Project Documentation: Real-Time Feedback Collector

Project Overview:

The goal of this project was to build a simple but effective real-time feedback collection system using AWS serverless services. The idea was to allow users to submit their feedback via a webpage, which would then be stored in DynamoDB for later analysis or processing. The project showcases a fully serverless architecture with no backend servers or traditional web servers involved. It is ideal for cloud computing practice, and provides hands-on exposure to Amplify, Lambda, API Gateway, IAM, and DynamoDB.

Initial Requirements:

The primary requirement was to create a feedback form hosted on a static webpage, capable of accepting a user's name (optional), their written feedback, and a rating from 1 to 5. This form needed to submit the data to a backend function, which would process and validate it before saving it to a DynamoDB table. The solution had to be serverless, lightweight, and fully managed using AWS services.

Frontend (AWS Amplify):

We started by creating a simple yet modern index.html file using HTML, CSS, and embedded JavaScript. This page contained a form with three inputs: a text field for the user's name, a textarea for feedback, and a dropdown for rating. Upon submission, JavaScript collected the form data and triggered a fetch() request to the API Gateway URL with the method set to POST and the body set as a JSON string. To host the webpage, we used AWS Amplify, which allows static website hosting via a drag-and-drop UI or GitHub repo integration. The index.html was uploaded to Amplify, making the webpage publicly accessible.

API Gateway Configuration:

Next, we created a REST API using API Gateway. Instead of defining a named resource like /submit-feedback, we chose to attach the POST method directly to the root (/) path for simplicity. We then integrated the POST method with a Lambda function and enabled Lambda proxy integration. This meant API Gateway would pass the request body as a stringified JSON inside the "body" key of the event object. We also created an OPTIONS method to support CORS, added appropriate response headers (Access-Control-Allow-Origin, Access-Control-Allow-Methods, and Access-Control-Allow-Headers), and deployed the API to a stage named dev. This ensured that the frontend could communicate with the API without browser-side CORS errors.

Lambda Function (Python):

The core logic of our backend was implemented in a Python-based Lambda function. Initially, we parsed the incoming request's body from event['body'], validated the fields (feedback, rating), and constructed a new feedback record using a UUID and timestamp. One of the key challenges encountered was that DynamoDB does not accept null values, so including 'name': None in the item caused silent failures. This was fixed by conditionally adding the name field to the item only if it was non-empty. We also added support for both API Gateway proxy and non-proxy formats by checking if event['body'] existed and falling back to the raw event object if not.

DynamoDB Setup:

We created a DynamoDB table named FeedbackCollector with a primary (partition) key of type String called ID. The table stores each feedback entry as an item with fields: ID, name (optional), feedback, rating, and timestamp. DynamoDB was chosen for its scalability and ease of integration with Lambda.

IAM Configuration:

The Lambda function was assigned an execution role that initially lacked the proper permission to

write to DynamoDB. This resulted in silent failures where CloudWatch logs showed the Lambda being invoked but no data was saved. We fixed this by attaching an inline policy to the Lambda's IAM role that granted dynamodb:PutItem and dynamodb:DescribeTable actions on the target DynamoDB table. Once applied, the Lambda was able to successfully insert records.

Testing & Debugging Process:

Throughout development, we used both the Lambda test feature and the actual frontend to test end-to-end functionality. One key finding was that Lambda test events send the body as a stringified JSON, while frontend calls via fetch() sent the body directly as JSON. This difference led to the Lambda parsing failing unless we added support for both cases. CloudWatch logs were instrumental in identifying such issues. We also observed that when frontend form inputs were invalid (like a blank feedback field or a rating sent as a string), the validation logic would reject the submission silently. To fix this, we enhanced the validation and logging to show whether or not the item was actually being saved.

Final System Behavior:

Once all parts were correctly configured:

- The frontend sends form data to the API Gateway.
- API Gateway invokes the Lambda function.
- Lambda parses, validates, and stores the data into DynamoDB.
- If successful, the frontend shows a popup confirmation.
- All operations are logged in CloudWatch.

Key Issues Faced & Resolved:

1. CORS Errors: Fixed by creating OPTIONS method and setting appropriate headers.

2. Missing /submit-feedback resource confusion: Removed unnecessary path segment and used root (/) endpoint.

3. DynamoDB insert silently failing: Caused by inserting None for name; fixed by conditional field inclusion.

4. IAM permissions: Lambda initially lacked PutItem; fixed via custom inline policy.

5. Event body format mismatch: Fixed by supporting both stringified and raw JSON request bodies.

Outcome:

The project is now fully functional and scalable. It is deployed using serverless architecture, incurs virtually no cost under the free tier, and can handle real-world feedback collection needs. It also serves as an excellent portfolio project to demonstrate your practical skills with AWS cloud services and real-world debugging experience.