Internship Project Report

Serverless Personal To-Do List Application

Cover Page

Title: Serverless Personal To-Do List Application

Name: Nikhil

Internship Organization: 1Stop.ai

Date: 22nd June 2025

Abstract

- In today's rapidly evolving digital ecosystem, cloud-native applications are becoming the backbone of modern software development. Traditional client-server architectures often require extensive resource management, cost overhead, and lack scalability for dynamic workloads. To address these challenges, this project presents a **Serverless Personal To-Do List Application** leveraging **Amazon Web Services (AWS)** to provide an efficient, scalable, and cost-effective task management solution.

- The core objective of this project is to design and develop a personal to-do list application that utilizes serverless computing principles to eliminate the need for server provisioning and maintenance. The application is built using a suite of AWS services, including **AWS Lambda** for backend compute logic, **Amazon API Gateway** for exposing RESTful APIs, and **Amazon DynamoDB** as a NoSQL database for persistent task storage. The frontend is optionally hosted on **Amazon S3**, providing a static and lightweight user interface.

- This system supports full **CRUD operations** (Create, Read, Update, Delete) on tasks, with Lambda functions written in JavaScript (Node.js). Each function is triggered via API Gateway endpoints, and securely accesses DynamoDB using **IAM roles and policies**, ensuring a robust security model. The serverless architecture enables the application to **auto-scale** based on incoming requests and follows a **pay-per-use** billing model, reducing both cost and complexity.

- Additionally, this project introduces a simple, intuitive user interface, allowing users to add, edit, view, and delete tasks with real-time feedback. The application architecture is designed with **modularity**, enabling easy future extensions such as user authentication, reminders, and notifications.

- The implementation successfully demonstrates the effectiveness of serverless computing for real-world applications, showing measurable benefits in terms of scalability, performance, and development efficiency. This project also serves as a foundational model for students and professionals aiming to explore **cloud-native application development** using serverless technologies.

- Overall, the Serverless Personal To-Do List Application represents a practical, efficient, and modern solution for task management, while showcasing the real-world applicability of AWS-based serverless design patterns in software engineering.

- To understand and implement serverless architecture using AWS.

- To build a scalable and cost-effective personal To-Do list application.

- To use Lambda for compute, API Gateway for communication, and DynamoDB for

storage.

- To gain hands-on experience with AWS services
  .
- To manage infrastructure as code using AWS SAM or the Serverless Framework.

Objective

The objective of this project is to **design and implement a serverless cloud-based To-Do List Application** using the Amazon Web Services (AWS) ecosystem. The project focuses on leveraging serverless technologies to develop a scalable, cost-efficient, and maintenance-free task management system that supports essential CRUD operations.

The goals of this project are as follows:

- Eliminate server management overhead by using AWS Lambda to run backend logic on-demand, without provisioning or maintaining infrastructure.
- Expose RESTful APIs through Amazon API Gateway to handle Create, Read, Update, and Delete operations on task data.
- Implement a fast and scalable NoSQL database using Amazon DynamoDB to store and manage to-do list items securely and efficiently.
- Follow modular and event-driven architecture, making the application flexible for future enhancements like user authentication, reminders, or analytics.
- Utilize modern cloud development tools such as AWS SAM or the Serverless Framework to deploy infrastructure as code and manage cloud resources programmatically.
- Perform end-to-end testing of all API endpoints using Postman to ensure reliability, data integrity, and proper error handling.
- Gain hands-on experience in serverless cloud-native application development, aligning with current industry trends and best practices.

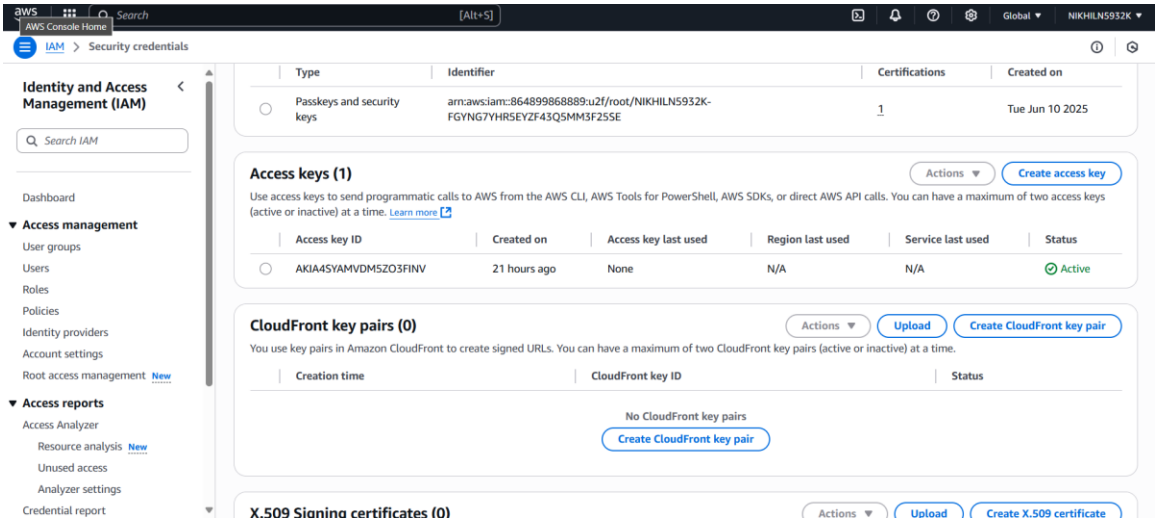
Introduction

In the evolving landscape of modern software development, serverless architecture has emerged as a revolutionary approach to building scalable and cost-efficient applications. Unlike traditional models that require developers to provision, manage, and monitor servers, serverless architecture abstracts infrastructure concerns entirely. This enables developers to focus purely on writing business logic while cloud providers handle operational aspects such as availability, fault tolerance, and auto-scaling. In essence, serverless computing shifts infrastructure responsibilities to platforms like **Amazon Web Services (AWS)**, allowing for more agile and efficient development workflows.

At the heart of serverless computing is **AWS Lambda**, a Function-as-a-Service (FaaS) offering that executes code in response to events and automatically manages the compute resources required. With services like **API Gateway** for routing HTTP requests and **DynamoDB** for data persistence, developers can build fully functional backend systems without managing any servers. These services scale automatically based on traffic and only incur charges for the actual execution time, making serverless solutions highly economical.

What is Serverless Architecture?

Serverless architecture allows developers to build and deploy applications without worrying about server provisioning or infrastructure management. In a serverless environment, the cloud provider dynamically allocates resources as needed, scaling up or down automatically in response to application demands. This architecture model promotes rapid development and operational efficiency by minimizing overhead.



Access key – IAM - user

Key Benefits of Serverless Computing:

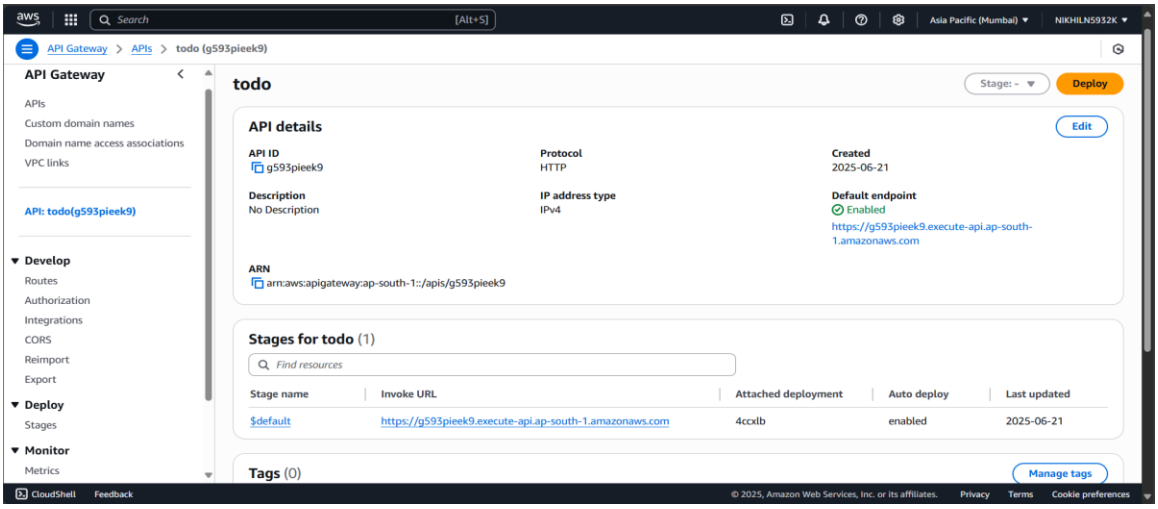
- **No Server Management:** Developers don't need to worry about provisioning or maintaining servers.
- **Built-in Scalability:** Applications automatically scale with traffic.
- **Reduced Operational Cost:** Pay-as-you-go pricing model significantly cuts down cost.
- **Faster Deployments:** Developers can ship features quickly by focusing solely on application logic.

Common Use Cases:

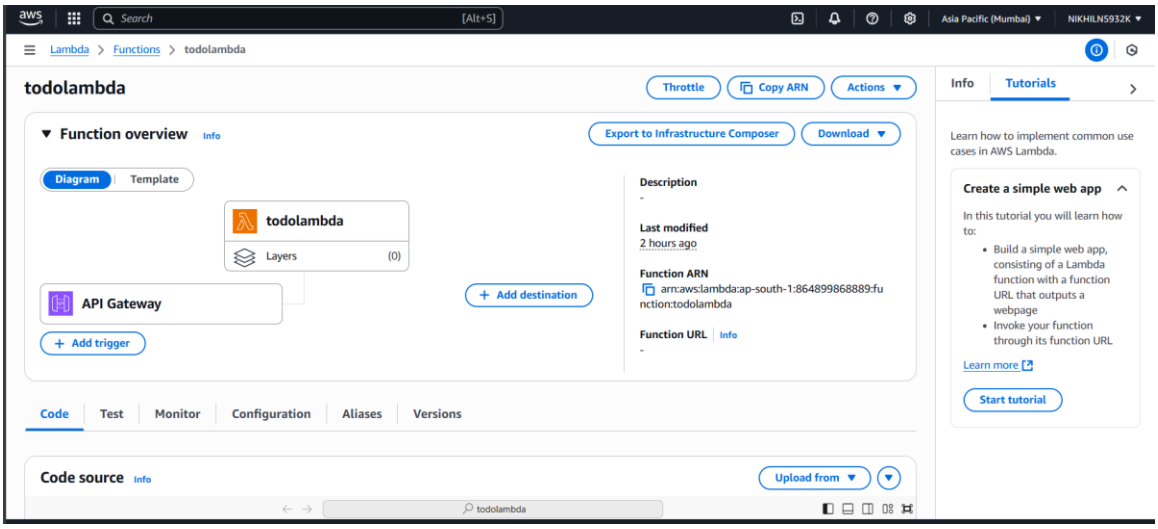
- **Real-time File Processing:** Automatically process uploaded files (e.g., images, logs).
- **Web Applications:** Backend logic for full-stack apps without managing infrastructure.
- **APIs and Microservices:** Easily create and deploy modular, scalable APIs.
- **IoT Backends:** Handle data ingestion from IoT devices in real time.

Project Relevance

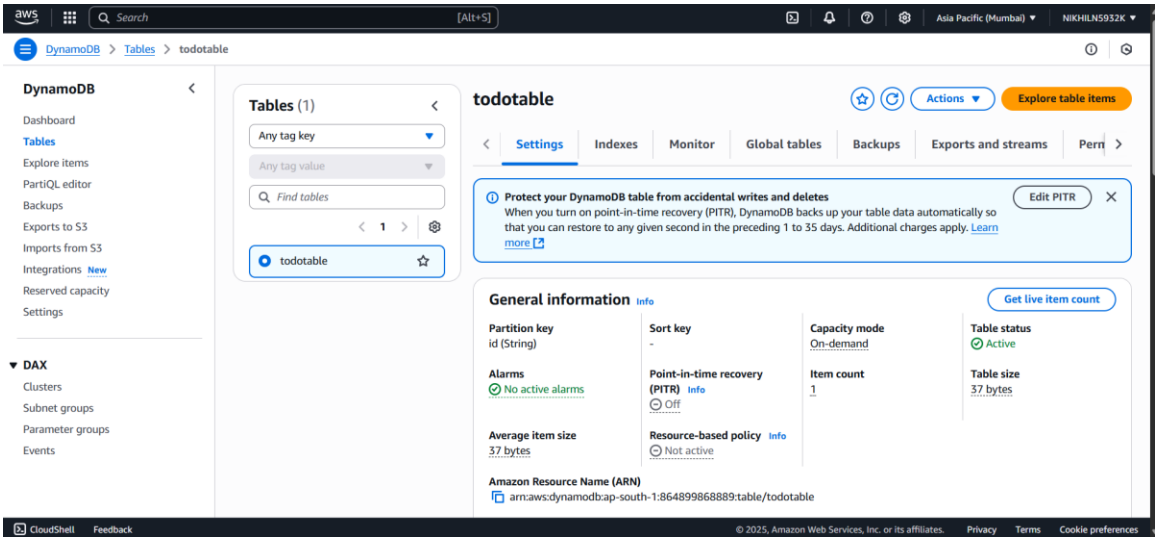
This project leverages the principles of serverless computing to build a **Personal To-Do List Application** that performs all core CRUD (Create, Read, Update, Delete) operations without the need for a traditional backend server. By using AWS Lambda for logic execution, Amazon API Gateway for routing requests, and Amazon DynamoDB for data storage, the application offers a fully cloud-native experience. The serverless design ensures automatic scalability, low maintenance, and minimal operational costs, making it an ideal solution for both personal productivity tools and enterprise-grade systems.



API – gateway used



Lambda – function



DynamoDB – table

Methodology

This project follows a serverless approach to building a cloud-native to-do list application, utilizing various AWS services and modern development tools. The methodology involves selecting the appropriate cloud components, defining API endpoints, implementing Lambda functions for backend logic, and testing the functionality using API testing tools. The following subsections detail the technologies used and the working architecture.

Tools and Technologies Used

To achieve the desired serverless functionality, the following tools and technologies were employed:

- **AWS Lambda**: Acts as the backend compute engine that executes code in response to HTTP requests without provisioning servers.
- **Amazon API Gateway**: Used to create and manage RESTful APIs that expose endpoints to interact with the Lambda functions.
- **Amazon DynamoDB**: A fully managed NoSQL database service used to store to-do tasks with fast performance and seamless scalability.
- **AWS SAM / Serverless Framework**: Used for packaging, deploying, and managing the infrastructure-as-code. It simplifies deployment and version control.
- **Postman**: A popular API client used to test HTTP requests and verify the functionality of each endpoint during development.

Workflow Overview

The development workflow involved the following key stages:

1. **API Design**: The API routes were planned following REST conventions. Endpoints like `POST /tasks`, `GET /tasks`, `PUT /tasks/{id}`, and `DELETE /tasks/{id}` were defined.
2. **Lambda Function Development**: Separate AWS Lambda functions were written in Node.js to handle each CRUD operation. These functions were configured with proper IAM roles for accessing DynamoDB securely.
3. **Database Schema**: DynamoDB was used with a simple table schema where each to-do task includes an `id`, `task`, and optional `status` or `timestamp`.
4. **Infrastructure Deployment**: AWS SAM or the Serverless Framework was used to package the application, define the API Gateway integrations, and deploy the stack.

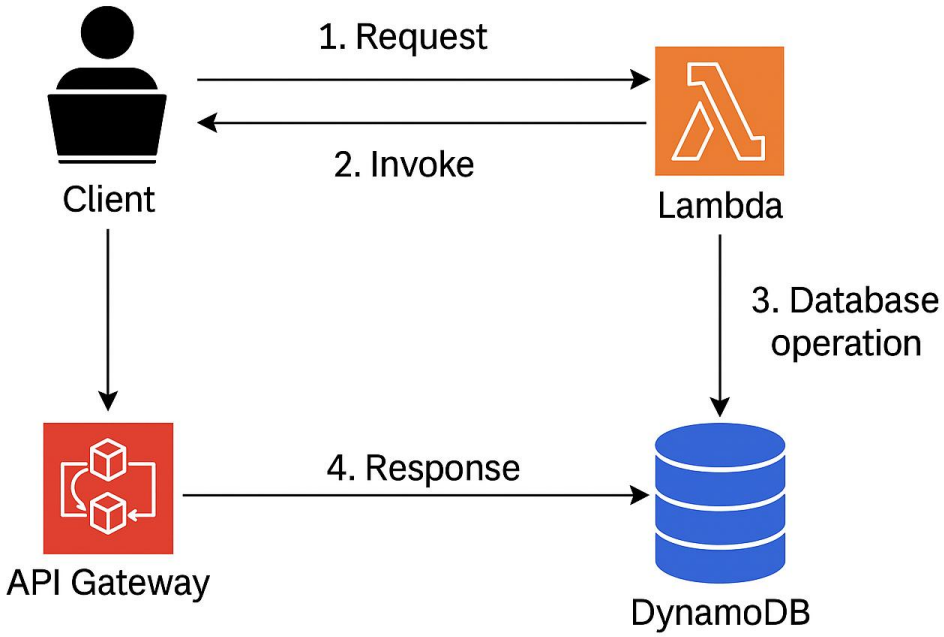
5. **Testing and Debugging**: Postman was used to test endpoints and validate request/response handling. AWS CloudWatch was used for debugging through logs.

Architecture Overview

The architecture of the application follows a modular and event-driven pattern using AWS-managed services. The steps below summarize the overall flow:

1. **Client Request**: The user initiates a request from a frontend or API client (like Postman).
2. **API Gateway Invocation**: The request hits an endpoint exposed by Amazon API Gateway.
3. **Lambda Function Triggered**: Based on the route and HTTP method, API Gateway invokes the corresponding Lambda function.
4. **Database Operation**: The Lambda function connects to DynamoDB and performs the relevant CRUD operation.
5. **Response Returned**: The Lambda function returns the output, which is relayed to the client via API Gateway.

Architecture Diagram:



Architecture Flowchart

Application Workflow

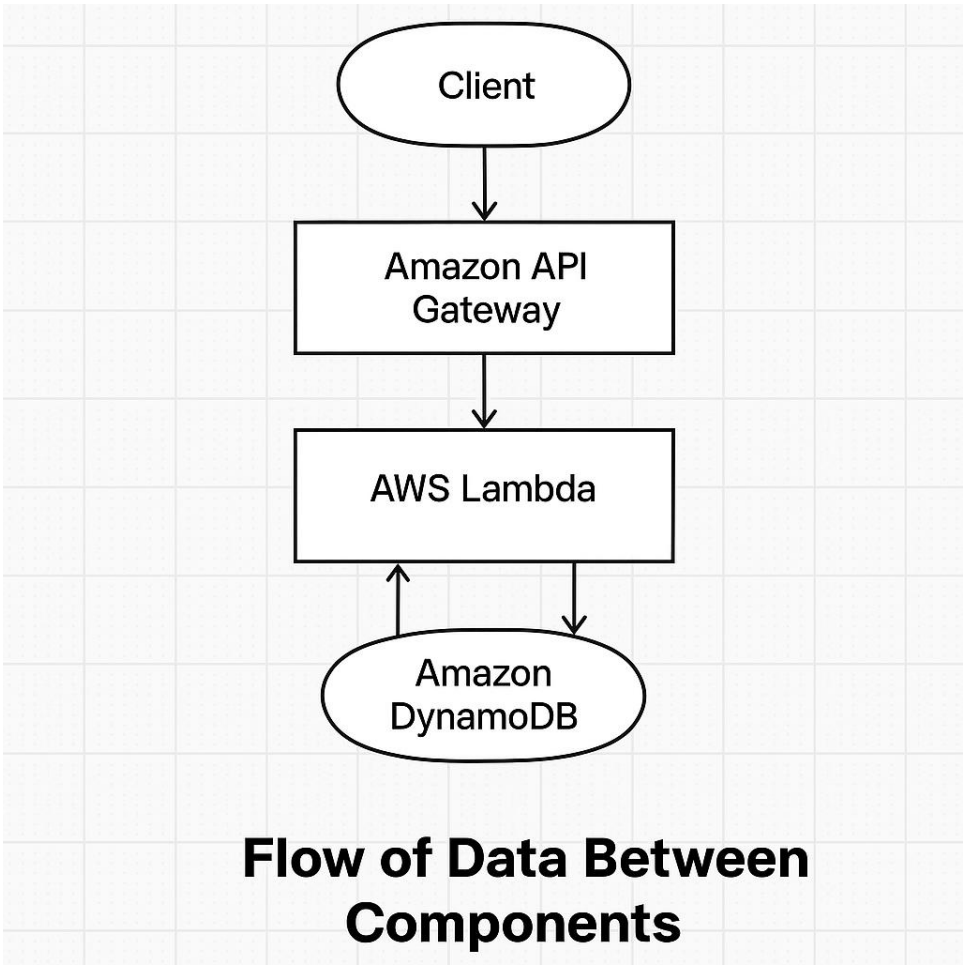
POST /todo – Add a new task to the list

GET /todo – Retrieve all tasks

PUT /todo – Update a task's details

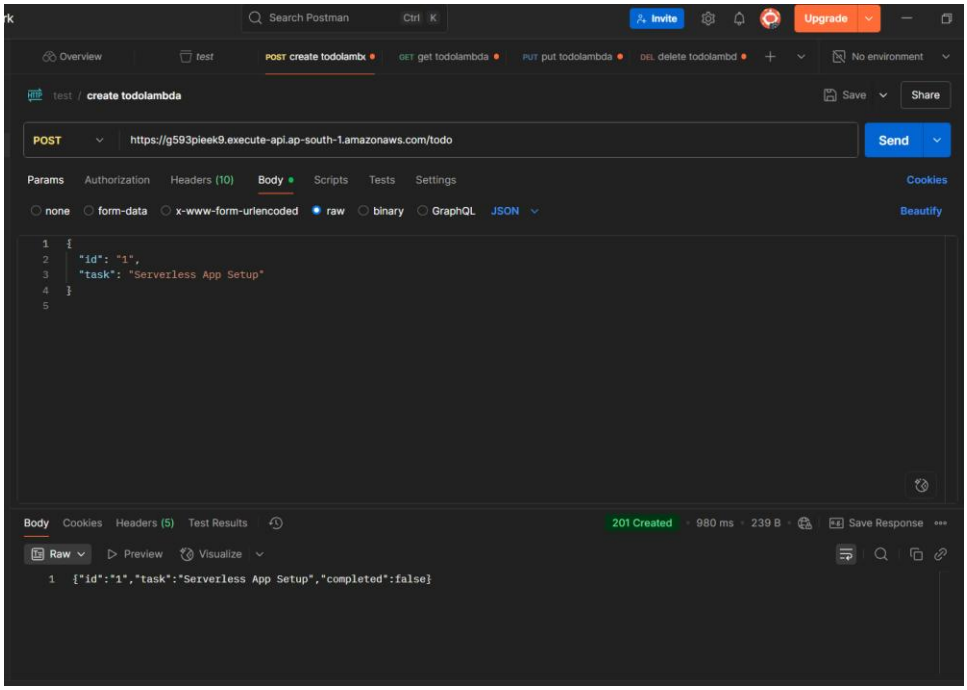
DELETE /todo?id=1 – Delete a task by ID

Flowchart:

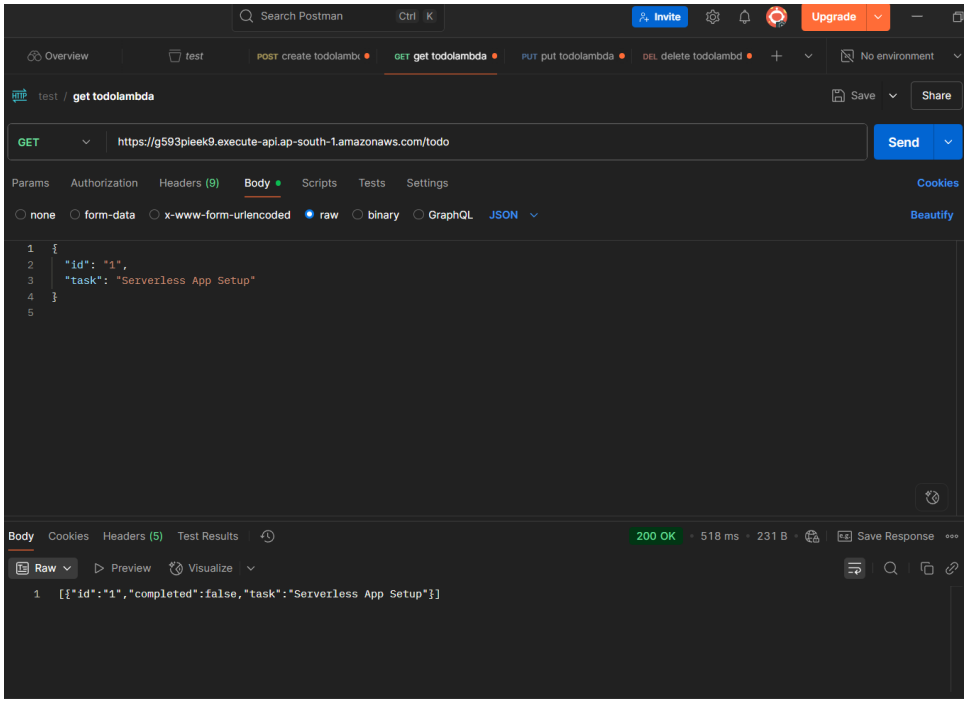


Architecture Flowchart

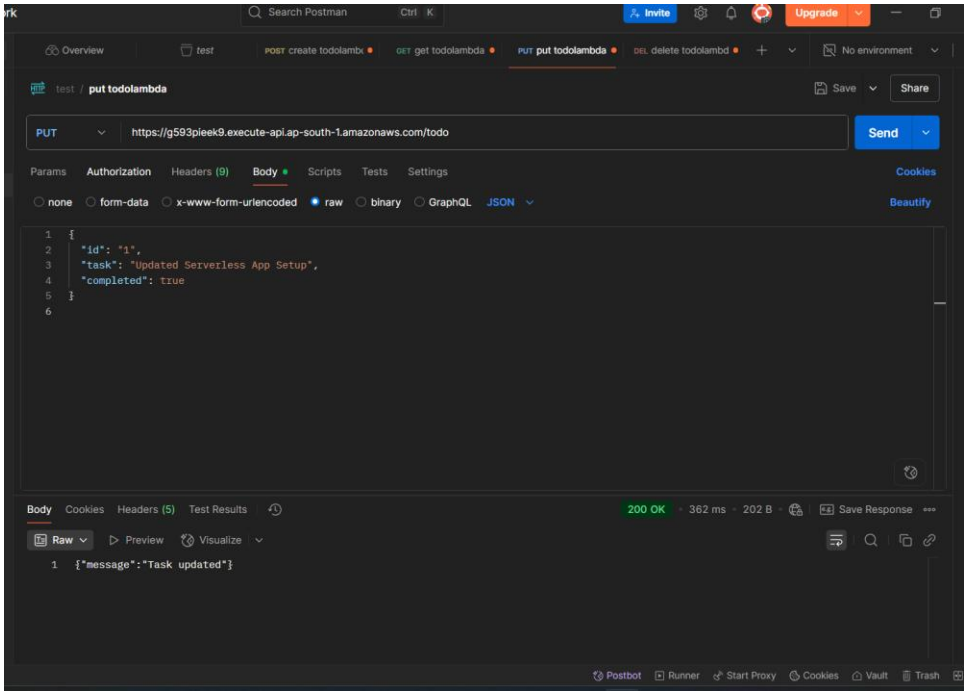
Screenshots of Results / Outputs



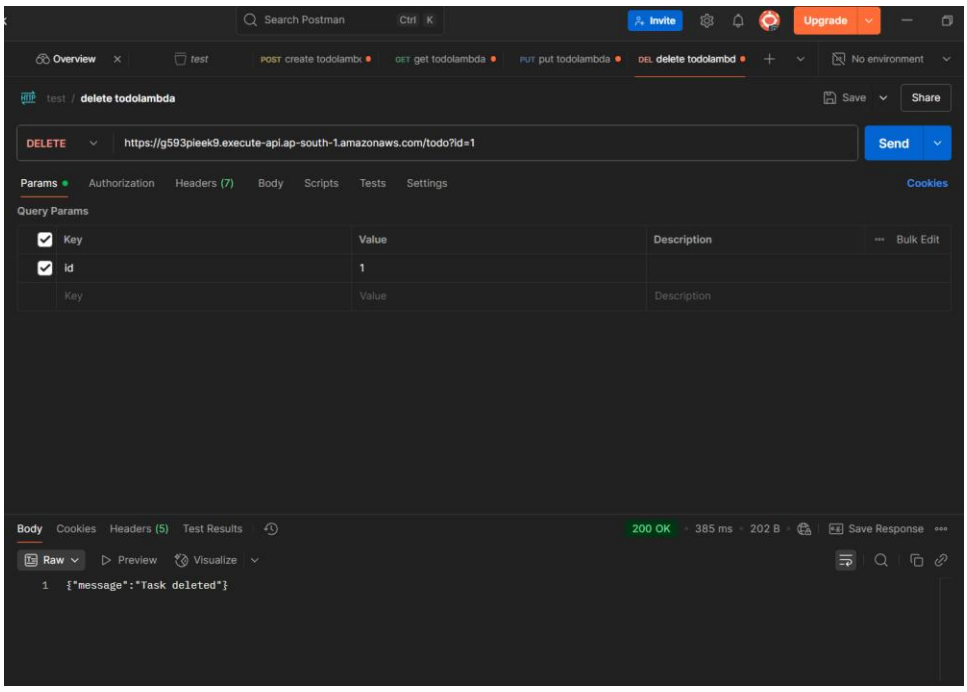
POST Task Screenshot



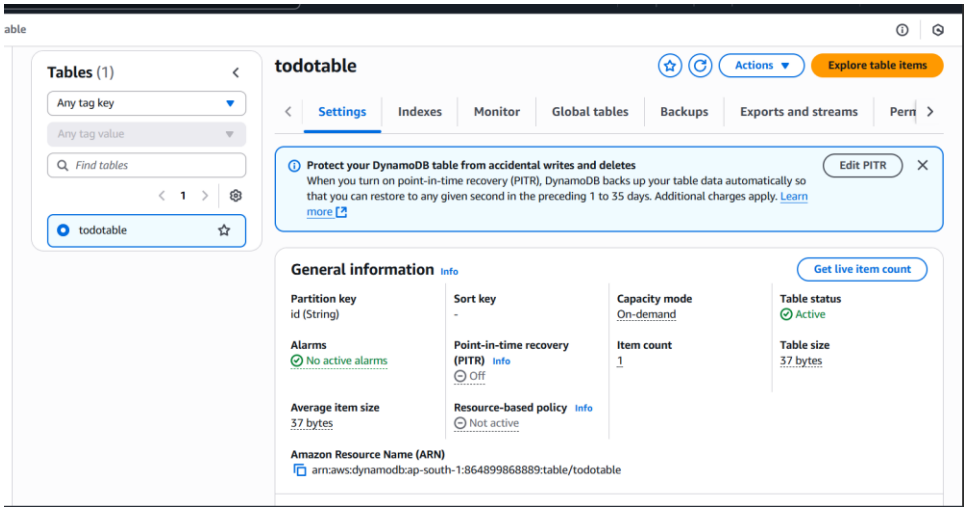
GET Task Screenshot



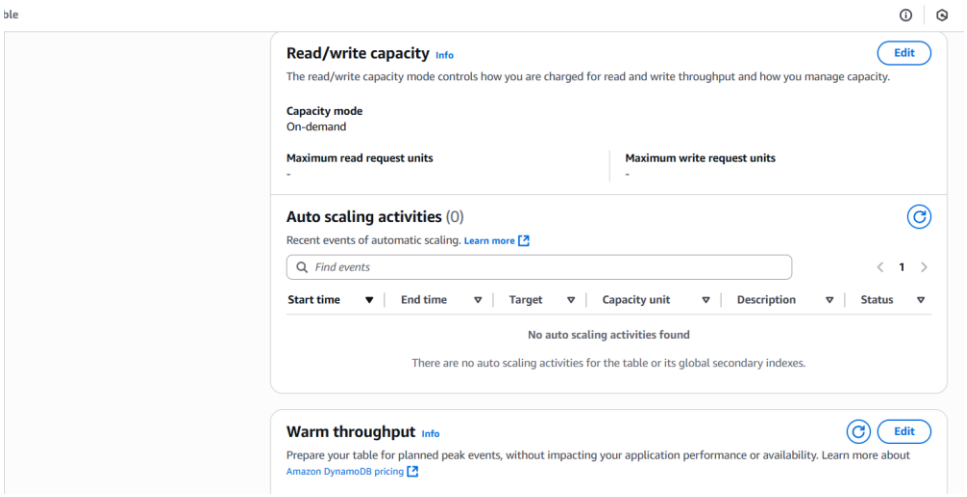
PUT Task Screenshot



DELETE Task Screenshot



DynamoDB Screenshot 1



DynamoDB Screenshot 2

Code Implementation

This project uses a single AWS Lambda function to handle all CRUD operations—Create, Read, Update, and Delete—on a DynamoDB table named `todotable`. The function is triggered via HTTP methods (`POST`, `GET`, `PUT`, `DELETE`) through Amazon API Gateway. The logic inside the Lambda uses the **`event.requestContext.http.method`** to determine the operation type and execute the appropriate DynamoDB query.

```
const AWS = require('aws-sdk');

const dynamoDB = new AWS.DynamoDB.DocumentClient();

const TABLE_NAME = "todotable";

exports.handler = async (event) => {

  const httpMethod = event.requestContext.http.method;

  const body = event.body ? JSON.parse(event.body) : {};

  const queryParams = event.queryStringParameters || {};

  switch (httpMethod) {


    // CREATE

    case "POST":

      if (!body.id || !body.task) {

        return {

          statusCode: 400,

          body: JSON.stringify({ message: "Invalid Request: Missing 'id' or 'task'" })

        };

      }

      const newTask = {

        id: body.id,

        task: body.task,

        completed: false

      };
```

```
    await dynamoDB.put({
        TableName: TABLE_NAME,
        Item: newTask
    }).promise();
    return {
        statusCode: 201,
        body: JSON.stringify(newTask)
    };

// READ
case "GET":
    const tasks = await dynamoDB.scan({
        TableName: TABLE_NAME
    }).promise();
    return {
        statusCode: 200,
        body: JSON.stringify(tasks.Items)
    };

// UPDATE
case "PUT":
    if (!body.id || body.task === undefined || body.completed === undefined) {
        return {
            statusCode: 400,
            body: JSON.stringify({ message: "Missing 'id', 'task', or 'completed'" })
```

```
      };
    }

    await dynamoDB.update({
      TableName: TABLE_NAME,
      Key: { id: body.id },
      UpdateExpression: "set task = :t, completed = :c",
      ExpressionAttributeValues: {
        ":t": body.task,
        ":c": body.completed
      }
    }).promise();
    return {
      statusCode: 200,
      body: JSON.stringify({ message: "Task updated" })
    };

  // DELETE
  case "DELETE":
    if (!queryParams.id) {
      return {
        statusCode: 400,
        body: JSON.stringify({ message: "Missing 'id' in query string" })
      };
    }
```

```
        await dynamoDB.delete({
            TableName: TABLE_NAME,
            Key: { id: queryParams.id }
        }).promise();

        return {
            statusCode: 200,
            body: JSON.stringify({ message: "Task deleted" })
        };

    // Unsupported method
    default:
        return {
            statusCode: 405,
            body: JSON.stringify({ message: `Method ${httpMethod} not allowed` })
        };
  }
};
```

Detailed Code Explanation

The Lambda function in this project is designed to handle all CRUD (Create, Read, Update, Delete) operations inside a **single file**, based on the incoming **HTTP method**. Below is a breakdown of each key part of the code:

---

1. AWS SDK Initialization

```
javascript
CopyEdit
const AWS = require('aws-sdk');
const dynamoDB = new AWS.DynamoDB.DocumentClient();
```

- The AWS SDK is imported to interact with AWS services.

- `DocumentClient` is used to perform high-level operations on **DynamoDB** using standard JavaScript objects instead of raw JSON.

---

2. Table Configuration

```
javascript
CopyEdit
const TABLE_NAME = "todotable";
```

- This defines the DynamoDB table name being used for storing tasks.
- The table is assumed to have `id` as its **primary key**.

---

3. HTTP Method Routing

```
javascript
CopyEdit
const httpMethod = event.requestContext.http.method;
```

- Extracts the HTTP method (GET, POST, PUT, DELETE) from the request.
- This determines which section of the `switch-case` block will execute.

---

4. Create Task – POST

```
javascript
CopyEdit
case "POST":
```

- Checks if both `id` and `task` are provided.
- If valid, it creates a new item in the DynamoDB table with default `completed: false`.
- Returns HTTP **201 Created** status with the new task.

---

5. Read Tasks – GET

```
javascript
CopyEdit
case "GET":
```

- Uses the `scan()` operation to fetch all items from the DynamoDB table.
- Returns HTTP **200 OK** with the full list of tasks.
- No input is required in the body for this operation.

---

6. Update Task – PUT

```
javascript
CopyEdit
case "PUT":
```

- Validates that `id`, `task`, and `completed` are all present in the request body.
- Uses `update()` to modify the task in DynamoDB.
- The update expression modifies both the task text and the completed status.
- Returns HTTP **200 OK** with a success message.

---

7. Delete Task – DELETE

```
javascript
CopyEdit
case "DELETE":
```

- Extracts the task `id` from the **query string parameters** (e.g., `/tasks?id=123`).
- Uses the `delete()` operation to remove the task from DynamoDB.
- Returns HTTP **200 OK** if successful.

---

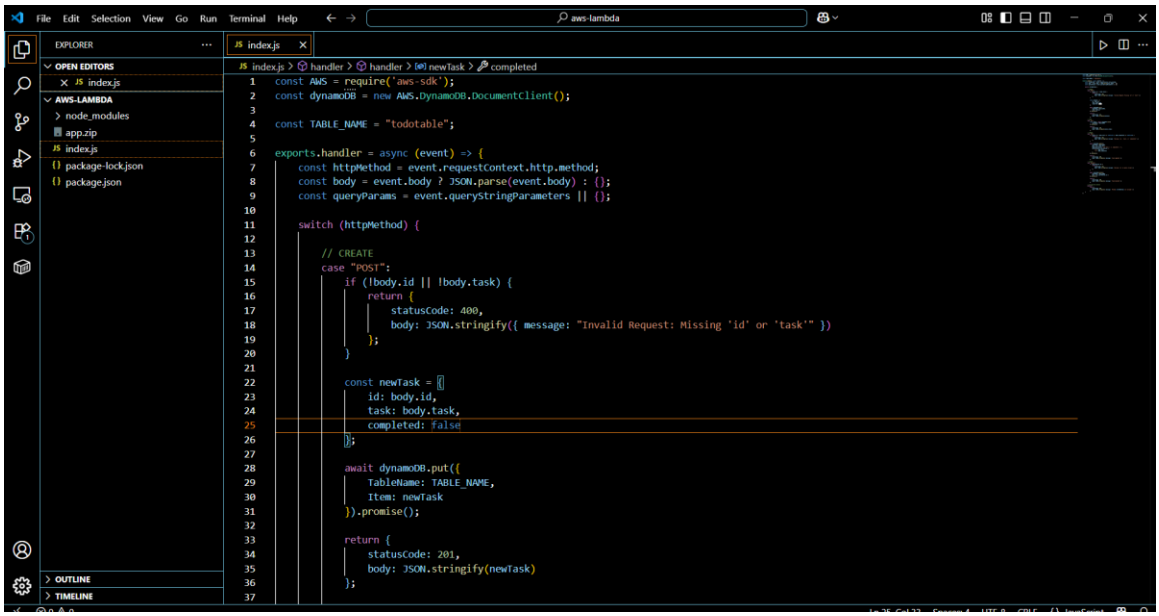
8. Error Handling – Unsupported Methods

```
javascript
CopyEdit
default:
```

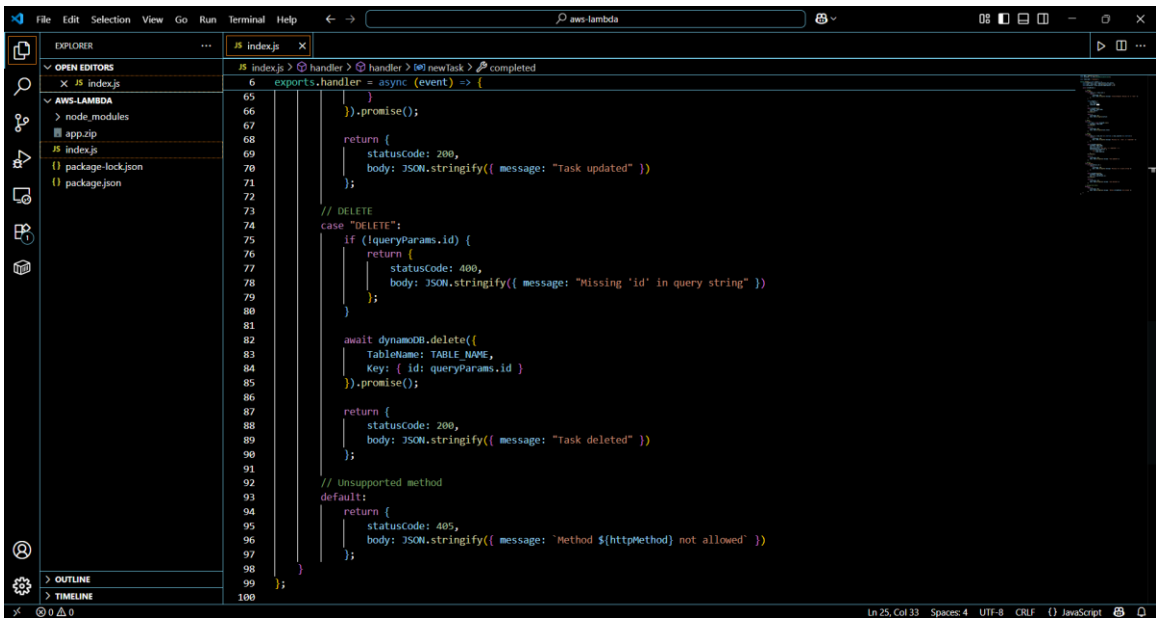
- If any HTTP method other than GET, POST, PUT, or DELETE is used, a **405 Method Not Allowed** response is returned.

Additional Notes:

- Basic validation is included (e.g., missing fields), but can be improved by adding a **schema validator** like Joi.
- You could log `event` and error details to **CloudWatch** for better observability.
- This function is **stateless**—it doesn't store anything in memory, which fits perfectly with serverless best practices.



Code snippet picture 1



Code snippet picture 2

Conclusion

This project successfully resulted in the development and deployment of a fully functional Serverless To-Do List Application using core AWS services such as Lambda, API Gateway, and DynamoDB. The implementation showcased the practical advantages of serverless computing—most notably, zero server management, automatic scalability, cost efficiency, and event-driven execution.

The project provided valuable hands-on experience in building and deploying cloud-native applications. By working directly with AWS Lambda to write backend logic, Amazon API Gateway to expose secure and scalable RESTful APIs, and Amazon DynamoDB for persistent storage, the application achieved its primary objective: to demonstrate how a task management system can be efficiently built using a serverless approach.

During the development process, several important concepts were reinforced, including:

- Setting up and securing AWS resources using **IAM roles and policies**,
- Designing and testing RESTful APIs,
- Understanding the **event-driven workflow** between services,
- Performing real-time debugging and monitoring using **AWS CloudWatch**.

However, the project was not without its challenges. One of the main difficulties was configuring proper **IAM permissions** to allow secure yet functional interactions between Lambda and DynamoDB. Additionally, managing event structure differences between local testing and actual API Gateway payloads required careful debugging. These challenges provided meaningful learning experiences in troubleshooting serverless applications.

The current version of the application supports basic **CRUD operations** through HTTP methods and offers a clean backend interface. While this meets the scope of the initial objective, there is significant room for further development.

References

1. AWS Lambda Documentation - https://docs.aws.amazon.com/lambda/
2. Amazon API Gateway - https://docs.aws.amazon.com/apigateway/
3. DynamoDB Developer Guide - https://docs.aws.amazon.com/amazondynamodb/
4. Serverless Framework - https://www.serverless.com/framework/docs/
5. AWS SAM - https://docs.aws.amazon.com/serverless-application-model/
6. Postman - https://www.postman.com/