Creating a serverless contact Form using AWS API gateway and AWS Lambda

Project Overview :

Introduction: The Serverless Contact Form project aims to provide a seamless and cost-effective solution for implementing a contact form on a website or application using AWS services, namely AWS API Gateway and AWS Lambda. Traditional contact forms often rely on server-side processing and hosting, which can be difficult to set up and maintain. By leveraging serverless architecture, this project offers a more efficient and scalable approach to handling contact form submissions

In this project, we have built a serverless contact form for a static website. The contact form allows visitors to submit their data and a message, which will be securely stored in a DynamoDB database.

Objectives:

- Build a Serverless Contact Form: Develop a contact form seamlessly integrated into a static website.
- Implement Secure and Scalable Backend: Utilize AWS Lambda, API Gateway, and DynamoDB to create a secure and scalable backend infrastructure.
- Store Form Data in DynamoDB: Persist submitted form data in a DynamoDB database for future reference and analysis, ensuring data integrity and accessibility.
- Enable Smooth Form Submission with JavaScript: Utilize JavaScript's fetch API to facilitate smooth form submission and efficient data handling between the client and server.
- Implement CORS for Secure Communication: Implement Cross-Origin Resource Sharing (CORS) to enable secure communication between the static website and the serverless backend, enhancing security and accessibility.
- Leverage AWS Services for Infrastructure Management: Utilize AWS services for infrastructure management to reduce operational overhead, ensuring efficient deployment and maintenance of the serverless architecture.
- Understand Serverless Architecture Principles and Benefits: Gain a deeper understanding of the principles and benefits of serverless architecture for web applications, including scalability, cost-effectiveness, and ease of development and deployment.

Scope: The project will focus on building a serverless backend infrastructure to handle contact form submissions. This includes:

- AWS API Gateway: Setting up API endpoints to receive form submissions securely.
- AWS Lambda: Creating Lambda functions to process incoming form submissions and send notifications.
- Data Storage: Optionally, integrating with AWS services like Amazon Simple Storage Service (S3) or Amazon DynamoDB to store form submissions for future reference.

Project Architecture :

Components:

- AWS API Gateway: Manages RESTful API endpoints for receiving form submissions securely.
- AWS Lambda: Executes serverless functions triggered by API Gateway events to process form submissions.
- Amazon S3: Used to host the static website files and serve the frontend to users.
- AWS IAM: Manages access control and permissions for services and resources within the architecture.
- Amazon DynamoDB: Stores form submission data if real-time database storage is necessary.
- Amazon CloudWatch: Monitors Lambda function invocations, API Gateway performance, and provides logs and metrics for monitoring and troubleshooting.

Networking Architecture:

- VPC Setup: Create a new VPC with a defined CIDR block.
- Subnets: Divide the VPC into public and private subnets.
- Routing: Configure route tables for internal and external traffic routing.
- Security Groups: Implement security groups to control inbound and outbound traffic for resources like Lambda functions and DynamoDB.

Compute:

For serverless contact form project on AWS, the compute resources primarily consist of AWS Lambda functions:

- AWS Lambda Functions: Lambda functions are used to handle form submissions and process incoming data from the contact form. These functions are invoked by events triggered by AWS API Gateway upon form submission.

Storage:

The primary storage solutions employed are Amazon S3 buckets and DynamoDB:

- Amazon S3 Buckets: Used to store static website files such as HTML, CSS, and JavaScript for hosting the contact form. The static website files hosted in S3 provide a URL for accessing the contact form.
- Amazon DynamoDB: Used as a NoSQL database to store submitted form data for future reference and analysis. The contact form Lambda functions can interact with DynamoDB to store and retrieve form submission data securely.

Security Measures:

- IAM Roles: Assign granular permissions to Lambda functions and other resources using IAM roles, adhering to the principle of least privilege.

Monitoring and Logging:

- Amazon CloudWatch: Monitors Lambda function invocations, API Gateway performance, and provides logs and metrics for monitoring and troubleshooting.

Scalability and High Availability:

- AWS Lambda: Automatically scales to accommodate varying levels of form submissions without manual intervention.

- AWS API Gateway: Scales automatically to handle increased traffic, ensuring high availability of the contact form endpoints.
- Amazon DynamoDB: Designed for high availability and scalability, providing consistent performance for read and write operations.

Implementation Details :

Configuration Steps:

- Build the Frontend:
    - Structure your project with folders for website files and Lambda function.
    - Create an index.html, style.css, and script.js file in the website folder.
    - Create an index.js and package.json file in the lambda folder.

```
- /
    - website/
        - index.html
        - style.css
        - script.js
    - lambda/
        - index.js
        - package.json
```

- Create the Lambda Function:
    - Write Lambda function code to process form submissions using AWS SDK to interact with DynamoDB.
    - Test the function locally to ensure correct functionality.
    - Zip the lambda folder, create an S3 bucket, and upload the zip file. Copy the object URL.
    - Upload the zip file to Lambda from the S3 location. Create a test event and save it for testing.
    - Check CloudWatch logs for any issues.

- Set up API Gateway and deploy the endpoint URL:
    - Create a Create a private REST API in the AWS Management Console.
    - Define a POST method and integrate it with the Lambda function.
    - Enable CORS to allow cross-origin requests.
    - Deploy the API to generate the endpoint URL.
- Create the DynamoDB Table:
    - Create a DynamoDB table to store form submission data in the AWS Management Console.
    - Define a string partition key and leave default settings.
    - Add a DynamoDB policy to the Lambda function's role to allow putting new items into the database.
    - Grant DynamoDB full access to the Lambda function.

- Upload the Website to S3:
    - Create an S3 bucket to host the static website.
    - Upload the website files to the bucket.
    - Replace the fetch line in script.js with the API Gateway's Invoke URL followed by /submit.

- Configure CORS(Cross Origin Resource Sharing) :
    - Configure CORS on the S3 bucket and API Gateway to allow requests from different origins.
    - Edit the CORS settings in the S3 bucket's Permissions section to include necessary headers.
    - Enable CORS on the API Gateway's /submit resource to allow requests from the S3 website domain.
    -
- Test the Application:
    - Access the static website using the provided S3 website endpoint.
    - Fill out the form and submit it.
    - Verify that the form data is stored in DynamoDB by checking the confirmation page or directly querying the table.

Challenges Faced:

- Integration Complexity:

Orchestrating multiple AWS services like API Gateway, Lambda, and DynamoDB posed integration challenges due to their diverse configurations.

- Cross-Origin Resource Sharing (CORS):

Configuring CORS settings in API Gateway for cross-origin requests from the S3-hosted frontend required careful consideration and testing.

Optimizations:

- Performance Improvement:
    - Optimized Lambda function code to minimize execution time and maximize resource utilization.
    - Fine-tuned DynamoDB operations to enhance query efficiency and reduce latency.

- Scalability Optimization:
    - Leveraged AWS auto-scaling features to automatically adjust resources based on demand, ensuring seamless scalability without overprovisioning.

Integration:

- Frontend and Backend Integration:
    - The frontend, consisting of the HTML, CSS, and JavaScript files hosted on Amazon S3, communicates with the backend Lambda function through the API Gateway.
    - When a user submits the contact form on the frontend, a POST request is sent to the API Gateway endpoint.
    - The API Gateway triggers the associated Lambda function, passing the form data as an event payload.

- Lambda Function and DynamoDB Integration:
    - The Lambda function, written in Node.js, receives the form data from the API Gateway.
    - It processes the data, validates inputs, and interacts with DynamoDB using the AWS SDK to store the form submission.
    - DynamoDB acts as the backend database, storing the form submission data securely.
- Cross-Origin Communication (CORS):
    - CORS is configured on both the API Gateway and the S3 bucket hosting the static website to allow secure communication between them.
    - This enables the frontend hosted on S3 to make cross-origin requests to the API Gateway endpoint without encountering browser restrictions.
- Logging and Monitoring:
    - CloudWatch is utilized for logging and monitoring across different components of the solution.
    - CloudWatch Logs captures logs generated by Lambda functions, API Gateway requests, and DynamoDB operations.
    - CloudWatch Metrics provides metrics for monitoring Lambda function invocations, API Gateway performance, and DynamoDB throughput.
- Permissions and Access Control:
    - IAM roles and policies are configured to control access to resources and services.
    - The Lambda function's IAM role is granted permissions to interact with DynamoDB, ensuring secure access to the database.
    - IAM policies are also used to define permissions for API Gateway endpoints, restricting access based on user roles and authentication.

Testing and Validation :

- **Functional Testting:**
    - Conducted thorough functional testing to ensure all features and functionalities of the solution meet the specified requirements.
    - Tested individual components such as Lambda functions, API Gateway endpoints, and DynamoDB operations to verify proper functionality.

- **User Acceptance Testing(UAT):**
    - Engaged end-users and stakeholders in user acceptance testing to validate the solution's usability, effectiveness, and alignment with business needs.
    - Gathered feedback through user testing sessions and surveys to identify any issues or areas for improvement before deployment.

Results and Evaluation :

Achievements:

- Successful implementation of a serverless contact form solution using AWS Lambda, API Gateway, DynamoDB, and S3.
- Seamless integration between frontend and backend components, allowing users to submit form data securely.

- Efficient handling of form submissions, with data stored reliably in DynamoDB for future reference and analysis.

Lessons Learned:

- Importance of thorough planning and requirements gathering to ensure a clear understanding of project objectives and scope.
- Significance of proper documentation and knowledge sharing to facilitate collaboration among team members and stakeholders.
- Understanding the nuances of AWS services and configurations to optimize performance, cost, and security.

Future Recommendations:

- Implement automated deployment pipelines using AWS CodePipeline and AWS CodeBuild for continuous integration and deployment (CI/CD).
- Explore serverless monitoring and observability tools such as AWS X-Ray or third-party solutions to gain deeper insights into system performance and behavior.
- Consider implementing user authentication and authorization mechanisms to restrict access to the contact form and ensure data privacy.

Conclusion :

In conclusion, this project exemplifies the implementation of a serverless contact form using AWS services, showcasing the seamless integration of a static website hosted on Amazon S3 with API Gateway and Lambda functions. By securely storing contact form data in DynamoDB, the project underscores the benefits of serverless architecture, including scalability, flexibility, and cost-efficiency.

Key Takeaways:

- Serverless Architecture: The project emphasizes the advantages of serverless computing, including automatic scaling and reduced operational overhead.
- AWS Services Integration: It demonstrates effective integration of Amazon S3, API Gateway, and Lambda to construct a serverless application.
- Data Storage: Utilization of DynamoDB ensures secure storage of contact form data.
- Frontend-Backend Separation: The project's separation of frontend and backend components facilitates easier maintenance and scalability.
- Security Considerations: Addressing security aspects such as CORS configuration and implementing necessary measures within API Gateway and Lambda functions ensures data protection and compliance.