A Project report on

SERVERLESS WEBSITE ON AWS

A Dissertation submitted to JNTU Hyderabad in partial fulfilment of the academic requirements for the award of the degree.

Bachelor of Technology

in

Computer Science and Engineering

Submitted by

G HARSHAVARDHAN 20H51A05G8

G SUPRIYA 20H51A05H0

M SRIKANTH 20H51A05L1

Under the esteemed guidance of

Dr. G.RAVI KUMAR

Associate Professor



Department of Computer Science and Engineering

CMR COLLEGE OF ENGINEERING & TECHNOLOGY

(An Autonomous Institution, Approved by AICTE, Affiliated to JNTUH, NAAC ’A+’)

KANDLAKOYA, MEDCHAL ROAD, HYDERABAD - 501401.

2020- 2024

### CMR COLLEGE OF ENGINEERING & TECHNOLOGY

KANDLAKOYA, MEDCHAL ROAD, HYDERABAD – 501401

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING



#### CERTIFICATE

This is to certify that the Mini Project-2 report entitled **"SERVERLESS WEBSITE ON AWS"** being submitted by G HARSHAVARDHAN(20H51A05G8), G SUPRIYA(20H51A05H0), M SRIKANTH(20H51A05L1)in partial fulfilment for the award of **Bachelor of Technology in Computer Science and Engineering** is a record of Bonafide work carried out his/her under my guidance and supervision.

The results embodies in this project report have not been submitted to any other University or Institute for the award of the Degree.

Dr. G. RAVI KUMAR Dr. Siva Skandha Sanagala

Associate Professor Associate Professor and HOD

Dept. of CSE Dept. of CSE

**Acknowledgement**

With great pleasure I want to take this opportunity to express my heartfelt gratitude to all the people who helped in making this project work a grand success.

We are grateful to **Dr. G. Ravikumar** Associate. Professor, Dept. of Computer Science and Engineering for his valuable technical suggestions and guidance during the execution of this project work.

We would like to thank **Dr. Siva Skandha Sanagala**, Head of the Department of Computer Science and Engineering, CMR College of Engineering and Technology, who is the major driving forces to complete my project work successfully.

We are very grateful to **Dr. Vijaya Kumar Koppula**, Dean-Academic, CMR College of Engineering and Technology, for his constant support and motivation in carrying out the project work successfully.

We are highly indebted to **Dr. V A Narayana**, Principal CMR College of Engineering and Technology, for giving permission to carry out this project in a successful and fruitful way.

We would like to thank the Teaching & Non- teaching staff of Department of Computer Science and Engineering for their co-operation.

Finally I express my sincere thanks to **Mr. Ch. Gopal Reddy**, Secretary, CMR Group of Institutions, for his continuous care. I sincerely acknowledge and thank all those who gave support directly and indirectly in completion of this project work.

**SIGNATURE**

###### 

###### G HARSHAVARDHAN 20H51A05G8

###### G SUPRIYA 20H51A05H0

M.SRIKANTH 20H51A05L1

**TABLE OF CONTENT**

|  |  |  |
| --- | --- | --- |
| **CHAPTER NO.** | **TITLE** | **PAGE NO.** |
|  | **ABSTRACT** |  |
| **1.** | **INTRODUCTION** | 1 |
| **2.** | **BACKGROUND WORK** | 2-6 |
|  | 2.1. Existing Systems | 2 |
|  | 2.2. Limitations | 2 |
|  | 2.3. Motivation Study | 2 |
|  | 2.4. Problem Analysis | 2 |
|  | 2.5. Literarure survey | 3-5 |
| **3.** | **PROPOSED SYSTEMS** | 6-8 |
|  | 3.1. Hardware requirements | 6 |
|  | 3.2. Software requirements | 6 |
|  | 3.3. User interfaces | 6 |
|  | 3.4. Software interfaces | 6 |
|  | 3.5.user characteristics | 7-8 |
| **4.** | **DESIGNING** | 9-12 |
|  | 4.1. Building a serverless website on aws | 9 |
|  | 4.2. The Website | 9 |
|  | 4.3.Dynamodb | 10 |
| **5.** | **RESULTS** | 14- 22 |
|  | 5.1. Implementation | 14-21 |
|  | 5.2. Result | 22 |
| **6.** | **CONCLUSION** | 21-22 |
|  | 6.1. Conclusion | 23 |
|  | 6.2. Future scope | 23 |
|  | 6.3. References | 24 |

**ABSTRACT**

This project aims to design and implement a serverless website using Amazon Web Services (AWS). The website will be built using AWS Lambda, Amazon API Gateway, Amazon S3, and Amazon DynamoDB. The primary goal of the project is to demonstrate the feasibility and benefits of using serverless architecture for web development.

The website will have a simple, yet functional design with features such as user authentication, dynamic content generation, and data storage. The authentication system will be implemented using Amazon Cognito, which provides user sign-up, sign-in, and access control. The dynamic content will be generated using AWS Lambda functions, which will interact with the database (DynamoDB) and other third-party APIs. The data will be stored in DynamoDB, a fully managed NoSQL database service, which provides high performance, scalability, and availability.

Overall, this project will showcase the power and flexibility of AWS serverless technology in building scalable, secure, and cost-efficient web applications.

# 

# 

# 

# 

# **1.** **INTRODUCTION**

**1.1 What Is Serverless?**

Serverless most often refers to serverless applications. Serverless applications are ones that don't require you to provision or manage any servers. You can focus on your core product and business logic instead of responsibilities like operating system (OS) access control, OS patching, provisioning, right-sizing, scaling, and availability. By building your application on a serverless platform, the platform manages these responsibilities for you. For service or platform to be considered serverless, it should provide the following capabilities:

**•** **No server management –** You don’t have to provision or maintain any servers. There is no software or runtime to install, maintain, or administer.

**•** **Flexible scaling –** You can scale your application automatically or by adjusting its capacity through toggling the units of consumption (for example, throughput, memory) rather than units of individual servers.

**• High availability –** Serverless applications have built-in availability and fault tolerance. You don't need to architect for these capabilities because the services running the application provide them by default.

**• No idle capacity –** You don't have to pay for idle capacity. There is no need to pre-provision or over-provision capacity for things like compute and storage. There is no charge when your code isn’t running. The AWS Cloud provides many different services that can be components of a serverless application. These include capabilities for:

• Compute – AWS Lambda

• APIs – Amazon API Gateway

• Storage – Amazon Simple Storage Service (Amazon S3)

• Databases –Amazon DynamoDB

• Interprocess messaging – Amazon Simple Notification Service (Amazon SNS) and Amazon Simple Queue Service (Amazon SQS)

• Orchestration – AWS Step Functions7 and Amazon CloudWatch Events.

• Analytics – Amazon Kinesis.

* 1. **PROJECT PURPOSE**

AWS Serverless Architecture is a way to build and run applications without managing infrastructure. AWS offers technologies for running code, managing data, and integrating applications, all without managing servers. Serverless technologies feature automatic scaling, built-in high availability, and a pay-for-use billing model to increase agility and optimize costs.**AWS Lambda** is an event-driven compute service natively integrated with over 200 AWS services and software as a service (SaaS) applications. With AWS serverless technologies, you can build and run applications without having to manage your own servers. Serverless applications are event-driven and loosely coupled via technology-agnostic APIs or messaging. AWS Lambda is a serverless compute service that is well suited to event-driven architectures. Lambda functions are triggered by events via integrated event sources such as Amazon Simple Queue Service (SQS), Amazon Simple Notification Service (SNS), and Amazon Kinesis that can be used to create asynchronous integrations.



Fig-1.1.a Serverless web Application

**2.** **BACKGROUND WORK**

**2.1. EXISTING SYSTEMS**

**Serverless Services on AWS: An Overview**

* AWS Lambda AWS Lambda has an event-driven charge system, which means you pay exclusively for what you use.
* AWS Fargate AWS Fargate runs containers on its own without your intervention.
* AWS DynamoDB AWS DynamoDB is a NoSQL database service hosted by AWS. .
* Amazon Aurora
* Amazon SNS
* Amazon SQS
* AWS CloudWatch
* Amazon QuickSight

**2.2. LIMITATIONS**

* Hard limit to execution time (from 5–15 minutes).
* No support for stateful applications.
* No local storage.
* Hard limit on invocation payload size (e.g. 128 KB for AWS Lambda).
* Cold starts due to instantiating new containers during scaling — potentially leading to latency.
* Lack of local testing options.
* Tooling limitations for deployment, management, and development.

**2.3. PRODUCT SCOPE**

1. User Interface (UI): Determine the design and layout of your website, including the pages, navigation structure, and user interactions. Consider using a front-end framework or library like React, Angular, or Vue.js for building a responsive and dynamic UI.
2. Static Content: Identify the static content of your website, such as HTML files, CSS stylesheets, JavaScript files, images, and other media assets. These assets will be stored in an Amazon S3 bucket and served to users directly from the bucket using S3's static website hosting feature.
3. Dynamic Functionality: Determine if your website requires any server-side processing or dynamic functionality. This could include features like user authentication, database access, form submissions, real-time updates, or integration with other APIs. For such functionality, you'll create AWS Lambda functions using a supported programming language.

**2.4. LITERATURE SURVEY**

1. AWS Documentation: The official documentation provided by AWS is a valuable resource for understanding the various AWS services related to serverless architectures. Start with the AWS Lambda documentation, Amazon API Gateway documentation, and Amazon S3 documentation. They provide detailed information, tutorials, and best practices.
2. Whitepapers and Case Studies: AWS publishes whitepapers and case studies that demonstrate the benefits, architecture patterns, and real-world examples of serverless applications on AWS. These resources can provide insights into best practices, cost optimization, and performance considerations.
3. Blogs and Developer Resources: AWS maintains a blog that covers a wide range of topics, including serverless architecture and best practices. Explore the AWS Compute Blog, AWS Architecture Blog, and AWS Developer Blog for articles, tutorials, and insights from AWS experts and community members.
4. Online Communities and Forums: Engaging with online communities and forums focused on AWS and serverless architecture can be beneficial. Platforms like Reddit (r/aws), Stack Overflow, and the AWS Developer Forums have active communities where you can find discussions, ask questions, and learn from others' experiences.

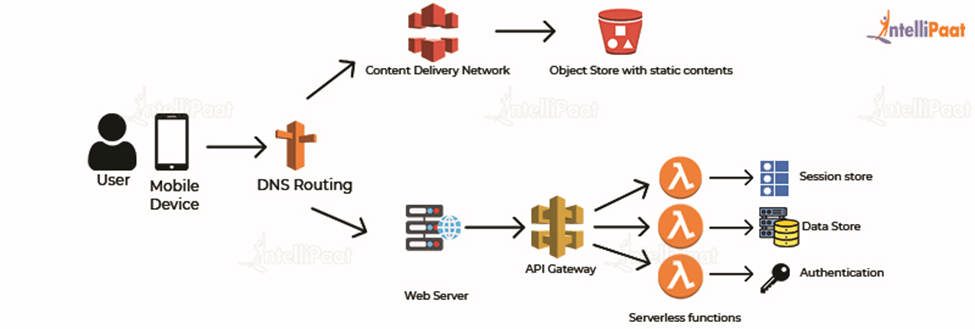


Fig-2.4.a Background Work

**3. PROPOSED SYSTEM**

**3.1. HARDWARE REQURIMENTS**

* AWS Lambda
* Amazon S3
* Amazon API Gateway
* Database and other services

Fig-3.1.a AWS Lambda Fig-3.1.b AWS Lamba Fig-3.1.c AWS Dynamodb

**3.2. SOFTWARE REQURIMENTS**

* Operating System
* Web Development tools
* AWS CLI
* Version Control
* Programming Languages and Frameworks
* Database Clients

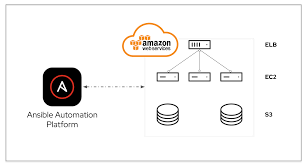


Fig-3.2.a Application Software

**3.3. USER INTERFACES**

* Static Website Hosting

Amazon S3

AWS Amplify

* Serverless Single-Page Applications (SPAs)

Amazon CloudFront

AWS ApplySync

* API Gateway and Backend Services

Amazon API Gateway

**3.4. SOFTWARE INTERFACES**

* AWS SDKs and APIs
* Event Sources and Triggers

AWS Lambda Triggers

Event-driven Services

* Service-to-Service Integration

API Gateway Integration

Database Integration

**3.5. USER CHARACTERISTICS**

When considering the user characteristics of a serverless website on AWS, it's important to focus on aspects such as scalability, performance, security, and accessibility. Here are some key user characteristics to consider:

1. **Scalability**: Serverless architectures on AWS are highly scalable, allowing your website to handle varying user loads. As the user base grows or experiences sudden spikes in traffic, AWS services like Lambda and API Gateway automatically scale to accommodate the demand. This ensures that your website remains responsive and available to users, regardless of the number of concurrent requests.
2. **Performance**: By leveraging AWS's global infrastructure and services like Amazon CloudFront, you can optimize the performance of your serverless website. CloudFront provides content delivery network (CDN) capabilities, caching, and edge locations to reduce latency and deliver content quickly to users around the world. This helps improve the overall user experience by reducing page load times and increasing responsiveness.
3. **Security**: AWS offers robust security measures to protect your serverless website and user data. Key security considerations include:
   * Identity and Access Management (IAM): You can use IAM to manage user access to AWS resources, controlling permissions and ensuring that only authorized users can interact with your website.
   * Encryption: AWS services like S3, API Gateway, and Lambda support encryption of data in transit and at rest. You can use SSL/TLS certificates to secure communications, and AWS Key Management Service (KMS) for encryption of sensitive data.
   * Web Application Firewall (WAF): AWS WAF helps protect your serverless website from common web application attacks, such as SQL injection and cross-site scripting (XSS). It allows you to define rules and policies to filter and block malicious traffic.
4. **Availability and Fault Tolerance**: AWS services are designed for high availability and fault tolerance. They handle infrastructure management, automatic scaling, and redundancy, ensuring that your serverless website remains accessible even in the event of failures or disruptions.
5. **Cost Efficiency**: Serverless architectures on AWS can offer cost advantages, as you only pay for the resources you actually use. AWS Lambda, for example, bills you based on the number of function invocations and the duration of each execution. This allows you to optimize costs by scaling resources based on actual demand.
6. **Accessibility:** Consider the accessibility of your serverless website to cater to users with disabilities. Ensure that your website adheres to accessibility guidelines, such as the Web Content Accessibility Guidelines (WCAG), by providing proper semantic markup, keyboard navigation support, alternative text for images, and other accessibility features.

These user characteristics are crucial to creating a successful serverless website on AWS. By considering scalability, performance, security, availability, cost efficiency, and accessibility, you can provide an optimal user experience and ensure that your website meets the needs of a diverse user base.

**4. DESIGNING**

**4.1 Building a serverless website in AWS**

Maintaining a web server for your website is not ideal. It’s tedious, takes a lot of time if done properly and distracts from what you really want to do: sell a product, share information, e.t.c. The more time and money you spend on the platform, the less is available to spend on the website and your core business. Unless you’re a hosting company of course, but then this blog post isn’t really for you.

Today, everyone can build a global, high-traffic website even when you have little to none ops experience. AWS provides many different managed services that when put together correctly, form a powerful architecture that allows you to do so much without managing a single operating system or server. In other words: we’re going to build a serverless website.

A quick word on the term serverless before we move on. If you’re not familiar with the term and would like to know more, Martin Fowler has published a lengthy article on the subject. For the context of this blog post, I’d define serverless as a managed service that abstracts away the entire concept of the function of a server. This includes managing the hardware and operating system, as well as a webserver such as Nginx or Apache, autoscaling, and more. The service can either be a database, such as DynamoDB as we’ll see in this blog post, but also executing code such as AWS Lambda.

**4.2.** **The Website**

The website that we’ll building will show a history of the S3 storage types prices. The prices will be stored in DynamoDB, and we’ll makes sure to keep the prices up to date by subscribing to the SNS topic provided by AWS. This topic will sent a notification when any prices for AWS have changed. The following diagram shows all components and their relations (click to enlarge). Combined, we have everything we need to build our serverless website.

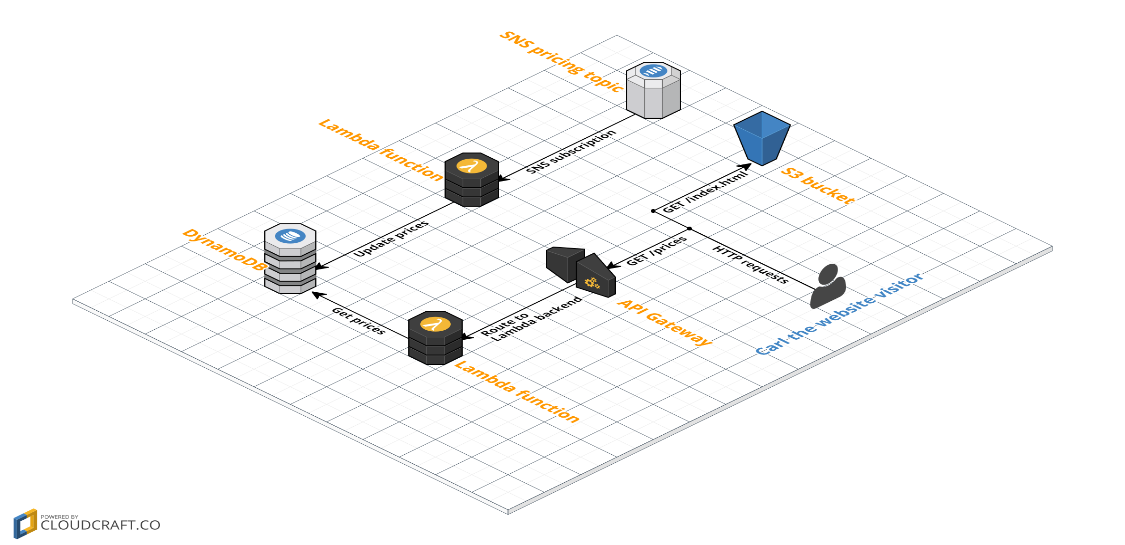
****

Fig-4.2.a Designing of Serverles Application

* **SNS pricing topic**. We will subscribe to an [SNS Topic](http://docs.aws.amazon.com/awsaccountbilling/latest/aboutv2/price-notification.html) that notifies us when AWS prices have been changed.
* **DynamoDB**. This is where we’ll store the pricing history for S3 storage.
* **Lambda functions**. We’ll use two different Lambda functions: one to add new prices to DynamoDB, one to fetch all prices.
* **API Gateway**. We’ll expose the Lambda function that fetches prices through a REST API. As we’ll see, this is easy with API Gateway.
* **S3**. We’ll use an S3 bucket to host our static HTML containing Javascript to grab and show the S3 storage prices.
* **Carl**. Carl is our website visitor. Carl is very anxious to learn about the history of S3 storage prices.

# **4.3. DynamoDB**

We will use DynamoDB, a managed NoSQL database. Setting it up is very easy and, just like all the components for our website, there is almost nothing that we will need to manage. The only thing we would have to manage is the capacity, which we shouldn’t set to high so we pay too much, but also not to low so clients’ requests will get throttled. For now though, the defaults will do fine.

Open up the [DynamoDB console](https://console.aws.amazon.com/dynamodb) and create a new table. Let’s call it s3\_storage\_prices. Set the primary key to id. Click Create to create the table.

We now have a DynamoDB table in which we can insert data. If you’re more experienced with an SQL database such as MySQL, you might expect that we need to create a schema. This is not required: as long as we add entries that have the id unique field, DynamoDB will accept whatever you put in it. With that said, it’s time to setup the Lambda functions that will put and get data to/from the table.

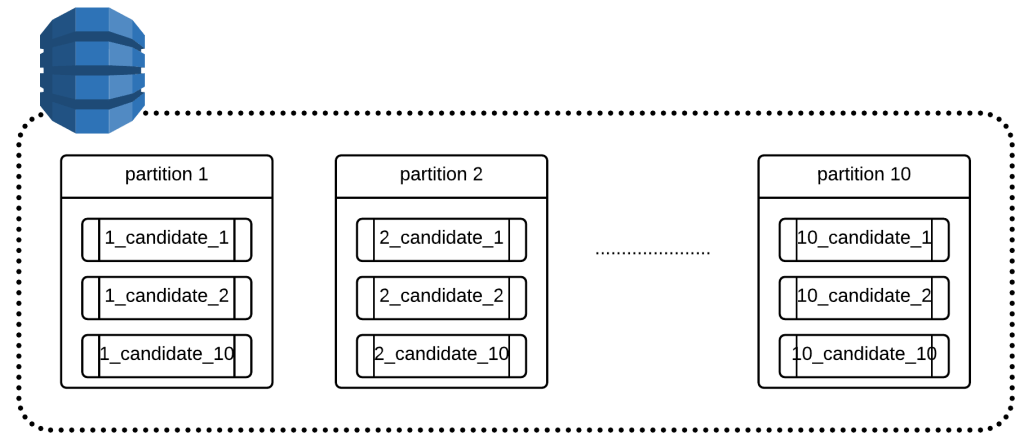


Fig-4.3.a AWS Dynamodb

# **4.4. Setup Lambda functions**

We glue together our components with two different Lambda functions. The first function will be invoked by the SNS topic when AWS notifies us of pricing changes. The function will then update the DynamoDB table to reflect the latest changes.

The second Lambda function will get the pricing information from DynamoDB and return it as a JSON array, to be processed by our frontend website that we built last.

Both functions are written for the NodeJS 6.10 runtime. They both need access to DynamoDB to get and insert the data from the table. Following the [least privilege strategy](http://docs.aws.amazon.com/IAM/latest/UserGuide/best-practices.html#grant-least-privilege) (granting exactly the permissions a function needs: no more and no less), we would create two separate IAM roles: and that can only read and one that can only write. For now though, let’s take a shortcut and create a single role that can do both. The following steps guide you through the process;

1. Open up the [IAM Console](https://console.aws.amazon.com/iam). On the left, click Roles and click the “Create new Role” button at the top.
2. Under the “AWS Service Role” tab, choose “AWS Lambda”. We’re going to create a role specifically for AWS Lambda.
3. Type “DynamoDB” in the search field now visible. Select the “AmazonDynamoDBFullAccess” policy (NOTE: this gives you much more permissions than required, so create a custom policy with only the required permissions if you’re doing this in a production environment).
4. Give the role a name such as “lambda-dynamodb-full-access” and click “Create role”.

## How to pass parameters or querystring from API Gateway to AWS Lambda? | by Aram Koukia | Koukia

Fig-4.4.a AWS Lambda

## **4.5. Application Architecture**

The application architecture uses  [AWS Lambda](https://aws.amazon.com/lambda/), [Amazon API Gateway](https://aws.amazon.com/api-gateway/), [Amazon DynamoDB](https://aws.amazon.com/dynamodb/), [Amazon Cognito](https://aws.amazon.com/cognito/), and [AWS Amplify Console](https://aws.amazon.com/amplify/). Amplify Console provides continuous deployment and hosting of the static web resources including HTML, CSS, JavaScript, and image files which are loaded in the user's browser. JavaScript executed in the browser sends and receives data from a public backend API built using Lambda and API Gateway. Amazon Cognito provides user management and authentication functions to secure the backend API. Finally, DynamoDB provides a persistence layer where data can be stored by the API's Lambda function.

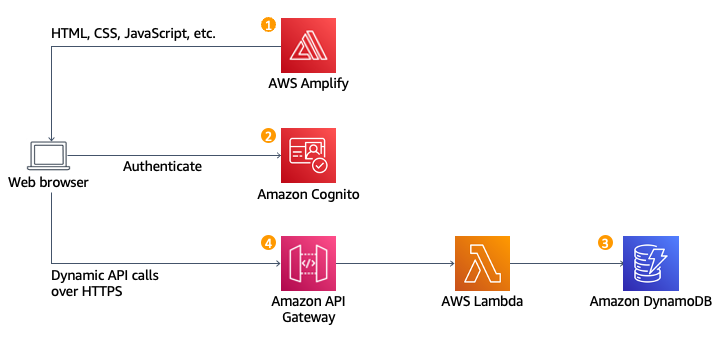


Fig-4.5.a Architechure

Static Web Hosting

AWS Amplify hosts static web resources including HTML, CSS, JavaScript, and image files which are loaded in the user's browser.

User Management

Amazon Cognito provides user management and authentication functions to secure the backend API.

**Serverless Backend**

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

**RESTful API**



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

Each service used in this architecture is eligible for the [AWS Free Tier](https://aws.amazon.com/free/?p=gsrc&c=ho_bswa). If you are outside the usage limits of the Free Tier, completing this tutorial will cost you less than $0.25\*.

Technologies used:

* An AWS account
* An [ArcGIS account](https://www.arcgis.com/home/signin.html) to add mapping to your app
* A text editor
* Recommended browser: The latest version of [Chrome](https://www.google.com/chrome/)
* [AWS Lambda](https://aws.amazon.com/lambda/)
* [Amazon API Gateway](https://aws.amazon.com/api-gateway/)
* [AWS Amplify](https://aws.amazon.com/amplify/)
* [Amazon DynamoDB](https://aws.amazon.com/dynamodb/)
* [Amazon Cognito](https://aws.amazon.com/cognito/)

**AWS Lambda**

AWS Lambda is a compute service that lets you run code without provisioning or managing servers.

Lambda runs your code on a high-availability compute infrastructure and performs all of the administration of the compute resources, including server and operating system maintenance, capacity provisioning and automatic scaling, and logging. With Lambda, all you need to do is supply your code in one of the language runtimes that Lambda supports.

**Amazon API Gateway**

Amazon API Gateway is an AWS service for creating, publishing, maintaining, monitoring, and securing REST, HTTP, and WebSocket APIs at any scale. API developers can create APIs that access AWS or other web services, as well as data stored in the [AWS Cloud](https://aws.amazon.com/what-is-cloud-computing/). As an API Gateway API developer, you can create APIs for use in your own client applications. Or you can make your APIs available to third-party app developers.

**AWS Amplify**

AWS amplify was released to help [front-end developers](https://intellipaat.com/blog/what-is-front-end-developer/) of both web and mobile applications convert their static applications to scalable full-stack applications. With the help of the AWS amplify, the developer can configure the app backends, connect their application in minutes and deploy static web apps in few clicks. It also enables the developer to manage the content outside the [**AWS console**](https://intellipaat.com/blog/what-is-amazon-aws-management-console/).

**Amazon DynamoDB**

Amazon DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability. DynamoDB lets you offload the administrative burdens of operating and scaling a distributed database so that you don't have to worry about hardware provisioning, setup and configuration, replication, software patching, or cluster scaling. DynamoDB also offers encryption at rest, which eliminates the operational burden and complexity involved in protecting sensitive data.

**Amazon Cognito**

Amazon Cognito is an identity platform for web and mobile apps. It’s a user directory, an authentication server, and an authorization service for OAuth 2.0 access tokens and AWS credentials. With Amazon Cognito, you can authenticate and authorize users from the built-in user directory, from your enterprise directory, and from consumer identity providers like Google and Facebook.

**5.** **RESULTS AND DISCUSSION**

**5.1. IMPLEMENTATION**

Implementing a serverless website on AWS typically involves utilizing services like AWS Lambda, Amazon S3, Amazon CloudFront, and Amazon Route 53. Here's a step-by-step guide to help you implement a serverless website on AWS:

1. **Static Website**: Begin by creating a static website, which consists of HTML, CSS, JavaScript, and any other assets your site requires. Ensure that your website is designed to work as a static site without any server-side processing.
2. **Amazon S3 Bucket**: Create an Amazon S3 bucket to host your static website files. Enable static website hosting on the bucket and configure the necessary permissions to allow public access to the files.
3. **Amazon CloudFront**: Set up an Amazon CloudFront distribution to serve your website's content globally. Create a new CloudFront distribution and configure the origin as your S3 bucket. CloudFront acts as a Content Delivery Network (CDN) to improve the performance and availability of your site.
4. **Amazon Route 53**: If you want to associate a custom domain name with your serverless website, use Amazon Route 53 to manage DNS records. Create a hosted zone for your domain and set up a new record pointing to your CloudFront distribution.
5. **AWS Lambda**: If your serverless website requires dynamic functionality or server-side processing, you can use AWS Lambda functions. For example, you might use Lambda to handle contact form submissions or process user authentication. Write your Lambda functions in the programming language of your choice and deploy them using the AWS Lambda service.
6. **API Gateway**: To connect your Lambda functions to your website, use Amazon API Gateway. Create an API in API Gateway and define the necessary routes and methods to trigger your Lambda functions. You can integrate API Gateway with CloudFront to serve your API endpoints and manage authorization and throttling.
7. **Optional Database**: If your serverless website requires persistent data storage, you can use Amazon DynamoDB or other AWS database services. DynamoDB is a NoSQL database that integrates well with serverless architectures. Configure the necessary permissions for your Lambda functions to access the database.
8. **Security and Permissions**: Configure appropriate security measures for your serverless website. Utilize AWS Identity and Access Management (IAM) to create IAM roles and policies to control access to AWS resources. Apply secure coding practices and implement necessary security measures to protect your website from common threats.
9. **Deployment and Monitoring**: Use AWS CloudFormation or other deployment tools like AWS Amplify or the Serverless Framework to manage the deployment of your serverless website. Monitor your website's performance, latency, and error rates using AWS CloudWatch and other monitoring services.

Top of Form


          Architecture diagram for a REST microservice. Client icon connects through an arrow to REST API resource icon with API Gateway service icon placed above it. REST API is connected by a double arrow to Lambda function resource icon with Permissions Policy resource icon placed above it, and Lambda service icon placed above both. Lambda function resource is connected through an arrow pointing to Users Table resource with DynamoDB service icon placed above it. Two dotted vertical lines separate the three services.
        

Fig-5.1.a Implementatiom of Serverless Web Application

**Serverless fundamentals learning path as text**

The following is a text representation of the key concepts in the preceeding diagram.

**Event Driven Architecture**

* Events are sent to services. Services produce additional events.
* Event 1 is processed by two services (A and B). Service A produces 'event 2' which is processed by Service B.

**Workshop: Intro to Serverless**

* Connect a REST API to a Lambda function to retrieve data from DynamoDB
* Services: API Gateway, Lambda, DynamoDB

**Development Workflow** **Create & Deploy**

* Management Console (essential)
* Infrastructure as Code (IaC)
* Note: The console gives you full manual control.
* Infrastructure as code topics and services: AWS SAM (important), AWS CloudFormation(essential), AWS CDK
* Note: Automate provisioning, code deploy, and more!

**Invoke & Test**

* Synchronous (essential)
* Asynchronous

**Update & Package**

* AWS SAM CLI can automate updates, packaging, and local testing.
* Packaging can include containers and layers - see Lambda for more info

**Secure**

* Least-priviledge
* Execution Role
* Resource-based Policy (RBAC)
* Services: IAM
* Note: If you start with less restrictive development permissions, always audit and use least-privilege for production.

**Monitor**

* Logs, Errors
* Services: CloudWatch, X-Ray

**Workshop: Synchronous Invoke**

* Note: Dive deeper into synchronous invocation with Serverless Workshop Module 2 and IaC using Serverless Application Model (AWS SAM).
* Services: API Gateway, Lambda, DynamoDB + AWS SAM CLI

**Development workflow**

Development workflow is similar regardless of programming language, tooling, or architecture. After you pick a language, app framework, and set up your local workstation, the DevSecOps cycle begins:

* **Plan** new features
* **Code** new features/fixes — write code, build, test locally, commit
* **Build** code continuously
* **Test** in staging environment
* **Release** to production
* **Deploy** infrastructure and updated code package
* **Operate** - sanity test, configure
* **Monitor** - scale, optimize, debug

**Serverless learning path**

The following is a text representation of the key concepts in the preceeding diagram.

Amazon Web Services Account (Essential)

* AWS Identity and Access Management: Root account, Admin account, Least-privilege
* Note: AWS Identity and Access Management (IAM) controls who is authenticated and authorized to use resources. IAM applies to developer accounts, roles, and resources in other services.

Programming Language

* Python (Important), JavaScript / TypeScript (Important)
* Java, C#, Go
* Ruby, PowerShell

Development Environment

* Local workstation or Cloud 9 IDE
* AWS CLI (Important) to create and manage services from command line
* AWS SAM CLI (Important) to speed up develop/deploy/test cycles from command line

**AWS Cloud Infrastructure**

* Region
* Availability Zones
* Management Console
* ARN - Amazon Resource Names uniquely identify AWS resources
* Note: Each region has multiple Availability Zones (AZs) with one or more datacenters.
* Tip: Pick a region close to your customers!

**Security Model**

* Shared Responsibility Model
* Security “of” the Cloud - Amazon is responsible for the security “of” the cloud, such as data center physical security.
* Security “in” the Cloud - You are responsible for security “in” the Cloud, such as your data, functions, and integrations. (Essential)

**Amazon Web Services Account**

Before getting started, you must have or create an Amazon Web Services (AWS) account.

If you are creating a new account, you will create a root account using an email address. The **root** account has unrestricted access, similar to root accounts for an operating system. As a best practice, you should create an administrative user too.

Warning: As you might imagine, granting administrative access to a user is still rather broad. An account with administrative level privileges will make getting started easier, but, for production systems, you should revisit and follow the principle of least-privilege. This means granting only the minimum access necessary to accomplish tasks.

* [Signing in to the AWS Management Console](https://docs.aws.amazon.com/signin/latest/userguide/console-sign-in-tutorials.html) provides an excellent step by step guide to account types and login steps.
* AWS Identity and Access Management (IAM) is the service to manage entities and resources authorized to use services and service resources.
* Sign up for an AWS account
* Create an administrative user

**Programming Language**

Amazon Web Service solutions are built for software developers. We assume that you have some experience with coding and deploying programs using one of the supported languages. This guide will **not** teach you how to program, but it will at times provide code samples.

Writing functions in an interpreted language like Python or JavaScript is easier because the code can be added directly through the Management Console web interface.

You can use one of the listed languages, or you can bring your own runtime container.

* Python, JavaScript / TypeScript — commonly used interpreted languages
* Java, C#, Go - compiled languages
* Ruby, PowerShell - less frequently used options

**Summary**

* You need an Amazon Web Services account to get started.
* Python and JavaScript/Typescript are popular programming languages for serverless. You will see these most frequently in examples, tutorials, and workshops.
* Java, C#, Go, Ruby, and PowerShell are available runtimes, but you can also bring your own.
* Set up your development environment with your preferred local IDE
* AWS data centers are organized into one or more Availability Zones located in multiple **regions** across the globe
* Region codes and ARNs are used to identify and connect to specific AWS services and resources
* Responsibility for security of serverless solutions is shared between you and AWS.

**5.2. Results:**

Implementing a serverless website on AWS offers several benefits and advantages:

1. 1.Scalability: Serverless architectures on AWS can automatically scale to handle varying levels of traffic and demand. Services like AWS Lambda and Amazon S3 automatically scale up or down based on the incoming requests, ensuring your website can handle high traffic without manual intervention.
2. Cost-effectiveness: With serverless architecture, you only pay for the actual usage of resources. AWS Lambda, for example, charges you based on the number of invocations and the duration of function execution, allowing you to optimize costs by avoiding idle server time. Additionally, services like Amazon S3 and CloudFront have cost-effective pricing models for hosting and delivering your website's static assets.
3. 3.Reduced Operational Overhead: Serverless architecture eliminates the need for managing and provisioning servers or infrastructure. AWS takes care of underlying server management, operating system updates, and capacity planning, allowing you to focus on developing your website and application logic.
4. 4.Improved Performance: Leveraging services like Amazon CloudFront, your website's content is cached at edge locations globally. This enables faster content delivery to users, reducing latency and improving the overall performance and user experience.
5. 5.High Availability: AWS provides high availability by default for serverless services. AWS Lambda and other serverless components are distributed across multiple availability zones, ensuring that your website remains available even if one zone experiences a failure. Additionally, CloudFront's global presence helps distribute traffic and minimize downtime.
6. 6.Flexibility and Agility: Serverless architectures allow you to quickly iterate and deploy updates to your website. You can deploy new features or fixes to individual functions without impacting the entire application, promoting agility and faster time-to-market.
7. 7.Integration with AWS Services: AWS offers a wide range of services that can be easily integrated into your serverless website. You can leverage services like Amazon DynamoDB for database storage, Amazon SES for email notifications, Amazon SNS for push notifications, and more, enhancing the functionality and capabilities of your website.
8. 8.Simplified Development: Serverless architectures promote modular and decoupled development. You can break down your application into smaller, more manageable functions, making it easier to develop, test, and maintain. Additionally, the use of AWS Lambda allows you to write functions in multiple programming languages, providing flexibility for your development team.
9. Overall, implementing a serverless website on AWS can provide scalability, cost savings, reduced operational overhead, improved performance, and a flexible development environment. It enables you to focus on building and delivering your website's core features while AWS handles the infrastructure management and scaling aspects.

**6.EXECUTION**

**>Html code**

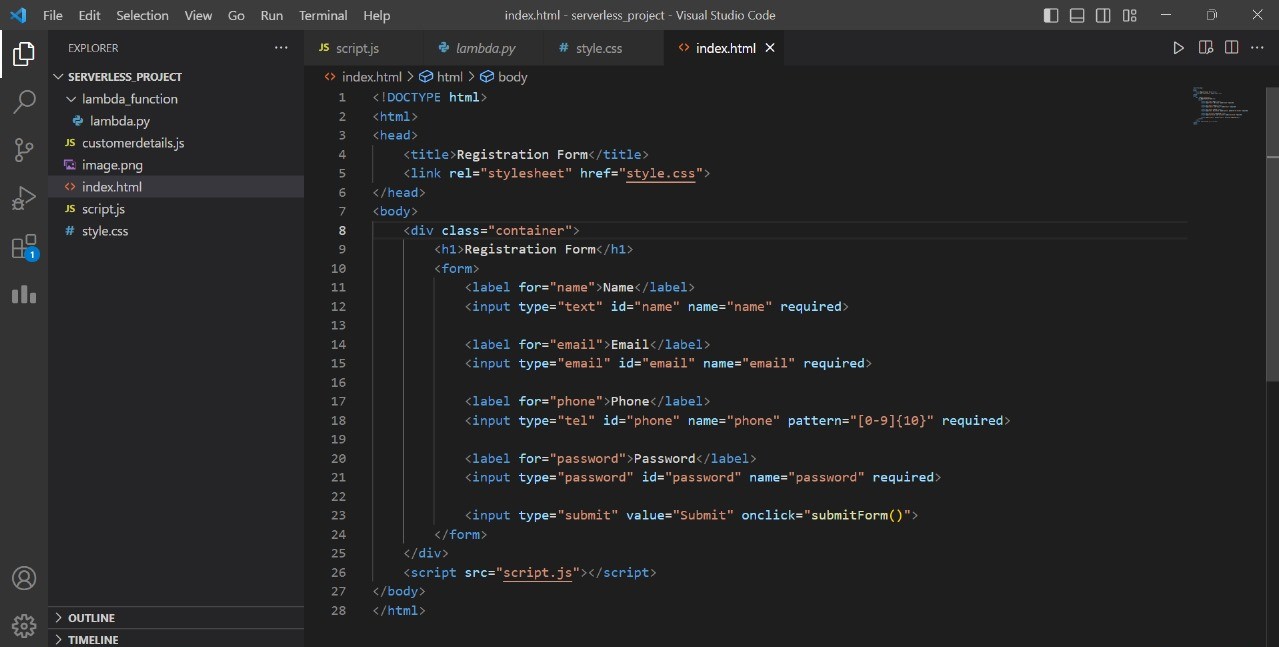
**s**

Fig-6.a Html code

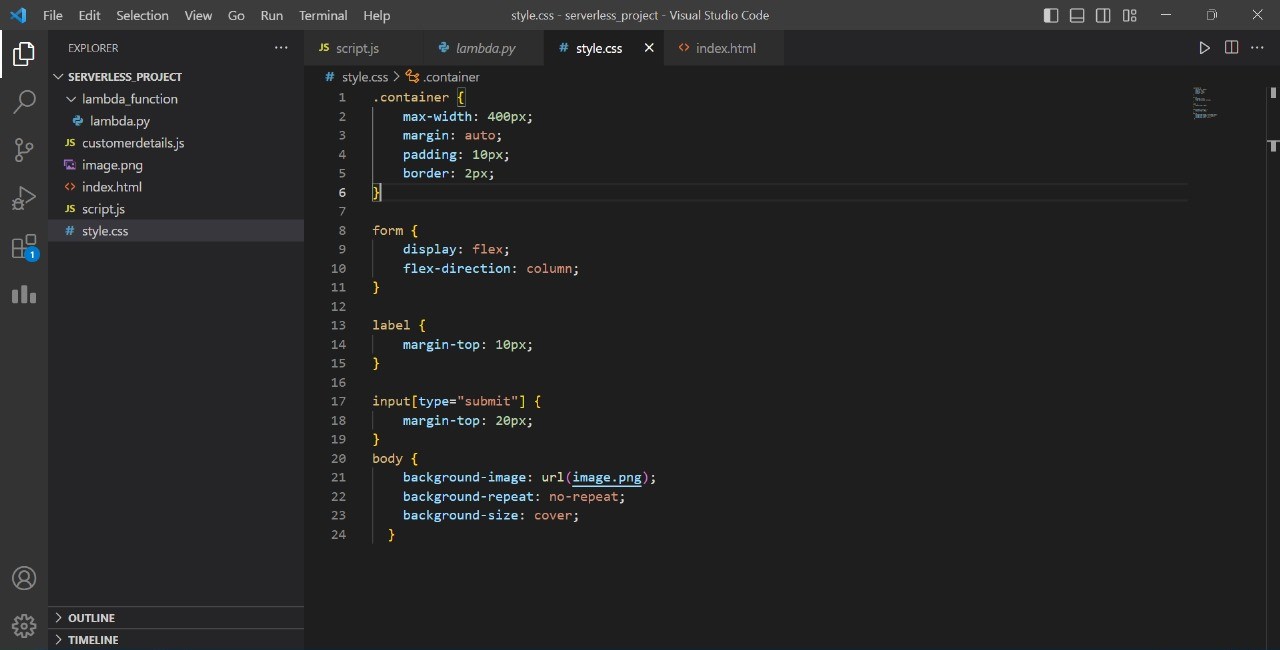
**>CSS code**

Fig-6.b CSS code

**>JavaScript code**

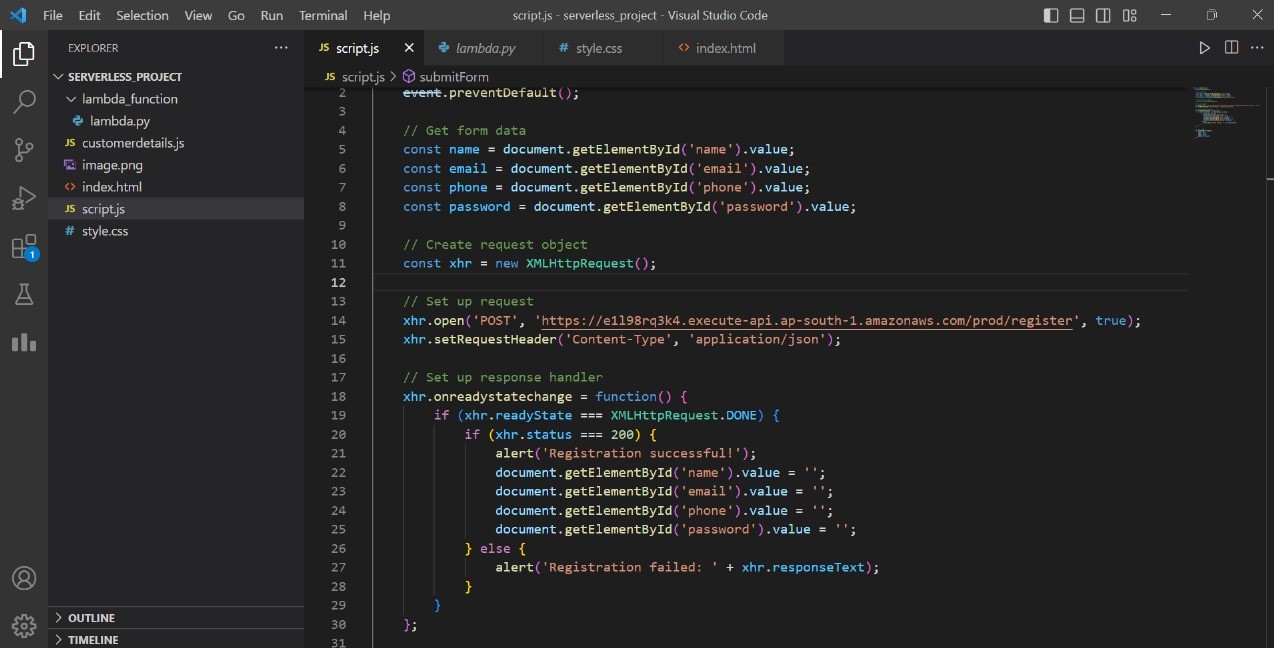
****

Fig-6.c JavaScript Code

**>Python code**

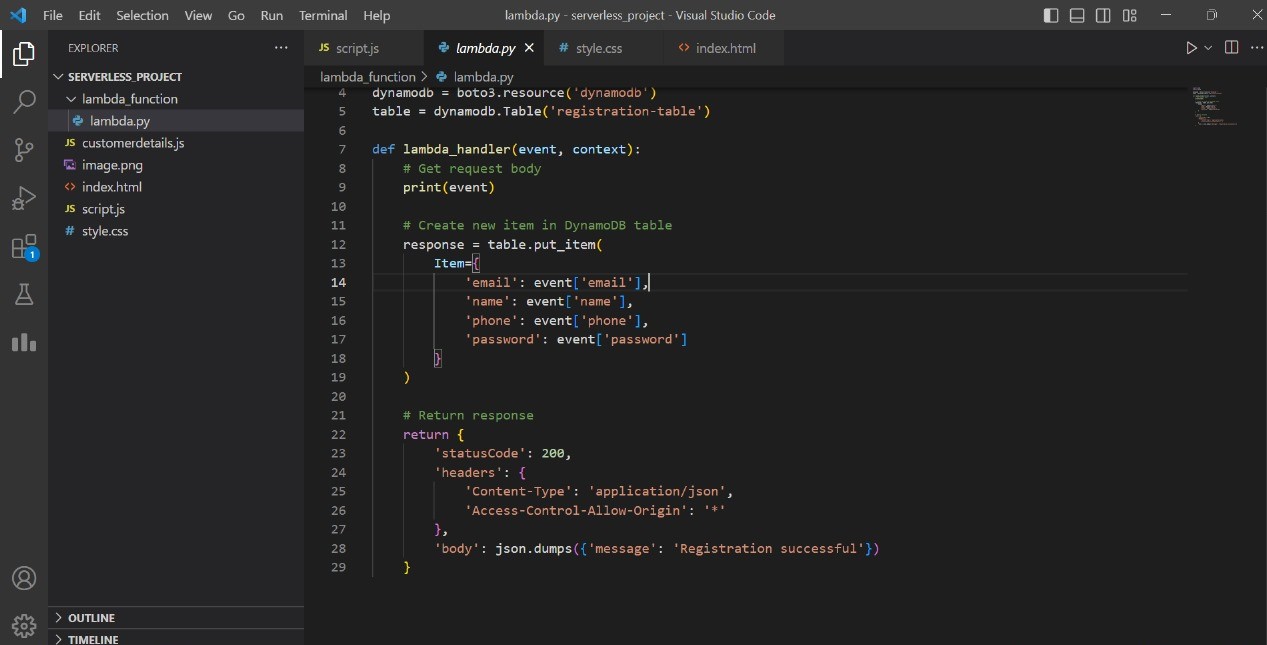
****

Fig-6.d Python Code

**>Registration form screen**

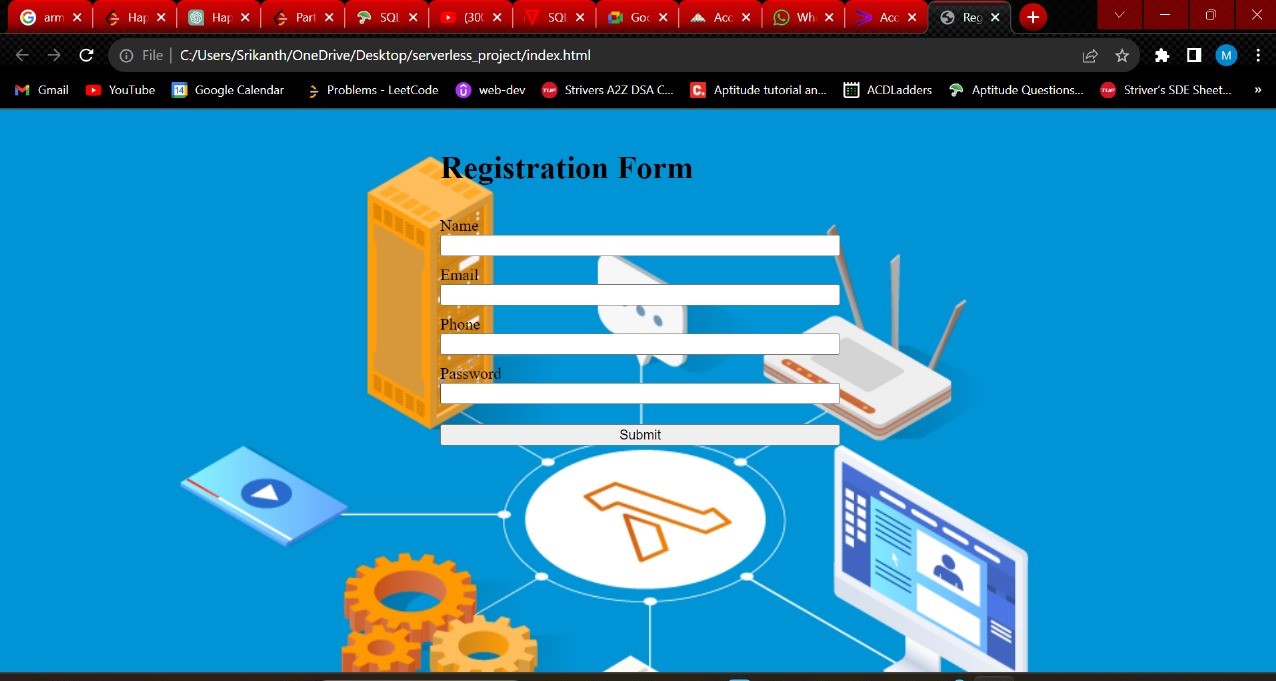
****

Fig-6.e Registration form

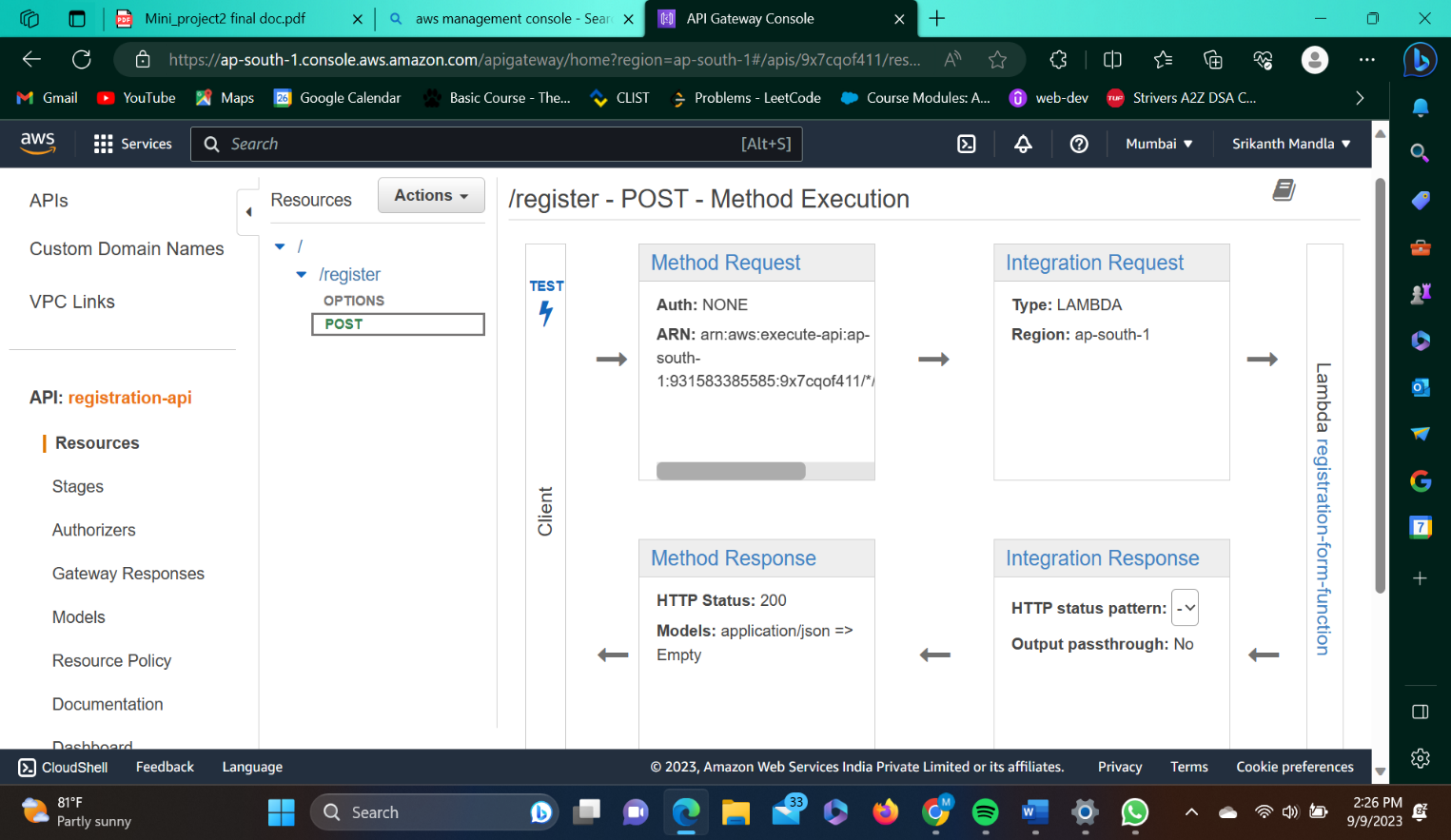
****

Fig-6.f API Gateway

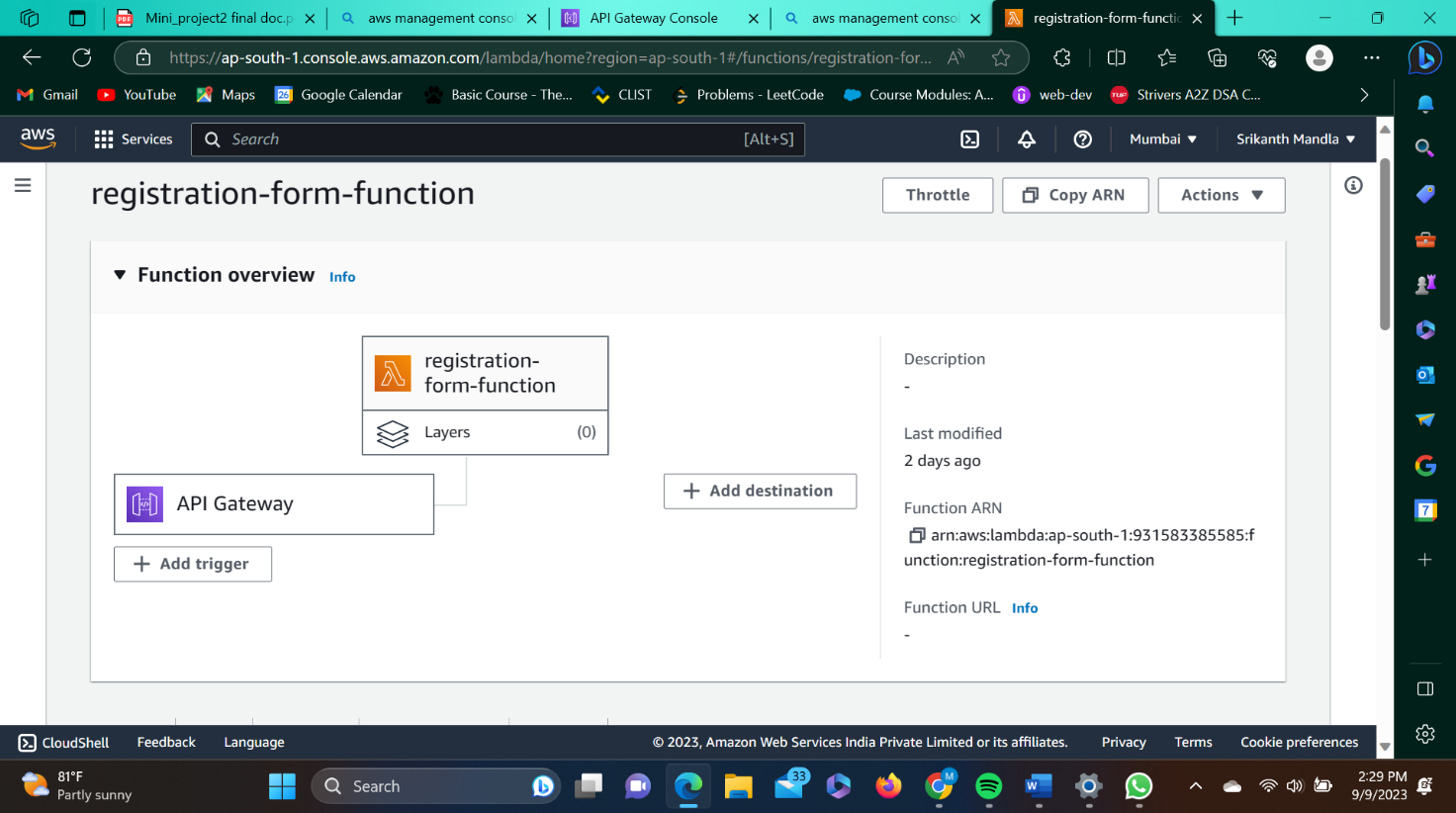
****

Fig-6.g AWS Lambda Function

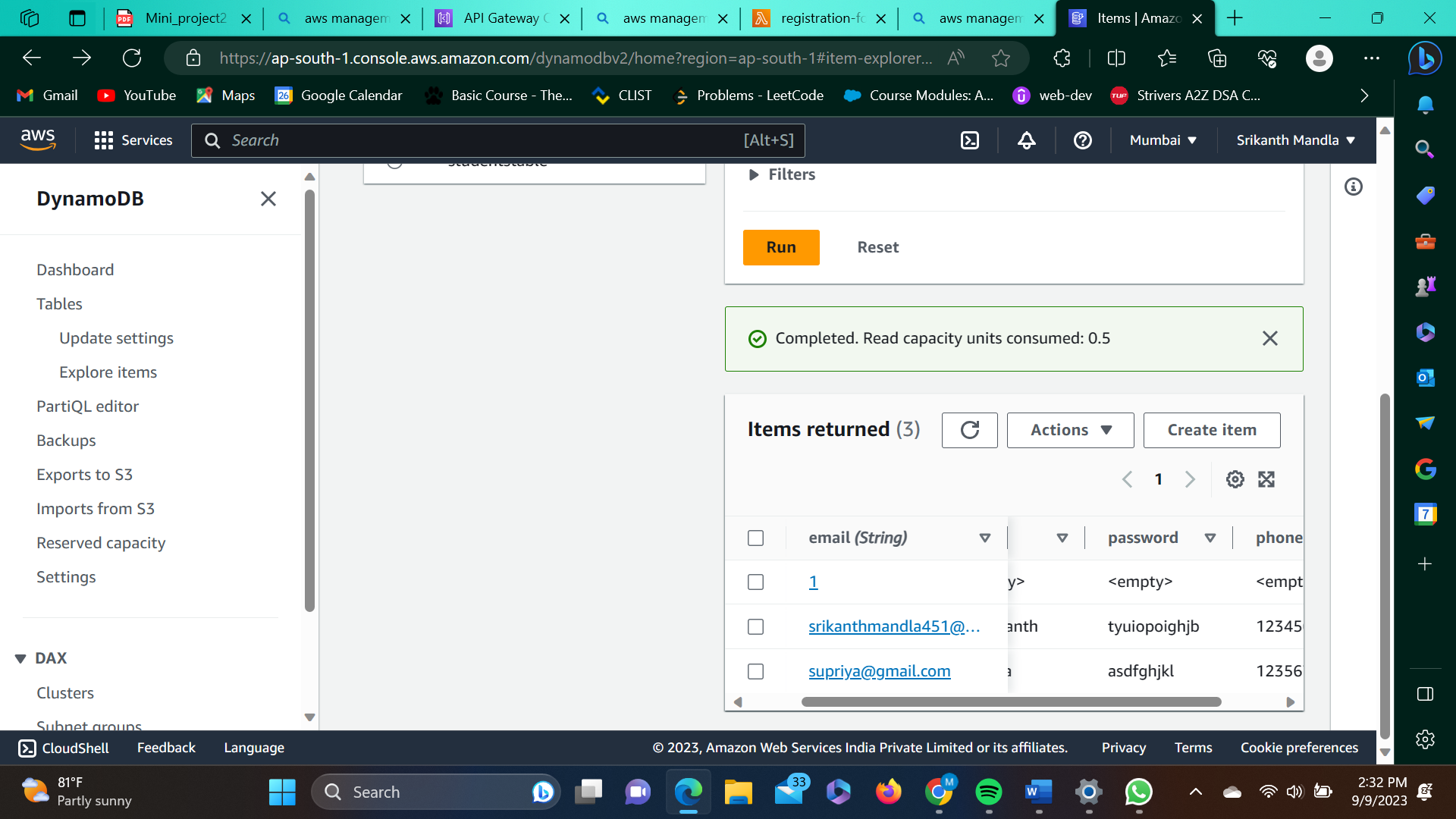


Fig-6.h Dynamodb

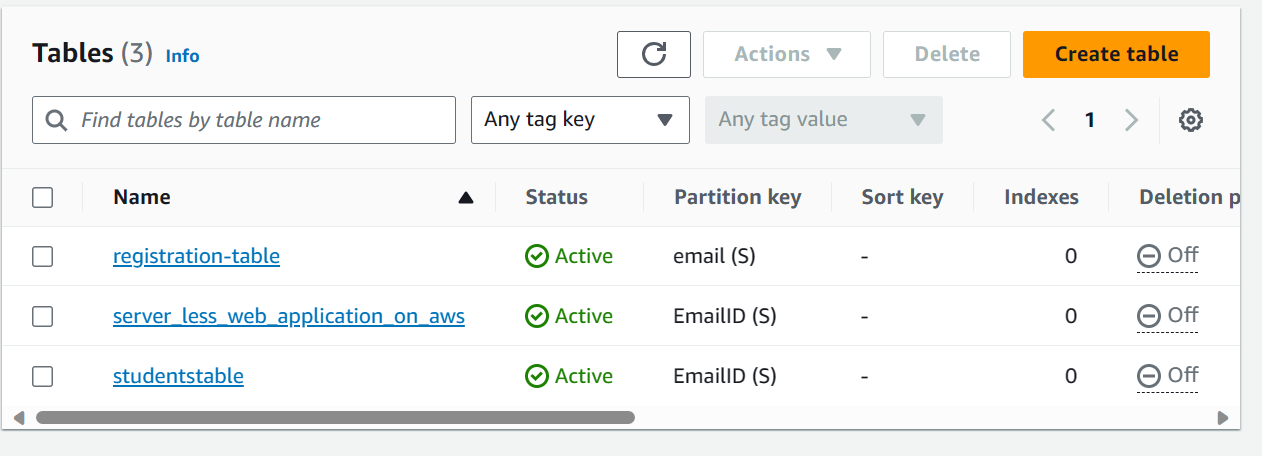
****

Fig-6.i Tables in Dynamodb

**7. CONCLUSION AND REFERENCES**

**7.1. CONCLUSION**

In conclusion, building a serverless website using AWS offers numerous advantages over traditional hosting models. By leveraging AWS services like AWS Lambda, Amazon S3, API Gateway, and others, developers can create highly scalable, cost-efficient, and easily manageable websites. The serverless architecture eliminates the need for server management, allowing developers to focus on coding and accelerating development cycles. Throughout the project, the proposed methodologies guide the development process, covering requirements gathering, architecture design, front-end and back-end development, database integration, API development, deployment, security, testing, and documentation. These methodologies ensure a systematic and efficient approach to building a serverless website.

**7.2. FUTURE WORK**

Future Work for a Serverless Website Using AWS:

1. Enhancing Functionality: Continuously work on adding new features and functionalities to the serverless website based on user feedback and evolving requirements. Explore additional AWS services that can be integrated, such as Amazon Cognito for user authentication or AWS Step Functions for building serverless workflows.
2. Optimizing Performance: Monitor and analyze the performance of the serverless website to identify areas for optimization. Implement performance tuning techniques, leverage caching mechanisms, and explore AWS services like AWS Elasticache for further performance improvements. Conduct load testing to validate the website's performance under heavy traffic conditions.
3. Advanced Security Measures: Strengthen the security of the serverless website by implementing advanced security measures. Consider using AWS Web Application Firewall (WAF) for protection against common web exploits and DDoS attacks. Implement additional encryption methods, such as AWS Key Management Service (KMS) for managing encryption keys.

**7.3. REFERENCES**

* <https://docs.aws.amazon.com/serverless/latest/devguide/serverless-dev-workflow.html>
* <https://chat.openai.com/>
* <https://www.pluralsight.com/guides/building-a-serverless-web-app-on-aws-services>
* <https://aws.amazon.com/blogs/compute/creating-serverless-applications-with-the-aws-cloud-development-kit/>
* <https://amonkincloud.com/>