**Contact Form Documentation**

**Serverless Contact Form Using AWS Services**

Jonathan Bernard, Ashan Viville, & Avi Patel  
Monroe College  
CS670 – Cloud Computing  
Instructor: Mahmud Islam

7/29/2025

**Step 1: Frontend Implementation – Contact Form Development**

The first phase of the group project focused on developing the **frontend user interface** for the serverless contact form application. The team implemented a responsive and accessible form using **HTML**, **CSS**, and **JavaScript**, laying the foundation for client-side interaction in the overall system architecture.

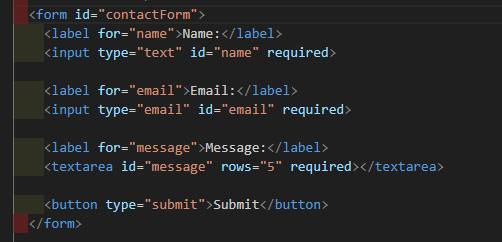
The form allows users to enter their **name**, **email**, and a **message**, and is designed to send data as a JSON payload via an HTTP POST request. While backend services such as **AWS API Gateway** and **Lambda** are not yet connected at this stage, the JavaScript logic simulates the request flow and provides real-time user feedback upon form submission.

The code was developed using **Visual Studio Code** and tested locally in a web browser. Inline **CSS styling** was included to ensure a user-friendly and aesthetically consistent interface, with proper alignment, spacing, and card-style design that enhances readability and accessibility. The form layout is optimized for both desktop and mobile experiences.

This frontend component will later be hosted on **Amazon S3** as a static website, enabling global access to the form. Subsequent steps will integrate the form with API Gateway, Lambda, and DynamoDB for full serverless functionality.

**Key Code Snippets:**

**HTML Form Structure**

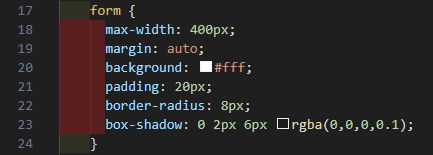


Defines the form layout and input fields. Each element is tied to a DOM reference for JavaScript data handling. The required attribute ensures client-side validation.

**JavaScript Submit Logic:**

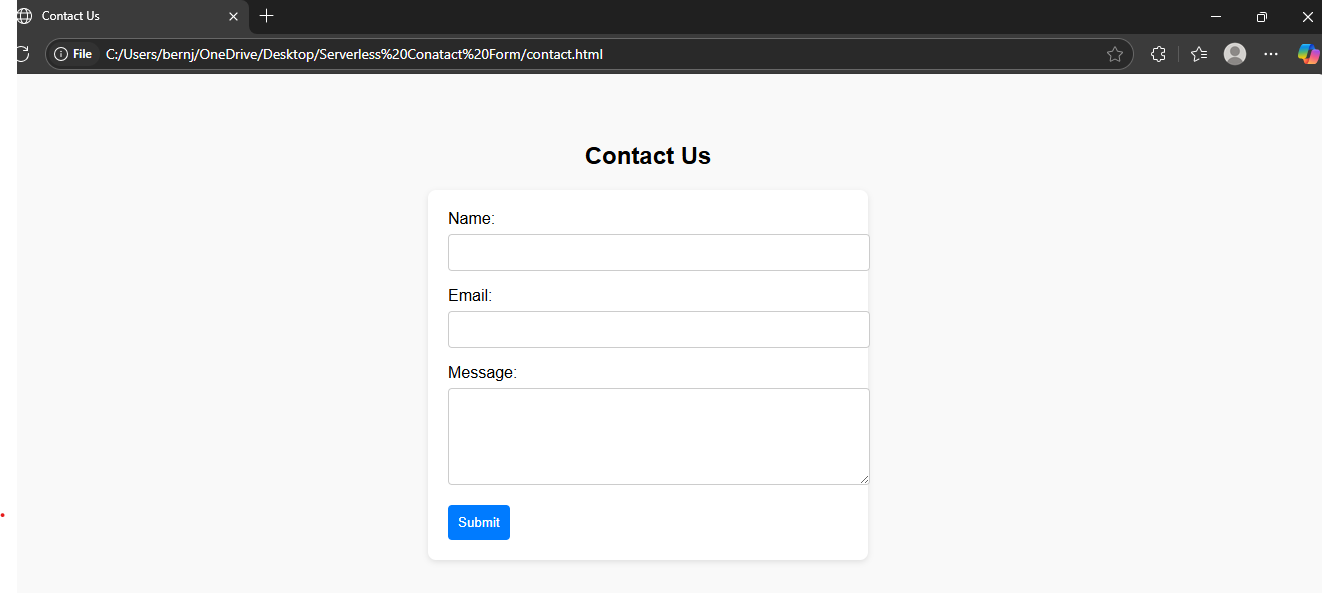
Implements asynchronous form submission via the Fetch API. Includes error handling and updates the DOM to display status messages for success or failure.

**CSS Styling Snippet:**

****

Applies a minimalist and accessible card-style layout. Improves readability and structure through spacing, borders, and centered alignment.

**Form Displayed in Local Browser**

****

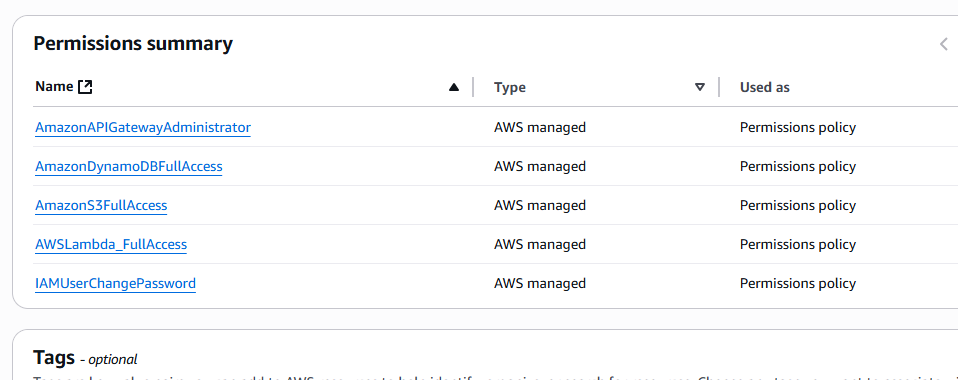
This screenshot shows the contact form rendered in a local browser after being built and saved as contact.html. The form is structured with clearly labeled input fields for name, email, and message, styled using internal CSS to appear centered and responsive. This visual confirmation demonstrates successful local testing before deployment to AWS.

**AWS IAM Setup for Group Collaboration (Step 2 - Part 1):**

To enable secure and collaborative development, a dedicated IAM group was created with custom permissions. Group members were given access to AWS Management Console using IAM Identity Center.  
Five key policies were attached:

* AWSLambda\_FullAccess: Deploy & manage serverless functions.
* AmazonDynamoDBFullAccess: Store and retrieve form data.
* AmazonAPIGatewayAdministrator: Integrate Lambda with API Gateway.
* AmazonS3FullAccess: (Optional) Storage for uploaded files.
* IAMUserChangePassword: Allow user-level credential management.

Each group member was added to this IAM group to ensure shared access to all AWS resources needed for the project. This setup supports version-controlled, serverless architecture development in a collaborative cloud environment.



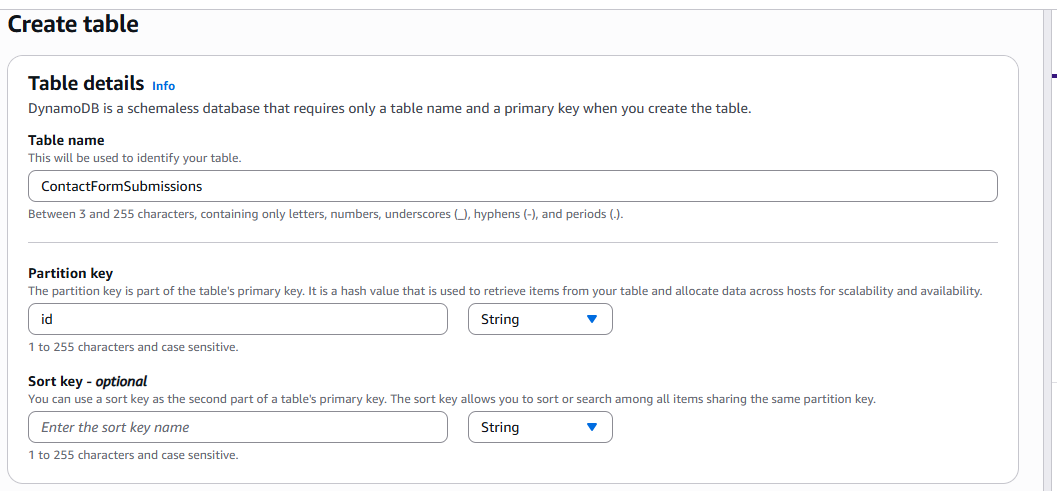
**Step 2 – DynamoDB Table Creation**

This step involved creating a serverless database layer for storing user-submitted contact form data using Amazon DynamoDB.

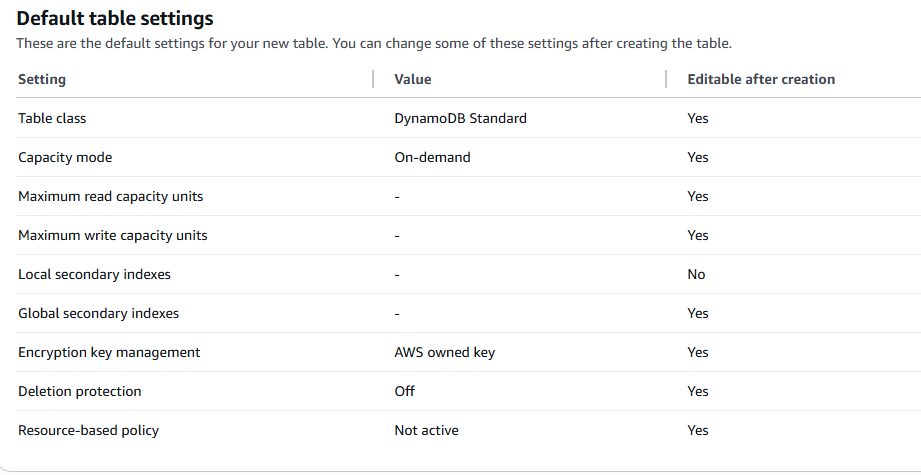
**Implementation Details**

* A DynamoDB table named ContactFormSubmissions was created to store contact form submissions from users.
* A single **partition key**, id (String), was defined to uniquely identify each submission.
* **On-Demand capacity mode** was selected to allow automatic scaling without manual provisioning.
* **AWS-managed encryption** was used to secure data at rest.
* **No sort key or secondary indexes** were required for the basic functionality of the application.

**Figure 1 Table Configuration Settings (Partition Key):**

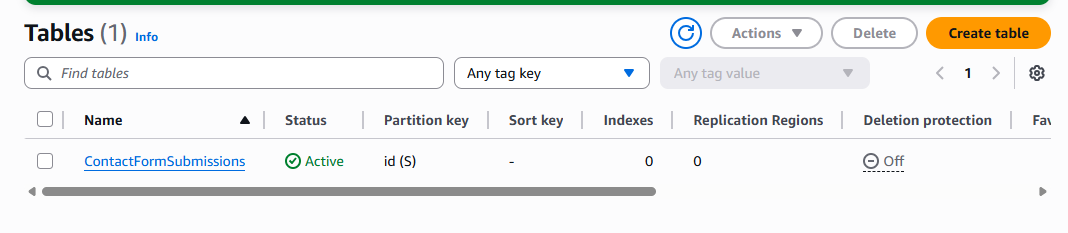


This screenshot shows the initial configuration of the DynamoDB table, where the table name ContactFormSubmissions and partition key id (String) were defined.

**Figure 2 – Default Table Settings Confirmation**

This shows the confirmation of default AWS settings such as on-demand capacity, encryption using AWS-managed keys, and no secondary indexes.

**Figure 3 – Table Status: Active**



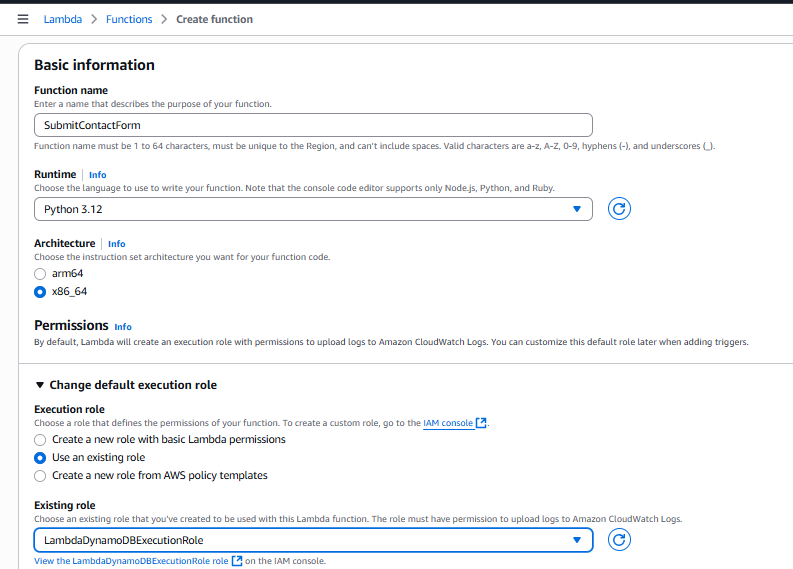
This confirms that the table was successfully created and is now in Active status, ready for Lambda and API Gateway integration in upcoming steps.

The creation of the DynamoDB table provides a secure, scalable, and serverless data backend for the contact form application. Using Amazon DynamoDB allows the system to handle submissions efficiently without managing any database infrastructure. This aligns with AWS best practices and satisfies the backend storage component of the project proposal. The table will be connected to a Lambda function in the next step to handle automatic storage of form submissions.

**Step 3: Lambda Function Configuration and Deployment**

To handle the backend processing of our serverless contact form, we created and configured an AWS Lambda function titled SubmitContactForm.

**Screenshot 3.1 – Lambda Function Creation Settings**

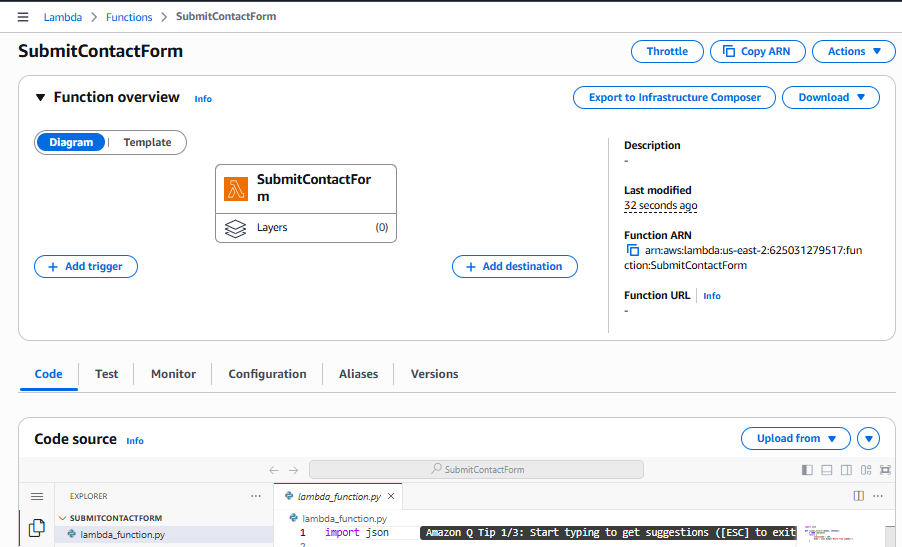


In the Lambda creation interface, we configured the following:

* Function Name: SubmitContactForm – clearly reflecting its role in processing form submissions.
* Runtime: Python 3.12, chosen for its compatibility with AWS Lambda and its extensive support for JSON parsing and AWS SDK integration (boto3).
* Architecture: We selected x86\_64, the default and broadly compatible processor architecture.

For permissions, we opted to use an existing IAM role named LambdaDynamoDBExecutionRole. This role was previously created and grants the Lambda function the necessary permissions to interact with Amazon DynamoDB and Amazon CloudWatch Logs. This avoids redundant role creation and ensures principle of least privilege is enforced by reusing scoped permissions.

**Screenshot 3.2 – Lambda Function Overview**



After creating the function, we were directed to the function dashboard where:

* The function **ARN (Amazon Resource Name)** is displayed, which will later be needed to integrate with the API Gateway.
* Under the **Code** tab, the default lambda\_function.py file was initialized. This file serves as the entry point for the backend logic that will receive and store contact form submissions into the DynamoDB table.

This completes the infrastructure provisioning of the Lambda compute service and prepares us to move into implementing and testing the actual function logic.

**Step 4: Implementation and Testing of AWS Lambda Function**

**Objective**

The primary goal of this step was to create a serverless AWS Lambda function that would process contact form submissions by receiving a JSON payload, extracting its fields, and storing the data in a DynamoDB table. This function acts as the backend logic for form handling in a scalable, event-driven architecture.

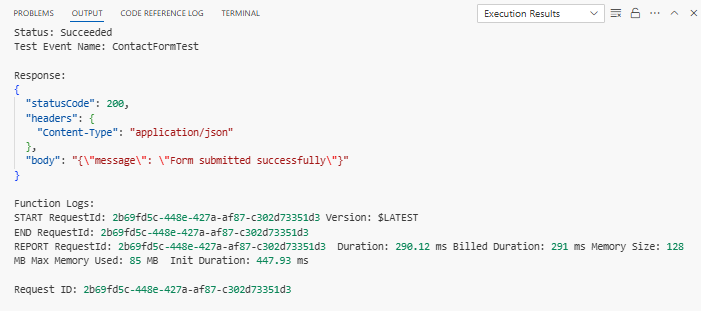
**Procedure**

The Lambda function was written using Python 3.12 and deployed directly from the AWS Console. The boto3 library was used to interact with DynamoDB, while the uuid module was included to assign a unique identifier to each form submission. The following sequence was executed:

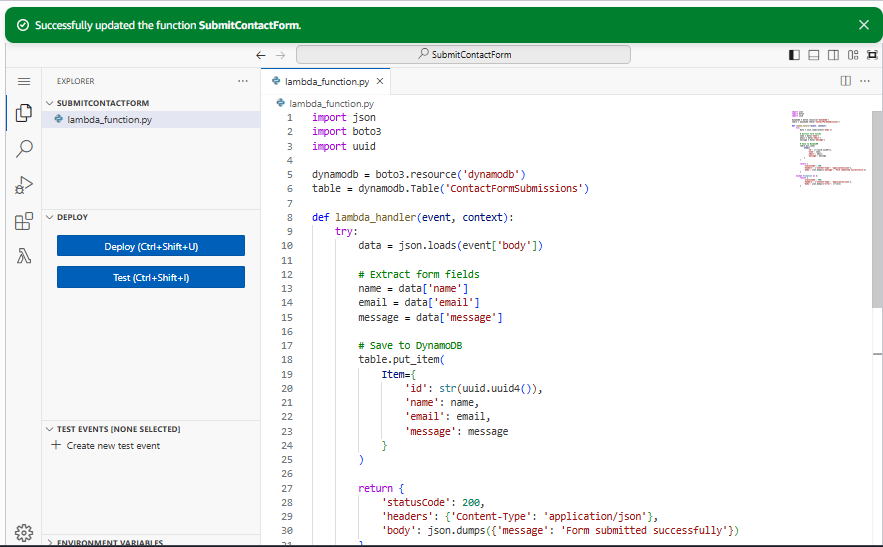
1. Lambda Code Creation  
   The function was developed inside the AWS Lambda code editor (lambda\_function.py). Upon invocation, it:
   * Parses the event body using the json library.
   * Extracts the name, email, and message fields from the incoming payload.
   * Generates a UUID for each submission.
   * Inserts the data into the DynamoDB table titled ContactFormSubmissions using the put\_item() method.
2. Test Event Setup  
   A test event was created using the “API Gateway AWS Proxy” template, which closely simulates the real-world data structure sent from an HTTP POST request via API Gateway. The body field was passed as a stringified JSON object to match API Gateway behavior.
3. Deployment and Invocation  
   After saving and deploying the function, the test event was executed. The Lambda function responded with a status code of 200 and a success message, verifying that the logic executed without errors and the data was stored correctly in DynamoDB.

**Results**

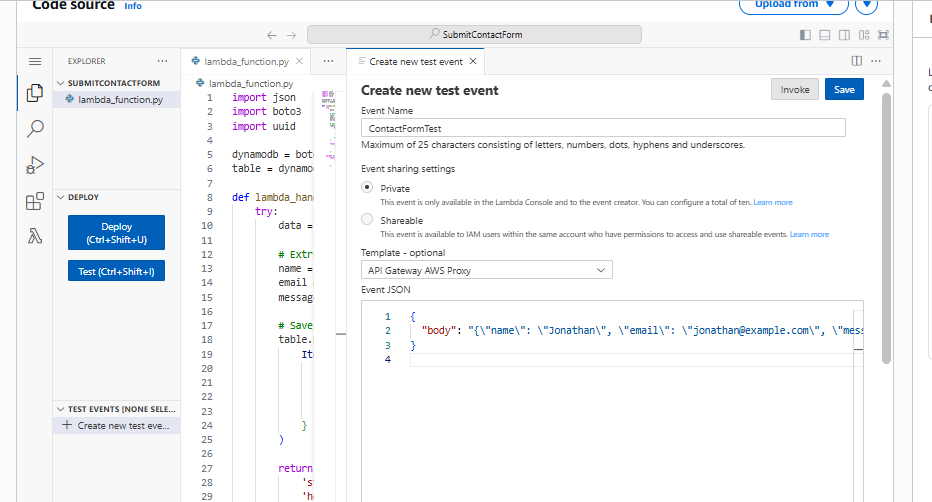
The execution logs confirmed successful processing with a memory footprint of 85 MB and execution time under 300 ms. The response returned:

****

This validated that the Lambda function was functioning correctly and integrated with DynamoDB as intended.

**Figure 1:**

Lambda function written in Python 3.12 showing DynamoDB integration.

**Figure 2:**

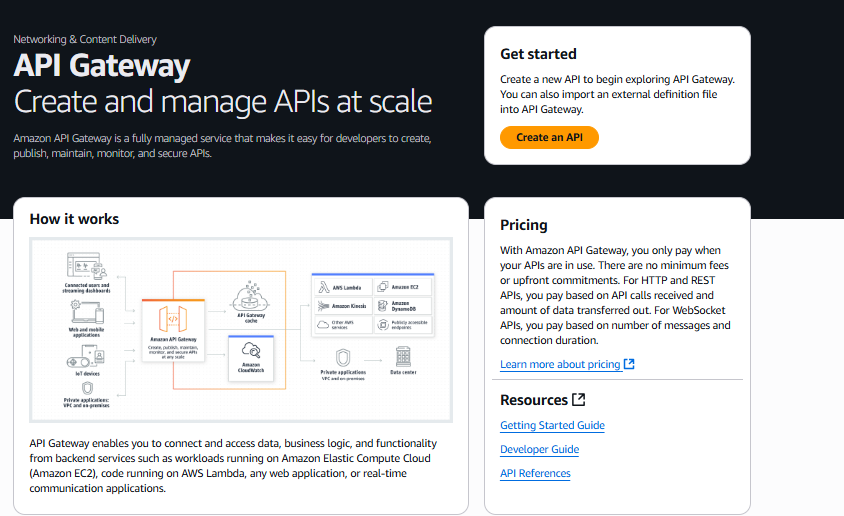
Execution result confirming the function successfully processed the test event.

This step was crucial in confirming the functionality of the backend Lambda logic before any frontend or API Gateway exposure. By simulating a real-world POST request, the development environment facilitated precise debugging and verification. This proactive validation approach reduces downstream errors and ensures reliability in production.

**Step 5: Create API Gateway to Trigger Lambda Function**

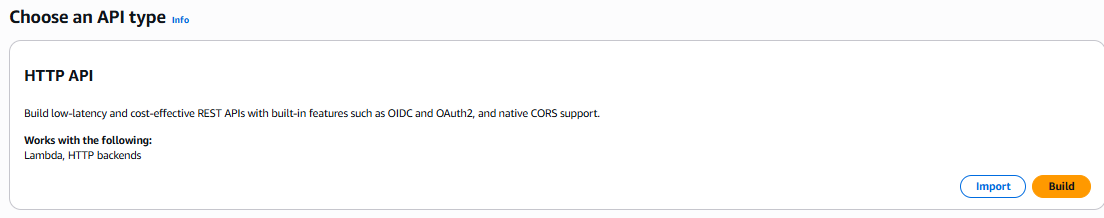
To enable frontend-to-backend communication for the contact form, we created and deployed an HTTP API using Amazon API Gateway. This service provides a scalable and secure entry point for client requests, which in this project are forwarded to a Lambda function for processing.

**Figure 1 : (API Gateway page)**



**Selecting the API Type**

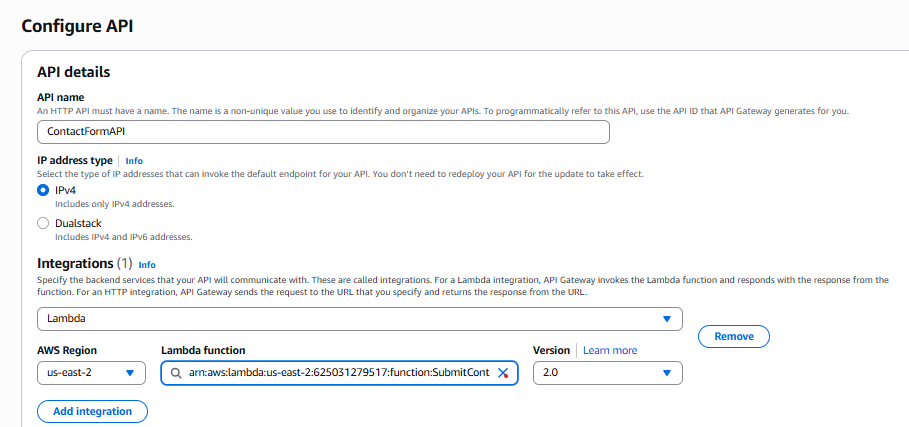
From the API Gateway console, we initiated the creation of a new API. As shown in **Figure 1**, the API Gateway dashboard emphasizes the platform’s ability to create, publish, and manage APIs at scale. We selected the **HTTP API** option (see **Figure 2**) for its low latency, native support for Lambda backends, and built-in CORS functionality—critical features for our serverless architecture.



**Configuring API Details**

In the configuration panel (**Figure 3**), we named the API ContactFormAPI and selected **IPv4** as the IP address type. We then integrated the Lambda function SubmitContactForm by referencing its Amazon Resource Name (ARN), which was auto-populated upon selecting the correct AWS Region (us-east-2). The integration ensures that incoming API requests are routed directly to our function.

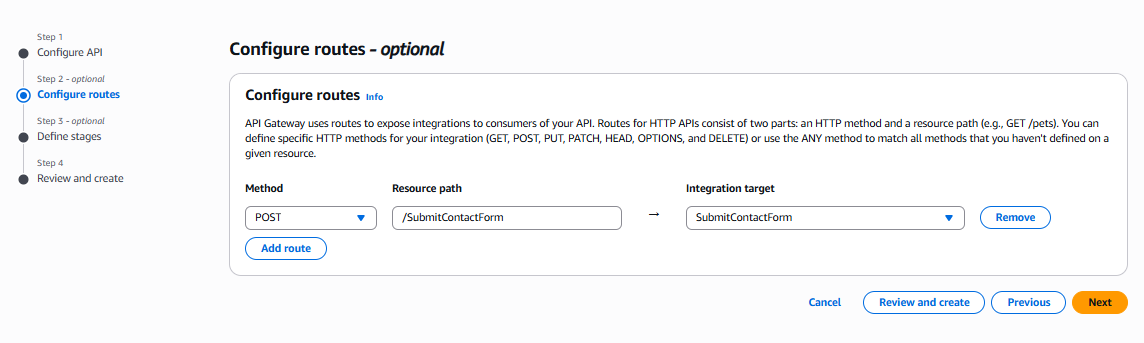
**Figure 3:**



**Setting Up Routes**

The next step involved defining API **routes** (see **Figure 4**). We added a POST route pointing to /SubmitContactForm and linked it to our SubmitContactForm Lambda function. This configuration matches the expected behavior of a contact form submission—where form data is transmitted via an HTTP POST request.

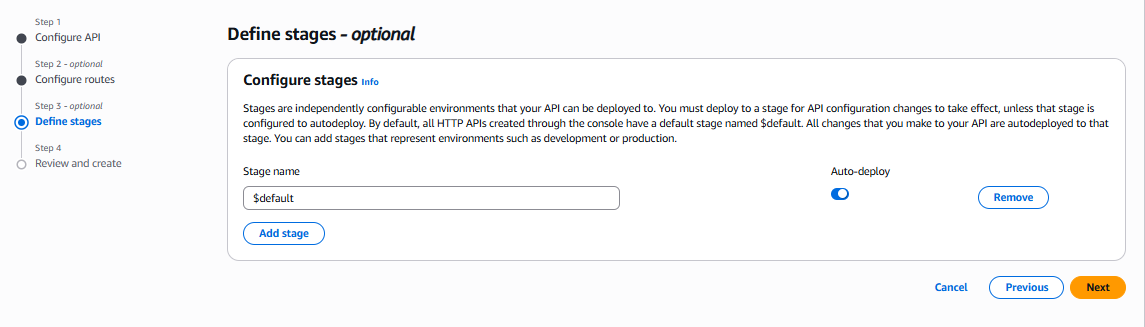
**Figure 4:**

****

**Defining Deployment Stage**

In Figure 5, we defined a deployment stage named $default with auto-deploy enabled. This option automatically applies configuration changes without requiring manual redeployment, supporting faster iteration during testing and production deployment.

**Figure 5:**

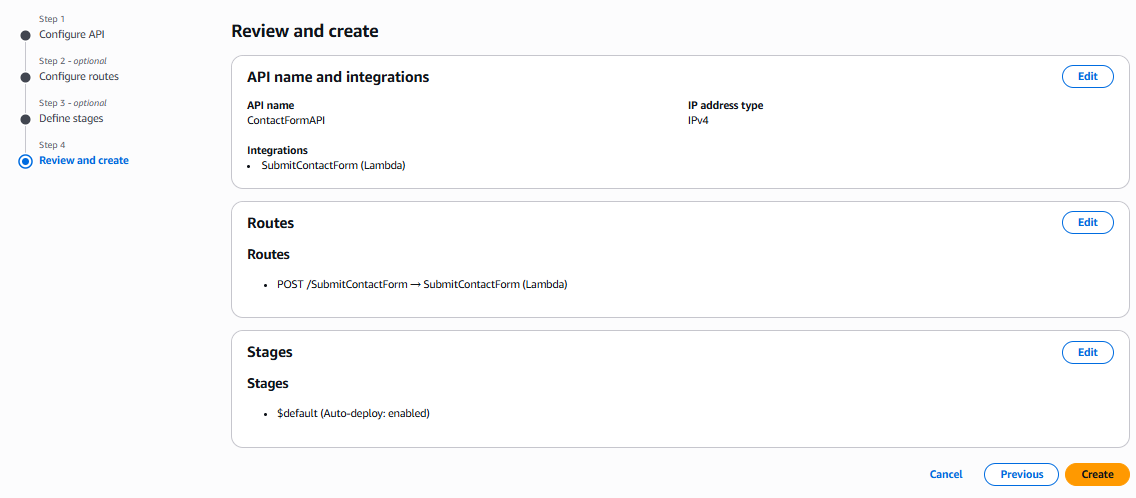
****

**Final Review and Deployment**

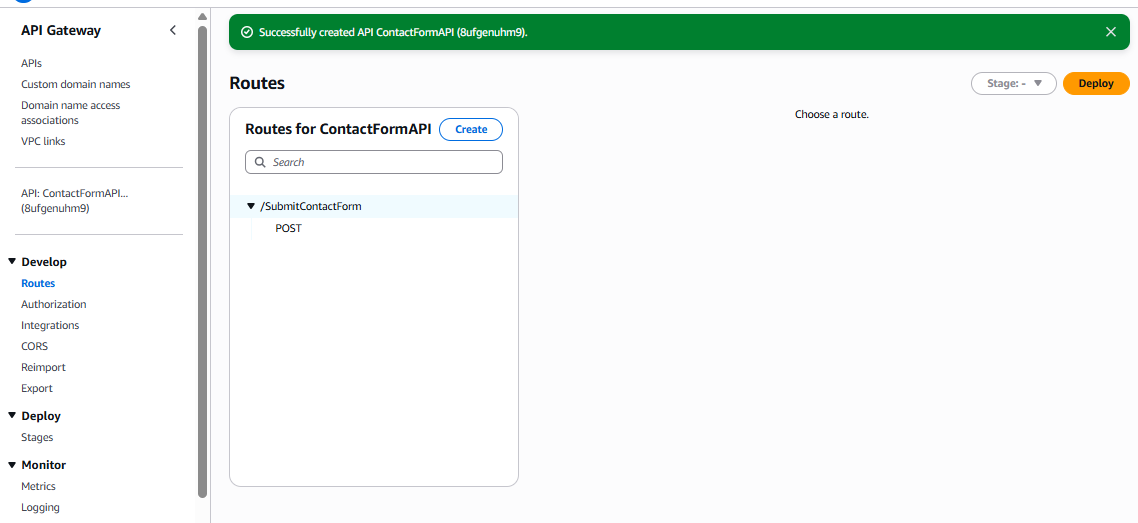
The final step was a comprehensive review (see Figure 6). We validated the API name, route method and path, Lambda integration, and deployment stage settings. Once confirmed, we deployed the API. As shown in Figure 7, the API Gateway successfully created and linked the /SubmitContactForm route to our backend, completing the deployment pipeline.

The deployed API Gateway now serves as a secure, scalable interface for transmitting form data from the frontend to the Lambda function. This integration is a core component of our serverless architecture and completes the backend connectivity needed for handling contact form submissions.

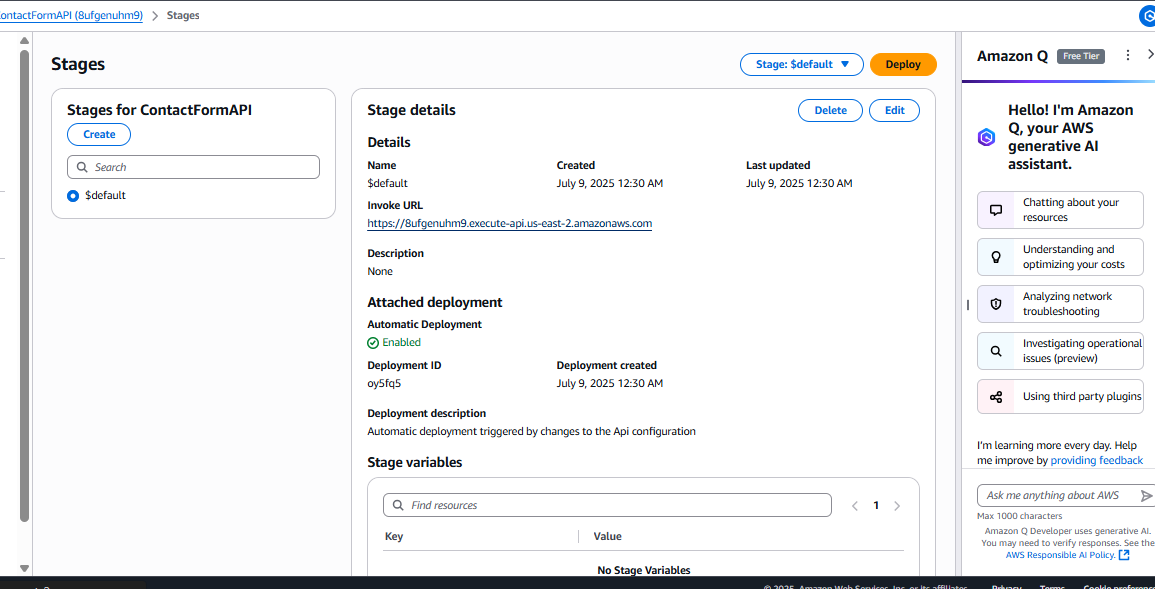
**Figure 6:**

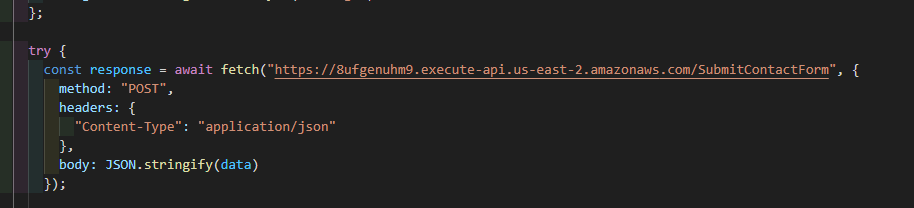
****

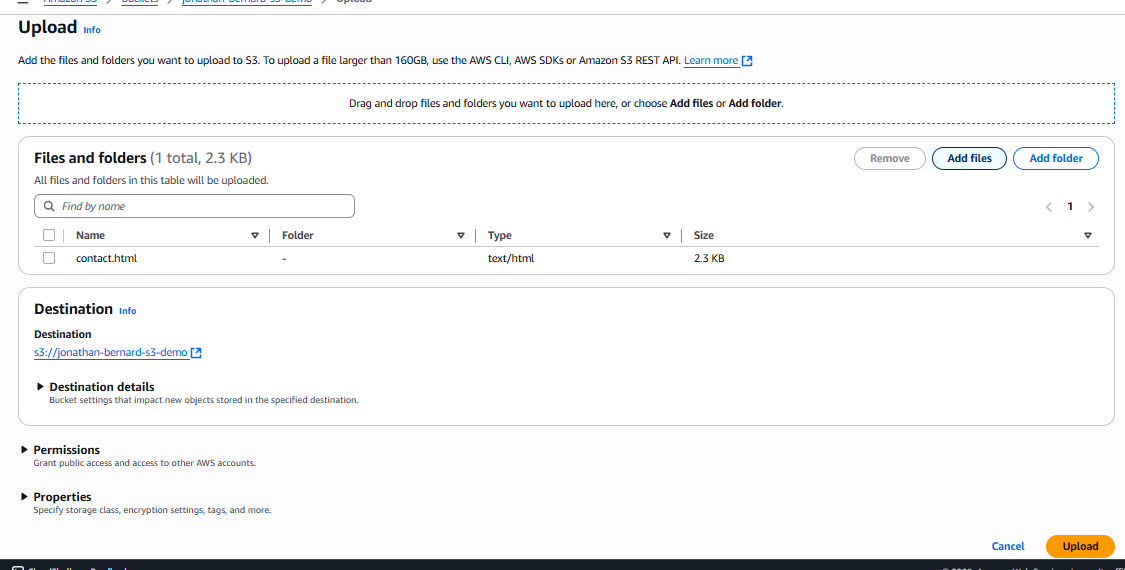
**Figure 7:**

****

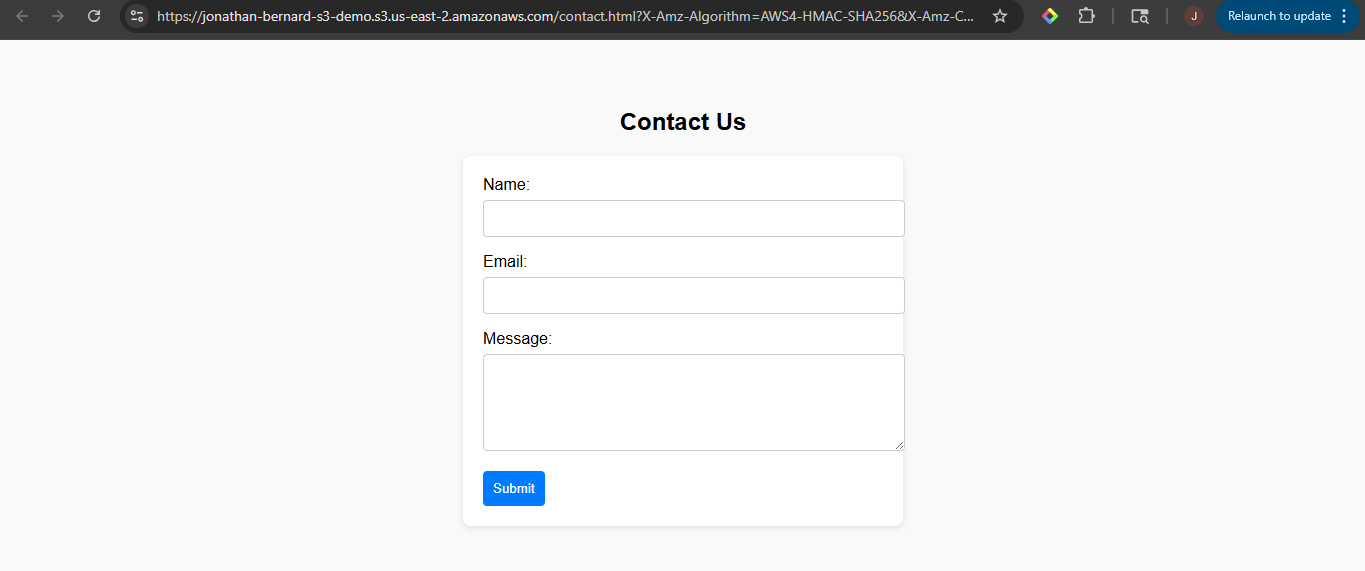
**Step 6: Connecting the Static Frontend to the API Gateway**

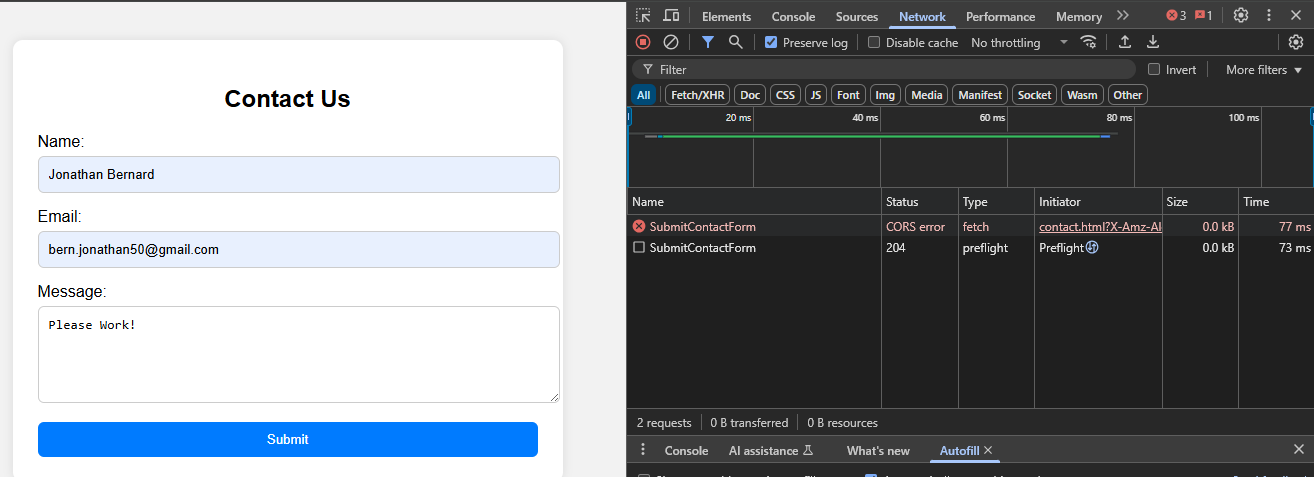
****

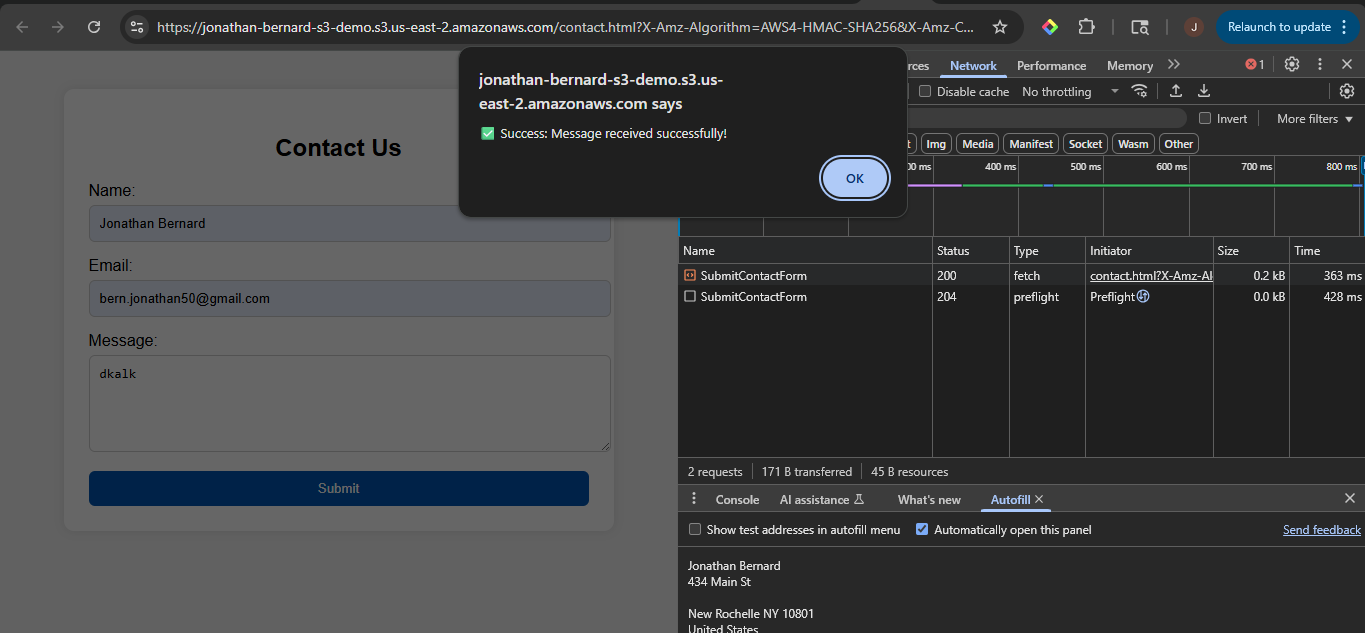
****

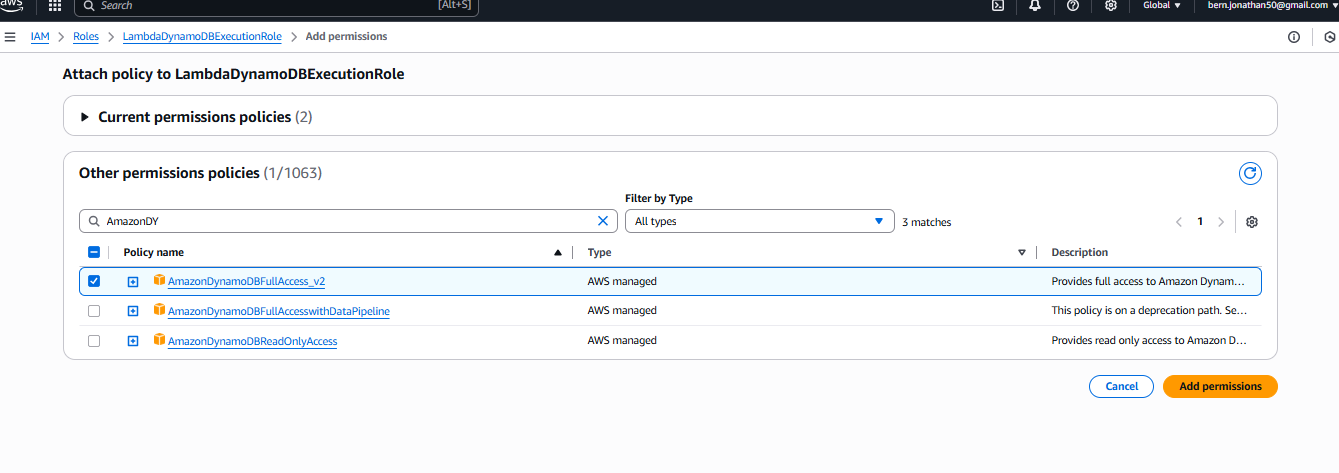
****

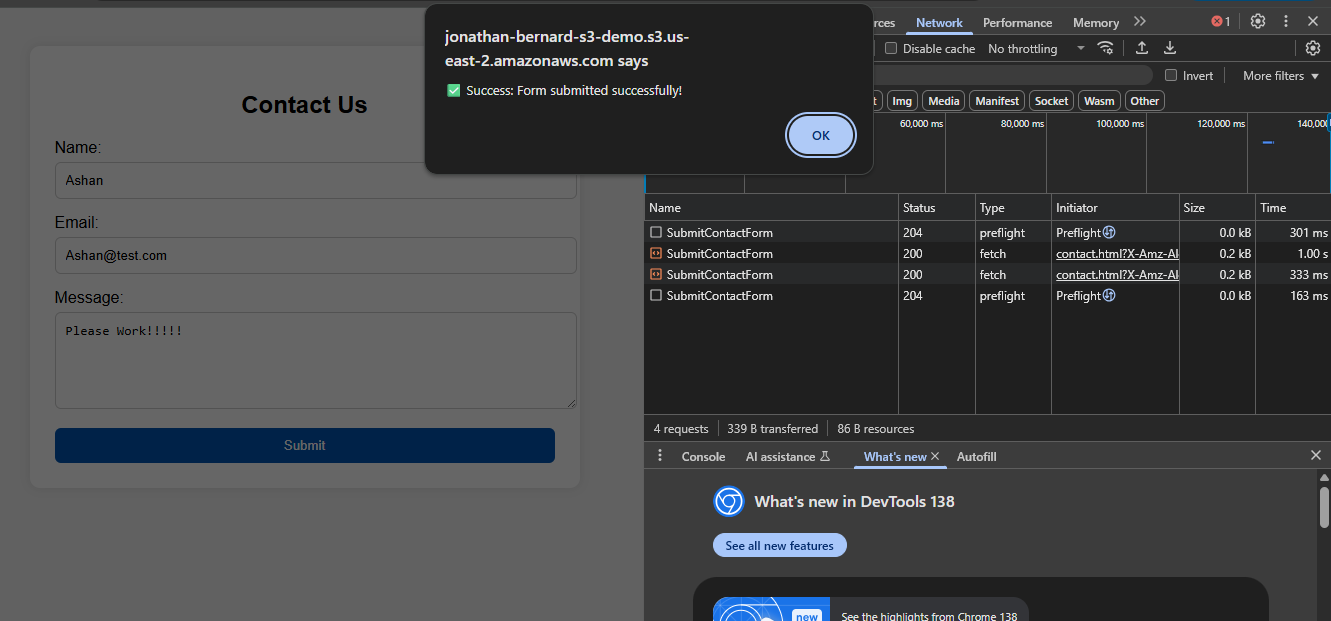
**Working Contact Form**

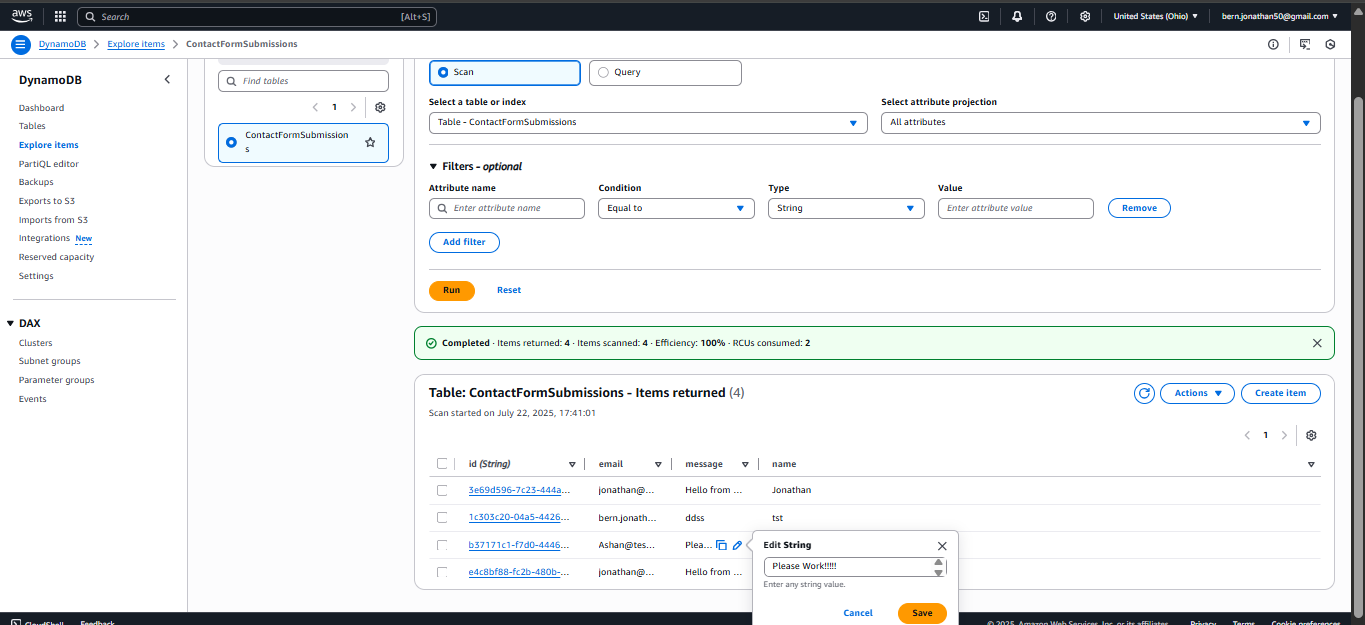
****

****

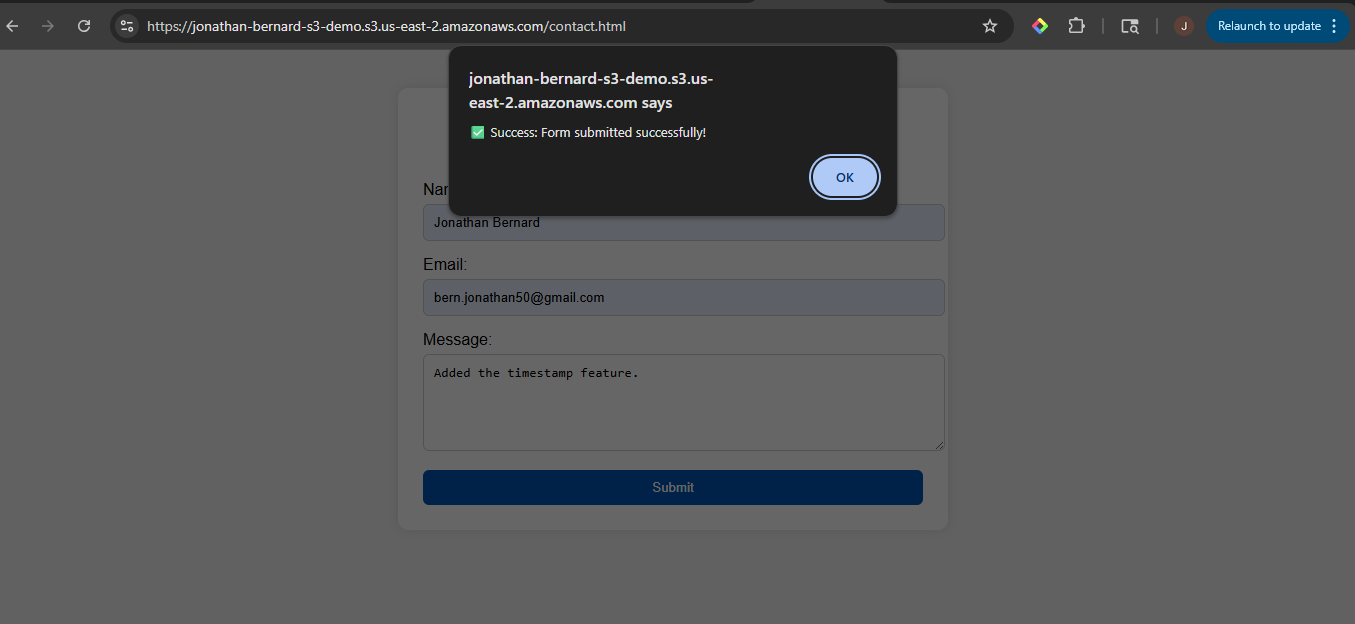
****

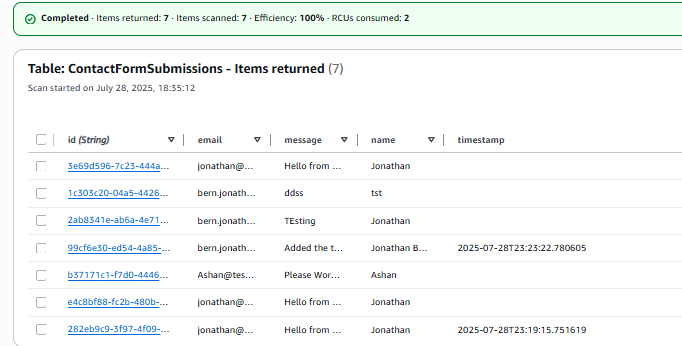
****

****

****

**Added Timestamp**

****

****

**Project Summary**

This project demonstrates a fully functional serverless contact form application built using AWS cloud services. The frontend is hosted on Amazon S3 as a static website, providing users with a simple and responsive contact form interface. When a user submits the form, the data is securely sent via API Gateway to an AWS Lambda function, which processes and stores the submission in Amazon DynamoDB. A timestamp feature has been added to each record for better tracking of submission times.

This architecture eliminates the need for dedicated servers, offering scalability, low cost, and minimal maintenance, making it ideal for lightweight applications. Future improvements could include email notifications (Amazon SES) and input validation for enhanced user experience and robustness.

The project serves as a practical example of how multiple AWS services can be integrated into a cohesive cloud-based solution.