Simple HTTP Relay Server with a WebApp and CI/CD Design HLD Part 1

Server Backend Design

1. Introduction

We want to have a simple user friendly web application that allows us to access messages over a web server. This design has three components:

1. ***(this document) A backend server for sending and getting messages on the cloud***
2. A web-application with an interface to the backend server
3. A CI/CD pipeline and development environment that makes development easier

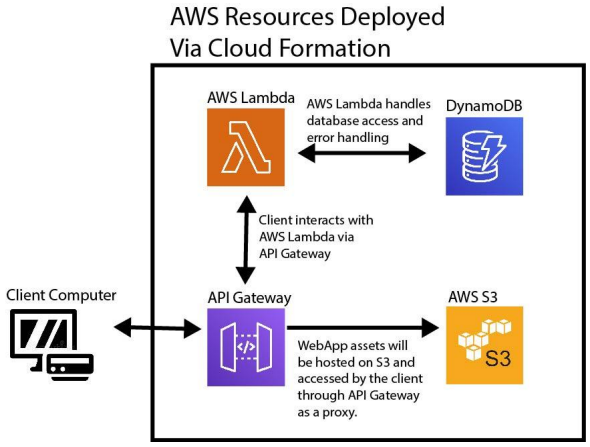
This document will focus only on the design and considerations of the backend server.

2. Backend Design

2.1 High Level Design

An AWS Lambda will access messages stored on DynamoDB, and all interactions by the client with the cloud will be through API Gateway.

For simplicity and clarity, the backend lambda will be written using python and with an import from Boto3. The API Gateway resources will be defined using OpenAPI to define the resources.



2.2 Alternatives / Considerations

1. **Python for server code:** Python is a simple language that is well supported with AWS Lambda and other build tools. Other languages like Java and TypeScript can have easier error handling. For the limited purposes of this demo I've chosen Python.
2. **API Gateway for all interactions:** By using API Gateway we free ourselves to change anything we want about the lambda later without any changes to the core API. If, for example, we want to change the backend to use Java, we can do that without disruption.

3. **AWS Resources over Azure:** I am unfamiliar with Azure, but I have worked at Amazon for more than two and a half years. AWS seems to also have the most support.
4. **DynamoDB for Message Storage:** I've chosen DynamoDB because it has simple API calls that can be used for each of the core backend requirements. We could use an S3 file to store the messages since we don't expect there to be that many and S3 persistent storage is much cheaper, but DynamoDB is the more scalable and direct option.

2.3 Detailed Design

2.3.1 API Gateway API Calls

Messages will be sent to the client in a message resource in json format. The following is the data stored in the resource.

Message REST Resource			
Variable Name	**Value Type**	**Description**	**Example**
message	String	string message stored in the system	"There's a frood who really knows where his towel is."
id	String	unique id for the message	"c45c851c-ff1c-11ed-be56-0242ac120002"

The API Gateway will handle the following API calls.

API Definition						
Verb	**Name**	**API Extension**	**Parameters**	**Returns**	**Success**	**Error**
POST	create message	"/messages"	message	Message resource	201	40x
GET	get message	"/messages/[id]"	message id	Message resource	200	40x
GET	get count of messages	"/messages"	-	Num messages	200	40x

Note: The **"get count of messages"** API call is the most unique API call because it does not return a resource. If we want to have a more consistent REST API, we should find a more universal Resource scheme for all resources***. For now that is out of scope.***

2.3.2 Lambda Event Handler

The Lambda event handler will expect one of these three events (identical to API above):

1. create message
2. get message

3. get count of messages

When the lambda is triggered it'll access the DynamoDB table and return a resource as defined above.

2.3.3 DynamoDB Table

The DynamoDB table will happen to store data entries that happen to be identical to the REST resource with the unique id as the primary key.

Message DynamoDB Resource			
Variable Name	**Value Type**	**Description**	**Example**
message	String	string message stored in the system	"There's a frood who really knows where his towel is."
id	String: **PrimaryKey**	unique id for the message	"c45c851c-ff1c-11ed-be56-0242ac120002"

2.3.4 Defining the Resources

We will define the cloud resources in a Cloud Formation template written in TypeScript. As of now, TypeScript has the most support and idiomatic building for CloudFormation templates.

3. Backend Development Timeline

Because this is an interview I've included conservative estimates for the time it will take for each stage of development.

Task	**Estimated Dev Time**	**Loose Deadline**
Research Project and Document Plan	3 hours	05/30 4:30 PM EST
Basic repo setup	30 minutes	05/30 5:00 PM EST
Simulate Python Lambda interacting with DynamoDB Triggered by API Gateway locally	3 hours	05/30 Midnight EST
Push API Gateway Interface	1 hour	05/31 11:00 AM EST
Push AWS Lambda Code	1 hour	05/31 11:00 AM EST
Push Cloud Formation Template *	3 hours	05/31 2:00 PM EST

*** It may be smartest to link github to cloud formation in a single commit instead of separate commits, so this action has been given extra time.**

Appendix

Project Requirements

In a single mono repo:

1. Develop a simple HTTP relay server using a programming language of your choice. The server should have three functionalities:
    - Receive a message, assign a unique number to it, and return the number.
    - Accept a unique number and return the corresponding message.
    - Display the total number of messages stored on the server.
2. Create a user-friendly web app for the server, enabling users to:
   View the total number of messages on the server.
    - Submit a new message and receive a unique number for it.
    - Enter a unique message number to retrieve the corresponding message.
3. Implement Continuous Integration (CI) and Continuous Deployment (CD) for the project. Ensure that the CI tests the applications.
4. Thoroughly document the deployment process, detailing how developers can deploy, debug, and test the project using local or isolated environments.

Assume that this repo will serve as a starting point for a team of developers and independent contributors. **Focus on providing the basic functionalities without incorporating additional features.** However, consider various aspects of the project's future development and best practices. Make provisions for the project's potential evolution and scalability, as its direction remains uncertain. Include these considerations in the documentation.