Serverless Web Application on AWS

A MINI PROJECT REPORT

Submitted by

G JAHNAVI [RA2011003011060]
SHIVAGURUPRIYAA G [RA2011003011080]
PRUDHIVI SAI GANESH [RA2011003011057]

Under the guidance of

Dr. V. Deepan Chakravarthy
(Associate Professor, Department of Computing Technologies)

In partial satisfaction of the requirements for the degree of

BACHELOR OF TECHNOLOGY
in

COMPUTER SCIENCE & ENGINEERING

SCHOOL OF COMPUTING
COLLEGE OF ENGINEERING AND TECHNOLOGY
SRM INSTITUTE OF SCIENCE AND TECHNOLOGY
KATTANKULATHUR - 603203

MAY 2023

COLLEGE OF ENGINEERING & TECHNOLOGY

SRM INSTITUTE OF SCIENCE & TECHNOLOGY

S.R.M. NAGAR, KATTANKULATHUR – 603 203

BONAFIDE CERTIFICATE

Certified that this project report "Serverless web application" is the bonafide work of "Shivagurupriyaa G (RA2011003011080), Prudhivi Sai Ganesh (RA2011003011057), G Jahnavi (RA2011003011060)" of III Year/VI Sem B.tech(CSE) who carried out the mini project work under my supervision for the course 18CSE316J - Essentials in Cloud and Devops in SRM Institute of Science and Technology during the academic year 2022-2023(Even sem).

SIGNATURE

Dr.V.Deepan Chakravarthy
Associate Professor, Department of Computing Technologies
School of computing

ABSTRACT

This serverless web application built on AWS is designed to allow users to perform CRUD operations on a DynamoDB database through a RESTful API provided by Amazon API Gateway. The front-end of the application is built using React, while the back-end is built using AWS Lambda, DynamoDB, and S3.

The application is deployed using AWS CloudFormation and AWS SAM, providing a scalable and secure infrastructure that can handle a large number of concurrent users. The system architecture is designed to be flexible and can be customized to meet the specific needs of the application.

The Agile methodology is used for the development of the application, ensuring that the development process is collaborative, flexible, and iterative. The continuous integration and deployment pipeline automates the testing, building, and deploying of the application to AWS, ensuring that the code is always in a deployable state.

Overall, this serverless web application built on AWS provides a flexible and scalable solution for building modern web applications that can handle a large number of concurrent users while minimizing the cost and complexity of managing the underlying infrastructure.

TABLE OF CONTENTS

LIST OF FIGURES

ABBREVIATIONS

AWS	Amazon Web Services
S3	Simple Storage Service
CRUD	Create, Read,Update, Delete
IAM	Identity And Access Management

CHAPTER 1

INTRODCUTION

1.1 Aim

The aim of building a serverless web application using AWS Lambda, DynamoDB, and S3 is to create a scalable, cost-effective, and highly available web application without the need for managing any servers. This architecture allows you to focus on building the application code and leave the infrastructure management to AWS. By using AWS Lambda, you can run your application code without provisioning or managing servers. DynamoDB provides a fully managed NoSQL database that can handle millions of requests per second and scales automatically. S3 is a highly scalable and durable object storage service that can host static websites. Together, these AWS services provide a powerful combination for building a serverless web application that can handle large amounts of traffic and data, while keeping costs low and minimizing operational overhead.

1.2 Background

Serverless architecture is a cloud computing model that allows you to run and manage applications without the need for provisioning or managing servers. Instead, the cloud provider manages the infrastructure and automatically scales the resources based on the application's usage. This results in lower costs and increased scalability, as you only pay for the resources that your application actually uses.

AWS Lambda is a compute service that allows you to run your application code in response to events, such as HTTP requests, file uploads, or database updates. You can write your code in several programming languages, including Python, Node.js, Java, and C#. AWS Lambda automatically scales the resources based on the number of requests and executes the code within milliseconds.

DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance at any scale. You can use DynamoDB to store and retrieve any amount of data, and it automatically scales the resources to handle millions of requests per second. You can define the schema of your tables and use the DynamoDB API to perform CRUD operations on your data.

S3 is a highly scalable and durable object storage service that allows you to store and retrieve any amount of data from anywhere on the web. You can use S3 to host static websites, store and distribute files, and back up data. S3 automatically scales the resources based on the amount of data stored and provides high availability and durability by replicating data across multiple locations.

By using these AWS services, you can build a serverless web application that is highly scalable, cost-effective, and easy to manage. You only pay for the resources that your application uses, and AWS manages the infrastructure for you, allowing you to focus on building your application logic.

1.3 Context of the Project

The context of the project is to build a serverless web application that allows users to perform CRUD (Create, Read, Update, Delete) operations on a DynamoDB table using a web interface. The DynamoDB table will store data related to the application, such as user information, products, or any other data relevant to the application.

The web application will be built using a combination of AWS Lambda, API Gateway, S3, and DynamoDB. Lambda functions will be responsible for handling incoming HTTP requests from the web interface and performing the necessary CRUD operations on the DynamoDB table. API Gateway will be used to expose the Lambda functions as a RESTful API that can be accessed from the web interface.

The web interface will be a static website hosted on S3, built using HTML, CSS, and JavaScript. The website will allow users to interact with the application by sending HTTP requests to the API Gateway and displaying the data returned from the DynamoDB table.

The overall goal of the project is to demonstrate the power and flexibility of serverless architecture and showcase the benefits of using AWS services to build highly scalable and cost-effective web applications. The project can be customized to fit a variety of use cases, such as e-commerce, social networking, or any other web application that requires storage and retrieval of data.

1.4 Objectives and Goals

The objectives and goals of the project can be broken down into several key areas:

1. Functionality: The primary objective of the project is to build a fully functional web application that allows users to perform CRUD operations on a DynamoDB table. The web interface should be intuitive and easy to use, and users should be able to create, read, update, and delete items from the table.

2. Scalability: Another key objective of the project is to demonstrate the scalability of the serverless architecture. The application should be able to handle a large number of requests and scale automatically based on the usage patterns. This will ensure that the application remains responsive and available even under heavy load.

3. Security: Security is a critical aspect of any web application. The project should implement best practices for securing the application and protecting user data. This includes implementing authentication and authorization mechanisms to ensure that only authorized users can access the application and perform CRUD operations on the DynamoDB table.

4. Cost-effectiveness: Serverless architecture is designed to be cost-effective, and the project should demonstrate this by keeping the costs of running the application as low as possible. This includes optimizing the use of AWS services and minimizing the resources needed to run the application.

5. Ease of deployment and management: Finally, the project should be designed to be easy to deploy and manage. This includes using tools such as AWS CloudFormation and AWS CodePipeline to automate the deployment process and simplify the management of the application over time.

By achieving these objectives and goals, the project will demonstrate the power and flexibility of serverless architecture and showcase the benefits of using AWS services to build scalable and cost-effective web applications.

1.5 Project scope and limitation

The project scope is to build a serverless web application that allows users to perform CRUD operations on a DynamoDB table using a web interface. The application will be built using a combination of AWS services, including Lambda, API Gateway, S3, and DynamoDB. The web interface will be a static website hosted on S3, and users will interact with the application by sending HTTP requests to the API Gateway.

The limitations of the project include:

1. Limited functionality: The project will focus on building a basic CRUD application. Additional functionality, such as search or filtering, may be beyond the scope of the project.

2. Limited testing: The project will focus on building the application and demonstrating its functionality. Extensive testing and validation may be beyond the scope of the project.

3. Limited customization: The project will provide a basic framework for building a serverless web application. Customization beyond the scope of the project, such as adding additional features or integrating with other services, may require additional work.

4. Limited security: While the project will implement basic security measures, such as authentication and authorization, it may not address all security concerns and may require additional security measures in a production environment.

5. Limited scalability: The project will focus on demonstrating the scalability of serverless architecture but may not address all scalability concerns for a production environment. Additional work may be required to optimize the application for scalability.

CHAPTER 2

LITERATURE SURVEY

2.1 Serverless Architecture:

Serverless architecture is a cloud computing model that allows developers to build and run applications without managing servers. In serverless architecture, the cloud provider manages the infrastructure, including the servers, operating system, and runtime environment. The developer is responsible only for the application code.

One of the primary benefits of serverless architecture is scalability. The cloud provider automatically scales the application based on demand, so developers do not need to worry about provisioning additional servers or managing capacity. This makes serverless architecture well-suited for applications with highly variable workloads, as well as applications that experience sudden spikes in traffic.

Another benefit of serverless architecture is cost-effectiveness. Developers only pay for the compute time used by their application, rather than paying for a fixed amount of server capacity. This means that developers can save money by building applications that are more efficient and only use resources when they are needed.

2.2 AWS services:

1. AWS Lambda: A serverless compute service that allows developers to run code in response to events, such as HTTP requests or changes to data in a database. Lambda is highly scalable and cost-effective, as developers only pay for the compute time used by their application.

2. API Gateway: A fully managed service that allows developers to create, publish, and manage APIs that can be used to trigger Lambda functions. API Gateway can be used to build RESTful APIs that provide a way for clients to interact with serverless applications.

3. Amazon S3: A highly scalable and durable object storage service that can be used to store and retrieve data, including static website files, images, and other content.

4. DynamoDB: A fully managed NoSQL database service that can be used to store and retrieve application data. DynamoDB is highly scalable and can handle large amounts of data, making it well-suited for serverless applications.

5. AWS CloudFormation: A service that allows developers to define and deploy infrastructure as code. CloudFormation can be used to create and manage AWS resources, including Lambda functions, API Gateway APIs, S3 buckets, and DynamoDB tables.

6. AWS IAM: A service that provides identity and access management for AWS resources. IAM can be used to manage permissions for AWS resources, including Lambda functions, API Gateway APIs, S3 buckets, and DynamoDB tables.

2.3 Best practices:

1. Design for scalability: Serverless architecture is highly scalable by nature, but you still need to design your application to handle spikes in traffic and large amounts of data. Use auto-scaling and other features provided by AWS services to handle spikes in traffic, and design your application to take advantage of distributed computing.

2. Minimize resource usage: Since you pay for the compute time used by your application, it's important to minimize resource usage wherever possible. Optimize your code for performance, use caching and other techniques to reduce the amount of compute time required, and use AWS services that are optimized for cost, such as S3 and DynamoDB.

3. Use a microservices architecture: Serverless architecture is well-suited for microservices, which are small, modular services that can be combined to build a larger application. Use AWS services like Lambda and API Gateway to build microservices that can be easily scaled and managed.

4. Use version control: Use version control to manage your application code and configuration. Use AWS CodeCommit or other version control tools to store your code and track changes, and use AWS CloudFormation to deploy and manage your infrastructure as code.

5. Secure your application: Use AWS IAM to manage access to your AWS resources, and use SSL/TLS to encrypt traffic to and from your application. Follow AWS security best practices to secure your application and protect your data.

6. Monitor your application: Use AWS CloudWatch to monitor your application and track metrics like CPU usage, memory usage, and request latency. Use CloudWatch alarms to alert you when certain thresholds are reached, and use AWS X-Ray to trace requests through your application and identify bottlenecks.

CHAPTER 3

SYSTEM DESIGN AND METHODOLOGY

3.1 System Requirements and Specifications

Hardware Requirements :

- Development machine with a modern processor, at least 8GB of memory, and a fast hard drive or solid-state drive
- Reliable internet connection to access AWS services and deploy the application
- Testing devices or emulators for various platforms, if needed
- Backup and disaster recovery plan to protect the application and data, using AWS services like S3, Glacier, and Disaster Recovery.

Software Requirements

- Node.js (v12 or higher)
- Amazon CLI
- AWS SDKs
- Git

3.2 Tools and Technologies used

1. AWS Lambda: A serverless compute service that lets you run code without provisioning or managing servers.

2. Amazon API Gateway: A fully managed service that makes it easy to create, publish, maintain, monitor, and secure APIs.

3. Amazon DynamoDB: A NoSQL database service that provides fast and predictable performance with seamless scalability.

4. Amazon S3: A highly scalable and durable object storage service that can store and retrieve any amount of data from anywhere on the web.

5. AWS CloudFormation: A service that provides a common language to describe and provision all the infrastructure resources needed for your application.

6. AWS SAM: AWS Serverless Application Model (SAM) is an open-source framework for building serverless applications on AWS.

7. AWS CLI: A unified tool to manage your AWS services from the command line.

8. AWS SDKs: Software development kits for various programming languages that provide APIs for interacting with AWS services.

9. Visual Studio Code: A popular code editor that has an extension for developing and deploying serverless applications on AWS.

10. Node.js: A popular runtime environment for building serverless applications on AWS Lambda.

3.3 System Architecture and Components

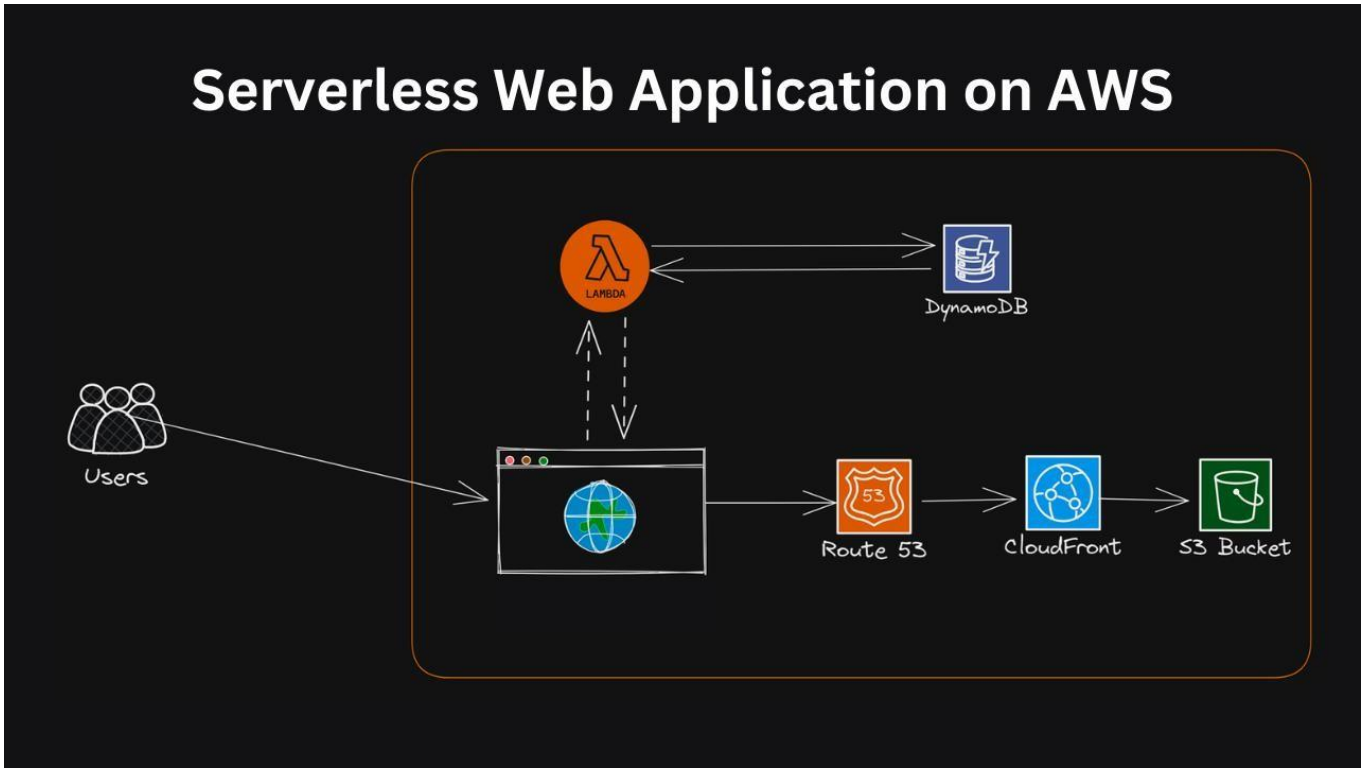


Fig 3.1 Architecture Diagram

3.4 Methodology Used in System Development

1. Requirements gathering: The requirements of the application are collected from the stakeholders, such as users, product owners, and developers.

2. Sprint planning: The requirements are broken down into smaller, manageable pieces called user stories. The user stories are prioritized and assigned to sprints, which are time-boxed periods of development, typically one to four weeks long.

3. Sprint execution: The development team works on the user stories assigned to the sprint, using iterative development techniques to build and test the application. The team holds daily stand-up meetings to discuss progress, identify issues, and plan for the day ahead.

4. Sprint review: At the end of the sprint, the team presents the completed user stories to the stakeholders for review and feedback.

5. Sprint retrospective: The team holds a retrospective meeting to reflect on the sprint, identify what worked well and what needs improvement, and plan for the next sprint.

6. Continuous integration and deployment: Throughout the development process, the application code is integrated and tested in a continuous integration and deployment (CI/CD) pipeline. This pipeline automates the testing, building, and deploying of the application to AWS, ensuring that the code is always in a deployable state.

CHAPTER 4

Configuration and Deployment Plan

In this project, you will build a serverless web application using AWS Lambda, DynamoDB, and S3. The application will allow users to create, read, update, and delete (CRUD) items from a DynamoDB table.

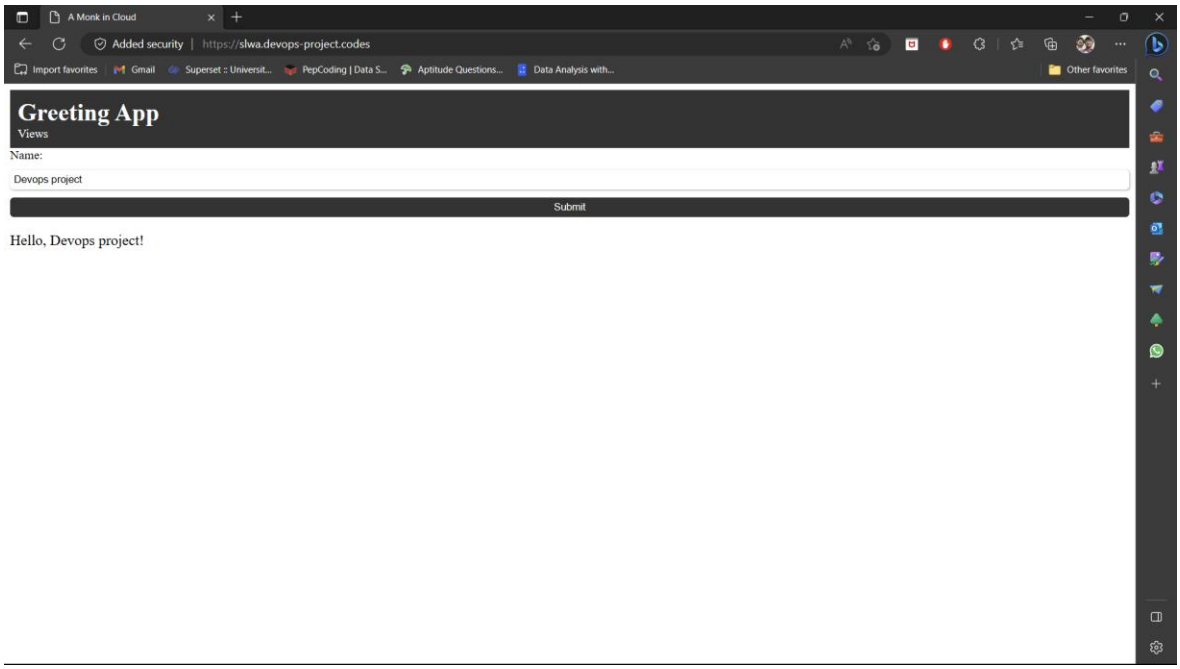


Fig 4.0 serverless web application

4.1 Set up AWS S3

As we are going to host a static web application we are going to store HTML, JavaScript, css files in s3 bucket.

Bucket Name: cloud-and-devops-project

Region: ap-south-1

Note: Keep rest of the settings as default in console.

Upload the HTML, CSS and JavaScript files into the s3 bucket.

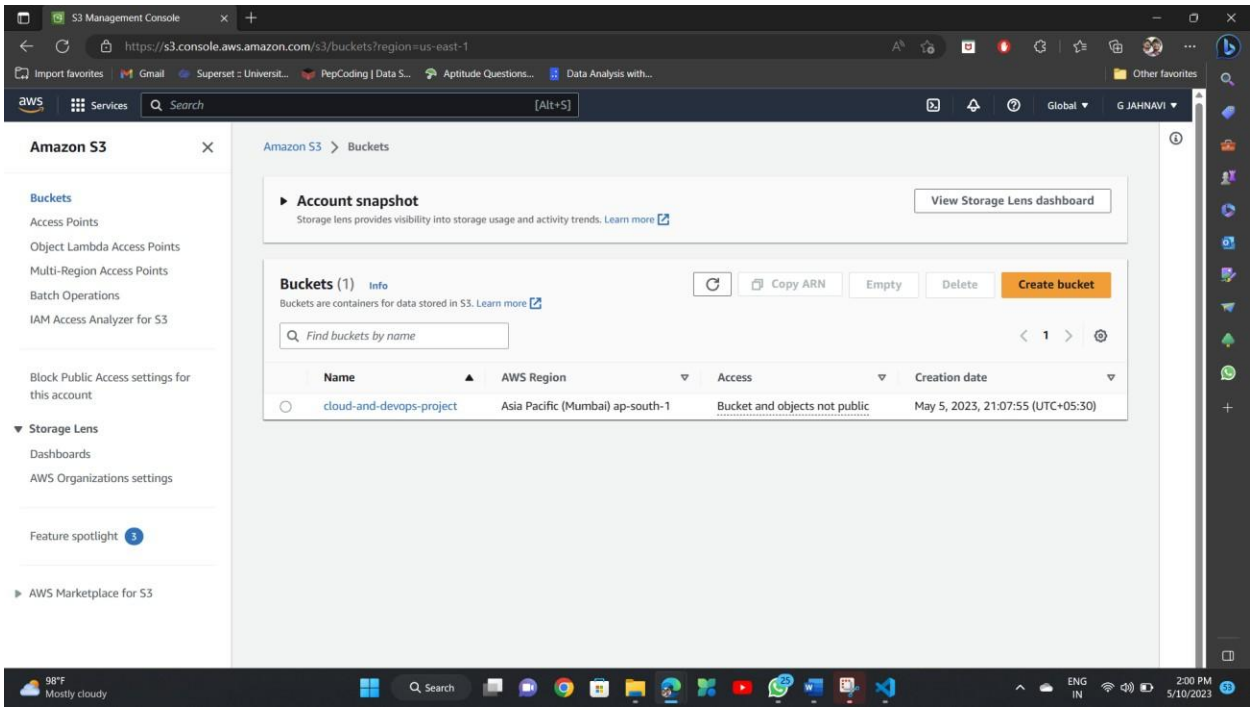
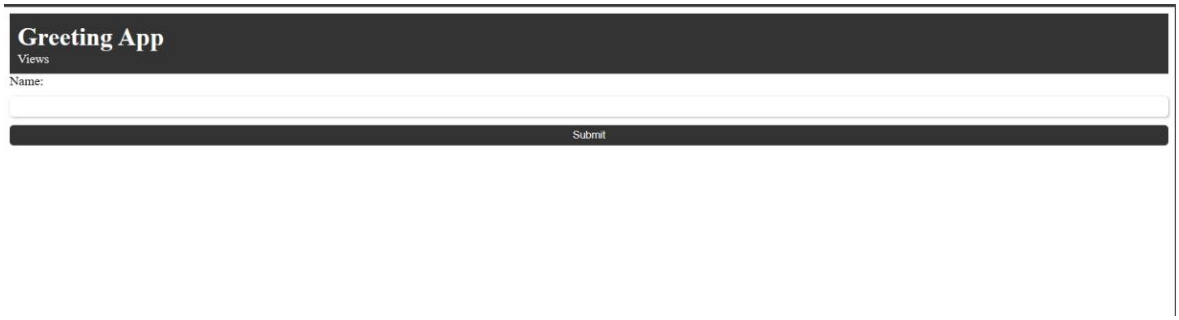


Fig 4.1. created S3 bucket

HTML File used:



Script.js:

```
const form = document.querySelector('form');
const greeting = document.querySelector('#greeting');

form.addEventListener('submit', (event) =>
    { event.preventDefault();
    const name = document.querySelector('#name').value;
    greeting.textContent = `Hello, ${name}!`;
```

```
});

const counter = document.querySelector(".counter-number");
async function updateCounter() {
    let response = await
        fetch( "https://o73g5ptpujgtclinhzhhneplje0jogoh.lambda-url.us-east-
        1.on.aws/"
    );
    let data = await response.json();
    counter.innerHTML = `Views: ${data}`;
}
updateCounter();
```

Style.css:

```
header {
    background-color: #333;
    color: #fff;
    padding: 10px;
}

h1 {
    margin: 0;
}

form {
    display: flex;
    flex-direction: column;
```

```
}

label {
    margin-bottom: 10px;
}

input[type="text"]
    { padding: 5px;
    margin-bottom: 10px;
    border-radius: 5px;
    border: none;
    box-shadow: 1px 1px 3px rgba(0, 0, 0, 0.3);
}

button {
    padding: 5px 10px;
    background-color: #333;
    color: #fff;
    border: none;
    border-radius: 5px;
    cursor: pointer;
}

#slwa {
    margin-top: 20px;
    font-size: 1.2rem;
}
```

4.2 Create AWS Cloudfront

Create a CloudFront distribution to serve the S3-hosted static files with low latency.

1. Go to the AWS Management Console and log in to your account.

2. Click on the "Services" dropdown menu, search for "CloudFront," and click on it.

3. Click on the "Create Distribution" button.

4. Select the type of distribution you want to create. There are two types: web and RTMP. For this example, we'll create a web distribution.

5. Configure your distribution settings. You'll need to set the following:

- Origin Domain Name: This is the domain name of the Amazon S3 bucket or web server where your content is stored.
- Origin Protocol Policy: This setting determines whether CloudFront uses HTTP or HTTPS to fetch content from your origin server. Select the one that fits your needs.
- Viewer Protocol Policy: This setting determines whether CloudFront delivers content over HTTP or HTTPS to your viewers. Again, select the one that fits your needs.
- Allowed HTTP Methods: Specify which HTTP methods are allowed for your content (GET, HEAD, OPTIONS, PUT, POST, PATCH, DELETE).
- Cached HTTP Methods: Specify which HTTP methods should be cached by CloudFront.
- Price Class: Choose the price class for your distribution, which determines the AWS regions where your content will be delivered.

6. Configure your cache behavior settings. Here are the important settings:

- Path Pattern: This is the pattern for the URL that you want CloudFront to cache. You can use wildcards to match multiple URLs.
- Viewer Protocol Policy: This setting determines whether CloudFront delivers content over HTTP or HTTPS to your viewers.

- Minimum TTL: This is the minimum time-to-live for your content, in seconds. CloudFront won't serve stale content that's older than this value.
- Maximum TTL: This is the maximum time-to-live for your content, in seconds. CloudFront won't serve content that's older than this value.
- Default TTL: This is the default time-to-live for your content, in seconds. CloudFront will use this value if it can't find a more specific TTL.

7. Configure your distribution's access settings. Here are the important settings:

- Restrict Viewer Access: Choose whether to restrict access to your content to specific IP addresses or HTTP referers.
- Trusted Signers: Specify which AWS accounts are allowed to sign URLs for your content.
- Alternate Domain Names (CNAMEs): Specify any alternate domain names that you want to use for your distribution.

8. Review your settings and click on the "Create Distribution" button.

9. Wait for your distribution to deploy. This can take up to 15 minutes.

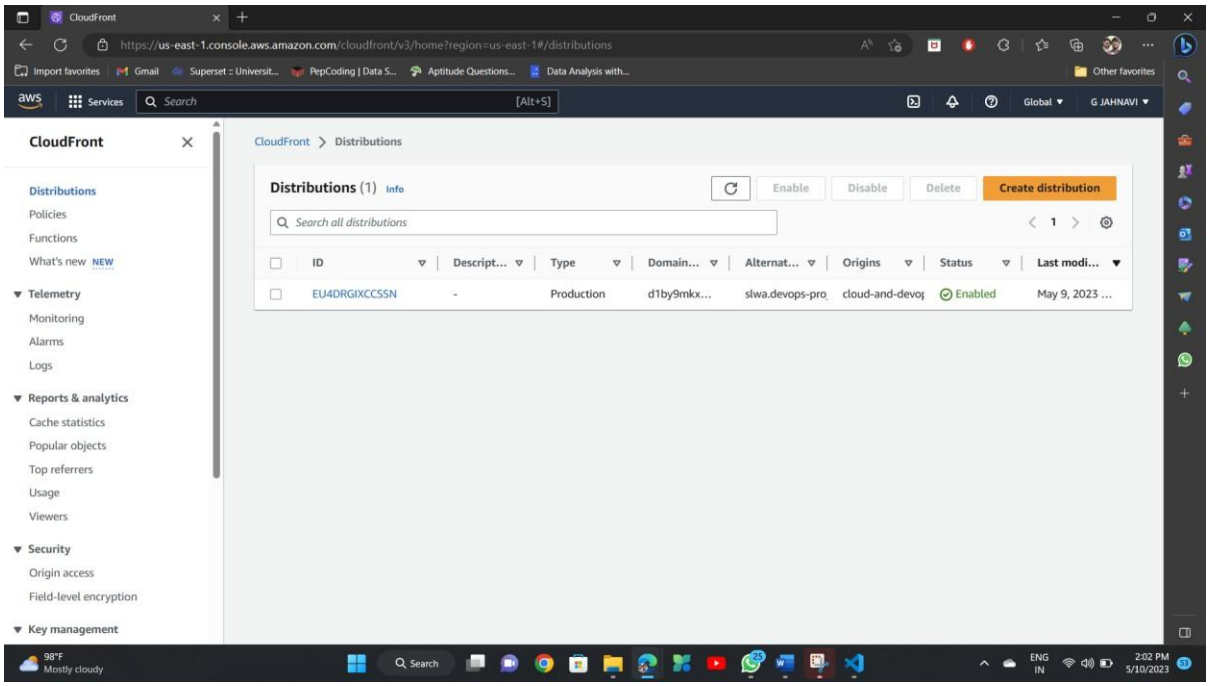


Fig 4.2 created cloudfront

4.3 Create route53

1. Go to the AWS Management Console and log in to your account.

2. Click on the "Services" dropdown menu, search for "Route 53," and click on it.

3. Click on the "Hosted zones" link on the left-hand side of the page.

4. Click on the "Create Hosted Zone" button.

5. Enter the domain name for your hosted zone, for example, "example.com". Note that you don't need to include the "www" part.

6. Choose the type of hosted zone that you want to create. There are two types: public and private. For this example, we'll create a public hosted zone.

7. Click on the "Create" button.

8. Once your hosted zone is created, you'll be taken to the hosted zone details page. Here, you'll see the name servers that you need to use for your domain name.

9. Go to your domain registrar's website and update your domain name's name servers to the ones provided by Route 53.

10. Wait for the changes to propagate. This can take up to 48 hours.

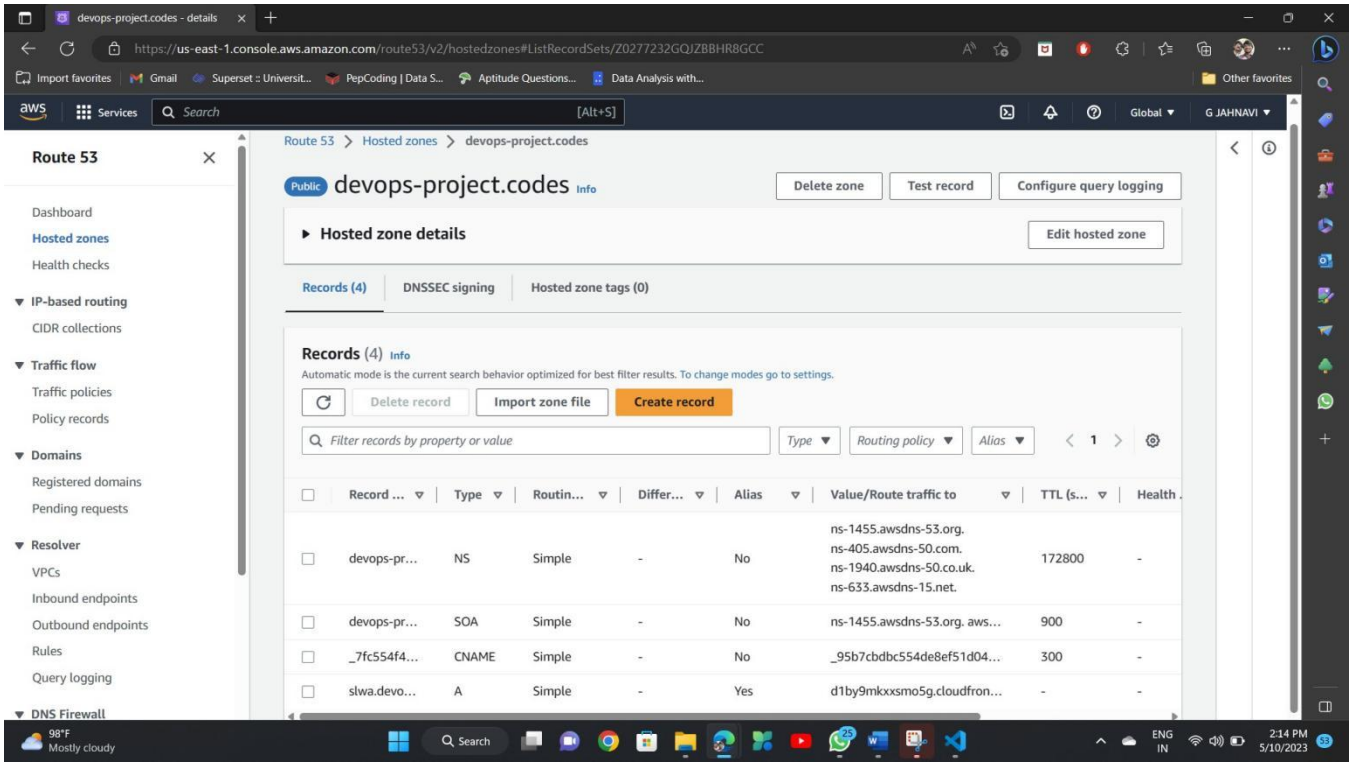


Fig 4.3. route53 created

4.4 Create DynamoDB

1. Go to the AWS Management Console and log in to your account.

2. Click on the "Services" dropdown menu, search for "DynamoDB," and click on it.

3. Click on the "Create table" button.

4. Enter the name of your table and the primary key. The primary key is used to uniquely identify each item in the table. You can choose to use a simple or composite primary key. A simple primary key consists of a partition key, while a composite primary key consists of both a partition key and a sort key.

5. Choose the read and write capacity for your table. This determines how much read and write throughput your table can handle. You can choose to use on-demand capacity, which automatically scales based on your workload, or provisioned capacity, which requires you to manually specify the capacity.

6. (Optional) Add any secondary indexes that you need. Secondary indexes allow you to query your data using different attributes than the primary key.

7. Click on the "Create" button.

8. Once your table is created, you can start adding items to it. To add an item, click on the "Items" tab in the DynamoDB console and then click on the "Create item" button. You can enter the attributes and values for your item in the pop-up window.

9. You can also use the DynamoDB console to query your data, create and manage backups, and set up alarms for your table.

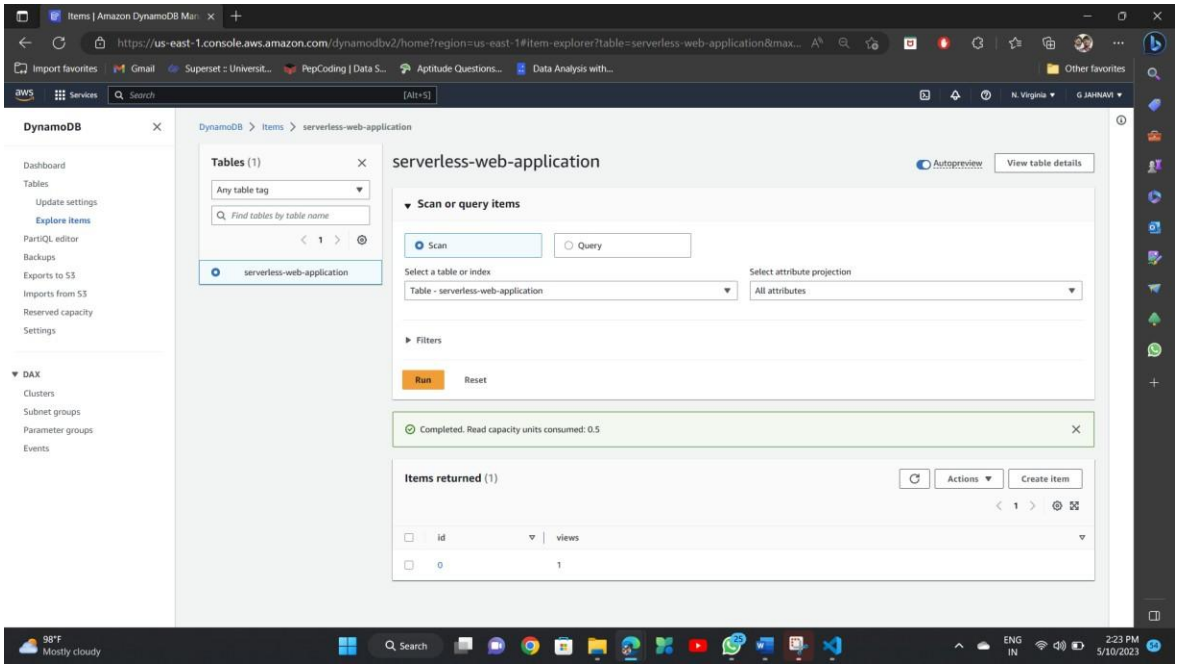


Fig.4.4. created dynmoDB

4.5 create IAM user

1. Go to the AWS Management Console and log in to your account.

2. Click on the "Services" dropdown menu, search for "IAM," and click on it.

3. In the IAM dashboard, click on the "Users" link on the left-hand side of the page.

4. Click on the "Add user" button.

5. Enter a name for your user.

6. Select the access type for your user. You can choose between programmatic access, which provides access to the AWS API and CLI, and AWS Management Console access, which provides access to the AWS Management Console.

7. (Optional) Set a password for your user if you are creating a user with AWS Management Console access. You can also choose to let the user create their own password.

8. Click on the "Next: Permissions" button.

9. Select the permissions that you want to grant to your user. You can choose to add your user to an existing group or create a new group with specific permissions.

10. (Optional) You can also choose to add tags to your user.

11. Click on the "Next: Review" button.

12. Review the information that you've entered, and click on the "Create user" button.

13. On the next page, you'll see the access key and secret access key for your user. Make sure to download the keys or copy them to a secure location as they won't be shown again.

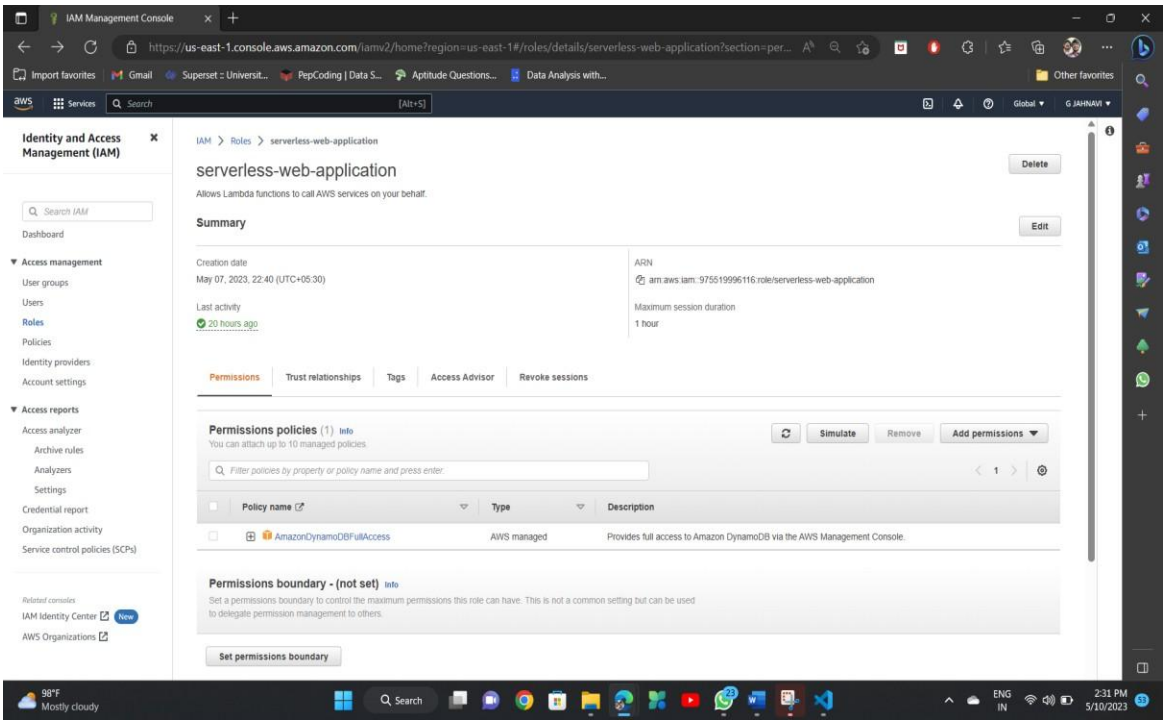


Fig 4.5 created IAM user

4.6 Create an AWS lambda

1. Go to the AWS Management Console and log in to your account.

2. Click on the "Services" dropdown menu, search for "Lambda," and click on it.

3. Click on the "Create function" button.

4. Choose the "Author from scratch" option.

5. Enter a name for your function.

6. Select the runtime that you want to use for your function. AWS Lambda supports several programming languages, including Node.js, Python, Java, and C#.

7. Choose the execution role for your function. This determines the permissions that your function has to access other AWS resources. You can either use an existing role or create a new one.

8. Click on the "Create function" button.

9. Once your function is created, you can write your function code in the "Function code" section of the AWS Lambda console. You can either edit your code directly in the console or upload a ZIP file containing your code.

10. You can also configure the trigger for your function in the "Designer" section of the console. AWS Lambda supports several trigger types, including API Gateway, S3, DynamoDB, and CloudWatch Events.

11. (Optional) You can also configure environment variables and advanced settings for your function.

12. Click on the "Save" button to save your changes.

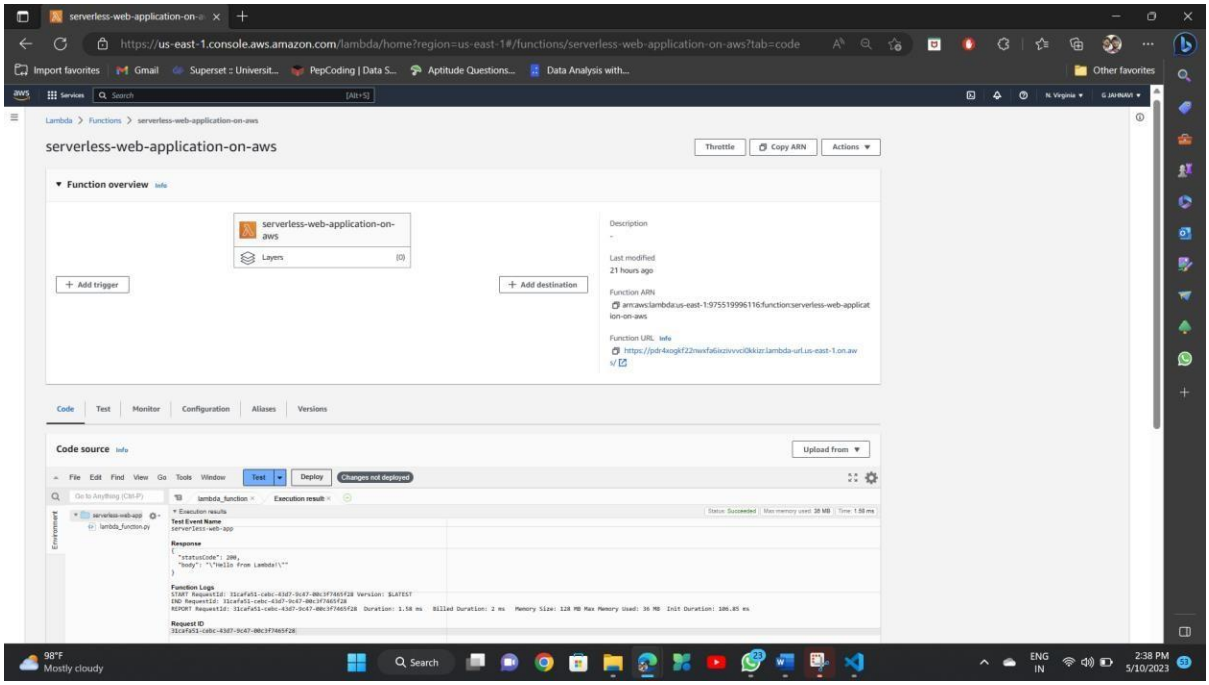


Fig 4.6. created lambda

Lambda function :

```
import json
import boto3
dynamodb = boto3.resource('dynamodb')
```

```
table = dynamodb.Table('serverless-web-application-on-aws')
def lambda_handler(event, context):
    response =
        table.get_item(Key={ 'id':'0'
    })
    views = response['Item']['views']
    views = views + 1

    print(views)

    response =
        table.put_item(Item={ 'id':'0',
        'views': views
    })

    return views
```

Execution Result:

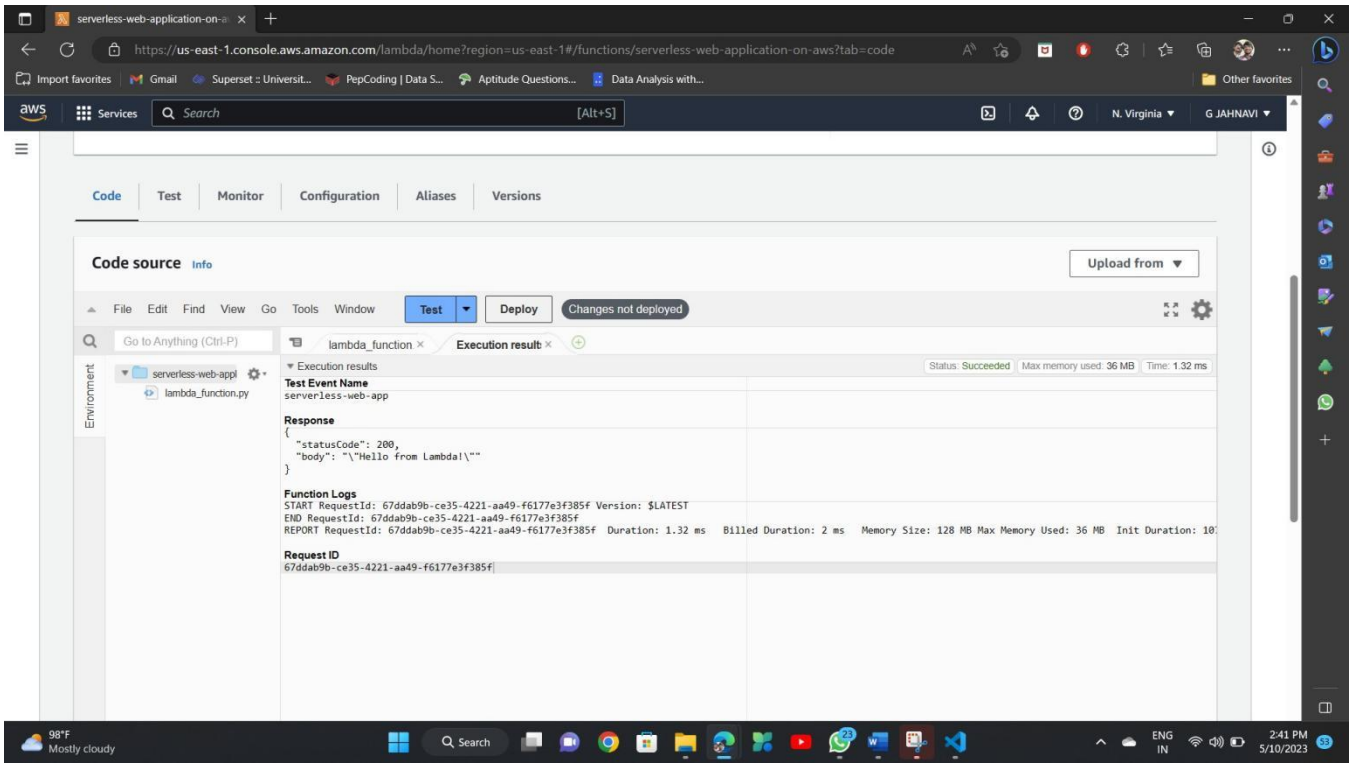
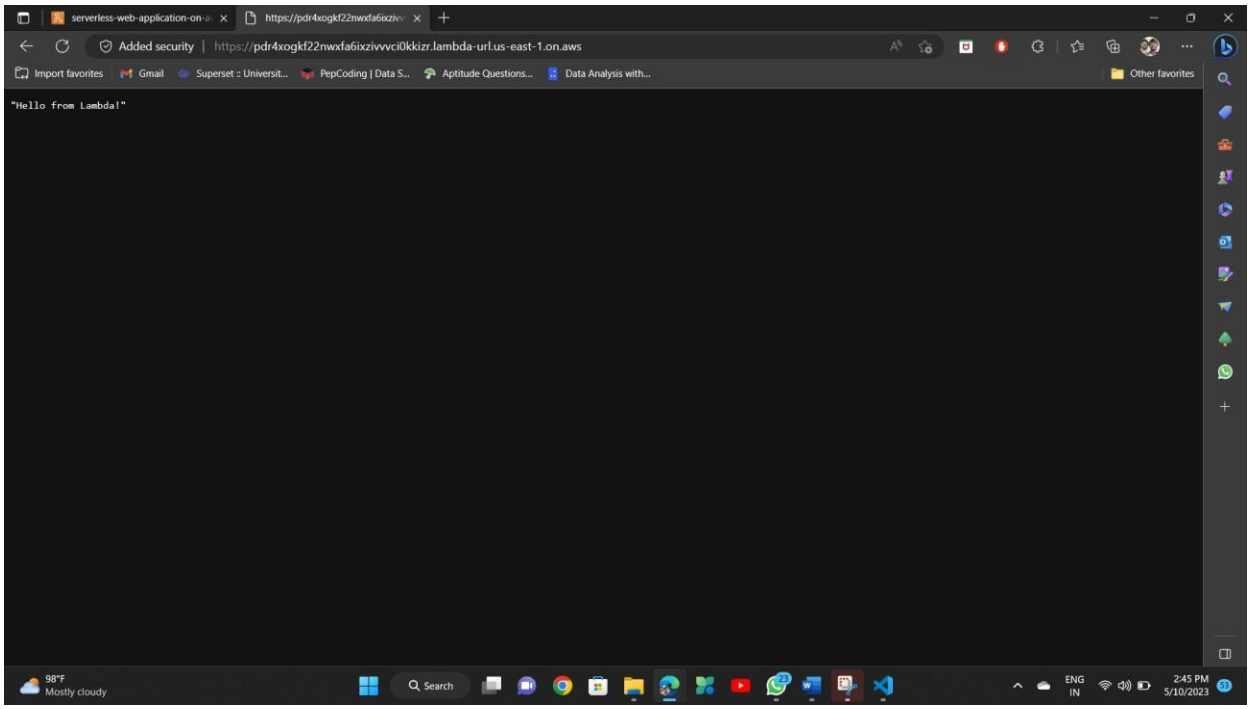


Fig.4.6.1.execution result


serverless-web-application-on-a
https://pdr4xogkf22nwxfa6ixzi
Added security | https://pdr4xogkf22nwxfa6ixzivvvci0kkizr.lambda-url.us-east-1.on.aws
Import favorites
Gmail
Superset :: Universit...
PepCoding | Data S...
Aptitude Questions...
Data Analysis with...
Other favorites
"Hello from Lambda!"
98°F
Mostly cloudy
Search
ENG
IN
2:45 PM
5/10/2023

CHAPTER 5
RESULTS AND DISCUSSIONS

Serverless web applications in AWS have become increasingly popular due to the many benefits provided by AWS services. Here are some of the key results and discussions on serverless web applications in AWS:

AWS Lambda: AWS Lambda is a key component of serverless web applications in AWS. Lambda allows you to run code without provisioning or managing servers. This provides automatic scaling and low costs, since you only pay for the compute time that you consume. Additionally, Lambda integrates with many other AWS services, making it easy to build complex serverless applications.

API Gateway: AWS API Gateway is another key component of serverless web applications in AWS. API Gateway provides a fully managed service for creating, publishing, and managing APIs. This allows you to expose your Lambda functions as RESTful APIs, making it easy to build serverless web applications.

DynamoDB: DynamoDB is a fully managed NoSQL database service in AWS. DynamoDB provides low-latency access to data at any scale, making it ideal for serverless web applications. Additionally, DynamoDB integrates with AWS Lambda and API Gateway, making it easy to build scalable and reliable serverless applications.

S3: Amazon S3 is an object storage service in AWS. S3 provides low-cost storage for data at any scale, making it ideal for serverless web applications. Additionally, S3 integrates with AWS Lambda, API Gateway, and other AWS services, making it easy to build serverless applications that store and retrieve data.

Challenges: While AWS provides many benefits for building serverless web applications, there are also some challenges to consider. For example, serverless web applications can be more difficult to debug and monitor, since the infrastructure is managed by AWS. Additionally, AWS provides many services and features, which can be overwhelming for beginners

CHAPTER 6

CONCLUSIONS AND RECOMMENDATIONS

In conclusion, serverless web applications in AWS offer numerous benefits, such as cost savings, scalability, faster development times, and automatic infrastructure management. AWS provides a wide range of services and features, such as Lambda, API Gateway, DynamoDB, and S3, which can be easily integrated to build complex and efficient serverless applications. However, there are also some challenges to consider, such as debugging and monitoring, and the complexity of the AWS ecosystem.

If you are considering using serverless web applications in AWS, it is important to evaluate the specific needs of your project and determine if a serverless approach is suitable. For projects with unpredictable or spiky traffic, or those with strict budget constraints, serverless web applications in AWS can be a great choice. However, if you need fine-grained control over your infrastructure or require specialized hardware or software, a traditional server-based approach may be more appropriate.

To successfully build and manage serverless web applications in AWS, it is recommended to have a good understanding of AWS services and features, as well as best practices for security, performance, and cost optimization. Additionally, it is important to design and architect your application with scalability and fault tolerance in mind, in order to ensure high availability and reliability.

Overall, serverless web applications in AWS offer numerous benefits and can be a powerful tool for building efficient and scalable web applications. With careful planning and execution, you can take advantage of the benefits of serverless technology while avoiding the pitfalls and challenges that come with it.

REFERENCES

1. https://aws.amazon.com/route53/
2. https://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/Introduction.html
3. https://aws.amazon.com/s3/
4. https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/Introduction.html
5. https://aws.amazon.com/lambda/