

Creating a Serverless Personal To-Do List Application : A Comprehensive Guide

Report By: Keshav Kumar

Date: 2nd May 2025

ABSTRACT

This report presents the end-to-end design, development, and deployment of a Serverless Personal To-Do List Application using Amazon Web Services (AWS). The solution adopts a fully serverless architecture to ensure scalability, high availability, and minimal operational overhead. AWS Lambda is used to handle the backend logic, Amazon API Gateway provides RESTful API interfaces for client interaction, and Amazon DynamoDB serves as the persistent NoSQL data store.

Infrastructure and deployment are managed through Infrastructure as Code (IaC) using AWS Serverless Application Model (SAM) or the Serverless Framework, enabling consistent, version-controlled deployments. The application supports standard CRUD operations—Create, Read, Update, and Delete—validated through Postman, as illustrated by the successful execution of task creation and deletion endpoints returning HTTP status codes 201 Created and 200 OK, respectively.

These test results confirm the robustness of the system and its readiness for real-world use. This project exemplifies the potential of serverless computing to deliver low-maintenance, cost-effective, and scalable applications for modern cloud-native workloads.

Key Highlights :

- 1. Fully Serverless Architecture -**
Built using AWS Lambda, API Gateway, and DynamoDB—offering auto-scaling and zero server maintenance.
- 2. Infrastructure as Code (IaC) –**
Deployment via AWS Serverless Framework ensuring consistency, automation, and portability.
- 3. Basic CRUD Support -**
Enables users to add, view, update, and delete tasks with clear API response handling.
- 4. Verified via Postman -**
API functionality validated using Postman, confirming endpoint accuracy and data integrity.
- 5. Scalable and Cost-Efficient Design -**
Optimized for minimal cost and automatic scaling, ideal for lightweight, high-performance apps.

Creating a Serverless Personal To-Do List Application : A Comprehensive Guide

OBJECTIVE

Task 1: Build a simple AWS-based To-Do List application using a serverless architecture.

- **1.1 Project Setup:**

- **Create Project Directory-**
 - Start by creating a dedicated folder for the project to organize source code and configuration files.
- **Initialize Serverless Project**
 - Use either:
 - AWS SAM CLI (sam init)
 - Or Serverless Framework (serverless create --template aws-nodejs)
- **Select Runtime-**
 - Choose a runtime for AWS Lambda, such as:
 - Python (e.g., python3.9)
 - Node.js (e.g., nodejs22.x)
- **Prepare for Component Creation**
 - Plan the creation of Lambda functions, a DynamoDB table, and RESTful APIs in the next phase.

- **1.2 Define Resources:**

- In your SAM template (template.yaml) or Serverless Framework configuration (serverless.yml), define the following resources:
- - **DynamoDB Table:** A table to store To-Do items. Define the primary key and any necessary attributes (e.g., id, title, description, createdAt, updatedAt).
 - **Lambda Functions:** Functions for each CRUD operation (CreateTodo, GetTodos, GetTodo, UpdateTodo, DeleteTodo).
 - **API Gateway:** Define the API endpoints and integrate them with the corresponding Lambda functions.
 - **Tested API Endpoints Using Postman:** Verified the functionality of create, read, update, and delete operations through Postman to ensure accurate responses and system behavior.

Creating a Serverless Personal To-Do List Application : A Comprehensive Guide

1.2 Define Resources -

- **DynamoDB Table –**
 - Create a table to store To-Do items.
 - Define attributes such as:
 - `id` (Primary Key)
 - `title` – Task title
 - `description` – Details of the task
- **Lambda Functions (CRUD Operations) -**
 - Implement 5 separate functions:
 - `CreateTodo` – Add a new task
 - `GetTodos` – Retrieve all tasks
 - `GetTodo` – Retrieve a single task by ID
 - `UpdateTodo` – Modify an existing task
 - `DeleteTodo` – Remove a task
- **API Gateway -**
 - Define RESTful API endpoints such as:
 - `POST /todos` → `CreateTodo`
 - `GET /todos` → `GetTodos`
 - `GET /todos/{id}` → `GetTodo`
 - `PUT /todos/{id}` → `UpdateTodo`
 - `DELETE /todos/{id}` → `DeleteTodo`
 - Integrate each route with its respective Lambda function.
- **Tested API Endpoints Using Postman -**
 - Verified each CRUD operation via Postman to ensure:
 - Correct HTTP response codes (201 Created, 200 OK, 404 Not Found, etc.)
 - Accurate request-response handling
 - Reliable data persistence in DynamoDB

Creating a Serverless Personal To-Do List Application : A Comprehensive Guide

Introduction

Title - Exploring the world of Cloud Computing .

1. Understanding Cloud Computing -

Cloud computing is the on-demand delivery of IT resources over the Internet with pay-as-you-go pricing. Instead of traditional infrastructure management, users access services like computing power, storage, and databases from a cloud provider as needed.

Features :

- **On-demand delivery of IT resources:** This means you can access computing resources (like servers, storage, software) whenever you need them, without needing to wait for a long setup process.
- **Over the Internet:** The resources are accessed through the internet, allowing you to use them from anywhere with an internet connection.
- **Pay-as-you-go pricing:** You only pay for the resources you actually use. This is similar to how you pay for electricity or water.
- **Instead of traditional infrastructure management:** Cloud computing eliminates the need for businesses to buy and maintain their own physical servers, data centers, and IT staff.
- **Users access services...from a cloud provider:** Companies like Amazon (AWS), Microsoft (Azure), and Google (Google Cloud) provide these cloud computing services.

Service Models:

- **IaaS (Infrastructure as a Service):** Provides virtualized resources (VMs, storage, networks). Users manage OS and applications.

Features:

- **Virtualized resources:** Instead of physical hardware, you get virtualized resources. For example, a virtual machine (VM) simulates a physical server.
- **VMs, storage, networks:** IaaS provides the basic building blocks of IT infrastructure. You get virtual servers, storage space, and networking capabilities.
- **Users manage OS and applications:** You have control over the operating system (like Windows or Linux) and the applications you run on the virtual infrastructure.
- **Cloud provider manages the infrastructure:** The cloud provider is responsible for maintaining the physical servers, data centers, and network hardware.
- **PaaS (Platform as a Service):** Offers development platforms and tools. Developers build and deploy applications without managing underlying infrastructure.
- **Development platforms and tools:** PaaS provides a complete environment for developers to build, test, and deploy software applications. This includes tools, libraries, and services.

- **Developers build and deploy applications:** PaaS is designed to make it easier for developers to create applications quickly.
- **Without managing underlying infrastructure:** Developers don't have to worry about the servers, operating systems, or network infrastructure. The cloud provider handles all of that.
- **SaaS (Software as a Service):** Delivers complete software solutions accessible via a web browser (e.g., email, CRM).

Features:

- **Complete software solutions:** SaaS provides fully functional applications that you can use without installing anything on your computer.
- **Accessible via a web browser:** You can access the software through a web browser, from any device with an internet connection.
- **Examples (e.g., email, CRM):** Examples of SaaS applications include Gmail, Salesforce (CRM), and Dropbox.

Deployment Models:

Public Cloud: Shared infrastructure operated by third-party providers.

Features:

- **Shared infrastructure:** In a public cloud, the cloud provider's hardware and resources are shared among multiple customers.
- **Third-party providers:** Companies like AWS, Azure, and Google Cloud own and operate public clouds.
- **Examples:** AWS, Microsoft Azure, Google Cloud Platform.
- **Private Cloud:** Dedicated infrastructure for a single organization.

Features:

- **Dedicated infrastructure:** A private cloud is set up for the exclusive use of a single company or organization.
- **Single organization:** The resources are not shared with other customers.
- **On-premises or hosted:** A private cloud can be located in the company's own data center or hosted by a third-party provider.
- **Hybrid Cloud:** A combination of public and private cloud environments.

Features:

- **Combination of public and private:** A hybrid cloud uses both public and private cloud resources.
- **Data and application sharing:** Data and applications can be moved between the public and private clouds.
- **Flexibility and options:** Hybrid clouds offer greater flexibility and allow organizations to optimize their infrastructure based on their specific needs.

Features:

- **Easily adjust resources:** Cloud computing allows you to quickly increase or decrease the amount of resources you are using (e.g., computing power, storage) as your needs change.
- **Changing demands:** If your website traffic increases, you can easily scale up your resources to handle the extra load. If traffic decreases, you can scale down to save money.
- **Cost Efficiency:** Pay-as-you-go pricing eliminates upfront investments and reduces operational expenses.
- **Pay-as-you-go pricing:** You only pay for the resources you use, which can be more cost-effective than buying and maintaining your own hardware.
- **Eliminates upfront investments:** You don't have to spend a lot of money upfront to buy servers and other infrastructure.
- **Reduces operational expenses:** Cloud computing can reduce the costs associated with running your own IT infrastructure, such as power, cooling, and IT staff.
- **High Availability:** Ensures continuous service availability through redundancy and fault tolerance.
- **Continuous service availability:** Cloud computing helps to ensure that your applications and data are always available.
- **Redundancy and fault tolerance:** Cloud providers use multiple data centers and redundant systems to minimize downtime in case of failures.
- **Agility and Speed:** Quickly provision and deploy resources, accelerating development and innovation.
- **Quickly provision and deploy resources:** You can quickly get the resources you need to set up and run applications, without long delays.
- **Accelerating development and innovation:** Cloud computing makes it easier and faster to develop and deploy new applications, which can help businesses innovate more quickly.
- **Global Reach:** Access resources and services from anywhere in the world.
- **Access resources...from anywhere:** Cloud computing allows you to access your applications and data from anywhere with an internet connection.
- **Global network of data centers:** Cloud providers have data centers around the world, which allows them to offer services to users globally.

Creating a Serverless Personal To-Do List Application :

A Comprehensive Guide

3.2. AWS Lambda -

AWS Lambda is a serverless computing service that lets you run code without provisioning or managing servers. You only pay for the compute time you consume - there's no charge when your code isn't running.

Features of AWS Lambda:

- **Serverless:**
 - Instead of provisioning virtual machines (like EC2 instances) or containers, you upload your code to Lambda, and AWS takes care of the underlying infrastructure.
 - This means you don't have to worry about server maintenance, operating system updates, patching, or scaling. AWS handles all of that.
 - "Serverless" doesn't mean there are no servers; it simply means that you, as the user, don't manage them.
- **Event-Driven:**
 - Lambda functions are designed to be triggered by events. An event is an occurrence or a change in state.
 - This event-driven architecture allows you to build applications that respond to various triggers in real-time.
 - Lambda integrates with many AWS services, allowing those services to trigger Lambda functions.

Examples of events:

- **Changes to data in Amazon S3:** When a new file is uploaded to an S3 bucket, or an existing file is modified, Lambda can be triggered to process that data (e.g., resizing an image, converting a file format).
 - **Messages arriving in Amazon SQS:** When a message is sent to an SQS queue, Lambda can be triggered to process that message (e.g., adding the message to a database, sending a notification).
 - **HTTP requests via Amazon API Gateway:** When a user makes an HTTP request to an API endpoint, API Gateway can trigger a Lambda function to handle that request (e.g., processing a form submission, retrieving data from a database).
 - **Other AWS Services:** Lambda can also be triggered by services like DynamoDB, CloudWatch, EventBridge, and more.
- **Functions:**
 - The code you write to be executed by Lambda is packaged as a "function".
 - A Lambda function is a stateless piece of code. This means it doesn't store data between invocations. If you need to store data, you need to use an external service like Amazon S3 or Amazon DynamoDB.
 - Lambda supports several programming languages, including:
 - Python
 - Node.js
 - PowerShell

- You are responsible for writing the code within your Lambda function to process the event and perform the desired actions.
- **Automatic Scaling:**
 - Lambda automatically scales the execution of your functions based on the number of incoming events.
 - When an event triggers your Lambda function, AWS Lambda automatically provisions the necessary resources to run your code.
 - If there are many events happening at the same time, Lambda will create multiple instances of your function to handle them concurrently. This scaling happens automatically and quickly.
 - When the events subside, Lambda scales down the number of function instances.
 - This automatic scaling is one of the key benefits of Lambda, as it allows your application to handle varying levels of traffic without requiring you to manage scaling yourself.

Working:

1. Upload Code:

- You create your Lambda function code and package it into a deployment package (e.g., a ZIP file or a container image).
- This package contains your code and any dependencies (libraries, etc.) that your code needs to run.
- You then upload this deployment package to AWS Lambda.

2. Configure Trigger:

- You configure an AWS service or an application to be the event source that triggers your Lambda function.
- This configuration tells Lambda when to execute your function.
- For example, you can configure an S3 bucket to trigger your Lambda function whenever a new object is created in the bucket.

3. Lambda Executes:

- When the configured event occurs, AWS Lambda detects the event and automatically executes your Lambda function.
- Lambda creates an execution environment, which includes the necessary resources (CPU, memory) to run your code.
- Your function code is then executed within this environment.

4. Pay for Execution Time:

- With AWS Lambda, you are charged based on the actual compute time your function consumes.
- You only pay for the time your code is running, measured in milliseconds.
- There is no charge when your function is not being executed.
- This pay-per-execution model can be very cost-effective, especially for applications with infrequent or variable workloads.

Benefits:

• No Server Management:

- Frees you from the operational overhead of managing servers.
- You don't have to worry about tasks like server provisioning, patching, scaling, and maintenance.

- This allows you to focus on writing your application code and business logic, rather than spending time on infrastructure management.
- **Scalability:**
 - Automatically scales your function executions to handle any level of demand.
 - Lambda can handle a few requests or thousands of requests concurrently.
 - You don't need to configure any scaling policies or rules. Lambda automatically handles scaling for you.
- **Cost-Effective:**
 - You only pay for the compute time you consume.
 - There are no charges when your function is idle.
 - This can result in significant cost savings compared to traditional server-based solutions, where you have to pay for the server even when it's not being used.
- **High Availability:**
 - Lambda runs your code on highly available infrastructure.
 - AWS Lambda is designed to provide high availability and fault tolerance.
 - Your functions are available even if there are hardware failures or other issues.
- **Integration:**
 - AWS Lambda integrates with a wide range of other AWS services.
 - This allows you to easily build applications that leverage the capabilities of multiple AWS services.
 - For example, you can use Lambda with S3, DynamoDB, API Gateway, SQS, SNS, and many others.

Use Cases:

- **Data Processing:**
 - Real-time data processing: Processing data as it is generated, such as processing log data, analyzing streaming data, or transforming data.
 - Batch data processing: Performing batch operations on large datasets, such as data transformation, data validation, or data aggregation.
- **Web Applications:**
 - Building serverless web applications: Creating dynamic web applications without managing any servers.
 - Handling HTTP requests, processing form submissions, authenticating users, and generating dynamic content.
- **Mobile Backends:**
 - Creating serverless backends for mobile apps: Building APIs that mobile apps can use to access data and functionality.
 - Handling user authentication, processing user input, and interacting with databases.
- **Automation:**
 - Automating tasks: Automating repetitive tasks, such as:
 - Running scheduled jobs (e.g., backups, maintenance tasks).
 - Responding to events (e.g., sending notifications, triggering workflows).
 - Integrating with other systems.

Creating a Serverless Personal To-Do List Application : A Comprehensive Guide

METHODOLOGY

This section provides a detailed explanation of the methodology used to develop a serverless To-Do list application. The methodology is structured to ensure a systematic approach, covering all phases from initial requirements gathering to ongoing maintenance. The core principle is to leverage the benefits of serverless architecture on AWS to create a scalable, cost-effective, and maintainable application.

Requirements Analysis and System Design-

1. Requirements Analysis:

- **Define the scope of the To-Do list application:**
 - The primary goal is to create a functional application that allows users to manage their tasks efficiently. This involves defining the boundaries of the application, clarifying what it will and will not do.
 - For this application, the scope is limited to basic To-Do list functionalities. More advanced features (like user authentication, tags, or sharing) are explicitly excluded to keep the development focused.
- **Identify necessary features (CRUD operations):**
 - The application will implement the fundamental CRUD (Create, Read, Update, Delete) operations. Each operation is mapped to a specific user action:
 - **Create:** Add a new To-Do item to the list.
 - **Read:** Retrieve a single To-Do item or the entire list.
 - **Update:** Modify an existing To-Do item's details.
 - **Delete:** Remove a To-Do item from the list.
- **Choose AWS services appropriate for a serverless architecture:**
 - Selecting the right AWS services is crucial for building a robust serverless application. The following services are chosen:
 - **AWS Lambda:** Provides the compute layer. Lambda functions will contain the application's business logic, executing code in response to events.
 - **Amazon DynamoDB:** A NoSQL database for storing To-Do items. DynamoDB is chosen for its scalability, performance, and ability to integrate seamlessly with Lambda.
 - **Amazon API Gateway:** Manages the API endpoints, allowing users to interact with the Lambda functions via HTTP requests.

Creating a Serverless Personal To-Do List Application : A Comprehensive Guide

2. System Design:

- **Design the API endpoints and their methods (GET, POST, PUT, DELETE):**

- A RESTful API is designed to expose the To-Do list functionalities. Each endpoint represents a specific resource (To-Do item), and the HTTP methods define the actions that can be performed on that resource. The following endpoints are defined:
 - POST /todos: Creates a new To-Do item.
 - GET /todos: Retrieves all To-Do items.
 - GET /todos/{id}: Retrieves a specific To-Do item by its ID.
 - PUT /todos/{id}: Updates a specific To-Do item by its ID.
 - DELETE /todos/{id}: Deletes a specific To-Do item by its ID.

- **Define Lambda function responsibilities:**

- Each API endpoint is associated with a specific Lambda function. This promotes a microservices approach, where each function has a single, well-defined responsibility:
 - CreateToDoFunction: Handles the POST /todos request. It receives the To-Do item data from the request body and stores it in DynamoDB.
 - GetToDoFunction: Handles the GET /todos and GET /todos/{id} requests. It retrieves To-Do items from DynamoDB and returns them in the response.
 - UpdateToDoFunction: Handles the PUT /todos/{id} request. It receives the updated To-Do item data and the ID, and updates the corresponding item in DynamoDB.
 - DeleteToDoFunction: Handles the DELETE /todos/{id} request. It receives the To-Do item ID and deletes the item from DynamoDB.

- **Design DynamoDB table schema to store To-Do items (e.g., id, title, description, status):**

- The structure of the DynamoDB table is defined to efficiently store and retrieve To-Do items. The following attributes are included:
 - id (String, Primary Key): A unique identifier for each To-Do item.
 - title (String): The title of the To-Do item.
 - description (String): A more detailed description of the To-Do item.
 - status (String): The current status of the To-Do item (e.g., "pending", "completed").
- The id attribute is chosen as the primary key because it ensures that each To-Do item can be uniquely identified.

Environment Setup and Implementation –

3. Environment Setup:

- **Configure AWS CLI and SAM/Serverless Framework:**

- To interact with AWS services, the AWS Command Line Interface (CLI) needs to be configured. This involves setting up credentials (access keys) that allow you to authenticate your requests to AWS.
- The Serverless Application Model (SAM) or Serverless Framework is used to simplify the deployment of serverless applications.
 - Both SAM and Serverless Framework provide abstractions and command-line tools that automate the process of packaging, configuring, and deploying the necessary AWS resources (Lambda functions, API Gateway, DynamoDB tables, etc.).
 - SAM is an open-source framework provided by AWS, while Serverless Framework is a third-party open-source framework.

- **Set up a local development environment for testing:**

- A local development environment is set up to facilitate development and testing before deploying to AWS. This typically includes:
 - Installing the chosen programming language runtime (e.g., Python, Node.js).
 - Installing necessary development tools and libraries.
 - Setting up a local version of DynamoDB (if needed) for local testing. SAM and Serverless Framework provide mechanisms for local testing.

4. Implementation:

- **Develop Lambda functions using Python/Node.js:**

- The core business logic of the application is implemented within the Lambda functions.
- Using the selected programming language (Python or Node.js), code is written to handle the requests for each API endpoint. This involves:
 - Receiving data from the API Gateway (e.g., request parameters, request body).
 - Interacting with DynamoDB to perform CRUD operations.
 - Constructing the appropriate response to be sent back to the client via API Gateway.

- **Define API Gateway configuration in `template.yaml` (SAM) or `serverless.yml` (Serverless Framework):**

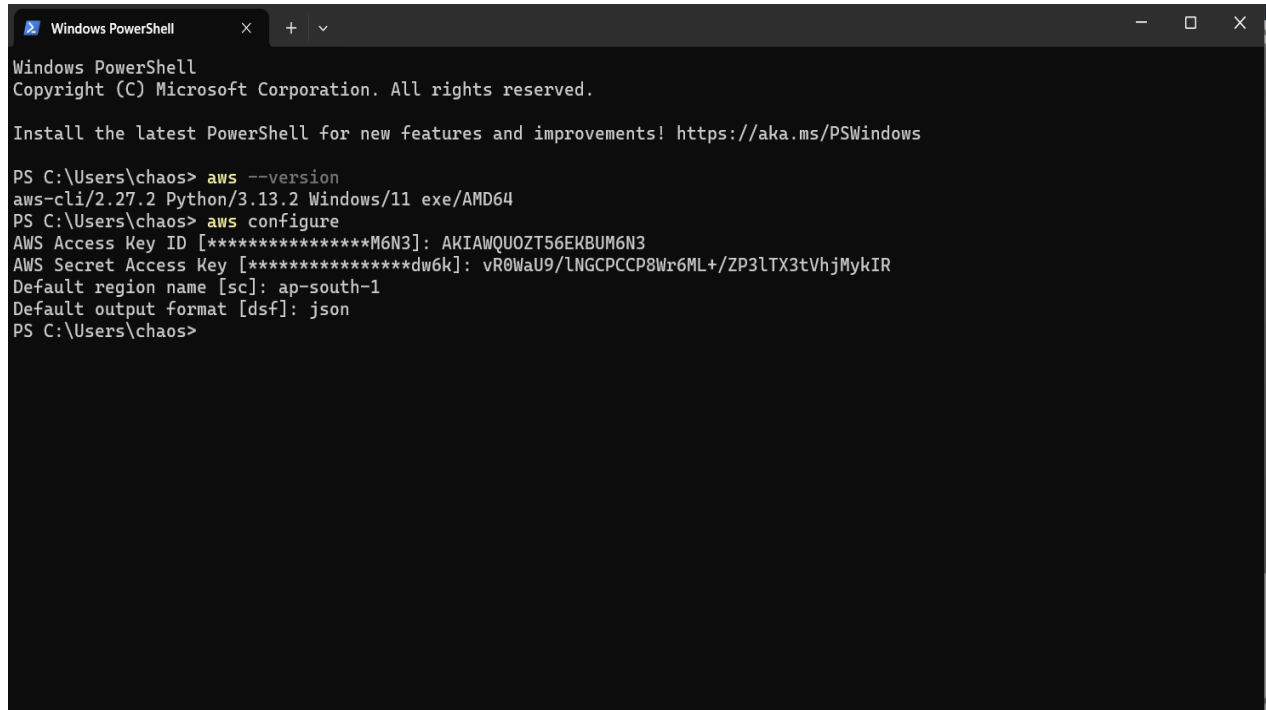
- The API Gateway is configured to define the API endpoints and how they are mapped to the corresponding Lambda functions.
- This configuration is done using either:
 - `template.yaml` for AWS SAM: A YAML-formatted file that defines the AWS resources for the application.
 - `serverless.yml` for Serverless Framework: A YAML-formatted file that defines the serverless application.
- The configuration includes:
 - Defining the API endpoints (e.g., `/todos`, `/todos/{id}`).
 - Specifying the HTTP methods for each endpoint (e.g., GET, POST, PUT, DELETE).
 - Mapping each endpoint and method to the corresponding Lambda function.

- **Write DynamoDB interaction code in each Lambda function:**

- The Lambda functions contain code that interacts with the DynamoDB table to perform the necessary database operations.
- This code uses the AWS SDK to make requests to the DynamoDB API.
- For example:
 - The `CreateToDoFunction` uses the `putItem` operation to add a new To-Do item.
 - The `GetToDoFunction` uses the `getItem` operation to retrieve a single To-Do item by its ID, or the `scan` operation to retrieve all items.
 - The `UpdateToDoFunction` uses the `updateItem` operation to modify an existing To-Do item.
 - The `DeleteToDoFunction` uses the `deleteItem` operation to remove a To-Do item.

Creating a Serverless Personal To-Do List Application : A Comprehensive Guide

SCREENSHOTS



```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\chaos> aws --version
aws-cli/2.27.2 Python/3.13.2 Windows/11 exe/AMD64
PS C:\Users\chaos> aws configure
AWS Access Key ID [*****M6N3]: AKIAWQUOZT56EKBUM6N3
AWS Secret Access Key [*****dw6k]: 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
Default region name [sc]: ap-south-1
Default output format [dsf]: json
PS C:\Users\chaos>
```

Create table

Table details [Info](#)

DynamoDB is a schemaless database that requires only a table name and a primary key when you create the table.

Table name

This will be used to identify your table.

Between 3 and 255 characters, containing only letters, numbers, underscores (_), hyphens (-), and periods (.).

Partition key

The partition key is part of the table's primary key. It is a hash value that is used to retrieve items from your table and allocate data across hosts for scalability and availability.

1 to 255 characters and case sensitive.

[Customize partition](#)

Creating a Serverless Personal To-Do List Application : A Comprehensive Guide

Create function [Info](#)

Choose one of the following options to create your function.

☒ Author from scratch

Start with a simple Hello World example.

☐ Use a blueprint

Build a Lambda application from sample code and configuration presets for common use cases.

☐ Container image

Select a container image to deploy for your function.

Basic information

Function name

Enter a name that describes the purpose of your function.

ToDoLambda

Function name must be 1 to 64 characters, must be unique to the Region, and can't include spaces. Valid characters are a-z, A-Z, 0-9, hyphens (-), and underscores (_).

Runtime

[Info](#)

Choose the language to use to write your function. Note that the console code editor supports only Node.js, Python, and Ruby.

Node.js 18.x

Architecture

[Info](#)

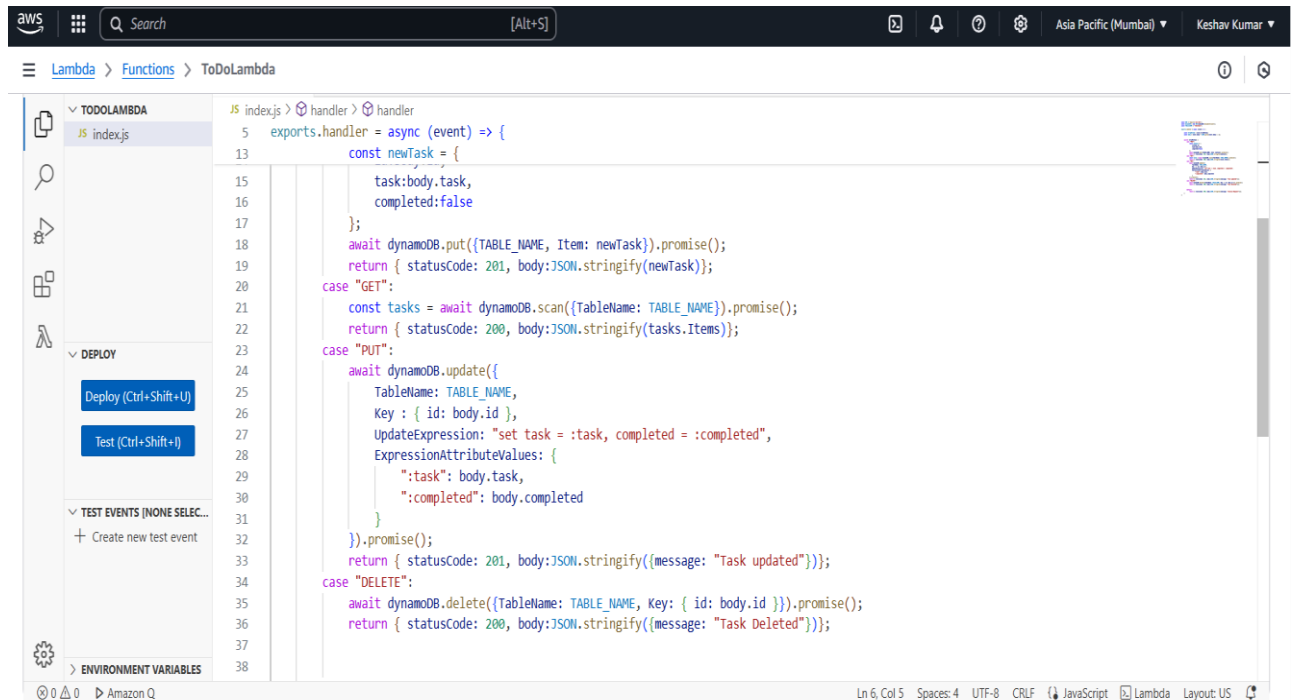
Choose the instruction set architecture you want for your function code.

☒ x86_64

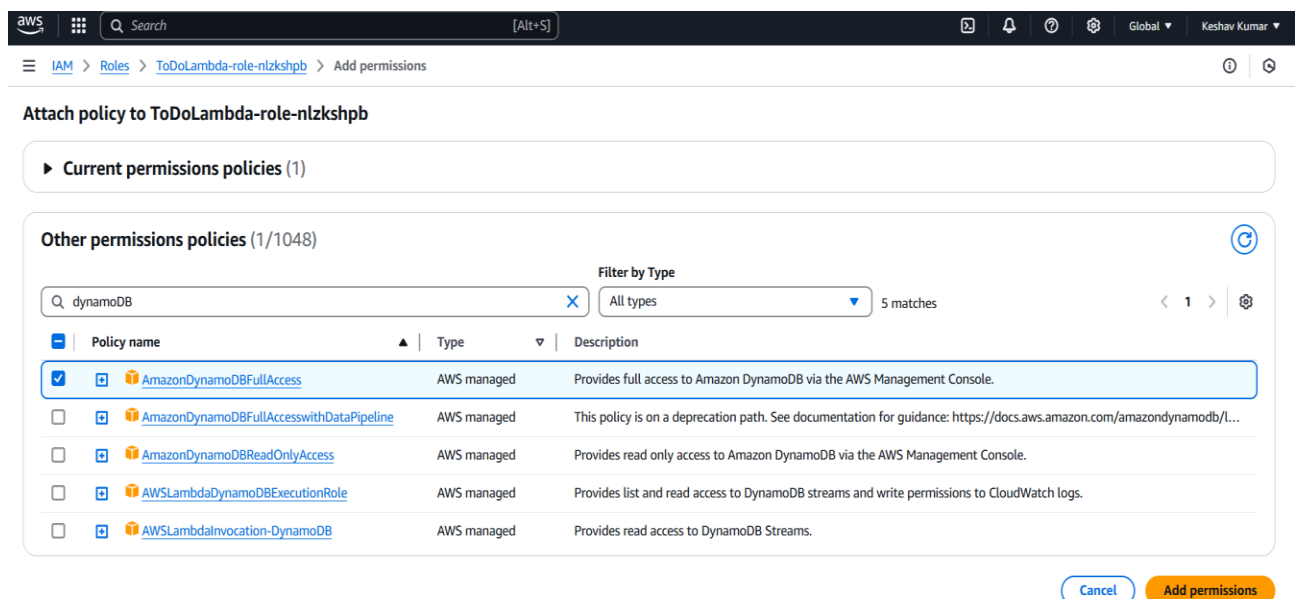
☐ arm64

```
JS index.js X
C:\> Xtra > Cloud > woRISGO > aws-serverless > JS index.js > handler > handler
1  const AWS = require('aws-sdk');
2  const dynamoDB = new AWS.DynamoDB.DocumentClient();
3  const TABLE_NAME = "ToDoTable";
4
5  exports.handler = async (event) => {
6      const httpMethod = event.httpMethod;
7      const body = event.body ? JSON.parse(event.body) : {};
8
9
10     switch (httpMethod) {
11         case "POST":
12             const newTask = {
13                 id:body.id,
14                 task:body.task,
15                 completed:false
16             };
17             await dynamoDB.put({ TableName: TABLE_NAME, Item: newTask }).promise();
18             return { statusCode: 201, body:JSON.stringify(newTask)};
19         case "GET":
20             const tasks = await dynamoDB.scan({TableName: TABLE_NAME}).promise();
21             return { statusCode: 200, body:JSON.stringify(tasks.Items)};
22         case "PUT":
23             await dynamoDB.update({
24                 TableName: TABLE_NAME,
25                 Key : { id: body.id },
26                 UpdateExpression: "set task = :task, completed = :completed",
27                 ExpressionAttributeValues: {
28                     ":task": body.task,
29                     ":completed": body.completed
30                 }
31             }).promise();
32             return { statusCode: 201, body:JSON.stringify({message: "Task updated"})};
33         case "DELETE":
34             await dynamoDB.delete({TableName: TABLE_NAME, Key: { id: body.id }}).promise();
35             return { statusCode: 200, body:JSON.stringify({message: "Task Deleted"})};
36
37         default:
38             return { statusCode: 400, body:JSON.stringify({message: "Invalid Request"})};
39     }
40 }
41 }
```

Creating a Serverless Personal To-Do List Application : A Comprehensive Guide



Code properties [Info](#)



Creating a Serverless Personal To-Do List Application : A Comprehensive Guide

Last activity	Maximum session duration
-	1 hour

Permissions	Trust relationships	Tags	Last Accessed	Revoke sessions
-------------	---------------------	------	---------------	-----------------

Permissions policies (2) [Info](#)

Simulate

Remove

Add permissions



You can attach up to 10 managed policies.

Q Search

Filter by Type

All types

< 1 >

<input type="checkbox"/>	Policy name	Type	Attached entities
<input type="checkbox"/>	 AmazonDynamoDBFullAccess	AWS managed	1
<input type="checkbox"/>	 AWSLambdaBasicExecutionRole-c7...	Customer man...	1

Stages for todo (1)

Q Find resources

Stage name	Invoke URL	Attached deployment	Auto deploy	Last updated
\$default	https://6famoitvtb.execute-api.us-east-1.amazonaws.com	g5udek	enabled	2025-05-02

Tags (0)

Manage tags

Q Find resources

< 1 >

Key	Value
-----	-------

No tags

No tags associated with the resource.

Manage tags

Creating a Serverless Personal To-Do List Application : A Comprehensive Guide

```
MINGW64:/c/Users/chaos

chaos@ZweiLiOUS MINGW64 ~
$ curl -X POST https://6famoitvtb.execute-api.us-east-1.amazonaws.com/todo \
> curl -X POST https://6famoitvtb.execute-api.us-east-1.amazonaws.com/todo \ -H
"Content-Type: application/json" \ -d '{"id":"1", "task":"Serverless App Setup"}'
{"message":"Internal Server Error"}curl: (6) Could not resolve host: curl
{"message":"Internal Server Error"}curl: (3) URL rejected: Malformed input to a
URL function
curl: (3) URL rejected: Malformed input to a URL function
curl: (3) URL rejected: Malformed input to a URL function
curl: (3) URL rejected: Port number was not a decimal number between 0 and 65535
curl: (3) URL rejected: Malformed input to a URL function

chaos@ZweiLiOUS MINGW64 ~
$
```

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\chaos> cd C:\Xtra\Cloud\woRISG0\aws-serverless
PS C:\Xtra\Cloud\woRISG0\aws-serverless> npm init -y
Wrote to C:\Xtra\Cloud\woRISG0\aws-serverless\package.json:

{
  "name": "aws-serverless",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "type": "commonjs"
}

PS C:\Xtra\Cloud\woRISG0\aws-serverless> npm install aws-sdk
npm warn deprecated querystring@0.2.0: The querystring API is considered Legacy. new code should use the URLSearchParams
API instead.

added 44 packages, and audited 45 packages in 6s
```

Creating a Serverless Personal To-Do List Application : A Comprehensive Guide

The screenshot shows a REST client interface with a GET request to `https://6famoitvtb.execute-api.us-east-1.amazonaws.com/todo`. The response is a 400 Bad Request with a status bar indicating 2.08 s and 214 B. The response body is raw JSON: `{ "message": "Invalid Request" }`.

HTTP Test / New Request

GET `https://6famoitvtb.execute-api.us-east-1.amazonaws.com/todo` Send

Params Authorization Headers (6) Body Scripts Settings Cookies

Query Params

Key	Value	Description
Key	Value	Description

Body Cookies Headers (5) Test Results 400 Bad Request • 2.08 s • 214 B Save Response

Raw Preview Visualize

```
1 { "message": "Invalid Request" }
```

The screenshot shows a REST client interface with a POST request to `https://k0wvy75ckl.execute-api.ap-south-1.amazonaws.com/ToDoList`. The response is a 500 Internal Server Error with a status bar indicating 1.10 s and 221 B. The response body is raw JSON: `{ "id": "1", "task": "Serverless App Setup" }`.

POST `https://k0wvy75ckl.execute-api.ap-south-1.amazonaws.com/ToDoList` Send

Params Authorization Headers (8) Body Scripts Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

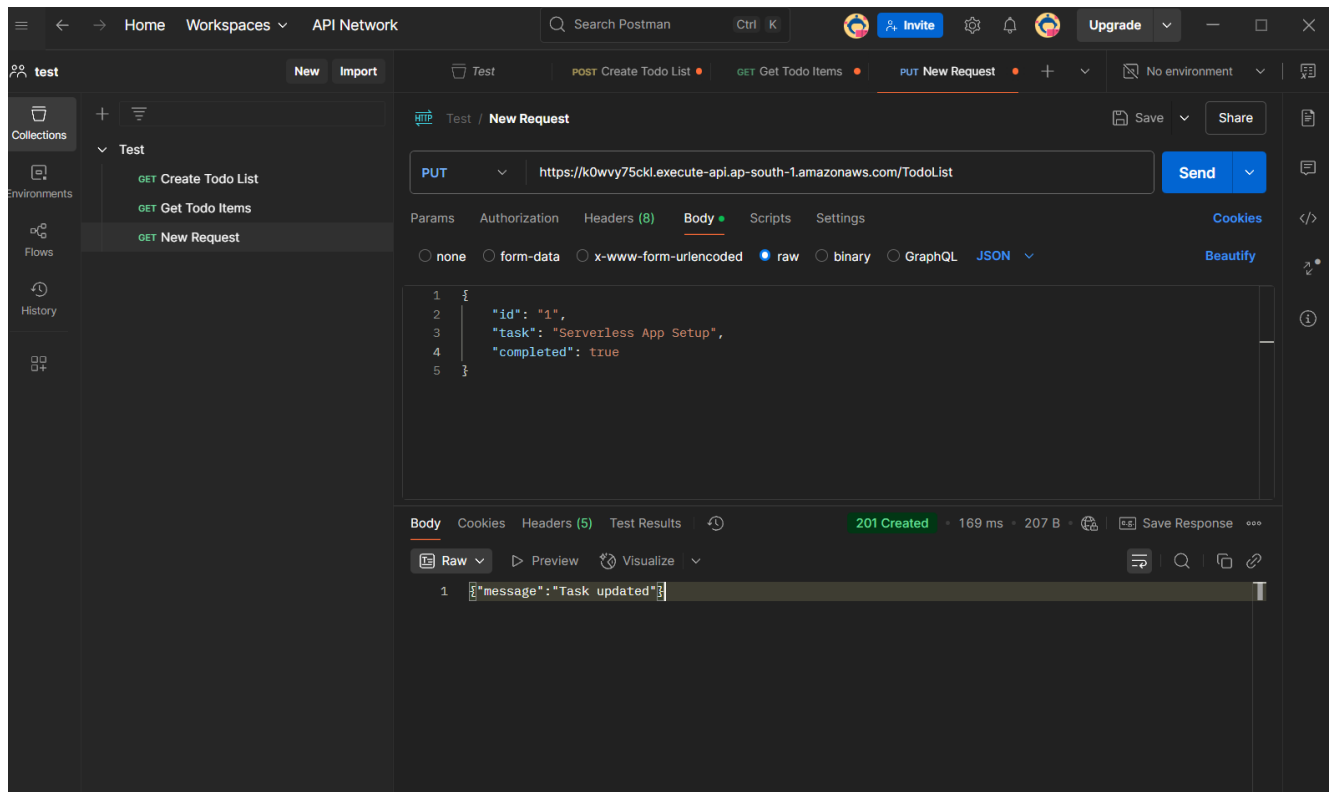
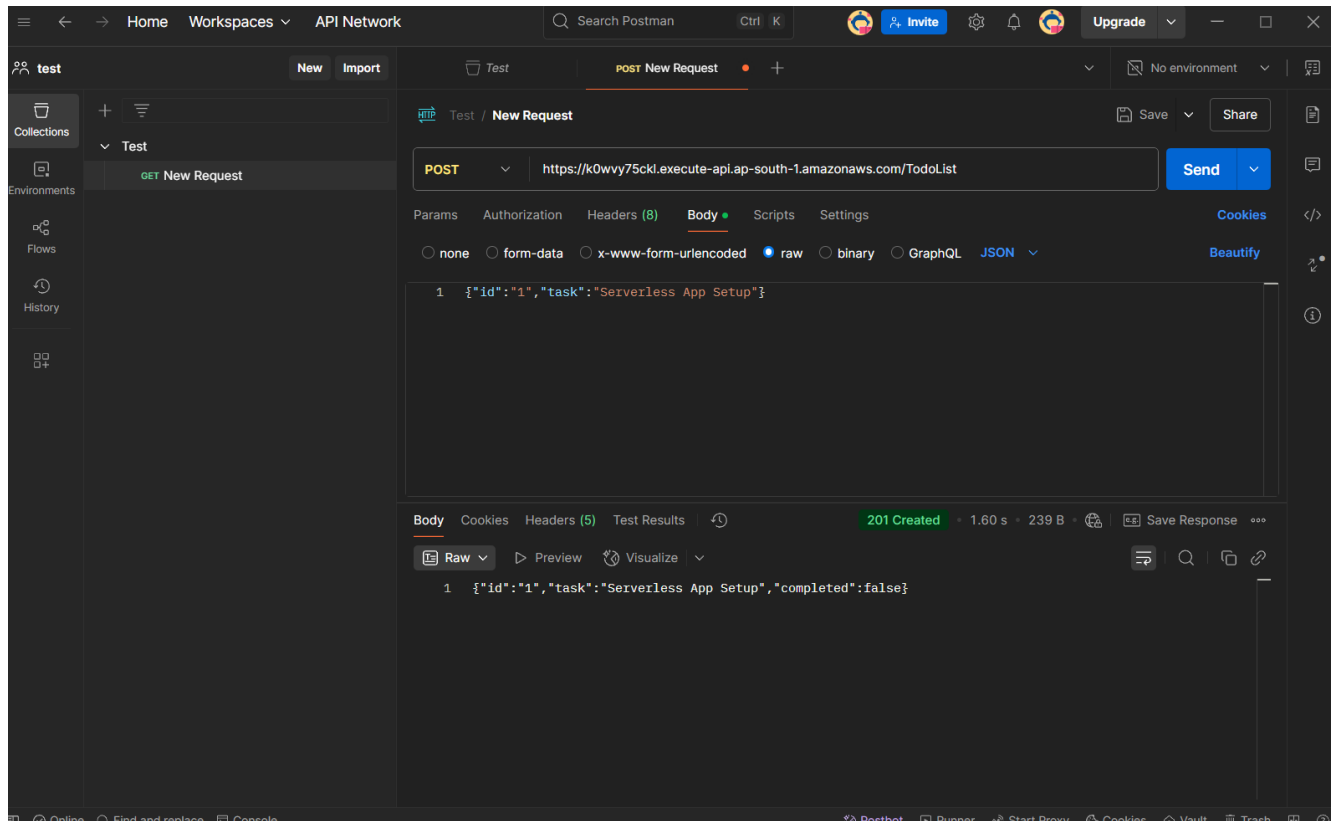
```
1 { "id": "1", "task": "Serverless App Setup" }
```

Body Cookies Headers (5) Test Results 500 Internal Server Error • 1.10 s • 221 B Save Response

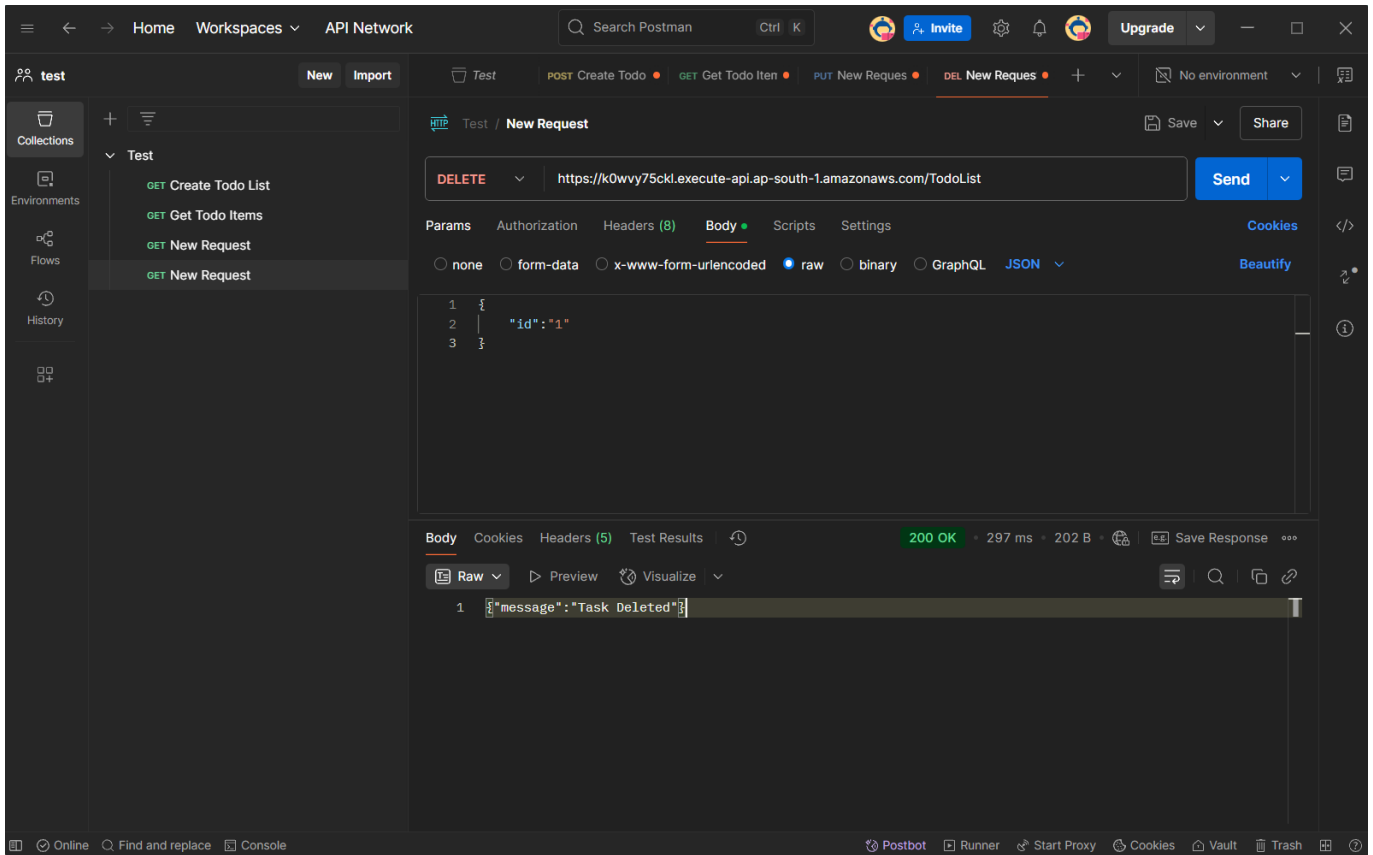
JSON Preview Visualize

```
1 {
2   "message": "Internal Server Error"
3 }
```

Creating a Serverless Personal To-Do List Application : A Comprehensive Guide

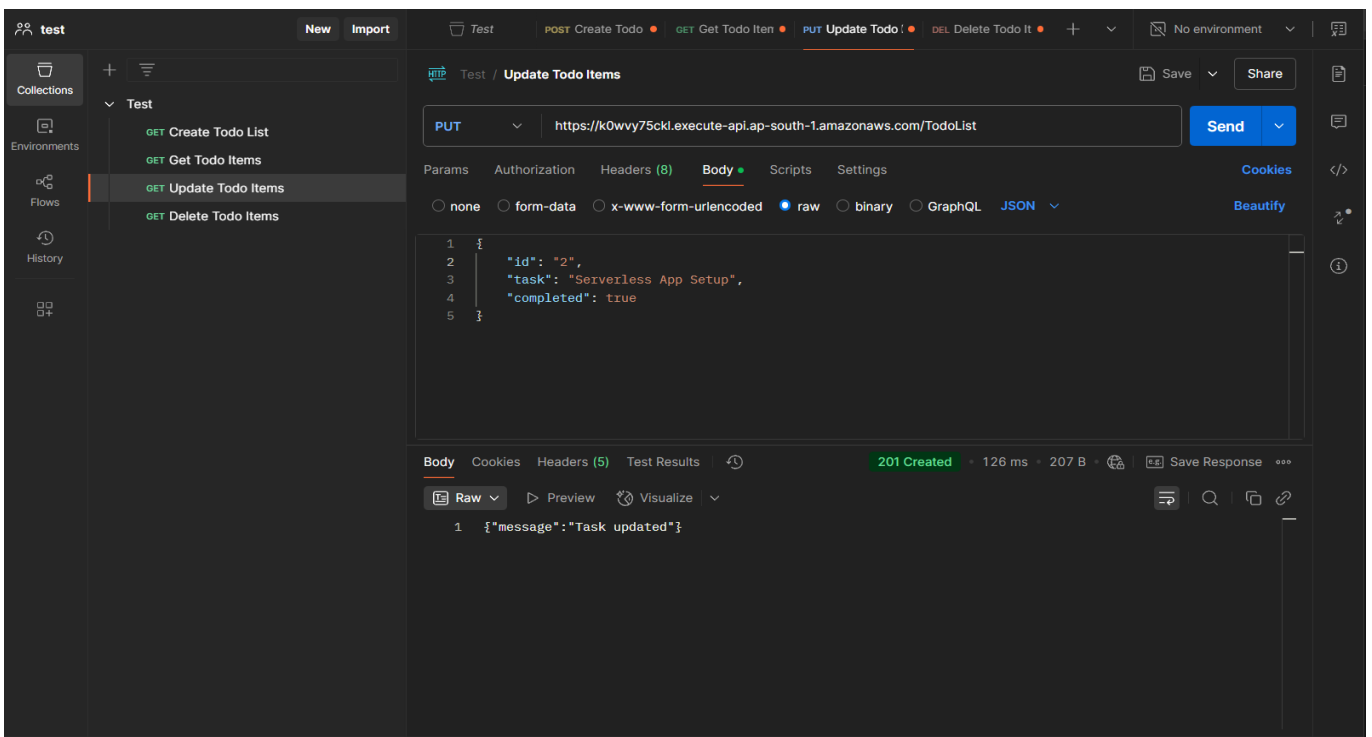
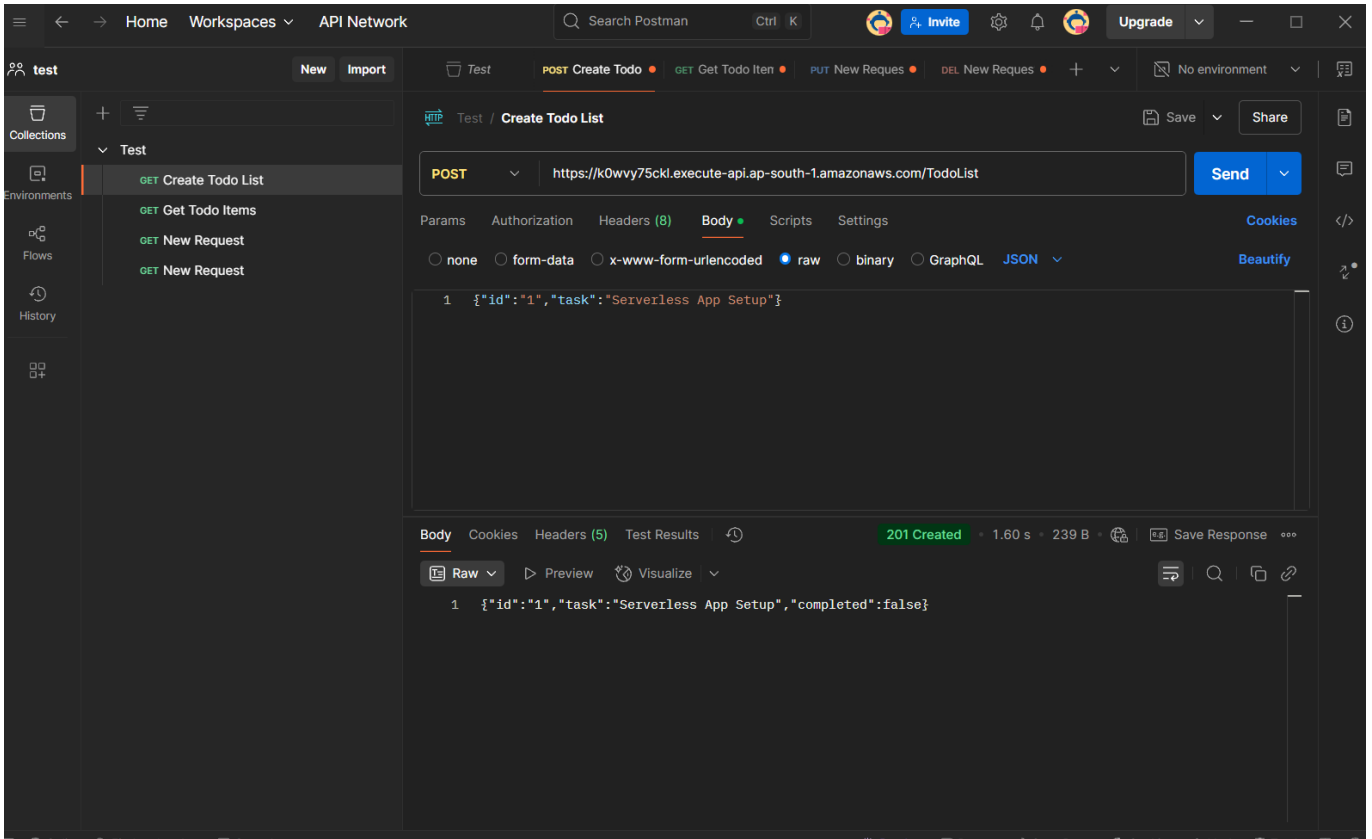


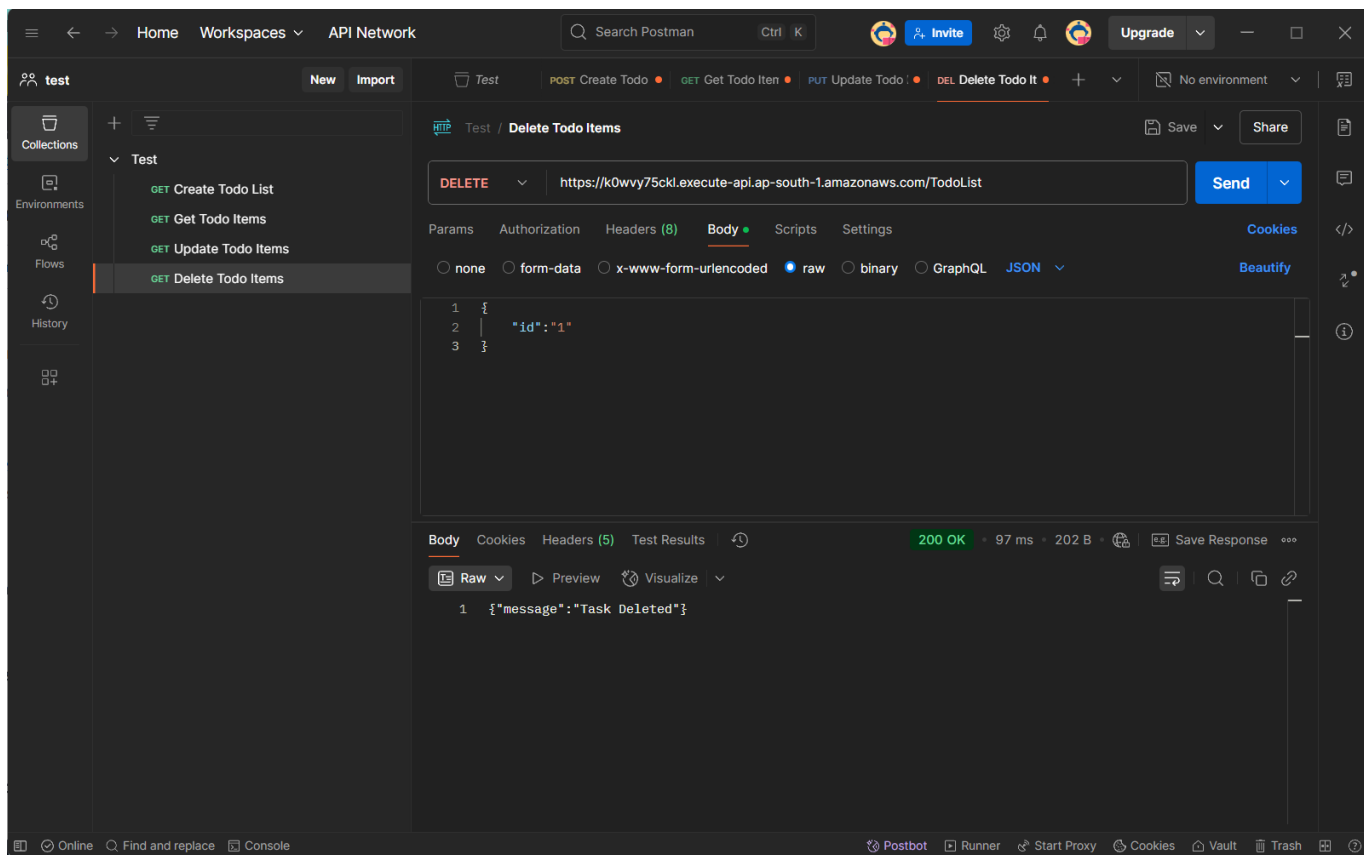
Creating a Serverless Personal To-Do List Application : A Comprehensive Guide



Creating a Serverless Personal To-Do List Application : A Comprehensive Guide

Final Outputs -





Creating a Serverless Personal To-Do List Application : A Comprehensive Guide

Conclusion -

In conclusion, the Serverless Personal To-Do List Application demonstrates the effectiveness and practicality of using AWS serverless services for developing scalable, maintainable, and cost-efficient applications. The project successfully achieved its objectives by adopting a cloud-native architecture, leveraging key AWS services like Lambda, API Gateway, and DynamoDB, and employing Infrastructure-as-Code for seamless deployment.

Through this implementation, it became evident that serverless computing not only reduces operational complexity but also enables rapid development and iteration. Moreover, the project has laid a solid foundation for future enhancements, including authentication, notifications, and frontend integration. This work showcases the potential of modern cloud platforms in building reliable applications with minimal overhead and serves as a reference for developers looking to explore serverless solutions for real-world use cases.

Conclusion Points –

- Demonstrates the power and efficiency of cloud-native solutions using AWS.
- Leverages serverless components: AWS Lambda, API Gateway, and DynamoDB.
- Illustrates building applications without traditional server infrastructure.
- Employs best practices in serverless architecture throughout the project lifecycle.
- Utilizes AWS SAM or Serverless Framework for rapid development and deployment.
- Reinforces the advantages of Infrastructure as Code (IaC).
- Implements core CRUD functionalities through RESTful APIs.
- Provides real-time to-do item management.
- Verifies API operations with Postman.
- Uses DynamoDB for a reliable, low-latency data store.
- Highlights cloud concepts: event-driven computing, pay-as-you-go pricing, automatic scaling, and high availability.
- Addresses security and monitoring with IAM roles and CloudWatch logs.
- Delivers a functional and maintainable productivity tool.
- Serves as a practical learning experience in cloud application development.
- Proves serverless computing is effective for building modern, scalable applications.