PARSHVANATH CHARITABLE TRUST'S

A.P. Shah Institute of Technology

Thane, 400615

Academic Year: 2021-22

Department of Computer Engineering

CSL605 SKILL BASED LAB COURSE: CLOUD COMPUTING

Mini Project Report

- **Title of Project** : Note Taking App using AWS CDK

- **Year and Semester** : T.E. (Sem VI)

- **Group Members :**

Name and Roll No.
Prathamesh Hambar (18)
Het Patel (44)
Riddhi Narkar (41)
Tejas Sheth (65)

Table of Contents

Problem Definition

This project aims to build a rudimentary note-taking application using the latest cloud technologies. In general, note-taking applications are slow, memory-intensive and too complex even for a simple task. Also, we need to always keep a server live 24x7 to make it available to the users. This leads to a high cost of allocating resources, even in the cloud.

We solve these problems by

- Implementing it with asynchronous functions with the concepts of async, await & Promise in TypeScript - React.
- To make it cost-efficient, we make use of serverless technologies.

Introduction

These days, making a simple CRUD application is rather more complex than executing deep learning algorithms due to the presence of multiple moving parts i.e. components that need to be perfectly running simultaneously for a smooth user experience.

JavaScript is the world's most popular programming language. With the React.js framework, we can exploit the reusable components to make the development process faster. This coupled with the static typing and Promise function of TypeScript make it a perfect choice for making lightweight web applications.

With the coming of cloud computing, developers were able to enjoy the use of virtual machines (VMs). But this came with a huge cost of keeping the VMs live. Thus, the concept of serverless was born. In this, consumers have to only pay for the amount of time their application has actually utilized the computing power which is usually in milliseconds per request. This drastically reduces the cost of running the application.

Infrastructure as Code (IaC) is a fairly new concept. Here the developer codes all the cloud infrastructure requirements and deploys them with a single command on the command-line interface (CLI). This reduces the time to provision resources for an application if the developer knows exactly what they want.

The concept of using 3 tiers for the application comes from the need to combine the components interdependent and related on each other together. It also helps as we only give access to the resources for the layers that actually utilize them instead of giving access to everything to everyone.

Description

Below we have explained all the services that are associated with our mini-project.

AWS CloudFormation

AWS CloudFormation is a service that helps you model and set up your AWS resources so that you can spend less time managing those resources and more time focusing on your applications that run in AWS. You create a template that describes all the AWS resources that you want (like Amazon EC2 instances or Amazon RDS DB instances), and CloudFormation takes care of provisioning and configuring those resources for you. You don't need to individually create and configure AWS resources and figure out what's dependent on what; CloudFormation handles that.

The following scenarios demonstrate how CloudFormation can help.

- Simplify infrastructure management
- Easily control and track changes to your infrastructure

Cross-Origin Resource Sharing (CORS)

Cross-Origin Resource Sharing (CORS) is an HTTP-header based mechanism that allows a server to indicate any origins (domain, scheme, or port) other than its own from which a browser should permit loading resources. CORS also relies on a mechanism by which browsers make a "preflight" request to the server hosting the cross-origin resource, in order to check that the server will permit the actual request. In that preflight, the browser sends headers that indicate the HTTP method and headers that will be used in the actual request.

An example of a cross-origin request: the front-end JavaScript code served from https://domain-a.com uses XMLHttpRequest to make a request for https://domain-b.com/data.json.

Amazon S3

Amazon Simple Storage Service (Amazon S3) is an object storage service that offers industry-leading scalability, data availability, security, and performance. Customers of all sizes and industries can use Amazon S3 to store and protect any amount of data for a range of use cases, such as data lakes, websites, mobile applications, backup and restore, archive, enterprise applications, IoT devices, and big data analytics. Amazon S3 provides management features so that you can optimize, organize, and configure access to your data to meet your specific business, organizational, and compliance requirements.

Features of Amazon S3

- Storage classes
- Storage management
- Access management
- Data processing
- Storage logging and monitoring
- Automated monitoring tools
- Manual monitoring tools
- Analytics and insights
- Strong consistency

Amazon CloudFront

Amazon CloudFront is a web service that speeds up the distribution of your static and dynamic web content, such as .html, .css, .js, and image files, to your users. CloudFront delivers your content through a worldwide network of data centres called edge locations. When a user requests content that you're serving with CloudFront, the request is routed to the edge location that provides the lowest latency (time delay), so that content is delivered with the best possible performance. If the content is already in the edge location with the lowest latency, CloudFront delivers it immediately.

If the content is not in that edge location, CloudFront retrieves it from an origin that you've defined—such as an Amazon S3 bucket, a MediaPackage channel, or an HTTP server (for example, a web server) that you have identified as the source for the definitive version of your content.

CloudFront speeds up the distribution of your content by routing each user request through the AWS backbone network to the edge location that can best serve your content. Typically, this is a CloudFront edge server that provides the fastest delivery to the viewer. Using the AWS network dramatically reduces the number of networks that your users' requests must pass through, which improves performance. Users get lower latency—the time it takes to load the first byte of the file—and higher data transfer rates. You also get increased reliability and availability because copies of your files (also known as objects) are now held (or cached) in multiple edge locations around the world.

Amazon API Gateway is a fully managed service that makes it easy for developers to create, publish, maintain, monitor, and secure APIs at any scale. APIs act as the "front door" for applications to access data, business logic, or functionality from your backend services. Using API Gateway, you can create RESTful APIs and WebSocket APIs that enable real-time two-way communication applications. API Gateway supports containerized and serverless workloads, as well as web applications.

API Gateway handles all the tasks involved in accepting and processing up to hundreds of thousands of concurrent API calls, including traffic management, CORS support, authorization and access control, throttling, monitoring, and API version management. API Gateway has no minimum fees or startup costs. You pay for the API calls you receive and the amount of data transferred out and, with the API Gateway tiered pricing model, you can reduce your cost as your API usage scales.

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.
With DynamoDB, you can create database tables that can store and retrieve any amount of data and serve any level of request traffic. You can scale up or scale down your tables' throughput capacity without downtime or performance degradation. You can use the AWS Management Console to monitor resource utilization and performance metrics.

AWS Lambda

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, code monitoring and logging. With Lambda, you can run code for virtually any type of application or backend service. All you need to do is supply your code in one of the languages that Lambda supports.

You organize your code into Lambda functions. Lambda runs your function only when needed and scales automatically, from a few requests per day to thousands per second. You pay only for the computer time that you consume—there is no charge when your code is not running.

You can invoke your Lambda functions using the Lambda API, or Lambda can run your functions in response to events from other AWS services.

For example, you can use Lambda to:

1) Build data-processing triggers for AWS services such as Amazon Simple Storage Service (Amazon S3) and Amazon DynamoDB.
2) Process streaming data stored in Amazon Kinesis.
3) Create your own backend that operates at AWS scale, performance, and security.
4) Lambda is a highly available service.

Lambda features:

- Concurrency and scaling controls
- Code signing.
- Lambda extensions
- Function blueprints
- Database access
- File systems access

IAM

AWS Identity and Access Management (IAM) is a web service that helps you securely control access to AWS resources. You use IAM to control who is authenticated (signed in) and authorized (has permissions) to use resources.

When you first create an AWS account, you begin with a single sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account root user and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you do not use the root user for your everyday tasks, even the administrative ones. Instead, adhere to the best practice of using the root user only to create your first IAM user. Then securely lock away the root user credentials and use them to perform only a few account and service management tasks.

CDK

The AWS Cloud Development Kit (AWS CDK) is an open-source software development framework to define your cloud application resources using familiar programming languages.

Provisioning cloud applications can be a challenging process that requires you to perform manual actions, write custom scripts, maintain templates, or learn domain-specific languages. AWS CDK uses the familiarity and expressive power of programming languages for modelling your applications. It provides high-level components called constructs that preconfigure cloud resources with proven defaults, so you can build cloud applications with ease. AWS CDK provisions your resources in a safe, repeatable manner through AWS CloudFormation. It also allows you to compose and share your own custom constructs incorporating your organization's requirements, helping you expedite new projects.

For customers who prefer Terraform, cdktf provides the CDK constructs for defining Terraform HCL state files in TypeScript and Python. For Kubernetes users, the cdk8s project allows you to use CDK constructs for defining Kubernetes configuration in TypeScript, Python, and Java. In addition, cdk8s can be used to define Kubernetes infrastructure running anywhere and can be used with the AWS CDK’s Amazon Elastic Kubernetes Service (Amazon EKS) construct library. Both cdk8s and cdktf are alpha releases. To find all of these CDKs in one place, check out Construct Hub, a place to discover and share construct libraries published by the open-source community, AWS, and partners.

Implementation

```
export class CcProjectStack extends Stack {
  constructor(scope: Construct, id: string, props?: StackProps) {
    super(scope, id, props);

    // The code that defines our stack goes here

    const table = new Table(this, 'NotesTable', {
      billingMode: BillingMode.PAY_PER_REQUEST,
      partitionKey: { name: 'pk', type: AttributeType.STRING },
      removalPolicy: RemovalPolicy.DESTROY,
      sortKey: { name: 'sk', type: AttributeType.STRING },
      tableName: 'NotesTable',
    });

    // The TypeScript Lambda functions need to be transpiled before runtime.
    // Thus, we NodejsFunction construct which uses esbuild, a very fast transpiler.
    const readFunction = new NodejsFunction(this, 'ReadNotesFn', {
      architecture: Architecture.ARM_64,
      entry: `${__dirname}/fns/readFunction.ts`,
      logRetention: RetentionDays.ONE_WEEK,
    });
    const writeFunction = new NodejsFunction(this, 'WriteNoteFn', {
      architecture: Architecture.ARM_64,
      entry: `${__dirname}/fns/writeFunction.ts`,
      logRetention: RetentionDays.ONE_WEEK,
    });

    // Grant table access to Lambda functions
    table.grantReadData(readFunction);
    table.grantWriteData(writeFunction);
```

```
// The base construct with a CORS configuration to create
the HTTP API,
    // for it to be served from a CloudFront domain.
    const api = new HttpApi(this, 'NotesApi', {
      corsPreflight: {
        allowHeaders: ['Content-Type'],
        allowMethods: [CorsHttpMethod.GET, CorsHttpMethod
.POST],
        allowOrigins: ['*'],
      },
    });

    // Integration constructs.
    const readIntegration = new HttpLambdaIntegration(
      'ReadIntegration',
      readFunction
    );
    const writeIntegration = new HttpLambdaIntegration(
      'WriteIntegration',
      writeFunction
    );

    // Assign a path/route to the integrations.
    api.addRoutes({
      integration: readIntegration,
      methods: [HttpMethod.GET],
      path: '/notes',
    });

    api.addRoutes({
      integration: writeIntegration,
      methods: [HttpMethod.POST],
      path: '/notes',
    });
```

```
// Storage for assets only
    const websiteBucket = new Bucket(this, 'WebsiteBucket', {
      autoDeleteObjects: true,
      blockPublicAccess: BlockPublicAccess.BLOCK_ALL,
      removalPolicy: RemovalPolicy.DESTROY,
    });

    // Grant read access to the distribution
    const originAccessIdentity = new OriginAccessIdentity(
      this,
      'OriginAccessIdentity'
    );
    websiteBucket.grantRead(originAccessIdentity);

    const distribution = new Distribution(this, 'Distribution', {
      defaultBehavior: {
        origin: new S3Origin(websiteBucket, { originAccessIdentity }),
        viewerProtocolPolicy: ViewerProtocolPolicy.REDIRECT_TO_HTTPS,
      },
      defaultRootObject: 'index.html',
      errorResponses: [
        {
          httpStatus: 404,
          responseHttpStatus: 200,
          responsePagePath: '/index.html',
        },
      ],
    });
```

```
const execOptions: ExecSyncOptions = {
      stdio: ['ignore', process.stderr, 'inherit'],
    };

    const bundle = Source.asset(join(__dirname, 'web'), {
      bundling: {
        command: [
          'sh',
          '-c',
          'echo "Docker build not supported. Please install esbuild."',
        ],
        image: DockerImage.fromRegistry('alpine'),
        local: {
          tryBundle(outputDir: string) {
            try {
              execSync('esbuild --version', execOptions);
            } catch {
              return false;
            }
            execSync('npx vite build', execOptions);
            copySync(join(__dirname, '../dist'), outputDir, {
              ...execOptions,
              recursive: true,
            });
            return true;
          },
        },
      },
    });
```

```
new BucketDeployment(this, 'DeployWebsite', {
      destinationBucket: websiteBucket,
      distribution,
      logRetention: RetentionDays.ONE_DAY,
      prune: false,
      sources: [bundle],
    });

    new AwsCustomResource(this, 'ApiUrlResource', {
      logRetention: RetentionDays.ONE_DAY,
      onUpdate: {
        action: 'putObject',
        parameters: {
          Body: Stack.of(this).toJsonString({
            [this.stackName]: { HttpApiUrl: api.apiEndpoint },
          }),
          Bucket: websiteBucket.bucketName,
          CacheControl: 'max-age=0, no-cache, no-store, must-revalidate',
          ContentType: 'application/json',
          Key: 'config.json',
        },
        physicalResourceId: PhysicalResourceId.of('config'),
        service: 'S3',
      },
      policy: AwsCustomResourcePolicy.fromStatements([
        new PolicyStatement({
          actions: ['s3:PutObject'],
          resources: [websiteBucket.arnForObjects('config.json')],
        }),
      ]),
    });

    // To get the generated endpoint
    new CfnOutput(this, 'HttpApiUrl', { value: api.apiEndpoint });
    new CfnOutput(this, 'DistributionDomain', {
      value: distribution.distributionDomainName,
    });
  }
}
```

```
// Initialize DB
const client = new Dynamo({ client: new DynamoDBClient({}) });

// Schema
const schema = {
  indexes: {
    primary: {
      hash: 'pk',
      sort: 'sk',
    },
  },
  models: {
    note: {
      type: {
        required: true,
        type: 'string',
        value: 'note',
      },
      pk: {
        type: 'string',
        value: 'note',
      },
      sk: {
        type: 'string',
        value: '${date}',
      },
      note: {
        required: true,
        type: 'string',
      },
      date: {
        required: true,
        type: 'string',
      },
      subject: {
        required: true,
        type: 'string',
      },
    },
  },
  version: '0.1.0',
  params: {
    typeField: 'type',
  },
  format: 'onetable:1.0.0',
} as const;

export type NoteType = Entity<typeof schema.models.note>;

const table = new Table({
  client,
  name: 'NotesTable',
  schema,
  timestamps: true,
});

export const Notes = table.getModel<NoteType>('note');
```

```
import { Notes } from './notesTable';

export const handler = async (): Promise<APIGatewayProxyResultV2> => {
  const notes = await Notes.find({ pk: 'note' }, { limit: 10, reverse: true });

// Adding the limit and reverse parameters means the query will return the ten
 most recent notes,
  // automatically sorted by the sort key.
  return { body: JSON.stringify(notes), statusCode: 200 };
};
```

```
import { Notes } from './notesTable';

export const handler = async (
  event: APIGatewayProxyEventV2
): Promise<APIGatewayProxyResultV2> => {
  const body = event.body;
  if (body) {
    const notes = await Notes.create(JSON.parse(body));
    return { body: JSON.stringify(notes), statusCode: 200 };
  }
  return { body: 'Error, invalid input!', statusCode: 400 };
};
```

```ts
import { NoteType } from '../fns/notesTable';

let url = '';

const getUrl = async () => {
  if (url) {
    return url;
  }
  const response = await fetch('./config.json');
  url = `${(await response.json()).CcProjectStack.HttpApiUrl}/notes`;
  return url;
};

export const getNotes = async () => {
  const result = await fetch(await getUrl());
  return await result.json();
};

export const saveNote = async (note: NoteType) => {
  await fetch(await getUrl(), {
    body: JSON.stringify(note),
    headers: { 'Content-Type': 'application/json' },
    method: 'POST',
    mode: 'cors',
  });
};
```

Learning Outcome

Thus, we learned -

1. Infrastructure as Code
2. 3-tier web application design
3. Serverless technologies
4. Containerization using Docker
5. To deploy a full-stack web application