

## Activity 1

**a. Aim of the Task and Application Functionality**

**Aim of the Task:** The objective was to develop a scalable, serverless note-taking application using AWS Services, meeting specific activity requirements. The goal of the app is to leverage AWS services to create an application that allows users to create, read, update, and delete notes efficiently, ensuring high availability and minimal maintenance overhead.

**Application Functionality:** The serverless note-taking application provides the following features:

- **Create Note:** Users can create new notes, which are stored in a DynamoDB table. Each note is assigned a unique ID and contains a text field.
- **Read Notes:** Users can retrieve all existing notes from the DynamoDB table.
- **Update Note:** Users can update the text of an existing note by providing its ID.
- **Delete Note:** Users can delete a note by providing its ID.
- **Logging:** Each CRUD operation is logged to an S3 bucket for tracking and audit purposes.

**b. Thought Process of Developing the Application****Design Considerations:**

1. **Scalability:**
  - The application is designed to scale automatically with user demand. Using AWS Lambda for backend logic ensures that compute resources are only used when necessary, reducing costs and handling high traffic seamlessly [1][2].
  - DynamoDB, with its provisioned throughput and scalability options, provides a reliable data store that can handle large volumes of data and high request rates [3][4].
2. **High Availability and Fault Tolerance:**
  - By leveraging AWS Lambda and DynamoDB, the application benefits from the high availability and fault tolerance provided by AWS services [5][6].
  - The use of S3 for logging ensures that logs are stored durably and are easily accessible for debugging and audit purposes [5][6].
3. **Cost Efficiency:**
  - The serverless architecture ensures that costs are minimized by only charging for actual usage (Lambda invocations, DynamoDB read/write operations, and S3 storage) [7][8].
  - Provisioned throughput for DynamoDB is initially set to handle moderate traffic, and can be adjusted as needed based on actual usage patterns [3][7].
4. **Security:**
  - The application uses an existing IAM role to grant necessary permissions to Lambda functions, adhering to the principle of least privilege [9][10].
  - API Gateway methods are configured with CORS headers to ensure secure cross-origin requests while enabling access from various client applications [9][10].

## Activity 1

**Development Process:****1. Defining Resources:**

- The primary resources required were identified: DynamoDB table for storing notes, S3 bucket for logs, Lambda functions for handling CRUD operations, and API Gateway for exposing the application endpoints.
- CloudFormation template was chosen to define these resources in a declarative manner, ensuring consistency and ease of deployment.

**2. Implementing Lambda Functions:**

- Four Lambda functions were implemented to handle the CRUD operations. Each function includes logic to interact with DynamoDB (at least twice) and log operations to S3.
- Node.js was selected as the runtime for its asynchronous capabilities and extensive library support.

**3. Configuring API Gateway:**

- API Gateway was configured to expose endpoints for each CRUD operation, integrating with the respective Lambda functions.
- Methods and resources were defined, ensuring proper request handling and response formatting.

**4. Handling Cross-Origin Requests:**

- CORS headers were configured for all API Gateway methods to allow requests from different origins, facilitating seamless interaction with the API from web clients.

**Challenges Faced:****1. Lambda Permissions:**

- Ensuring the correct permissions for Lambda functions to interact with DynamoDB and S3 required careful configuration of IAM roles and policies.
- Addressing issues related to API Gateway invoking Lambda functions due to permission misconfigurations.

**2. Error Handling:**

- Implementing robust error handling within Lambda functions to provide meaningful error messages and ensure a smooth user experience.
- Logging errors to S3 to aid in troubleshooting and monitoring the application's health.

**3. CORS Configuration:**

- Configuring CORS headers correctly to allow cross-origin requests.

**Activity Requirements Fulfillment****1. Minimum of Four Lambda Functions:**

- The application includes four Lambda functions:
  - CreateNoteFunction: Handles creating notes.
  - GetNotesFunction: Handles retrieving notes.
  - UpdateNoteFunction: Handles updating notes.
  - DeleteNoteFunction: Handles deleting notes.

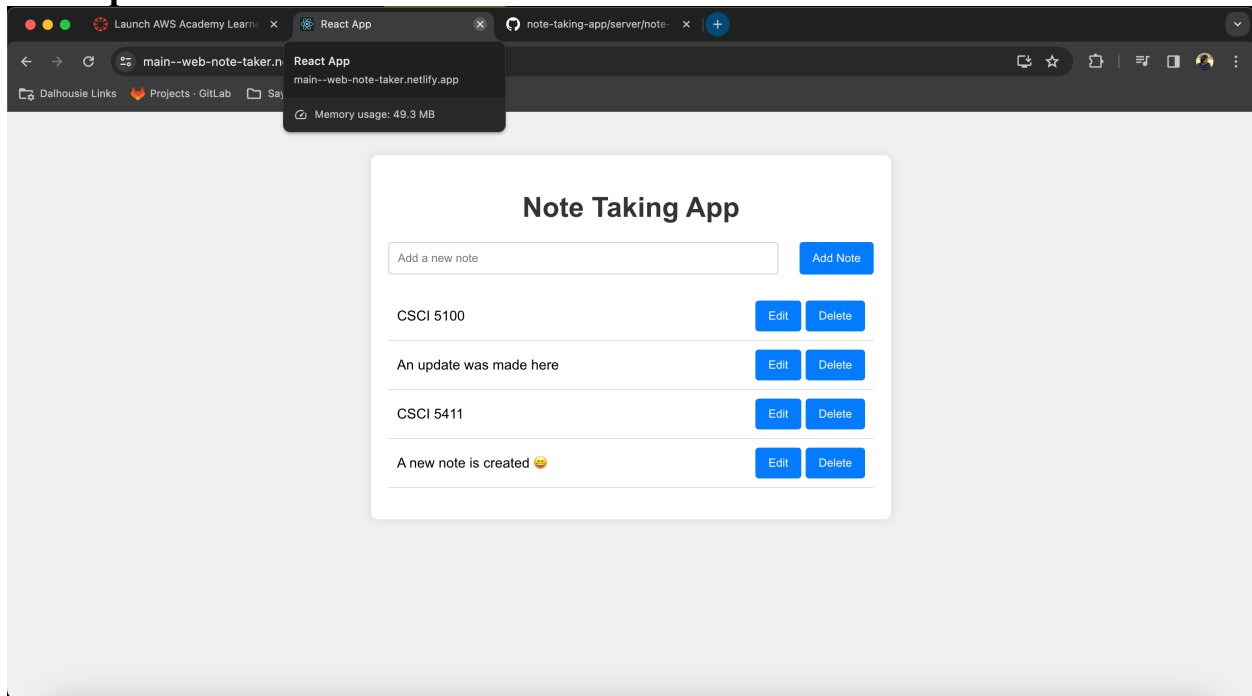
**2. Minimum of One Use-Case of S3 for Storage:**

## Activity 1

- The application uses an S3 bucket (NotesBucket) for storing logs of all CRUD operations, implemented through the Lambda functions.
- 3. **Minimum of Two Use Cases Interacting with DynamoDB:**
  - The application interacts with DynamoDB in multiple use cases:
    - Creating a note involves inserting an item into the NotesTable.
    - Reading notes involves scanning the NotesTable.
    - Updating a note involves updating an item in the NotesTable.
    - Deleting a note involves removing an item from the NotesTable.
- 4. **Technology for Development and Deployment:**
  - The backend of the application is developed using AWS services including Lambda, DynamoDB, S3, and API Gateway, managed through a CloudFormation script.
  - The frontend is a React App hosted on Netlify.

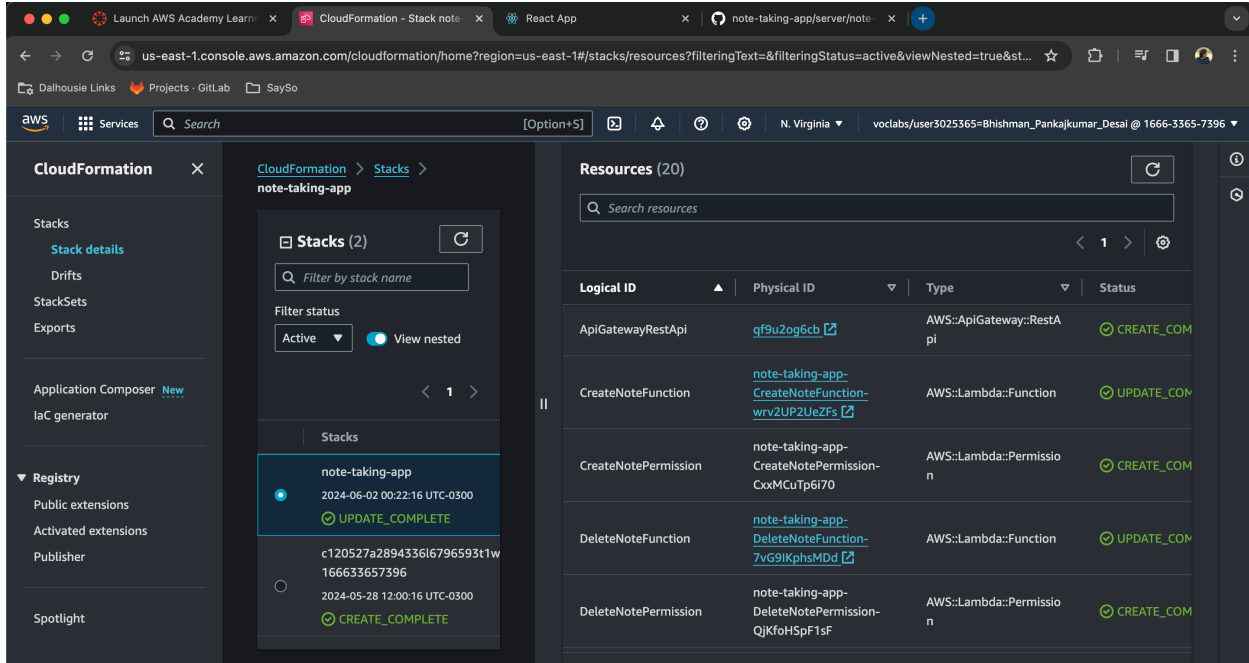
**c. Deployed URL**

<https://main--web-note-taker.netlify.app/>

**d. Output Screenshots:**

*Figure 1: Frontend of application*

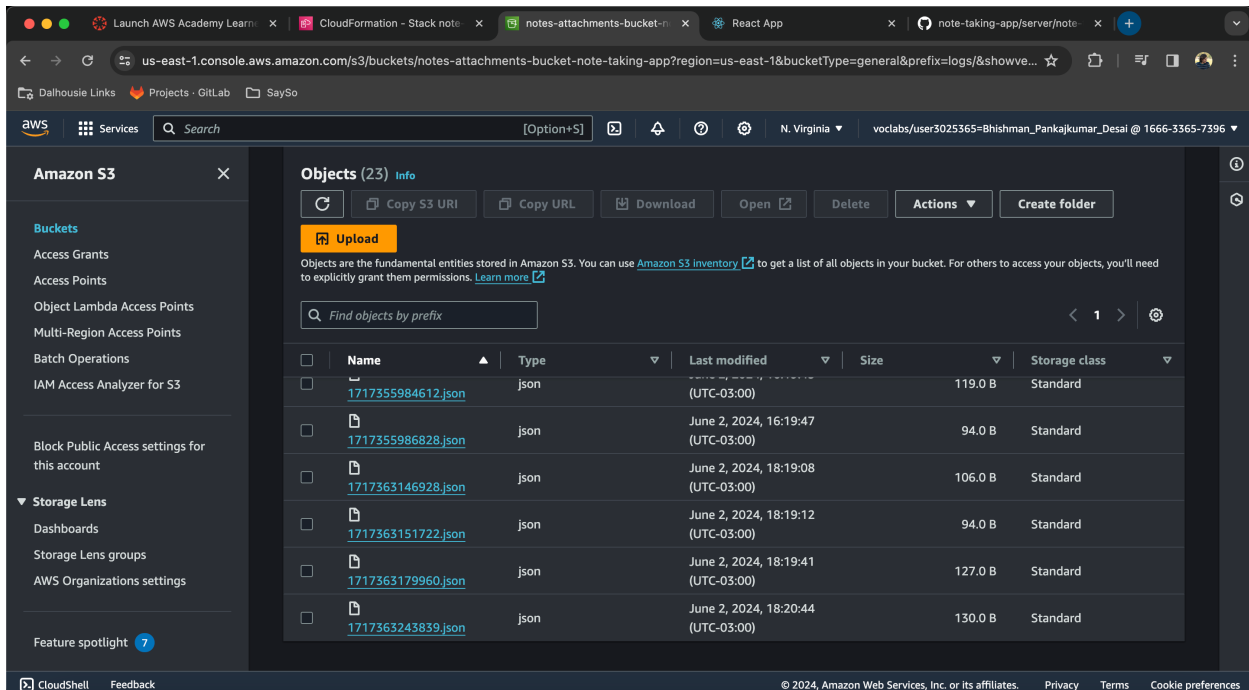
## Activity 1



The screenshot shows the AWS CloudFormation console for the 'note-taking-app' stack. The stack is in the 'UPDATE\_COMPLETE' state. The left sidebar shows the 'Stacks' section with 'Stack details' selected. The main area displays the 'Resources' section with 20 resources listed in a table.

Logical ID	Physical ID	Type	Status
ApiGatewayRestApi	qf9u2og6cb	AWS::ApiGateway::RestApi	CREATE_COMPLETE
CreateNoteFunction	note-taking-app-CreateNoteFunction-wrv2UP2Ue2Fs	AWS::Lambda::Function	UPDATE_COMPLETE
CreateNotePermission	note-taking-app-CreateNotePermission-CxxMCUtp6I70	AWS::Lambda::Permission	CREATE_COMPLETE
DeleteNoteFunction	note-taking-app-DeleteNoteFunction-7vG9IKphsMDd	AWS::Lambda::Function	UPDATE_COMPLETE
DeleteNotePermission	note-taking-app-DeleteNotePermission-QJkfoHSpF1sF	AWS::Lambda::Permission	CREATE_COMPLETE

Figure 2: Cloudformation stack with resources

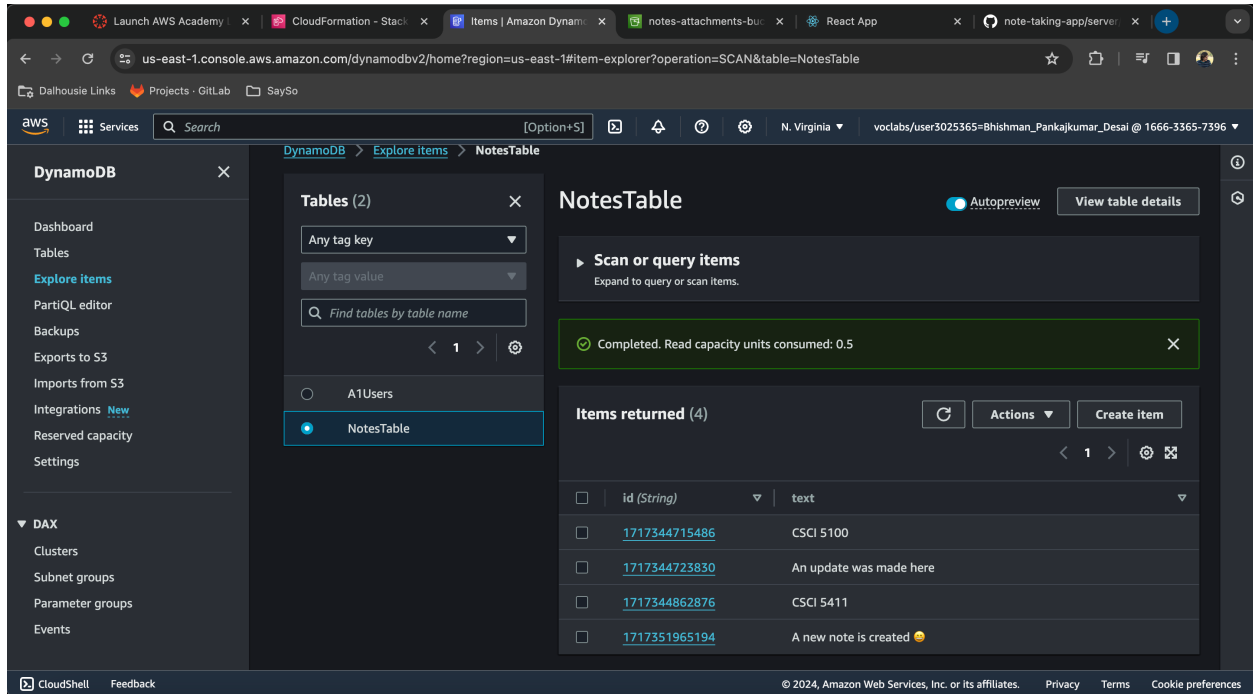


The screenshot shows the Amazon S3 console for the 'notes-attachments-bucket-note-taking-app' bucket. The bucket is in the 'us-east-1' region. The left sidebar shows the 'Buckets' section with 'Access Grants' selected. The main area displays the 'Objects' section with 23 objects listed in a table.

Name	Type	Last modified	Size	Storage class
1717355984612.json	json	June 2, 2024, 16:19:47 (UTC-03:00)	119.0 B	Standard
1717355986828.json	json	June 2, 2024, 16:19:47 (UTC-03:00)	94.0 B	Standard
1717363146928.json	json	June 2, 2024, 18:19:08 (UTC-03:00)	106.0 B	Standard
1717363151722.json	json	June 2, 2024, 18:19:12 (UTC-03:00)	94.0 B	Standard
1717363179960.json	json	June 2, 2024, 18:19:41 (UTC-03:00)	127.0 B	Standard
1717363243839.json	json	June 2, 2024, 18:20:44 (UTC-03:00)	130.0 B	Standard

Figure 3: Logs in S3

## Activity 1



The screenshot shows the AWS Management Console interface for the 'NotesTable' in DynamoDB. The left sidebar contains navigation links for 'DynamoDB', 'DAX', and 'CloudShell'. The main content area displays the 'NotesTable' with a 'Scan or query items' button and a 'Completed' status message. Below this, a table lists 4 items returned, each with an 'id (String)' and a 'text' field.

id (String)	text
1717344715486	CSCI 5100
1717344723830	An update was made here
1717344862876	CSCI 5411
1717351965194	A new note is created 🌟

Figure 4: DynamoDB entries

## Activity 1

**References:**

- [1] "HTTP APIs with Amazon DynamoDB," Amazon Web Services, Inc. [Online]. Available: <https://docs.aws.amazon.com/apigateway/latest/developerguide/http-api-dynamo-db.html>. [Accessed: June 2, 2024].
- [2] "Using AWS Lambda with DynamoDB for Serverless Applications," Reintech. [Online]. Available: <https://reintech.io/blog/using-aws-lambda-dynamodb-serverless-applications>. [Accessed: June 2, 2024].
- [3] "Amazon DynamoDB Auto Scaling: Performance and Cost Optimization at Any Scale," Amazon Web Services, Inc. [Online]. Available: <https://aws.amazon.com/blogs/database/amazon-dynamodb-auto-scaling-performance-and-cost-optimization-at-any-scale/>. [Accessed: June 2, 2024].
- [4] A. Obregon, "Deep Dive into AWS DynamoDB: Understanding the Core Features," Medium. [Online]. Available: <https://medium.com/@AlexanderObregon/deep-dive-into-aws-dynamodb-understanding-the-core-features-dc39cb3e14f2>. [Accessed: June 2, 2024].
- [5] "Disaster Recovery and Resiliency," Amazon Web Services, Inc. [Online]. Available: <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/disaster-recovery-resiliency.html>. [Accessed: June 2, 2024].
- [6] "Security and Resilience," Amazon Web Services, Inc. [Online]. Available: <https://docs.aws.amazon.com/lambda/latest/dg/security-resilience.html>. [Accessed: June 2, 2024].
- [7] "Serverless Architecture Design Patterns," CloudZero. [Online]. Available: <https://www.cloudzero.com/blog/serverless-architecture-design-patterns/>. [Accessed: June 2, 2024].
- [8] "Serverless Cost Efficiency," CloudZero. [Online]. Available: <https://www.cloudzero.com/blog/serverless-cost-efficiency/>. [Accessed: June 2, 2024].
- [9] "Security in Amazon DynamoDB," Amazon Web Services, Inc. [Online]. Available: <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/security.html>. [Accessed: June 2, 2024].
- [10] "AWS Lambda Security," Amazon Web Services, Inc. [Online]. Available: <https://docs.aws.amazon.com/lambda/latest/dg/lambda-security.html>. [Accessed: June 2, 2024].