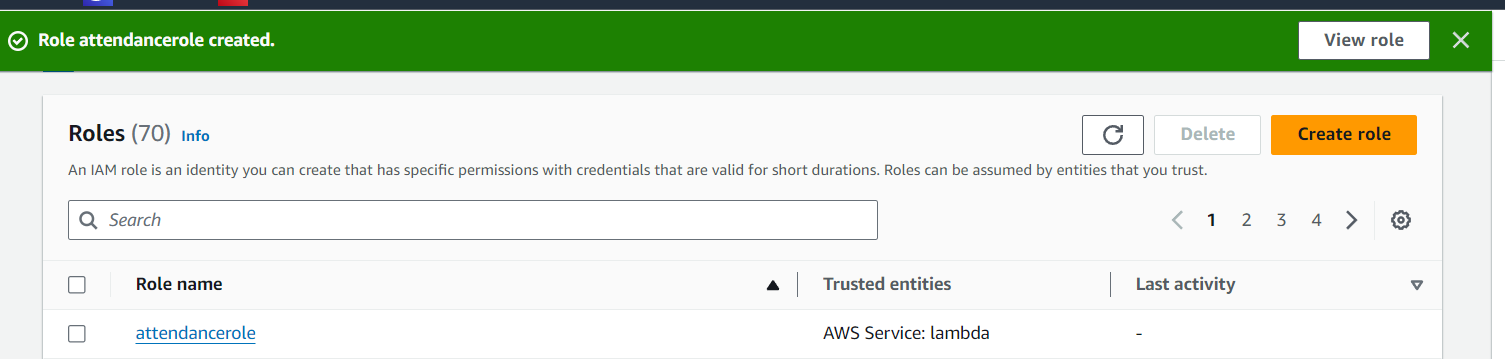
**Step 5 :** Now create a new role called“attendancerole” & usecase as Lambda



**Step 6 :** Under “Add Permission” add “[AmazonDynamoDBFullAccess](https://us-east-1.console.aws.amazon.com/iam/home?region=us-east-1#/policies/details/arn%3Aaws%3Aiam%3A%3Aaws%3Apolicy%2FAmazonDynamoDBFullAccess)”

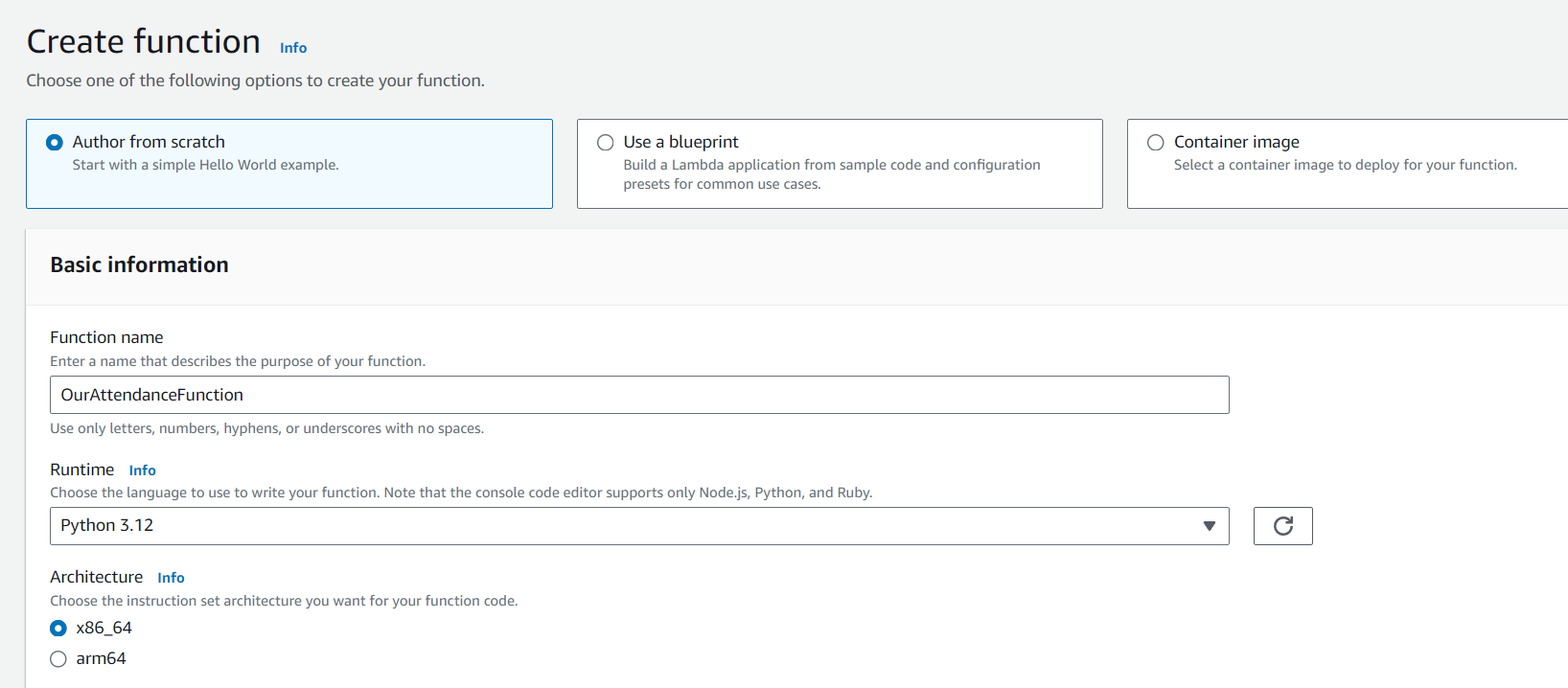


**Step 7 :** Give a role name and create role



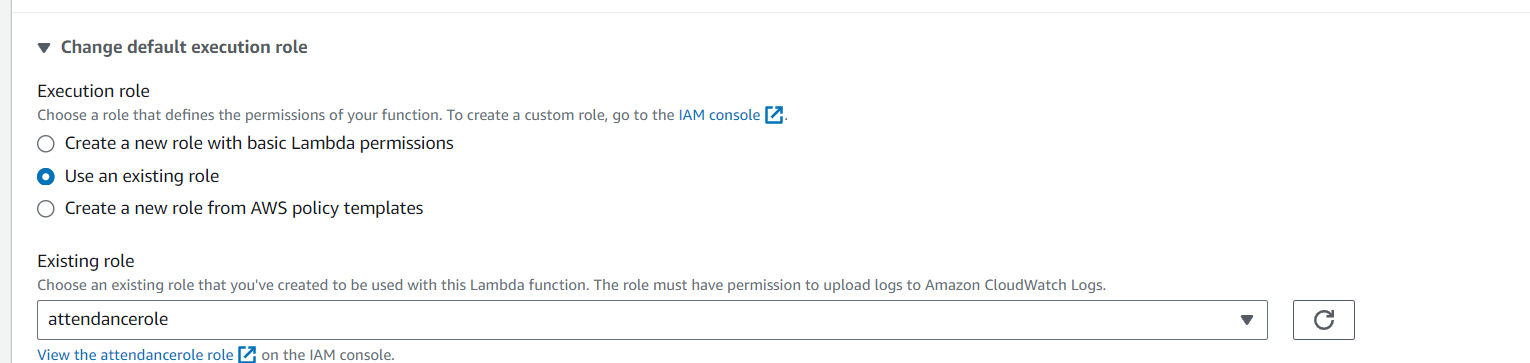
**Part 2 [Creation of Lambda Function ]**

**Step 8 :** Now go to AWS Lambda , and create a new function called “OurAttendanceFunction” in the environment Python 3.12 and architecture as x86\_64

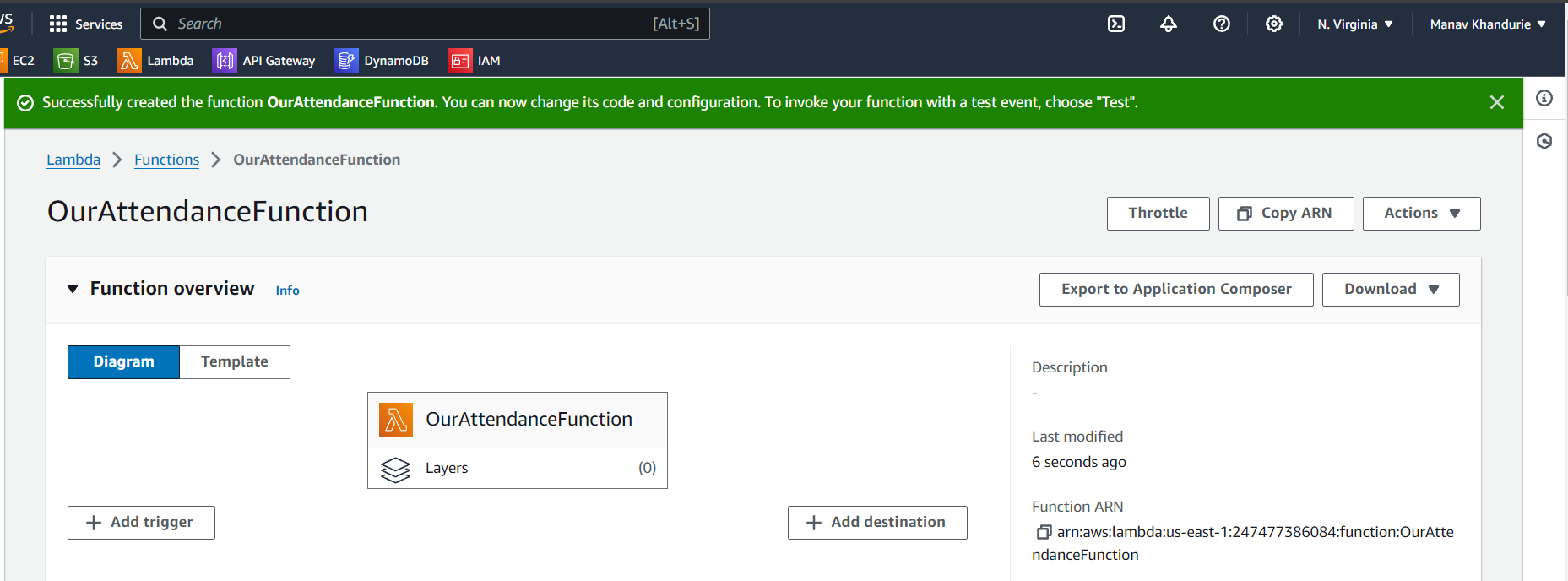


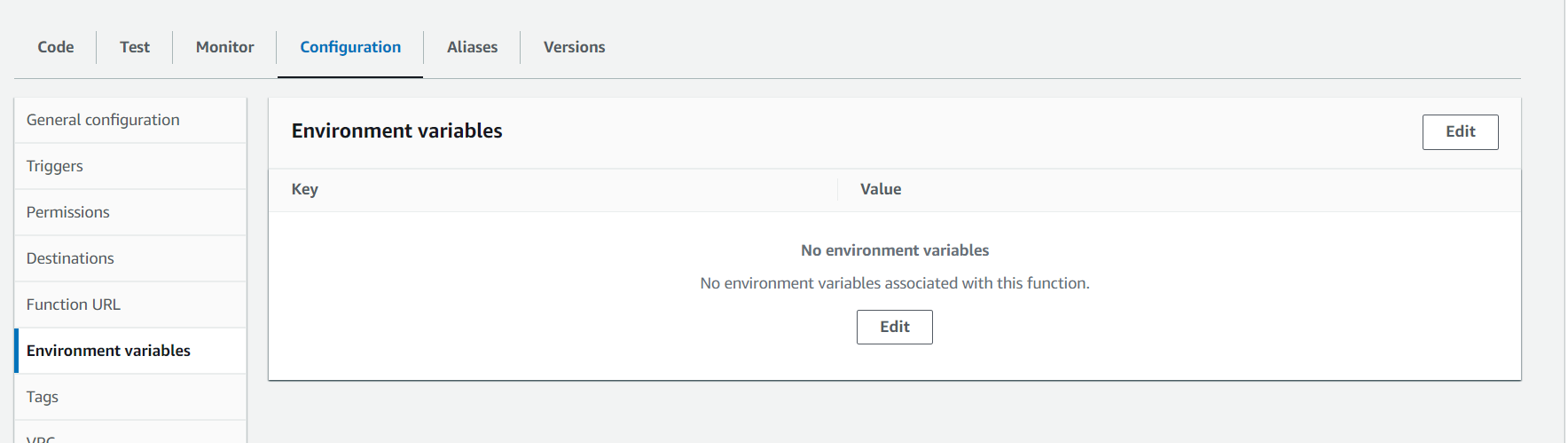
**Step 9 :** Under the default execution role , we select “Use an existing role”

And we select the role “attendancerole” that we created in step 7

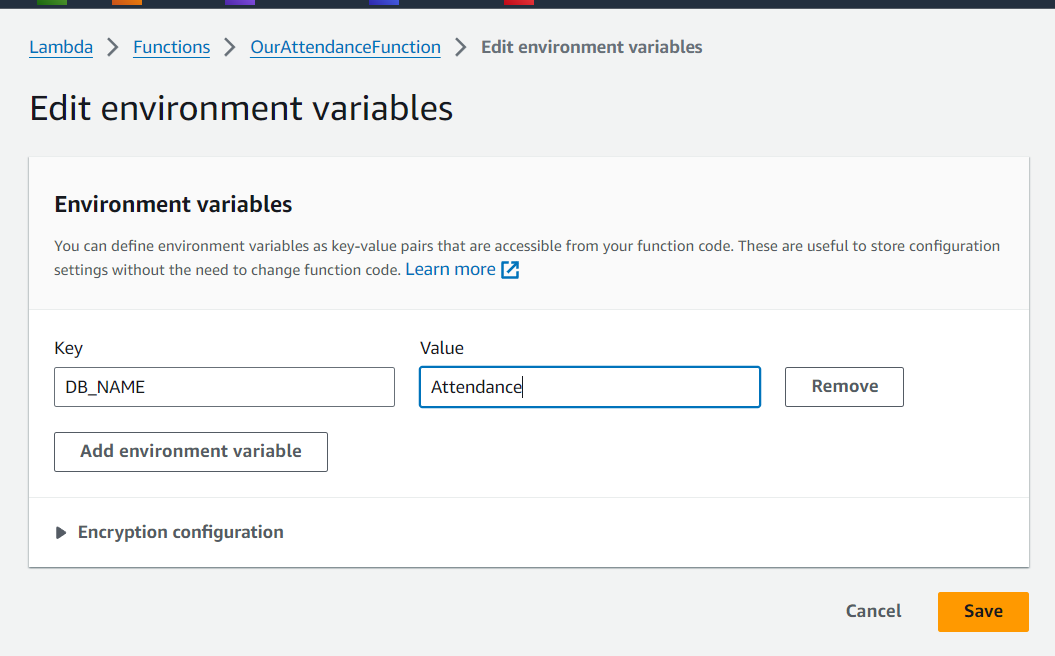


**Step 10:** Finally we click on create the function

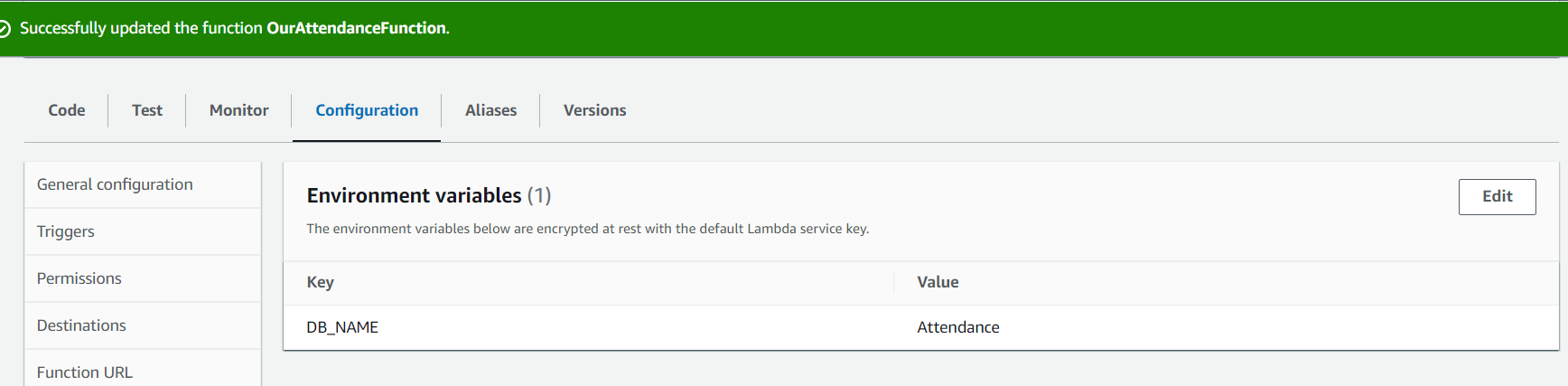


**Step 11:** Under the heading “configuration” we goto environment variables  


**Step 12:** Here we click on edit and add a new variable named DB\_NAME



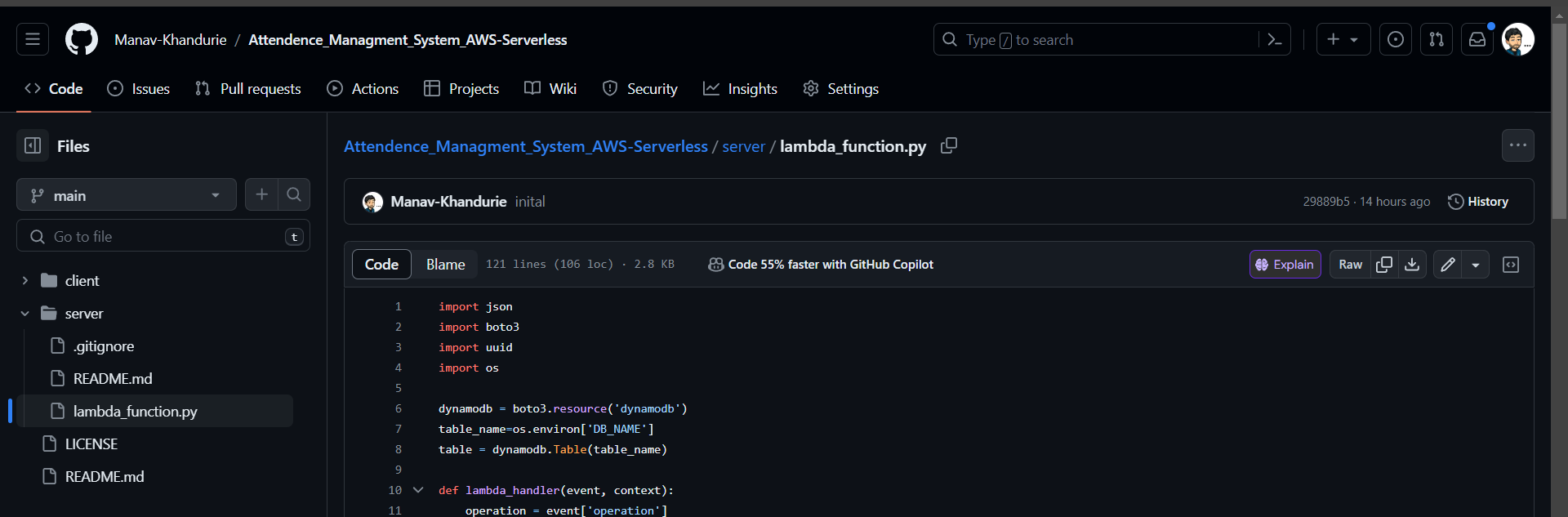
**Step 13 :** Click on save to store the env variable



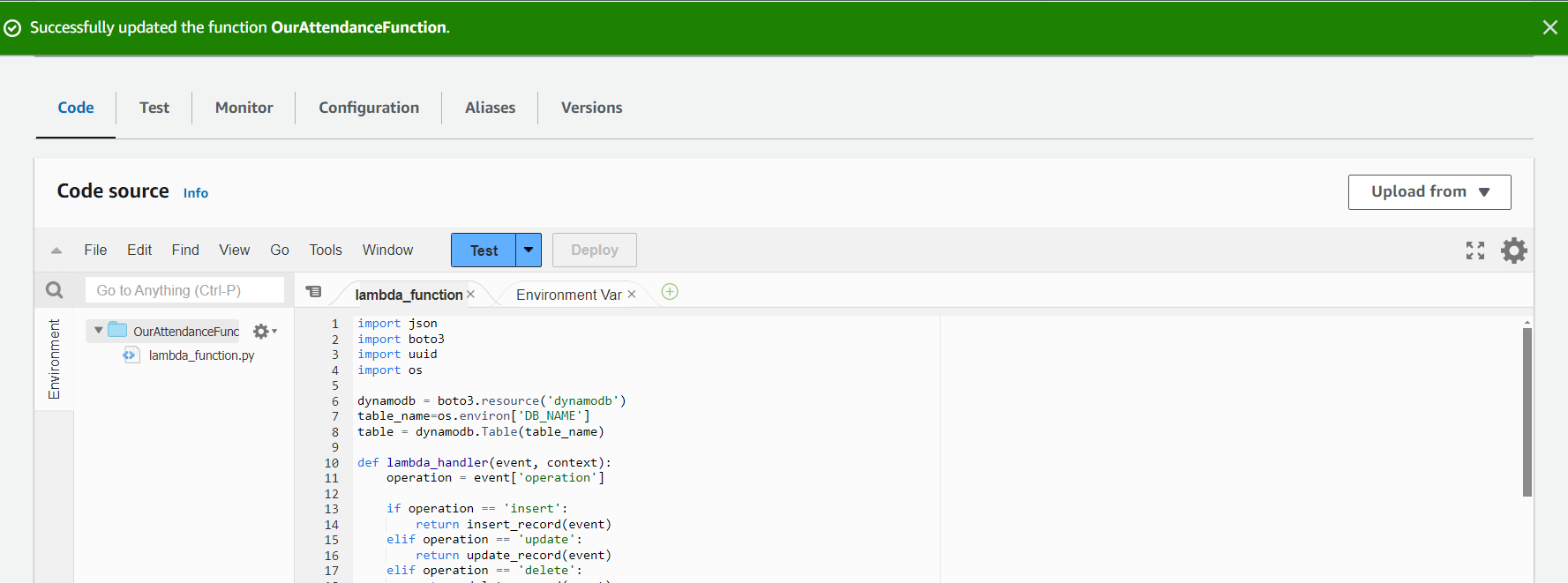
**Step 14 :** In the section Code 🡪 lambda\_function.py

We add the code that’s provided in the source code present at

Server/lambda\_function.py , simply copy and paste the code in the browser editor



After pasting the code click on deploy



**Step 15 :** Now we start by testing the lambda function , so we goto test , we assign a eventname called “inserttest” and add a requestbody as given below , finally click save & test

`{

  "operation": "insert",

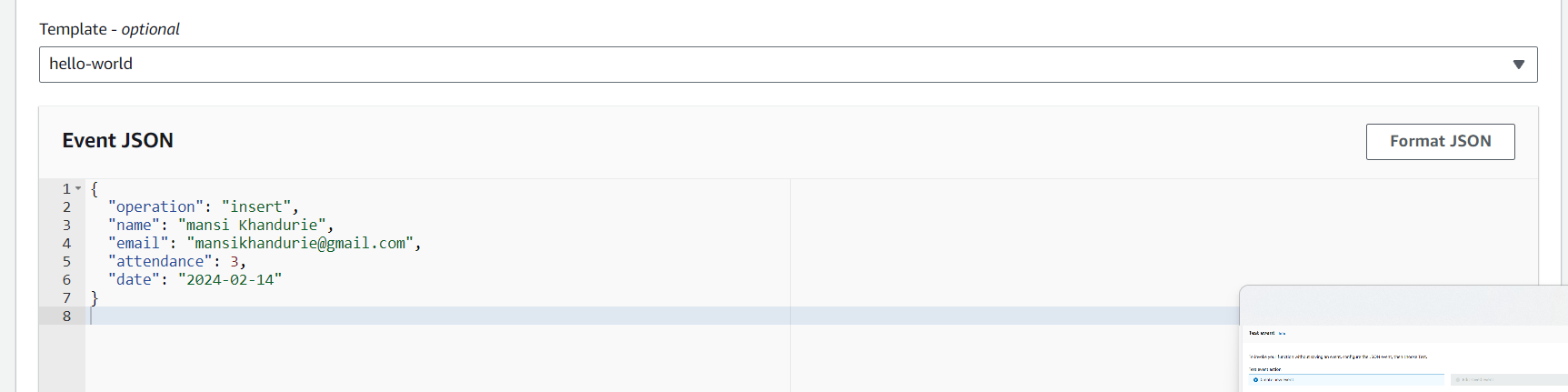
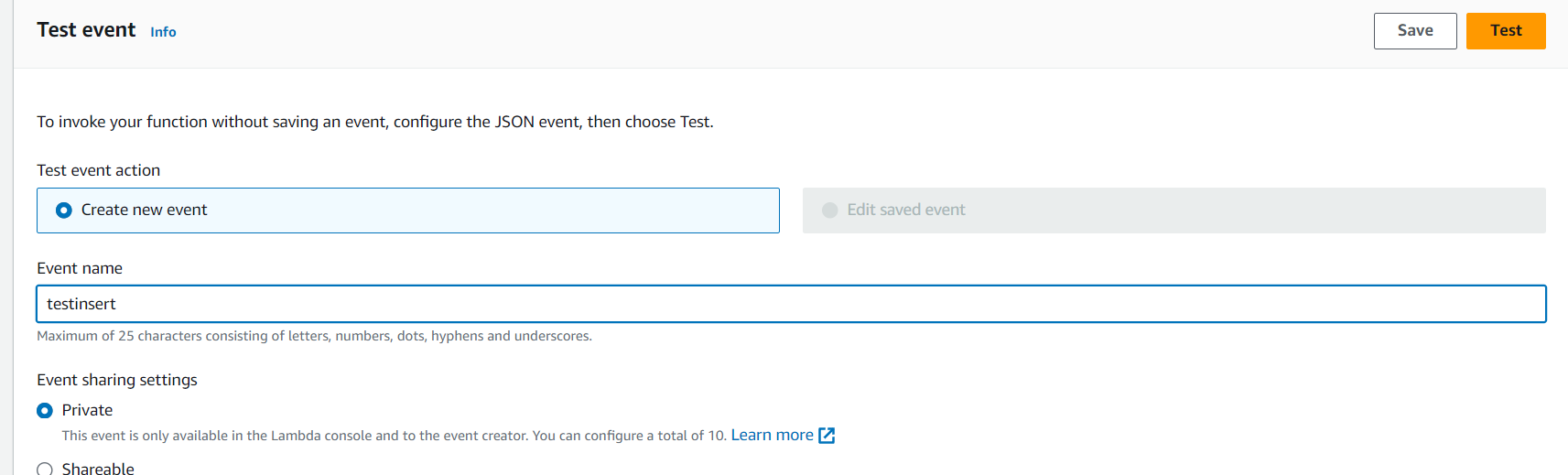
  "name": "manav Khandurie",

  "email": "manavkhandurie@gmail.com",

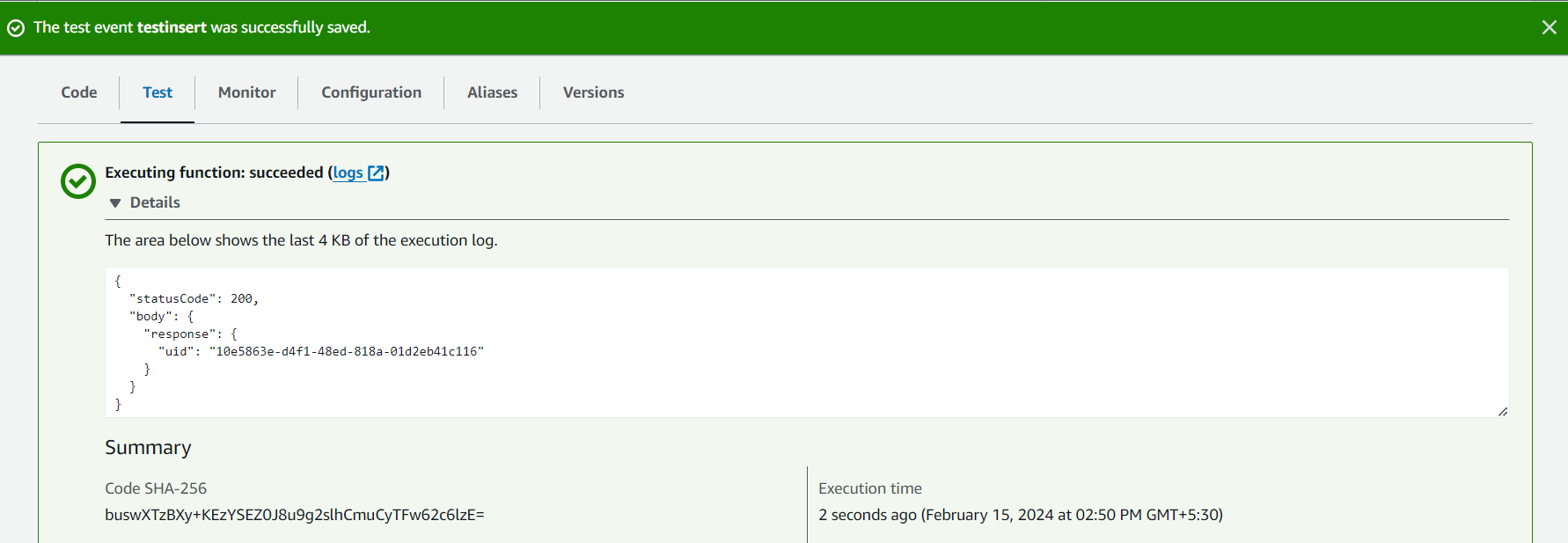
  "attendance": 3,

  "date": "2024-02-14"

}



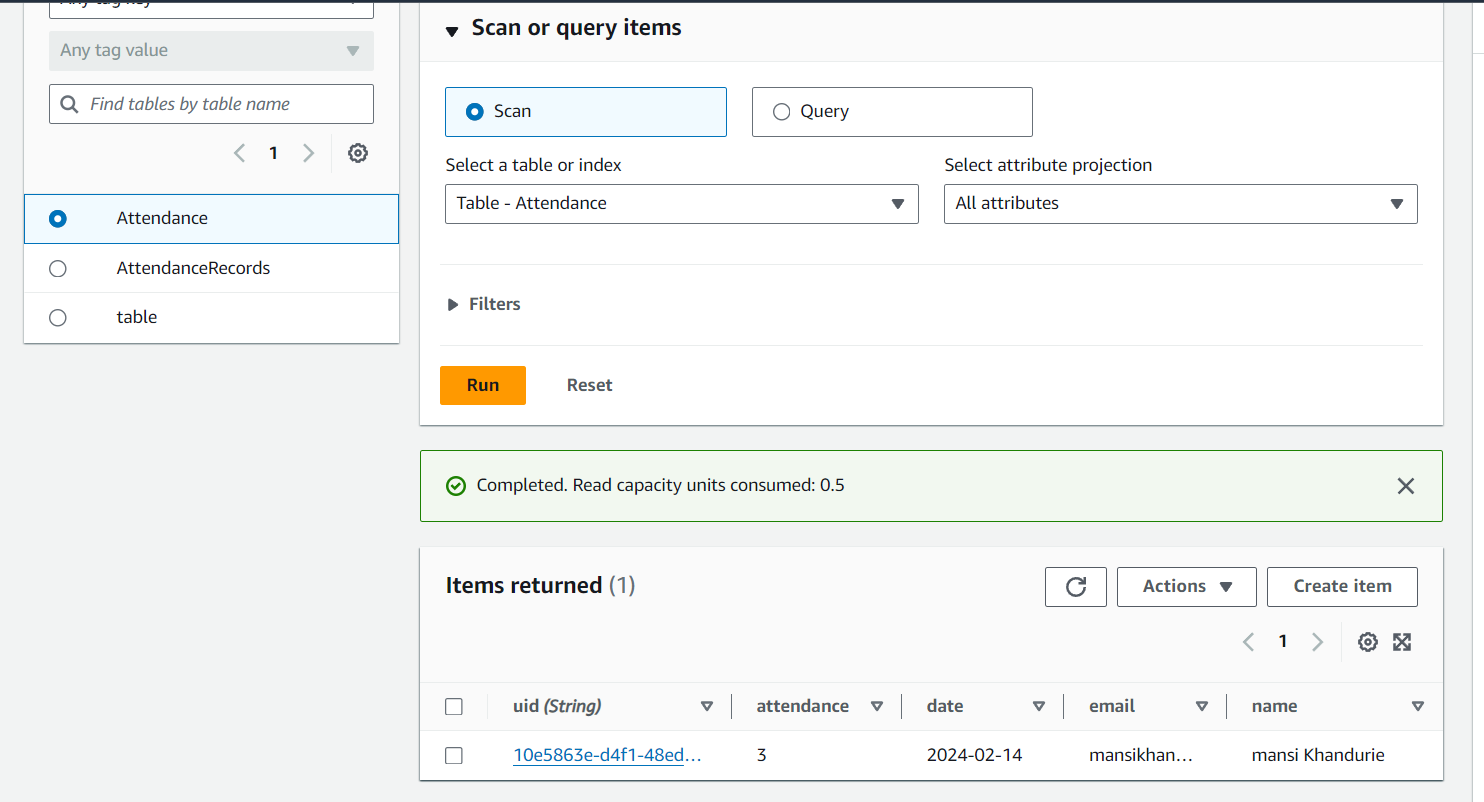
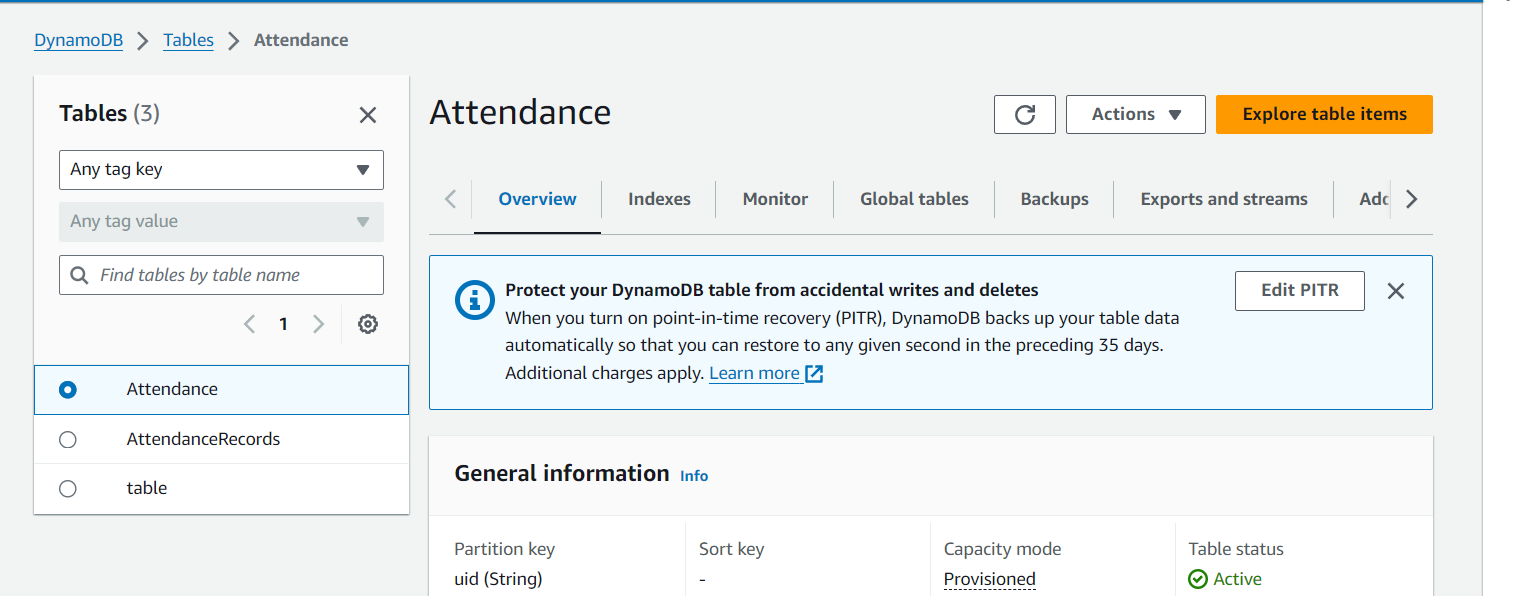
**Step 16 :** The output should be a statuscode 200 as below



Note :- If there is an error go through steps 1 -16 again and check

Also tally at the dynomoDb Table

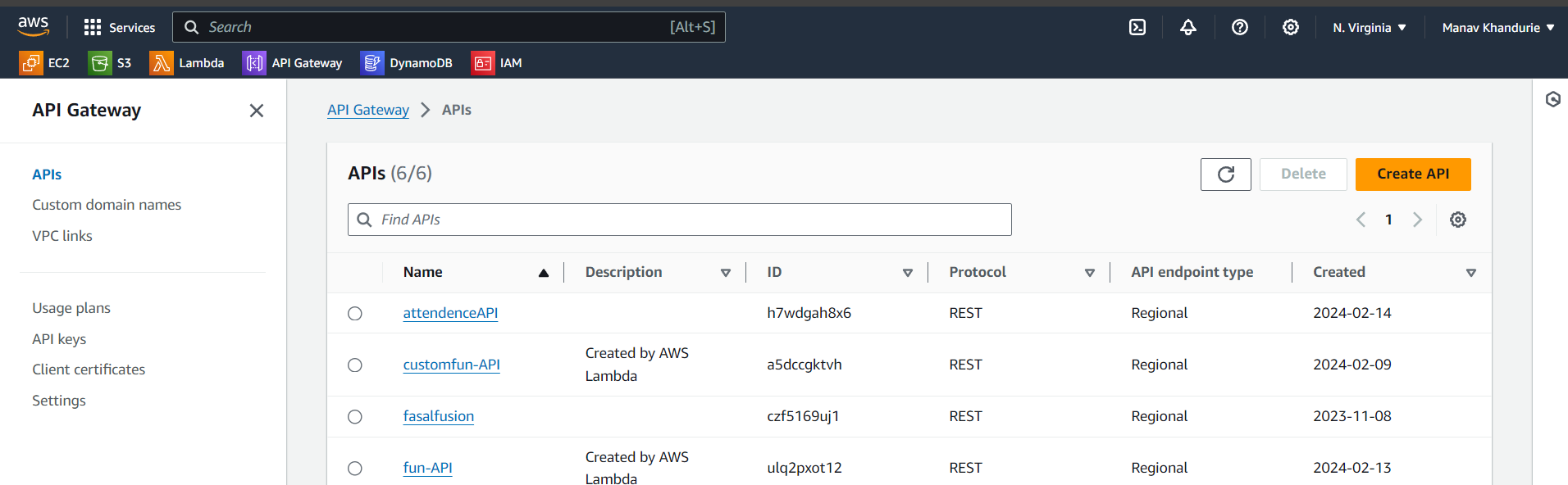
Select your table and click explore table items

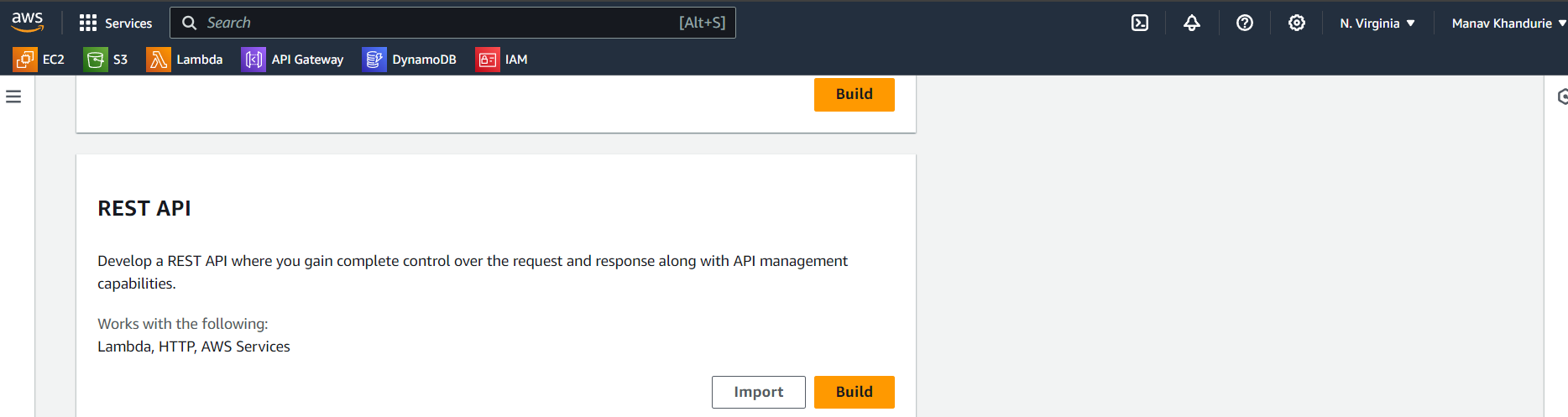


As we can see a record has being inserted so our table , i.ee lambda function works as expected.

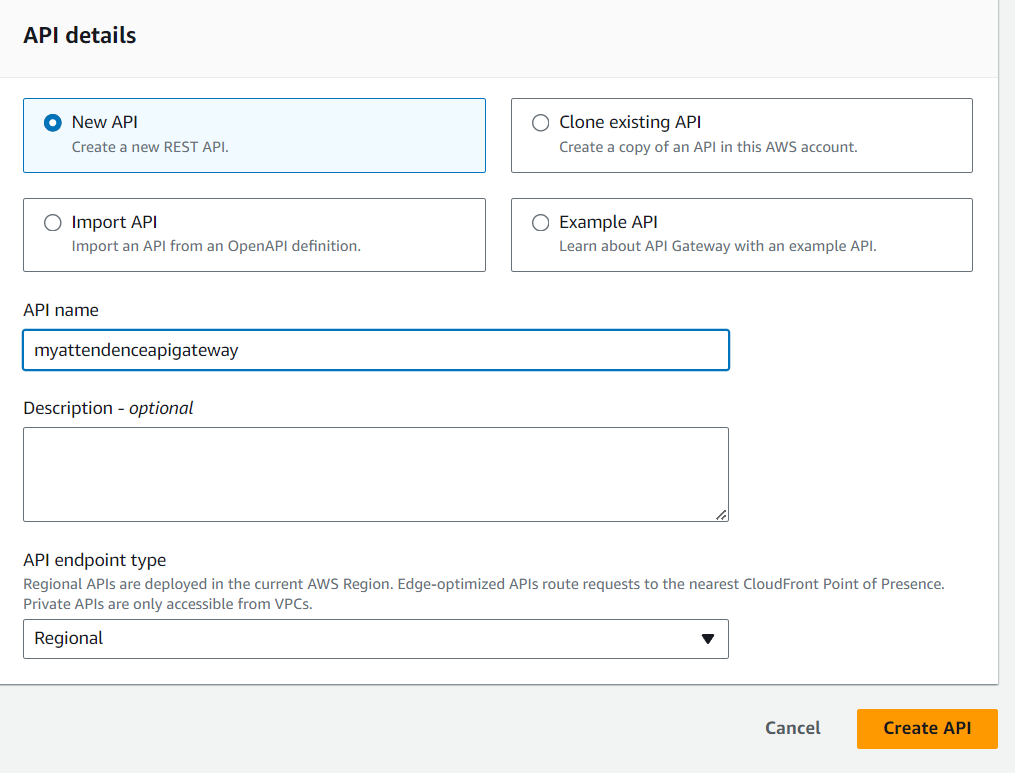
**Part 3 [Creation of API GATEWAY]**

**Step 17 :** After creation of Lambda , we goto API Gateway 🡪 Create 🡪 Rest API Creation

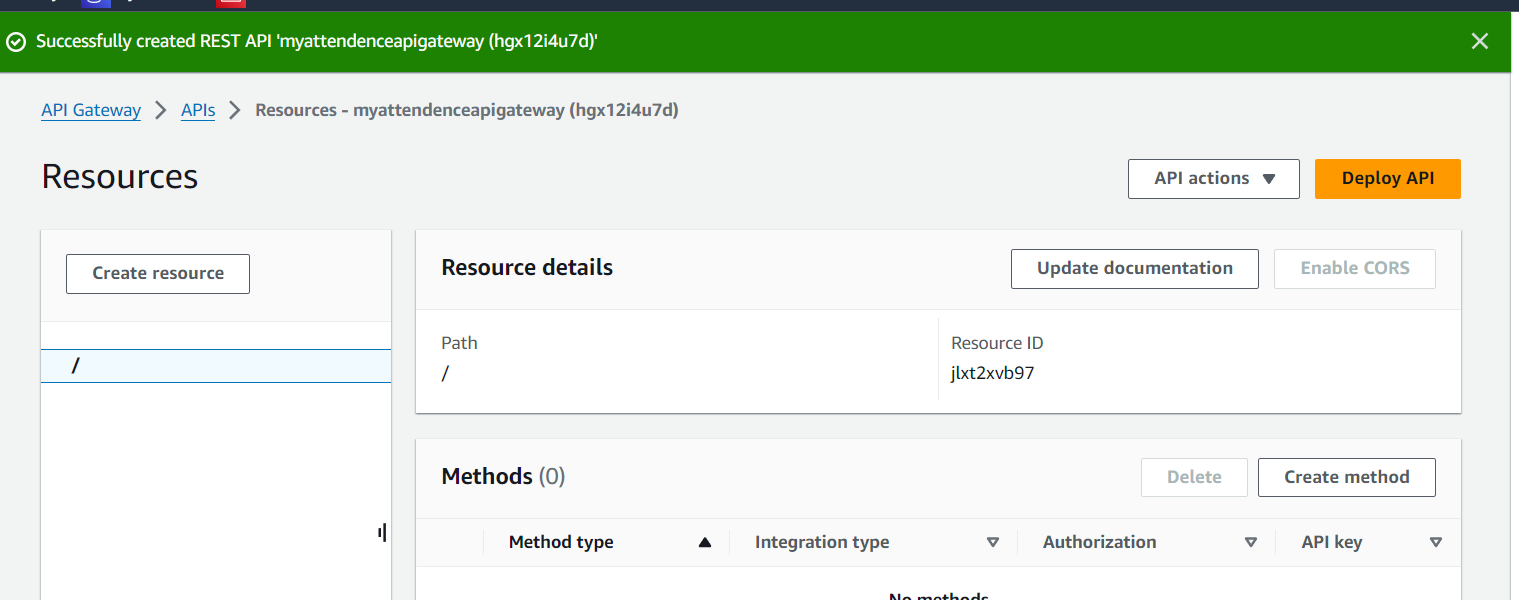




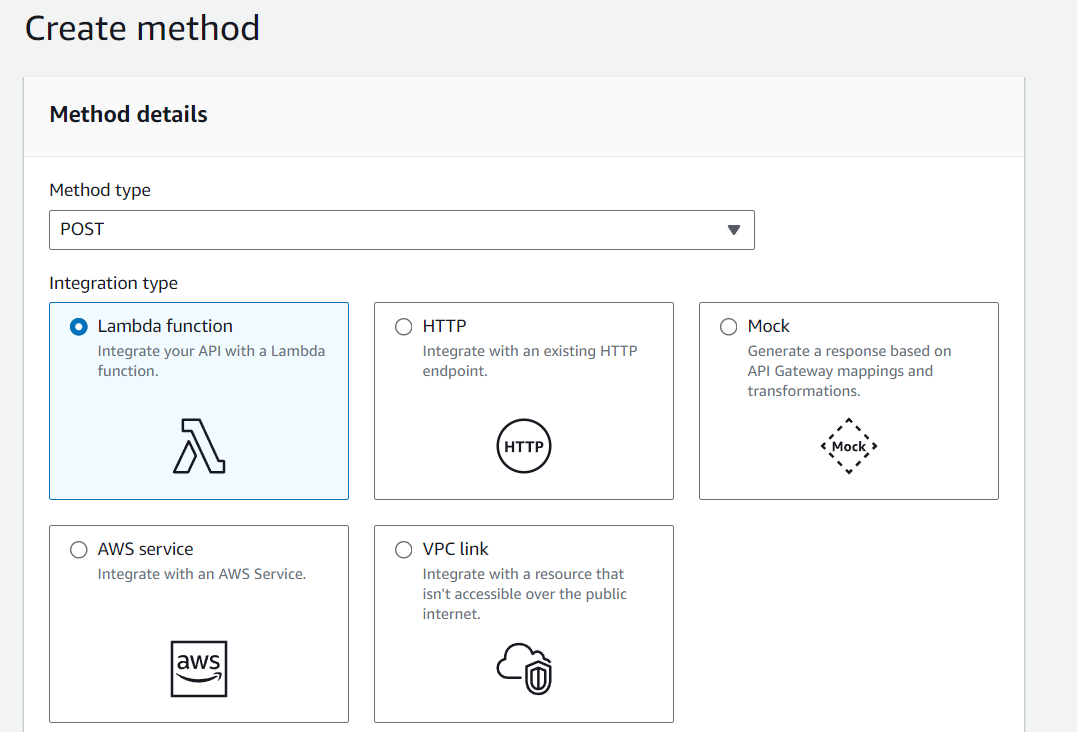
**Step 18 :** We assign a name to the gateway



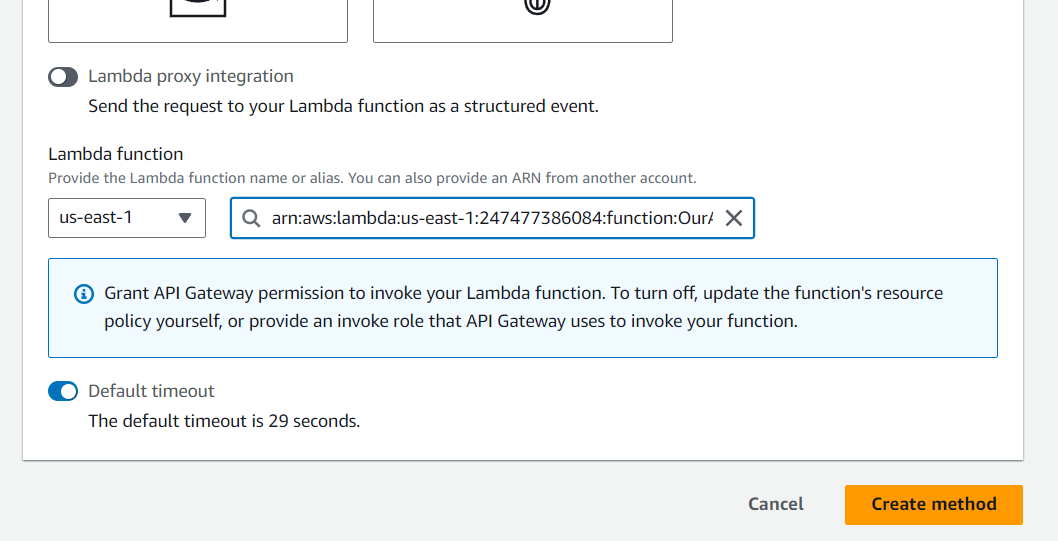
**Step 19 :** Under the ‘/’ Path we create a new method



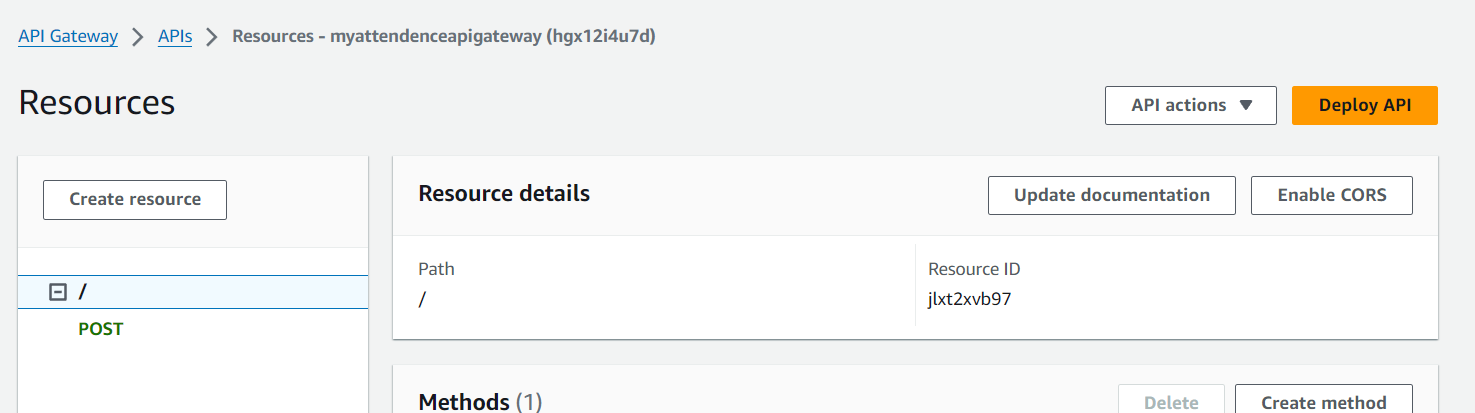
**Step 20:** We select the method POST & integration as lambda

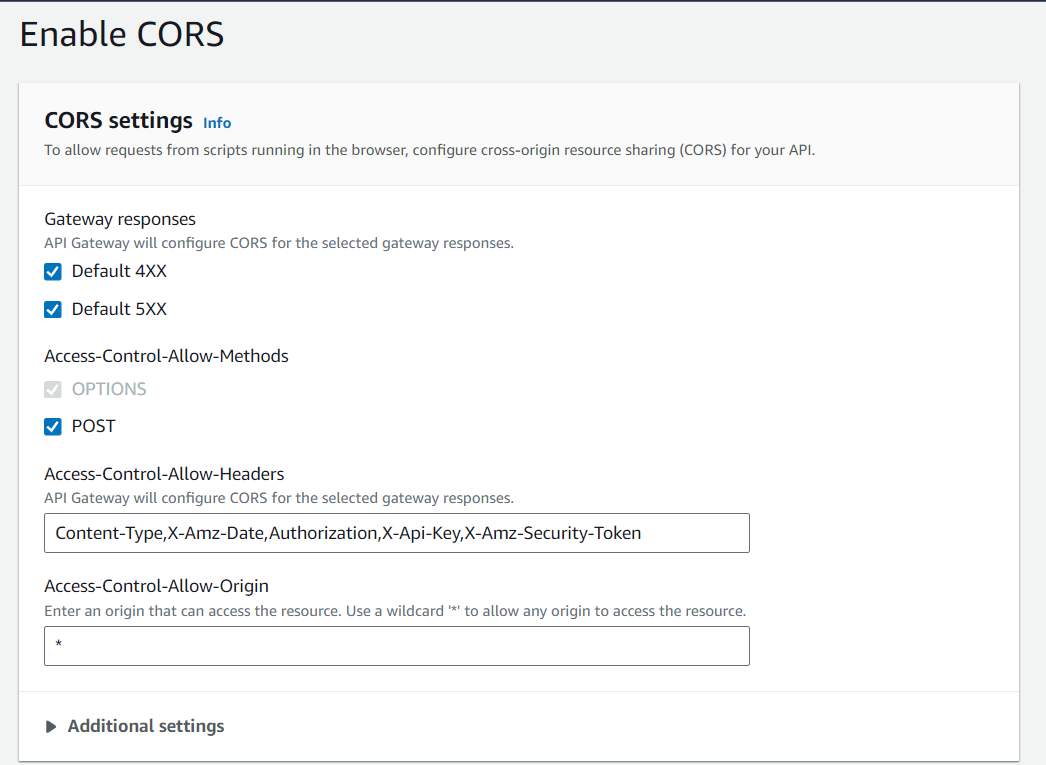


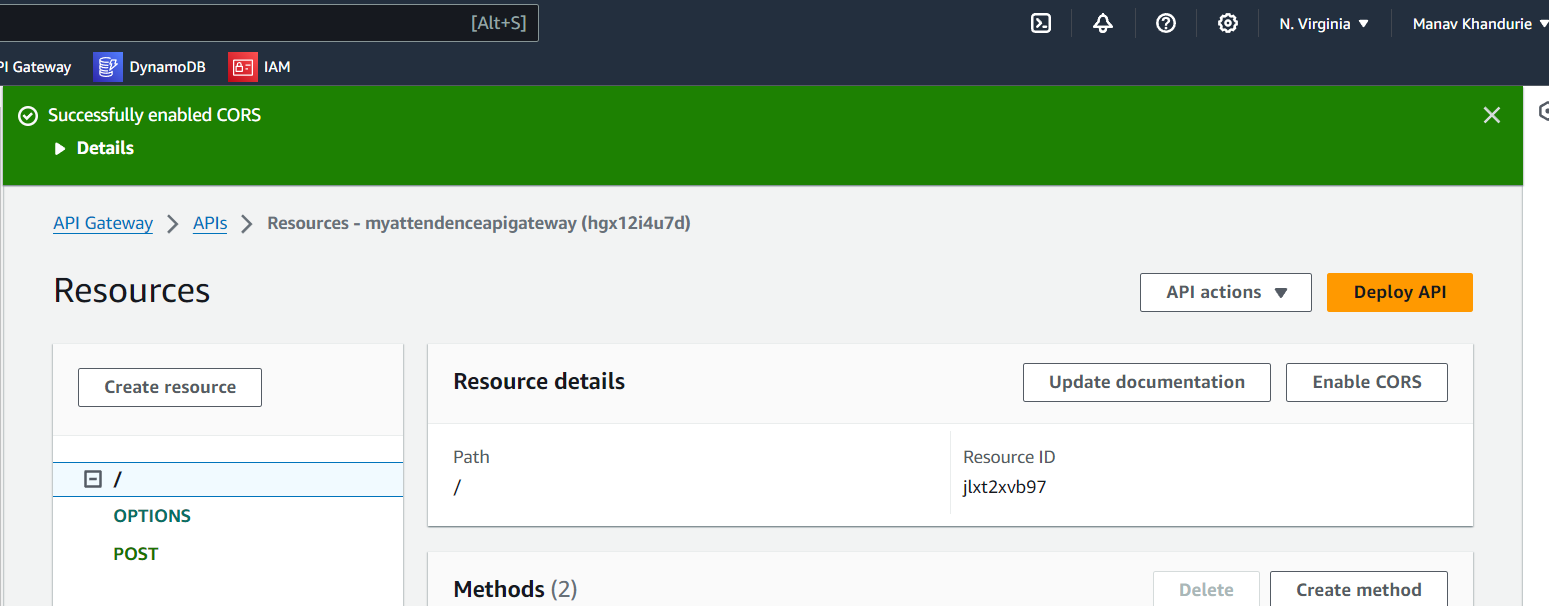
**Step 21 :** We also select lambda function we created earlier & click create



**Step 22 :** We would also enable CROS by selecting ‘/’ Path & select POST







**Step 23 :** Now to test the API Select POST Method under ‘/’ Path and goto Test and use the following testcase

{

  "operation": "insert",

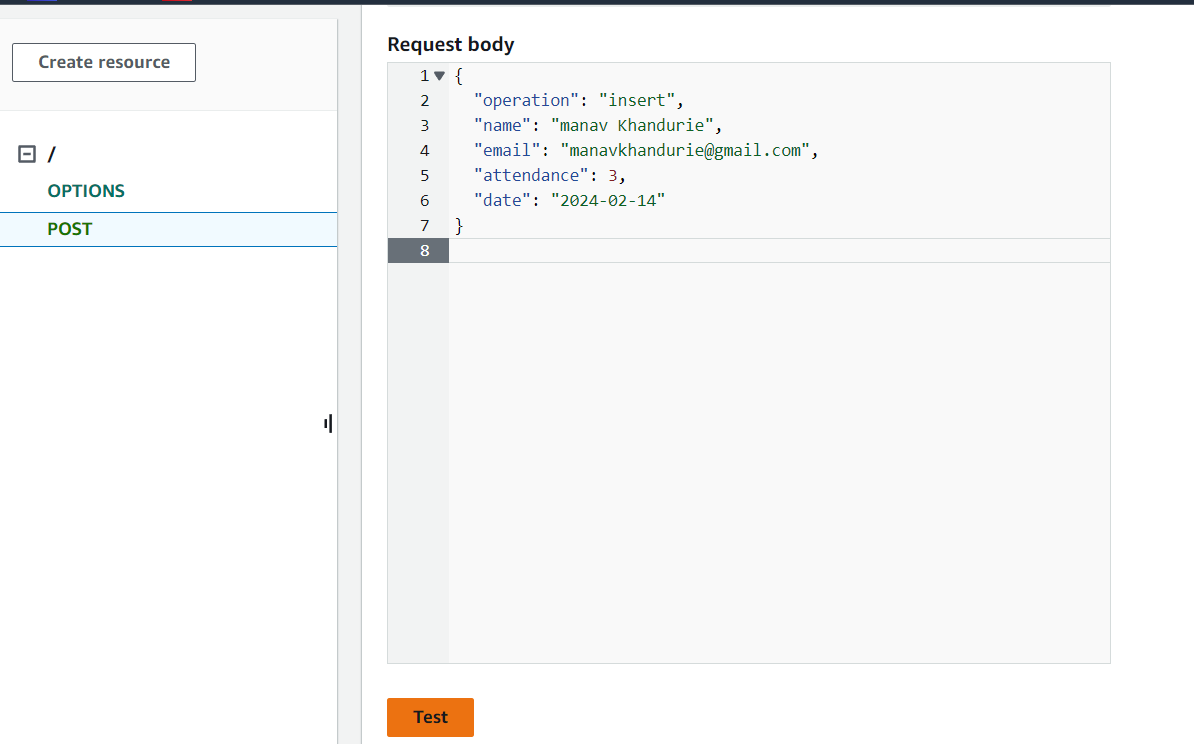
  "name": "manav Khandurie",

  "email": "manavkhandurie@gmail.com",

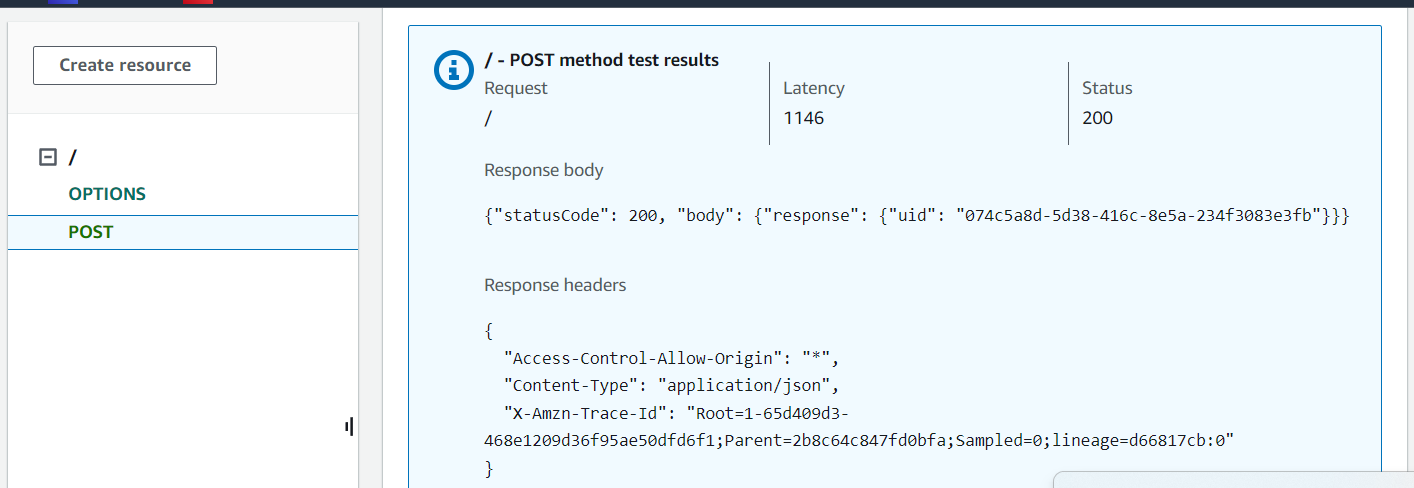
  "attendance": 3,

  "date": "2024-02-14"

}

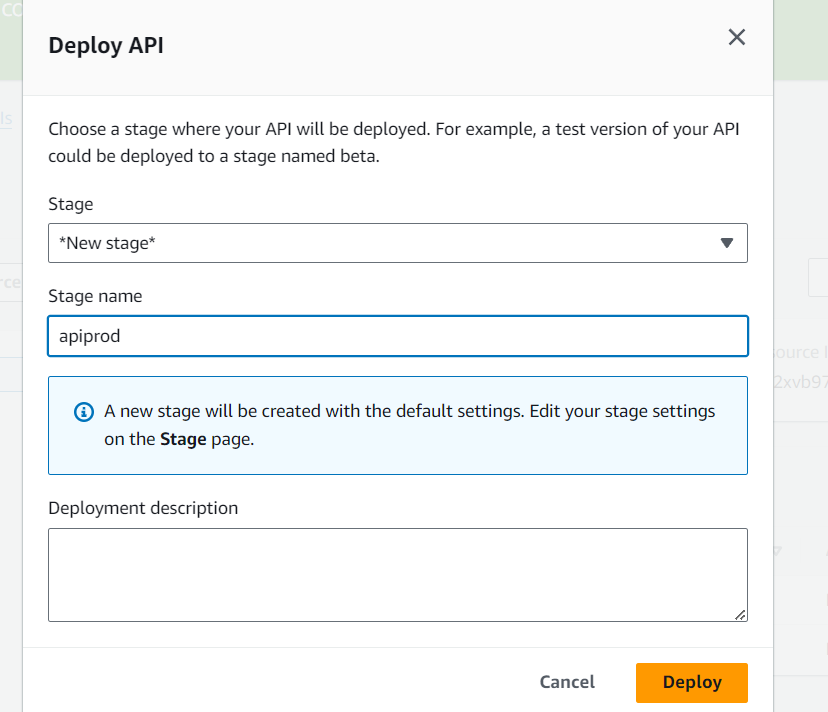


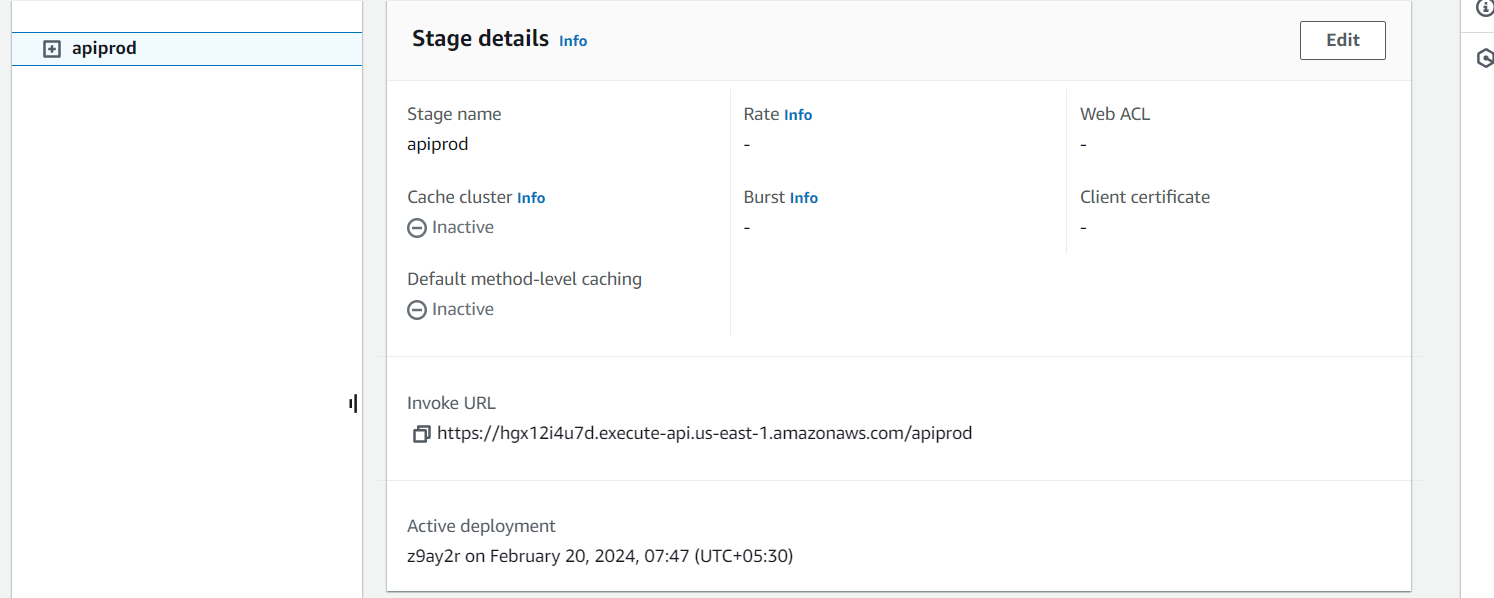
**Step 24:** Finally we click Test , and see the response



Note : 🡪 If you face error at these steps it is probably because of the configuration of post method under the resource ‘/’ go to steps 19 to 25 and retry the Part

**Step 25:** Finally Deploy API in a new Stage [ Where we find invoke Url]



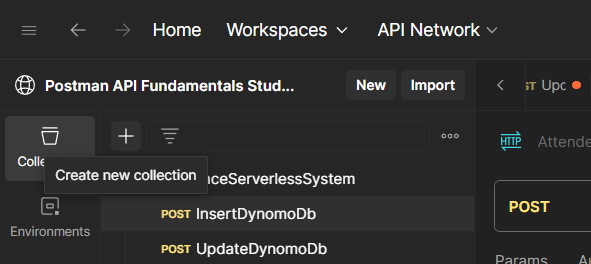


**Part 4 [Testing with POSTMAN]**

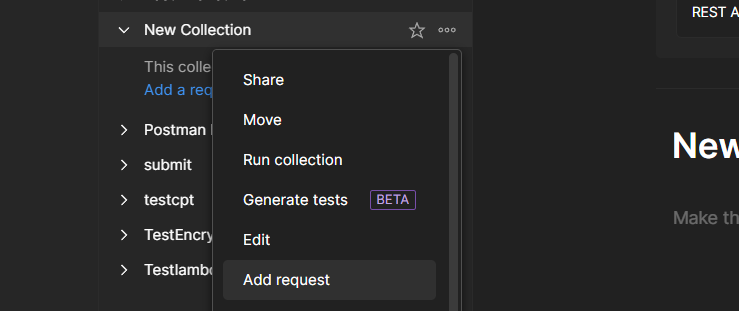
Note : Now this part is optional because you have already tested our API in the previous part however it is one of the best practises to test our API with Postman before deployment so if you wish to skip this part feel free to jump to part 5

***Perquisites:*** - Install Postman from <https://www.postman.com/downloads/>

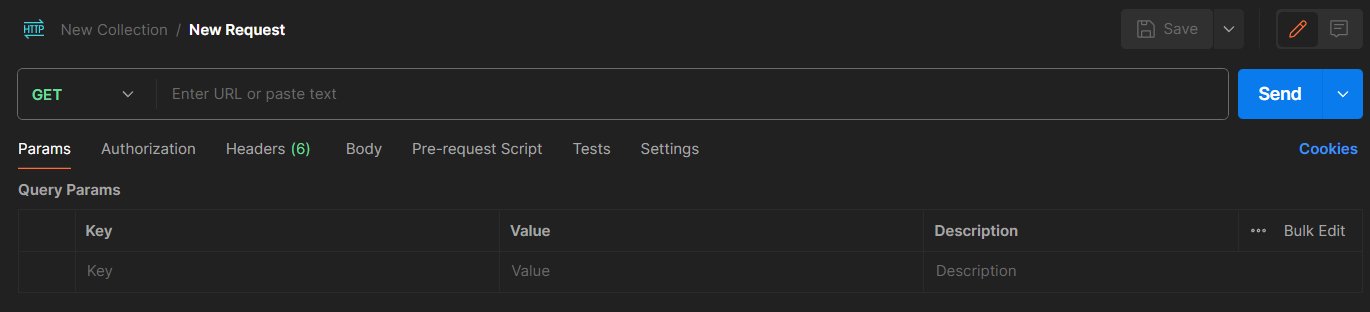
**Step 26:** Open PostMan 🡪 Create Collection

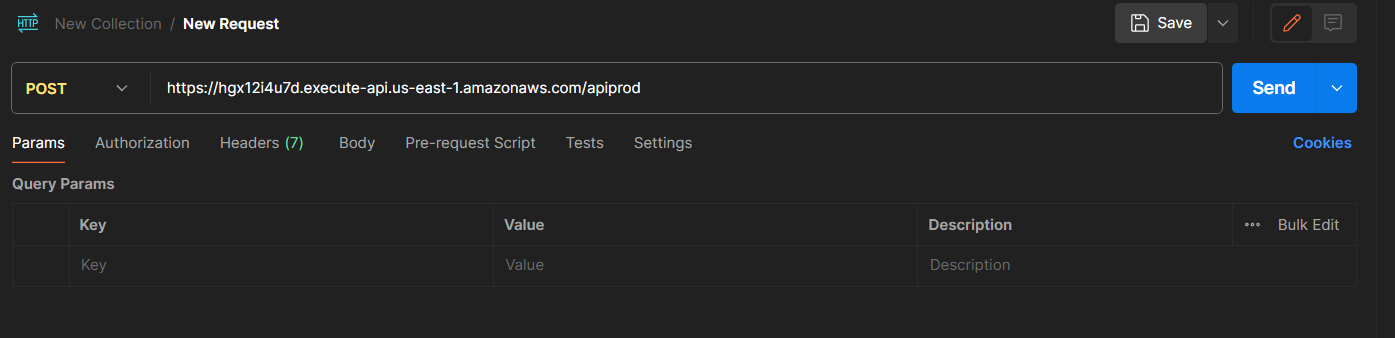


**Step 27:** Create a new Blank Collection and goto New Request

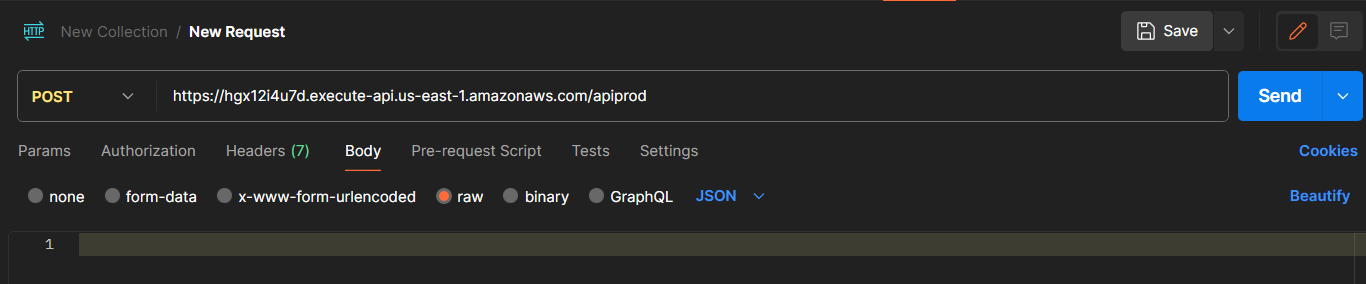


**Step 28:** In the new Request , select the POST option paste the invoke URL of API Gateway





**Step 29:** Goto Body and pate the following body, under RAW, using JSON



{

  "operation": "insert",

  "name": "manav Khandurie",

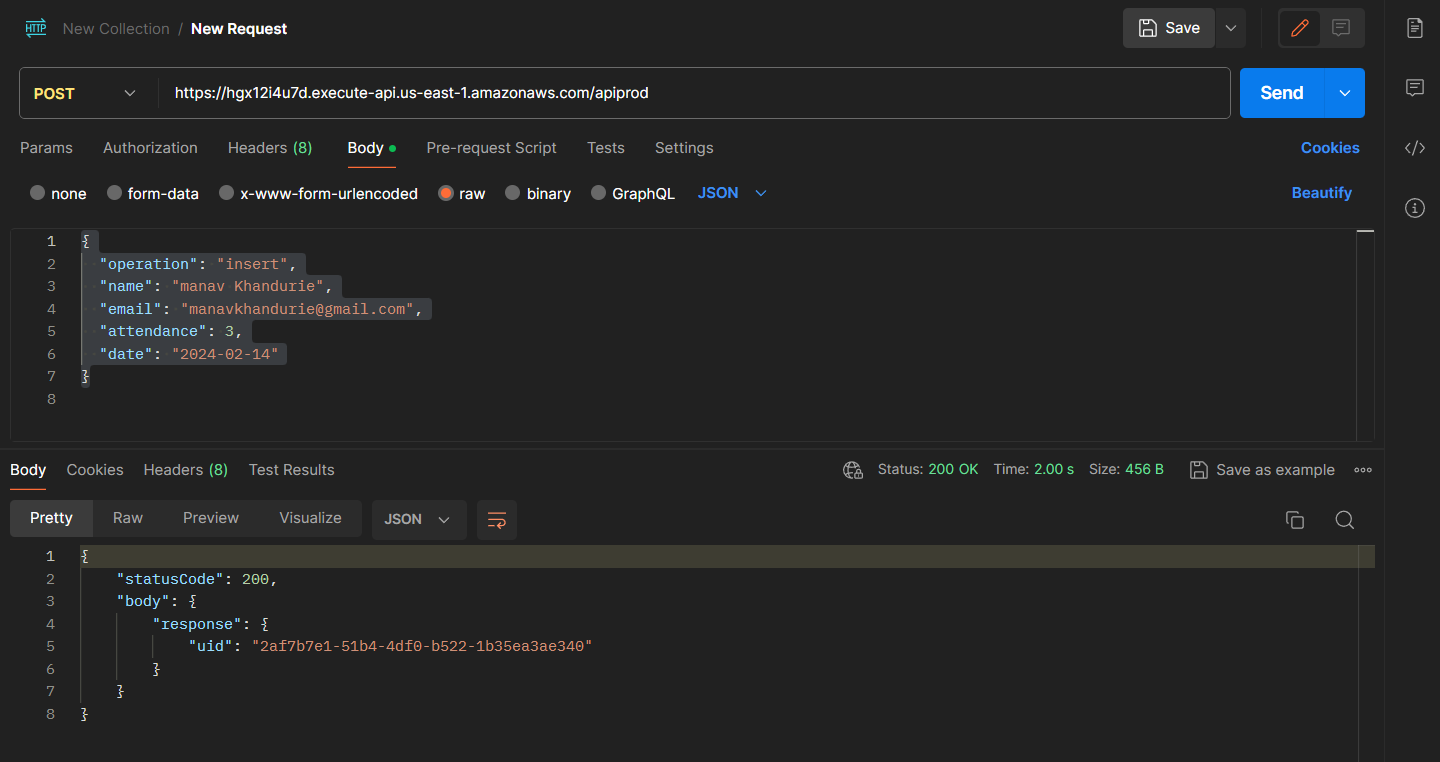
  "email": "manavkhandurie@gmail.com",

  "attendance": 3,

  "date": "2024-02-14"

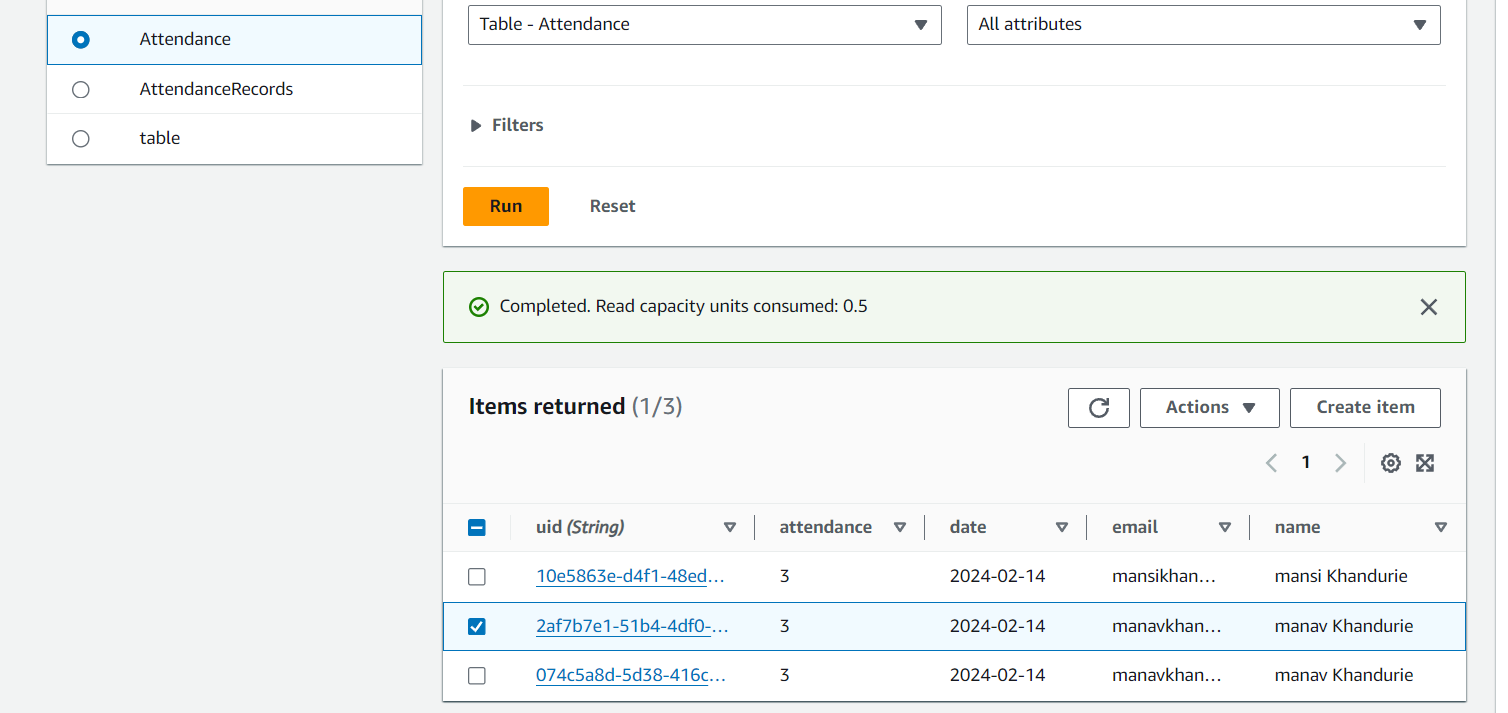
}

**Step 30:** Click on Send , and we should get a 200 Status Response



Also we will check at the db side ,

We can see that item under 2af7b7e1-51b4-4df0-b522-1b35ea3ae340 uid is added

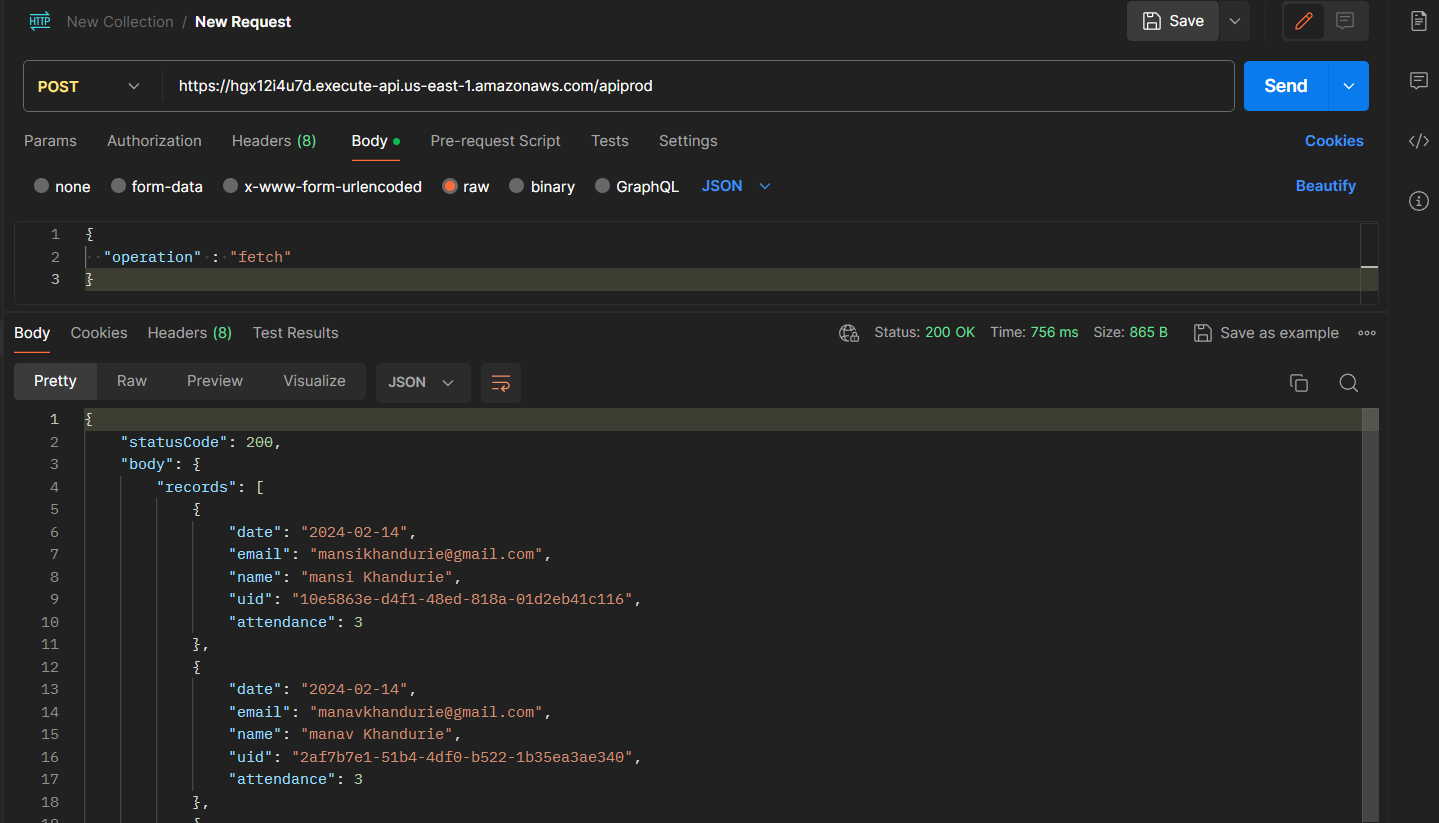


**Step 31:** Subsequently we also check fetch & delete [ Use your UID in del ]

{

  "operation" : "fetch"

}

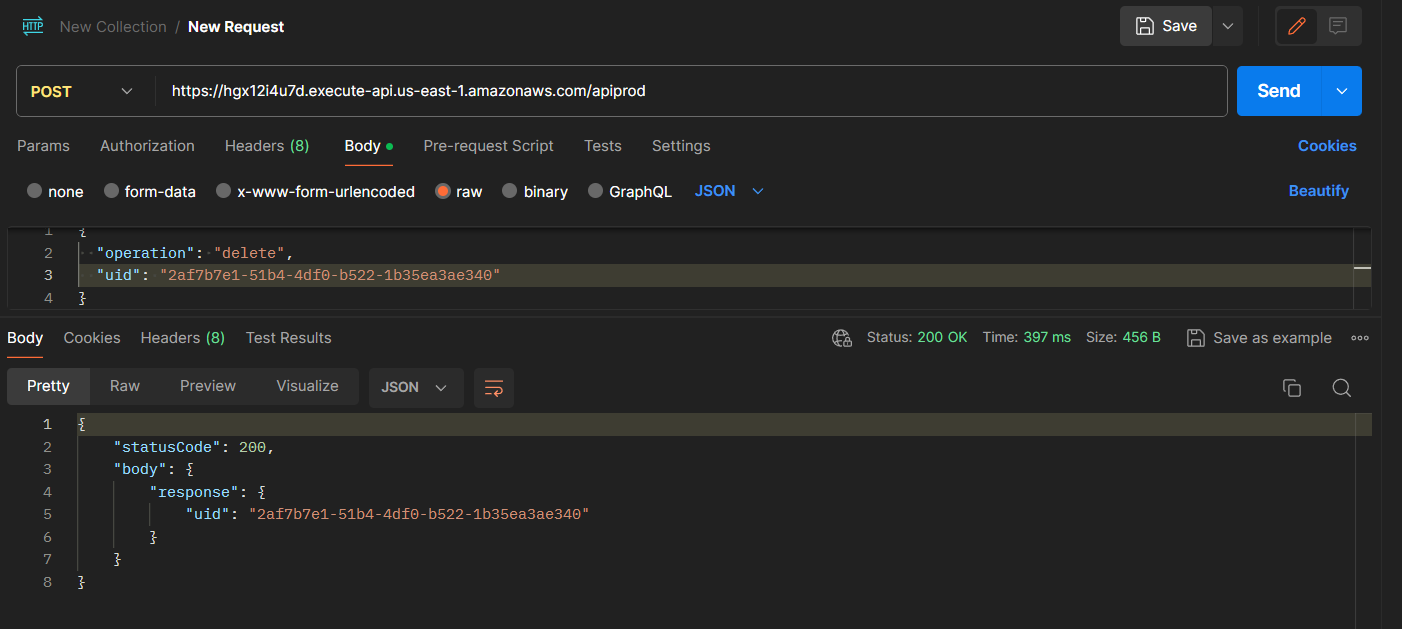


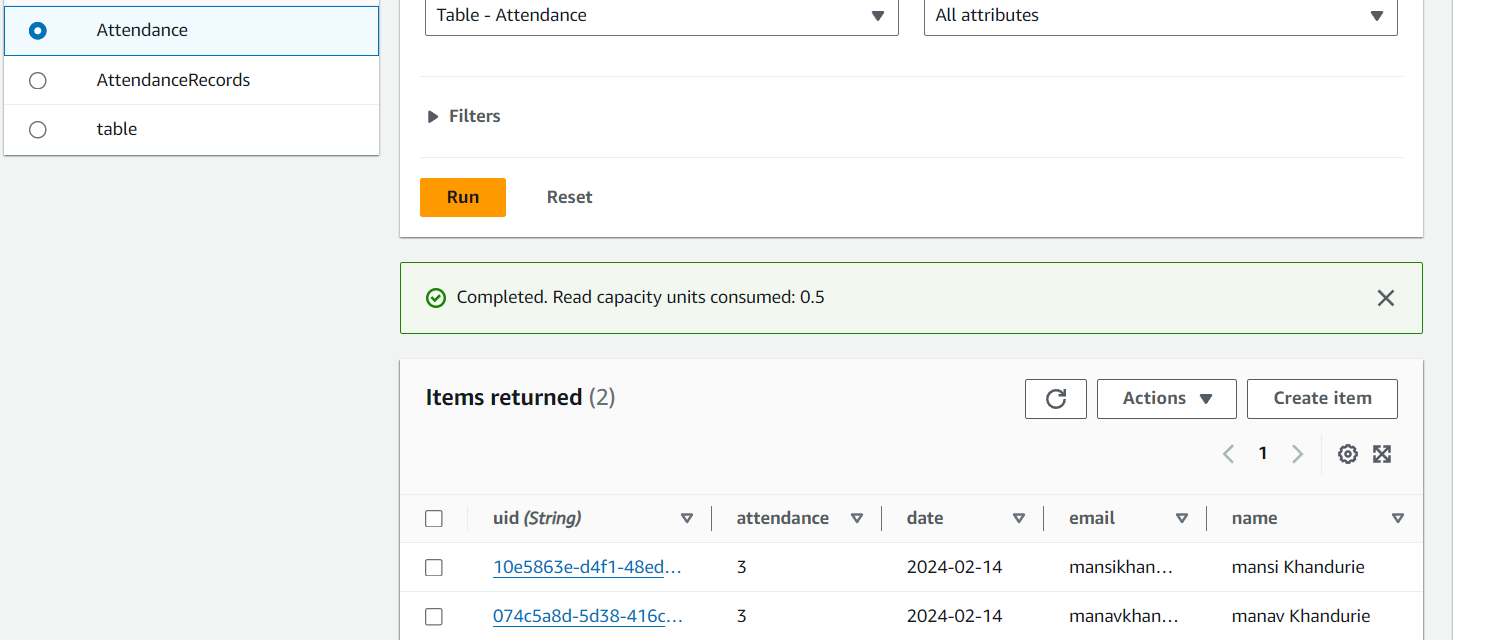
{

  "operation": "delete",

  "uid": "e29815aa-6688-41b6-8ce6-4ea614c22e10"

}

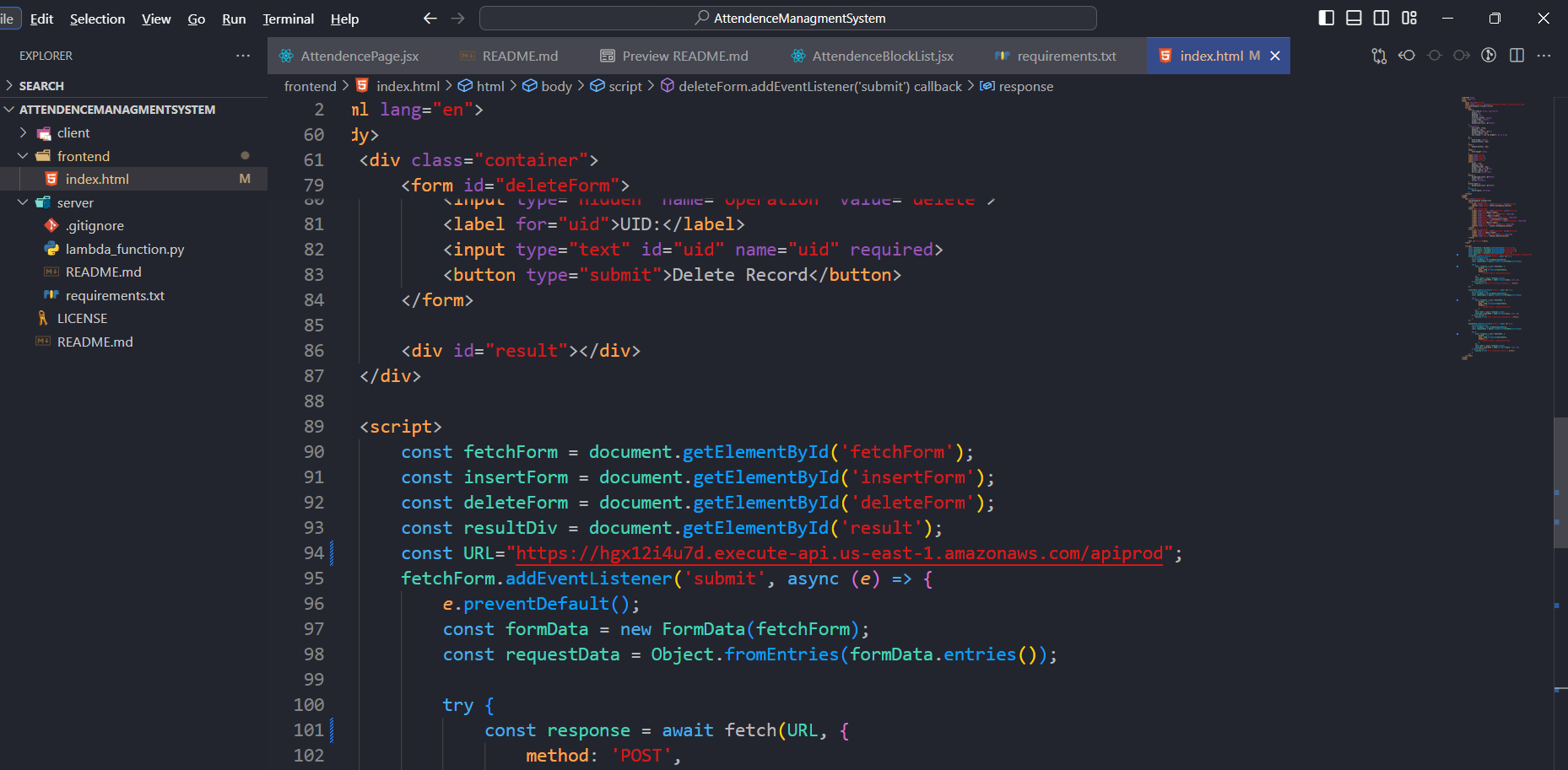




[ ID with 2af7b7e1-51b4-4df0-b522-1b35ea3ae340 deleted ]

**Part 5 [Frontend Deployment]**

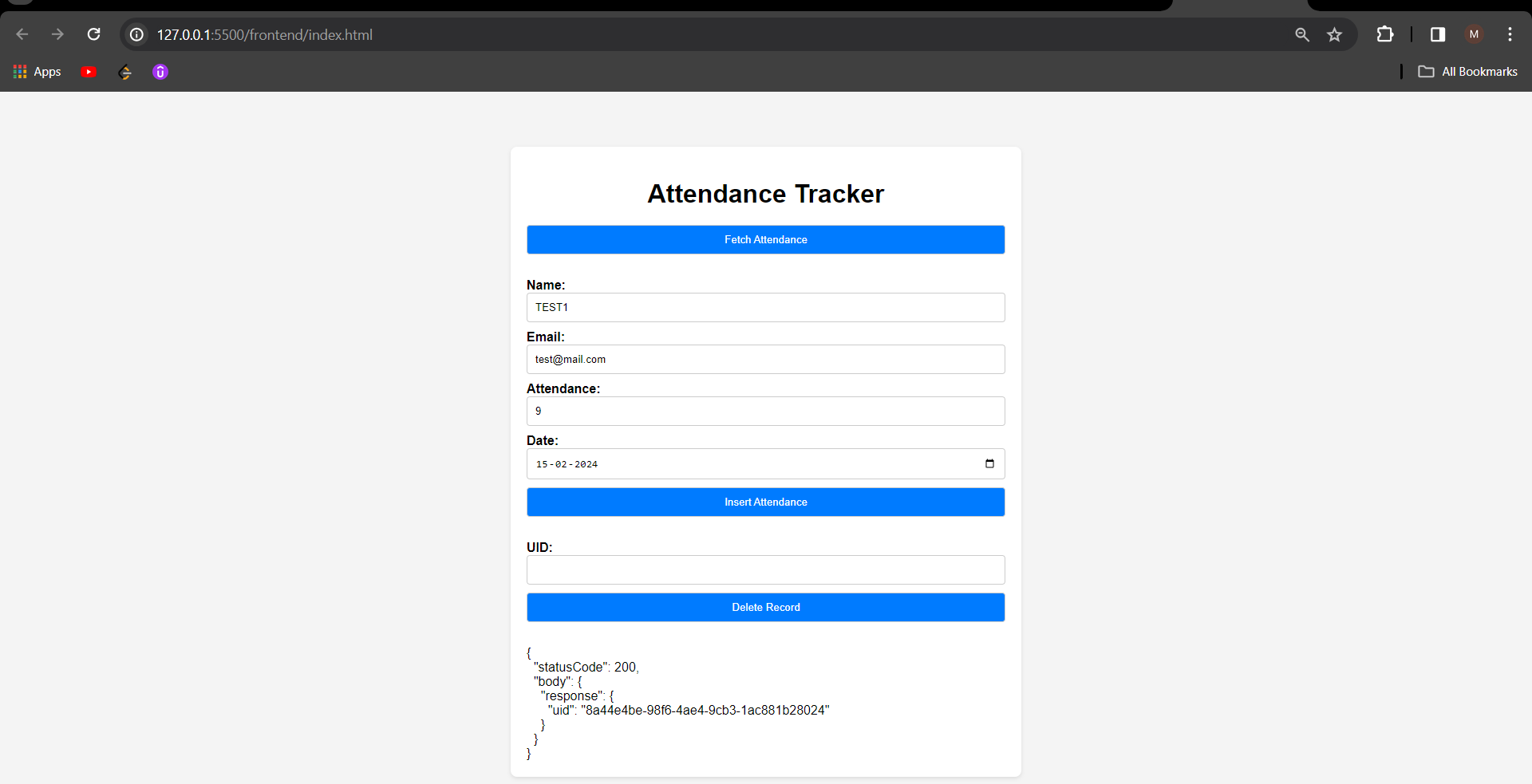
**Step 32:** We fetch the frontend from Source Code🡪 Frontend & view using VS Code

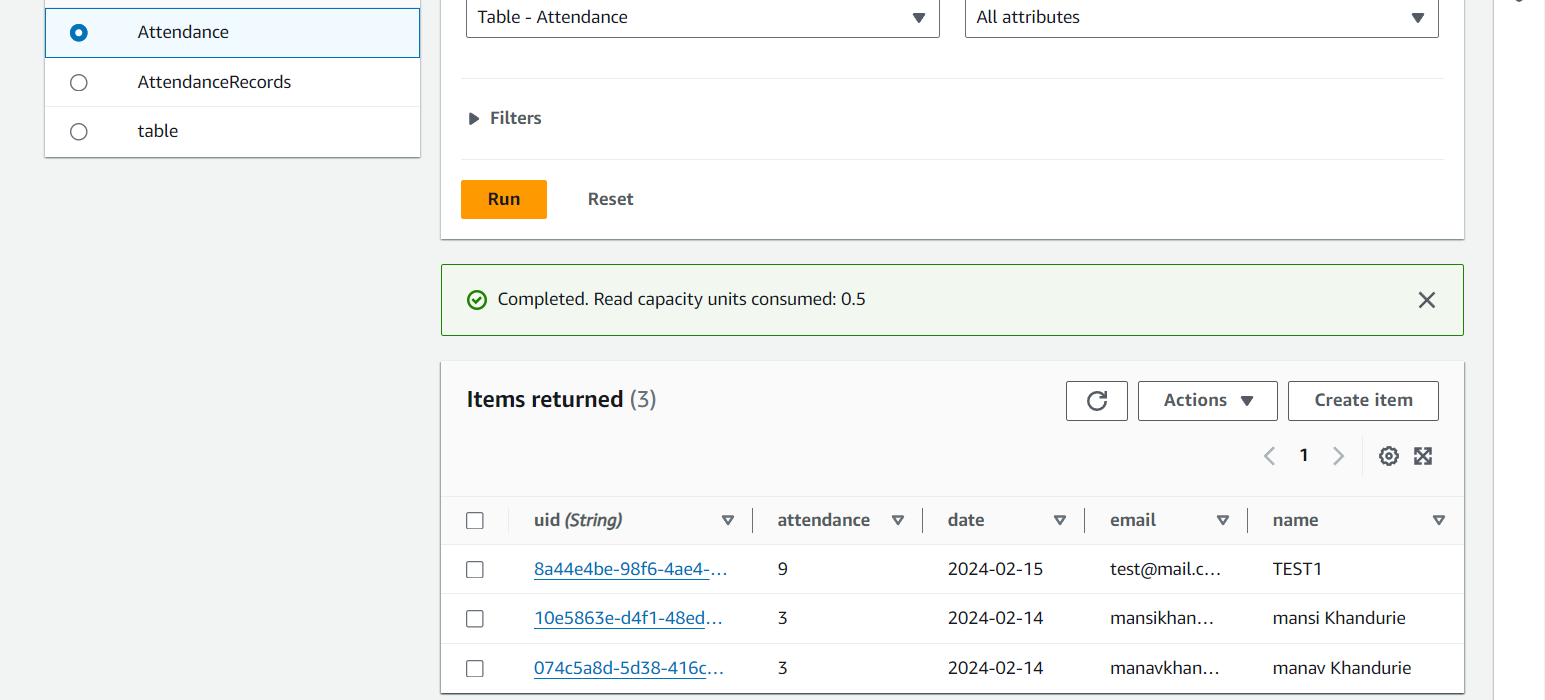


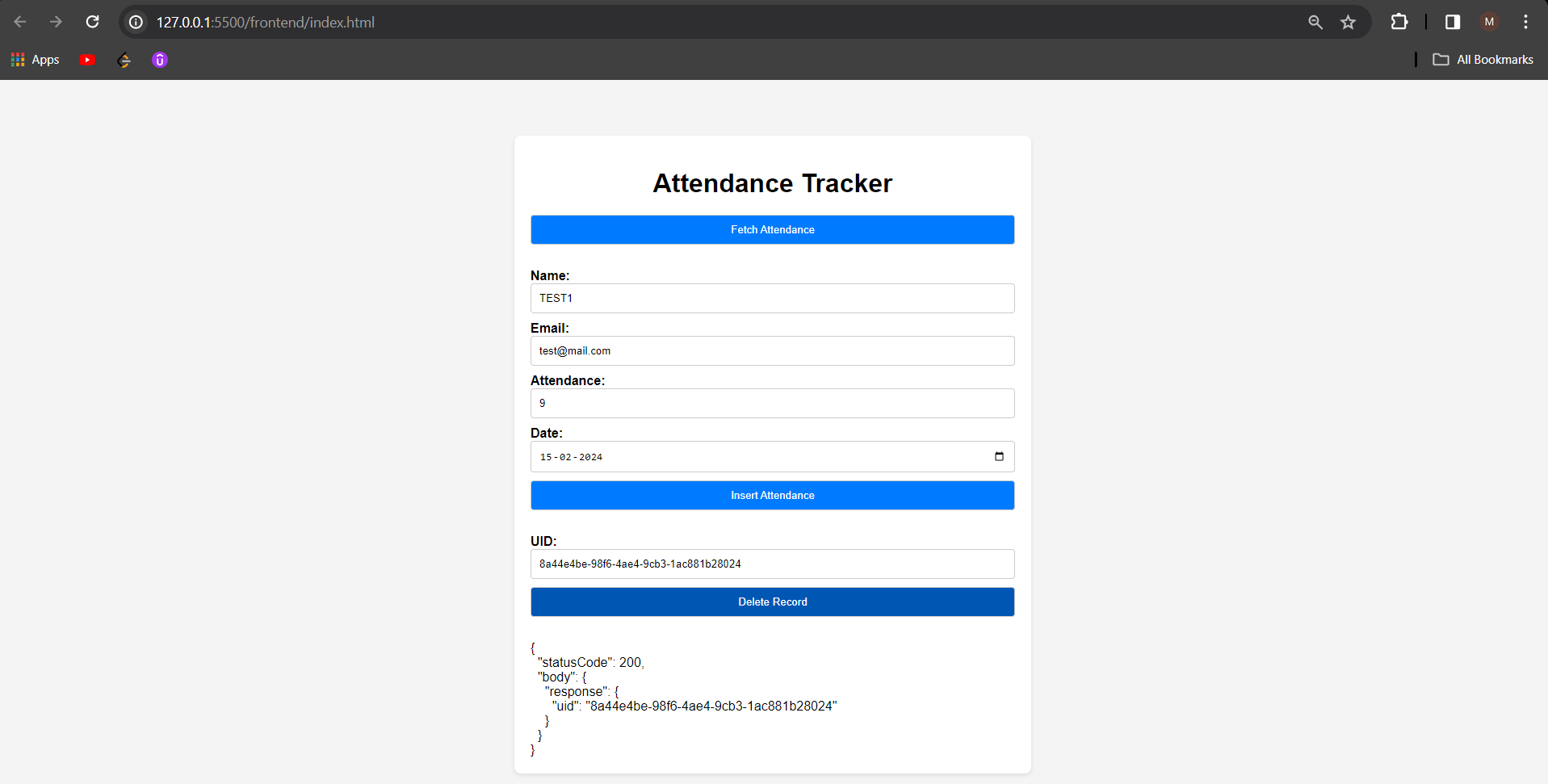
**Step 33:** In the source code change the URL variable to your API

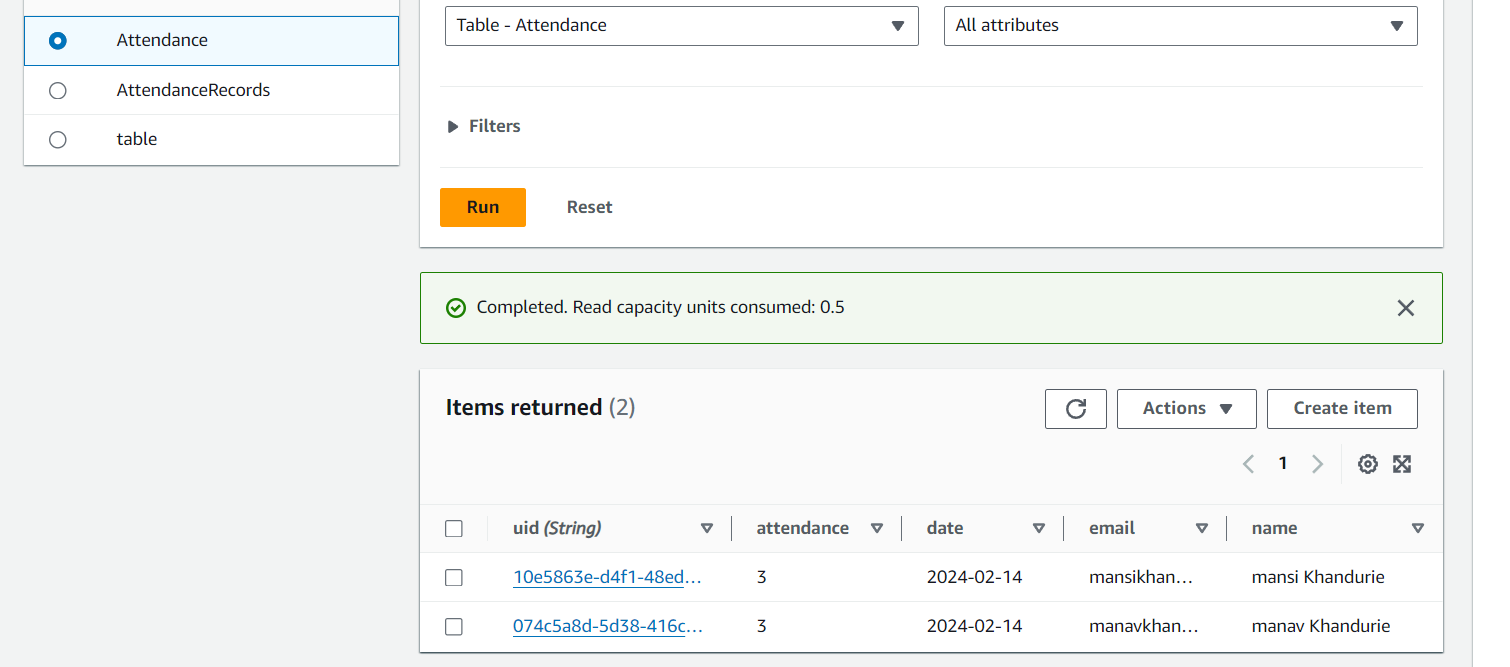
**Step 34:** Test the site locally , use live server or simply open the html page using chrome

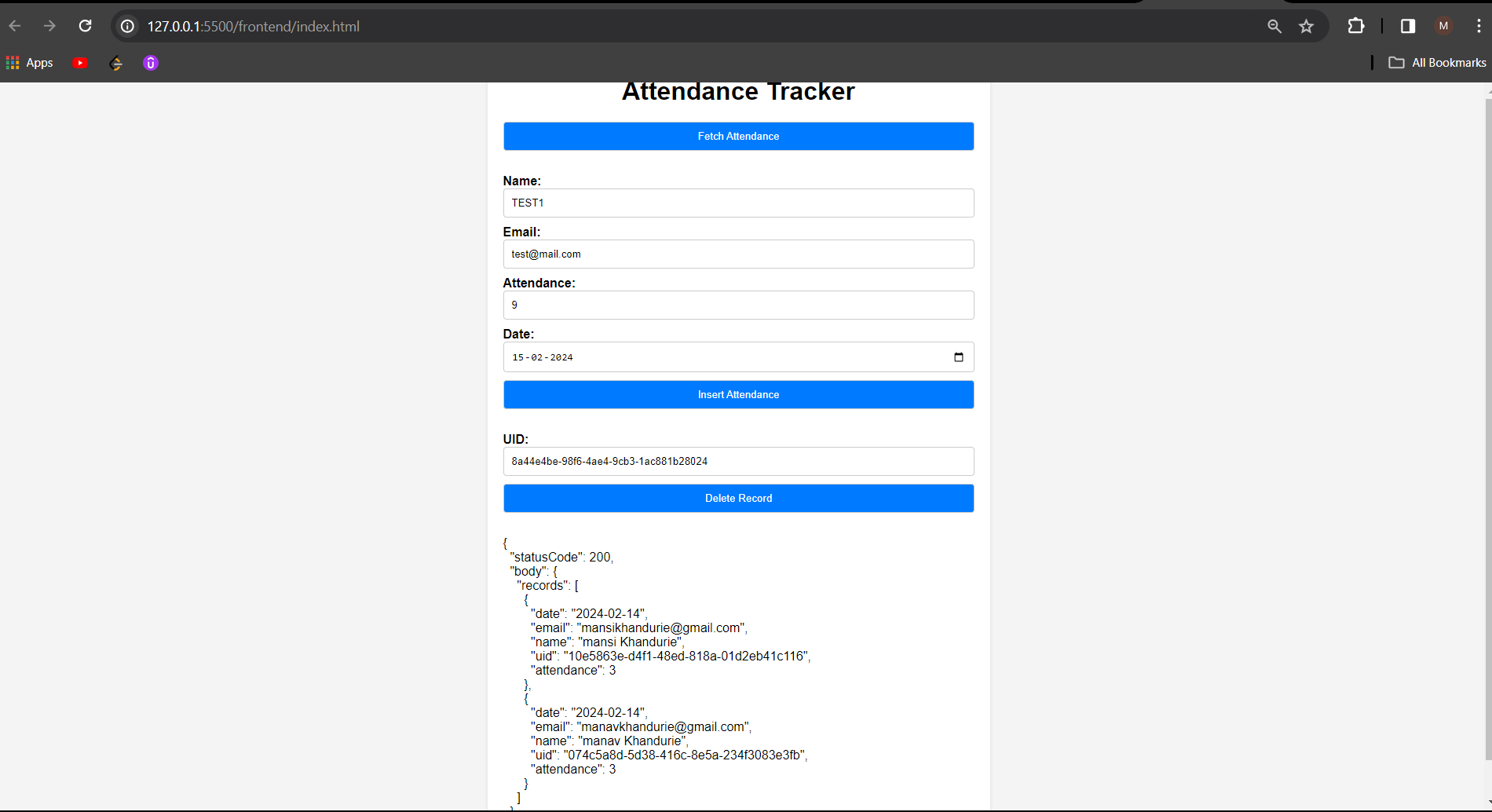
Here we check all Functionalities insert delete and fetch



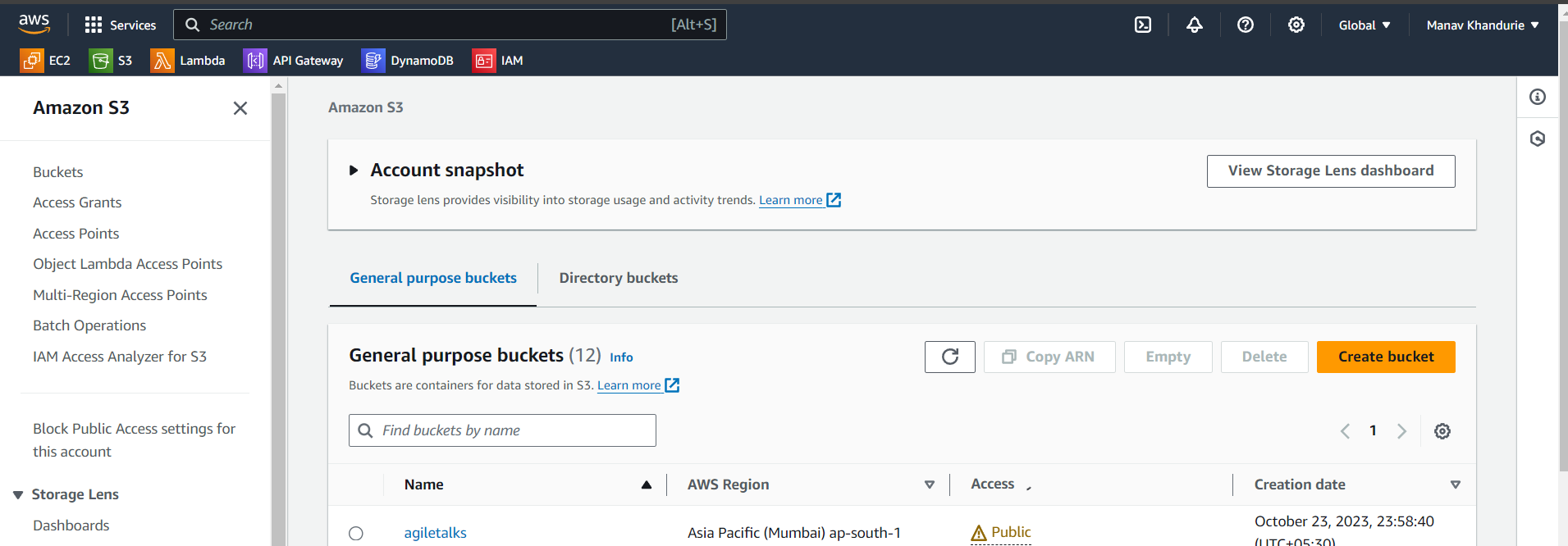




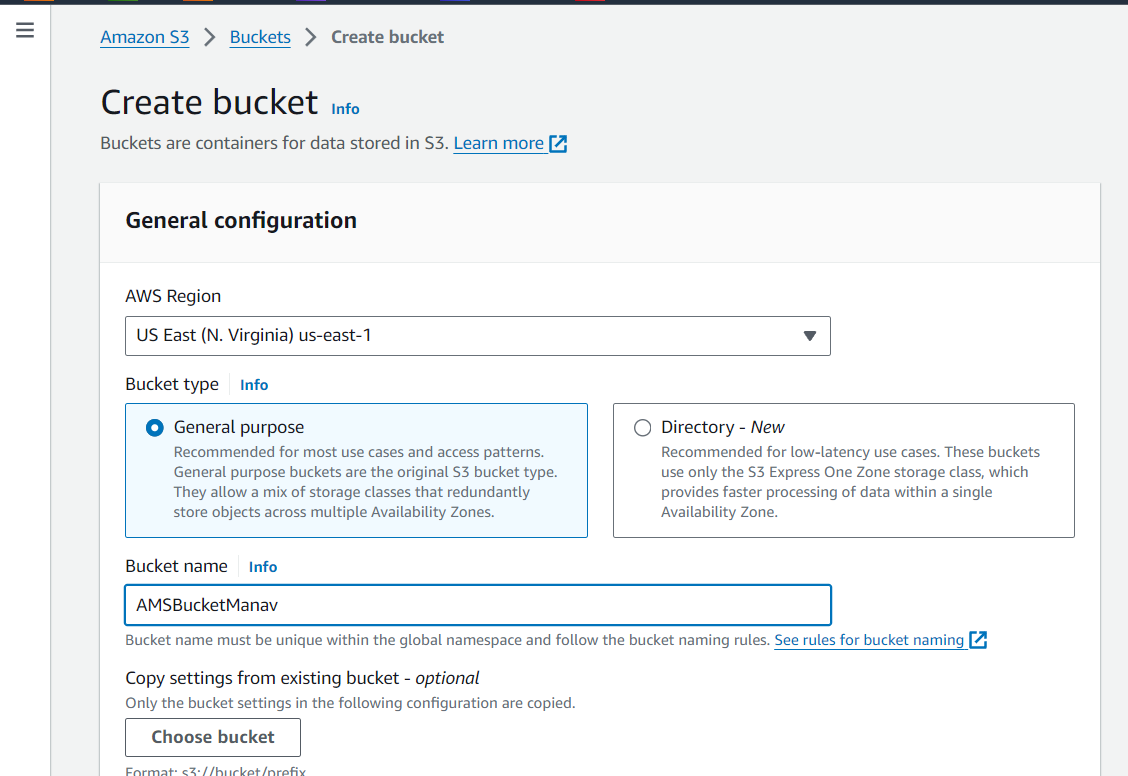




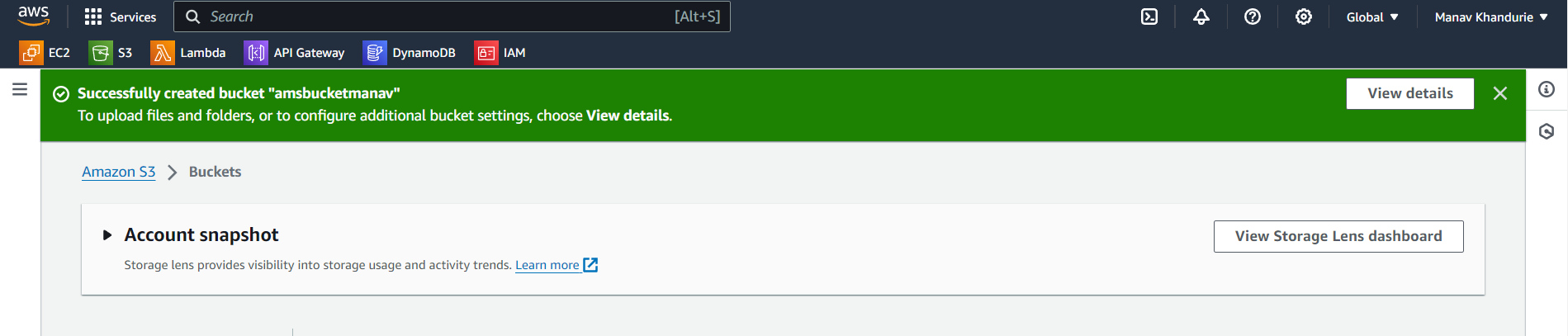
**Step 35:** Now we goto S3 , and create a bucket



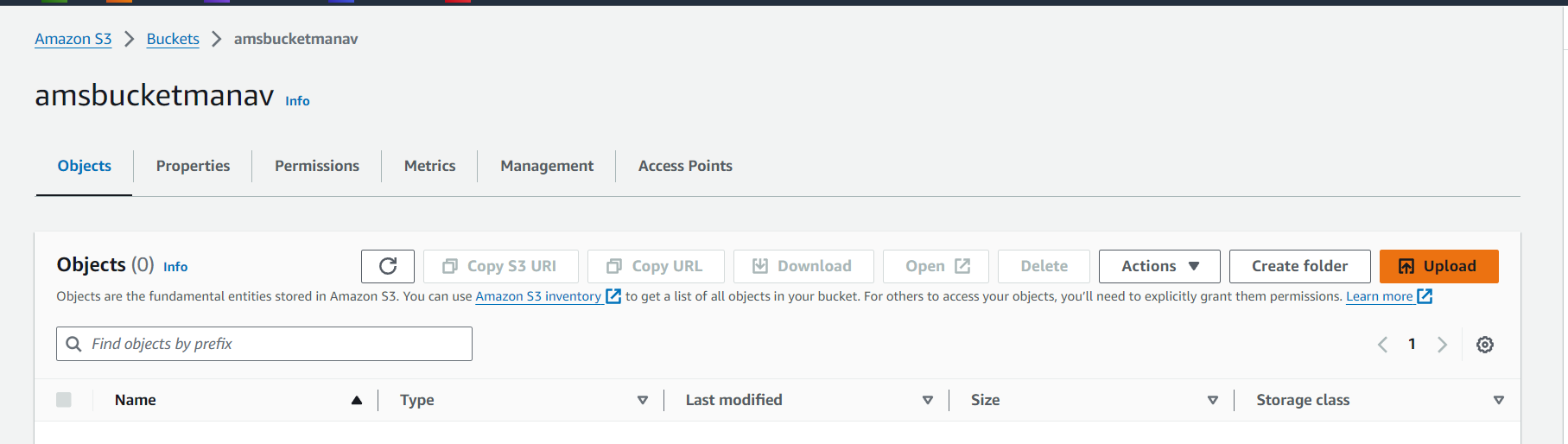
**Step 36:** We give a unique name to the bucket , unblock the public acess & create

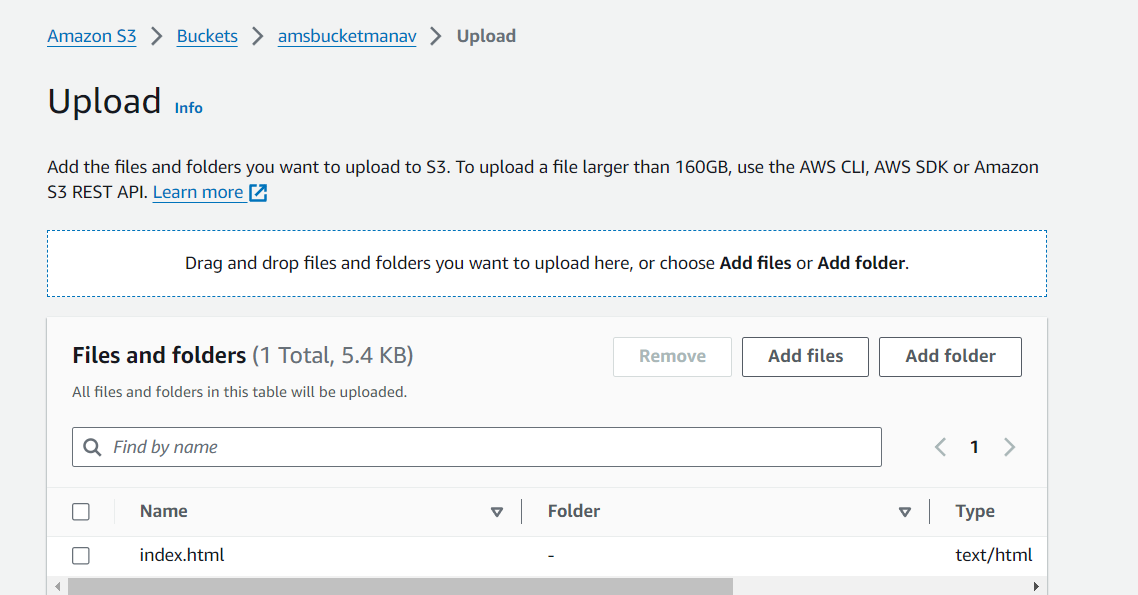


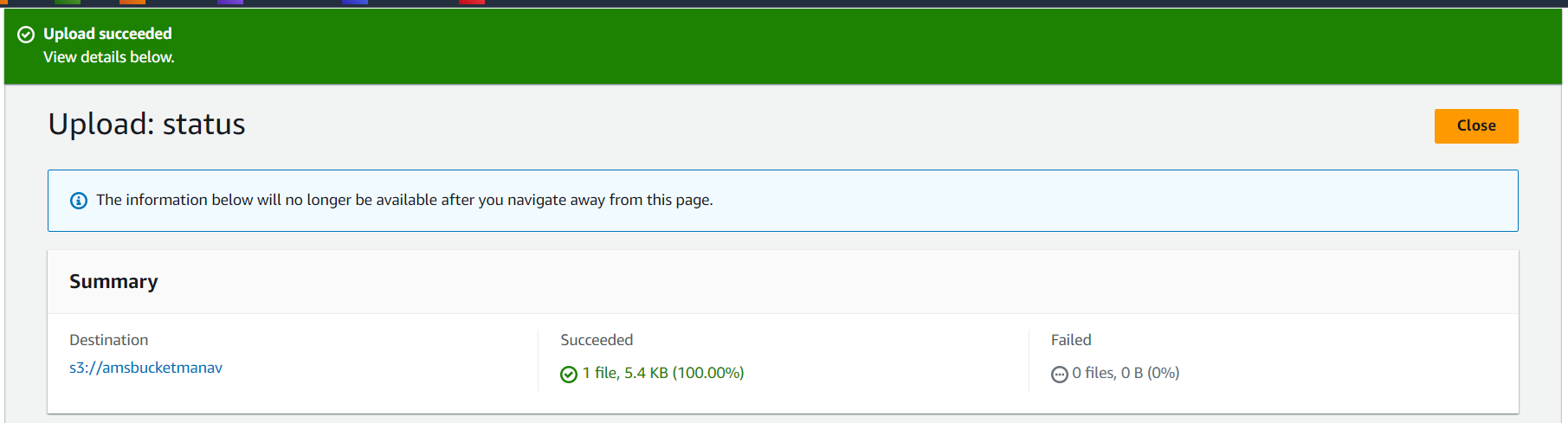




**Step 37:** We goto the bucket we created and add the HTML page we created before , goto 🡪 Upload , Upload file , select the File and Upload









**Step 38:**  Goto Permission 🡪 Bucket Policy 🡪 Edit and use the following Policy & save. Note do change the Resouce to your Bucket ARN shown

“<Your-resource-arn>/\*”

{

"Version": "2012-10-17",

"Statement": [

{

"Effect": "Allow",

"Principal": "\*",

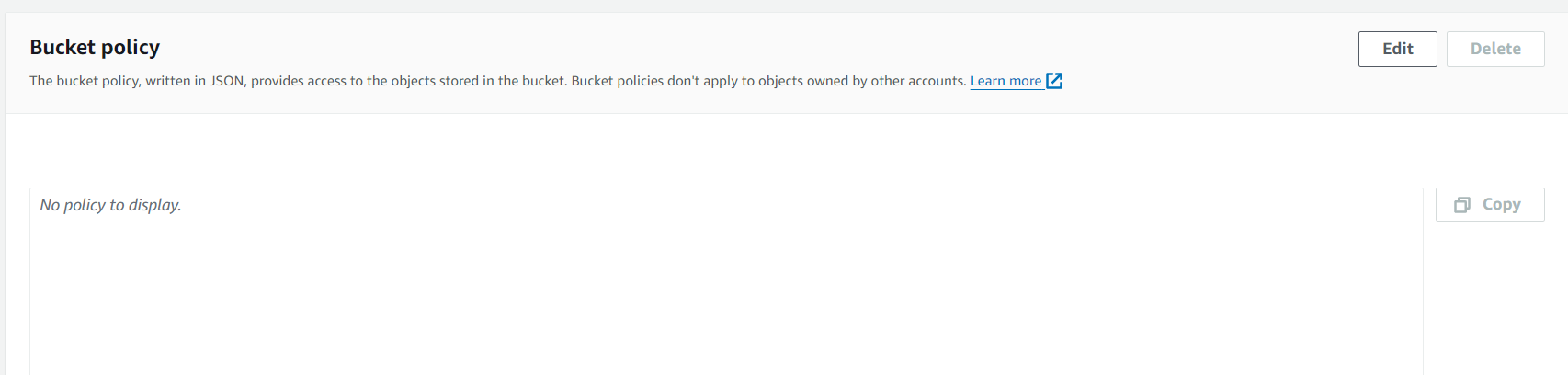
"Action": "s3:GetObject",

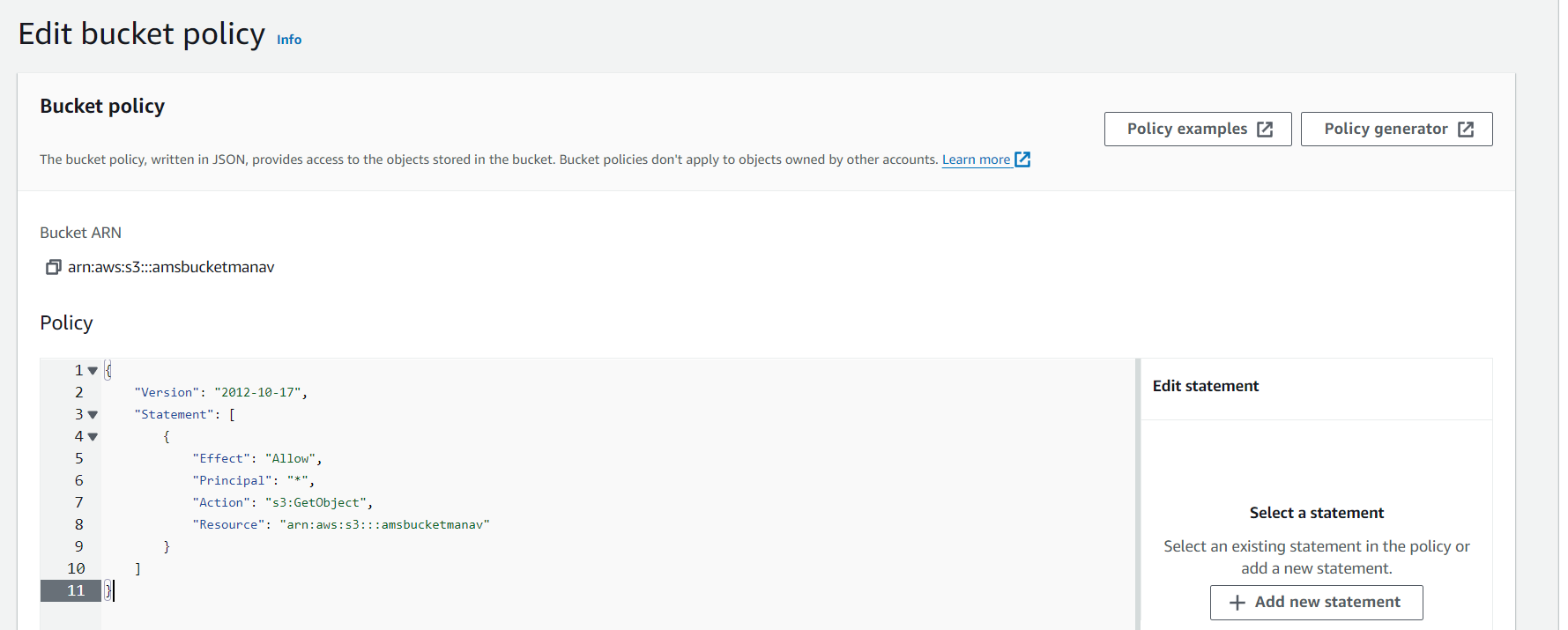
"Resource": "arn:aws:s3:::agiletalks/\*"

}

]

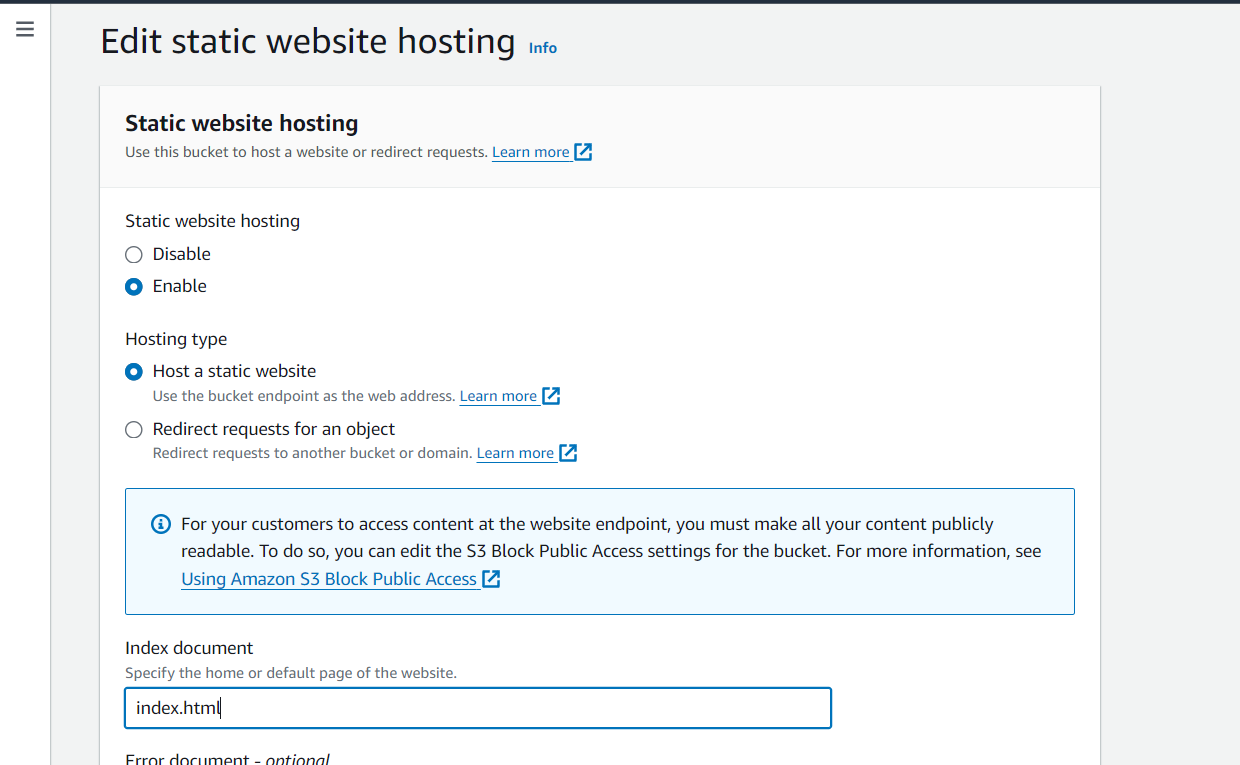
}

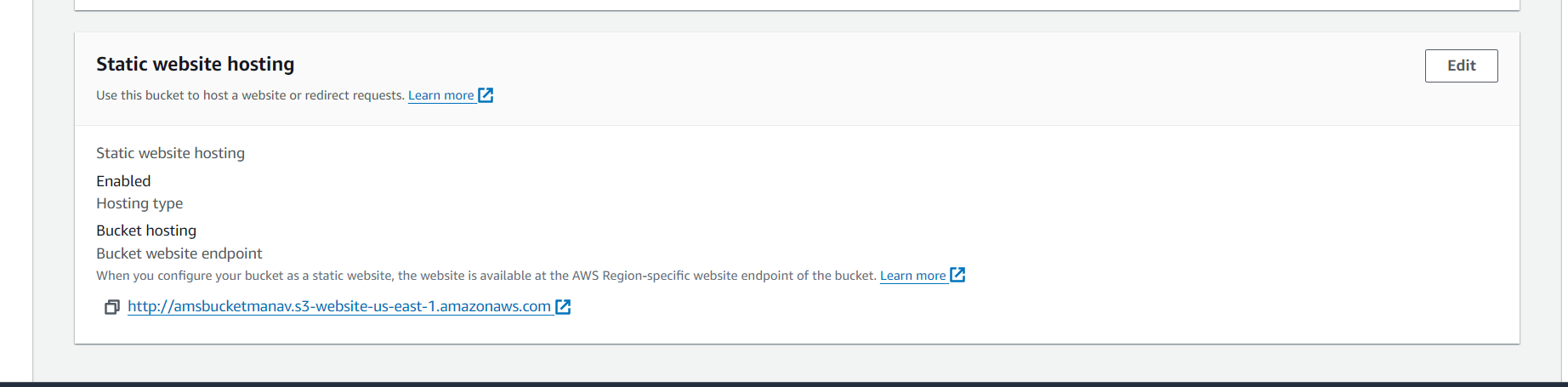






**Step 39:** Now goto Staic Site hosting under Properties and enable it , also set the index document as index.html [ P.S. This is your html file’s name that you uploaded] , we ll get a bucket endpoint url





**Step 40:** Open the URL in the incognito mode , well see the application test it out as you wish

