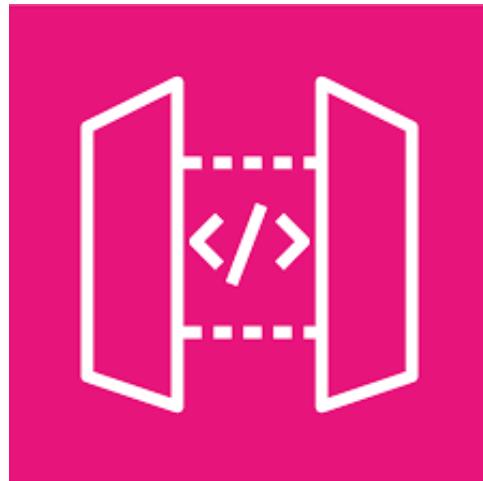


# Práctica 8

## API GATEWAY

### Chat

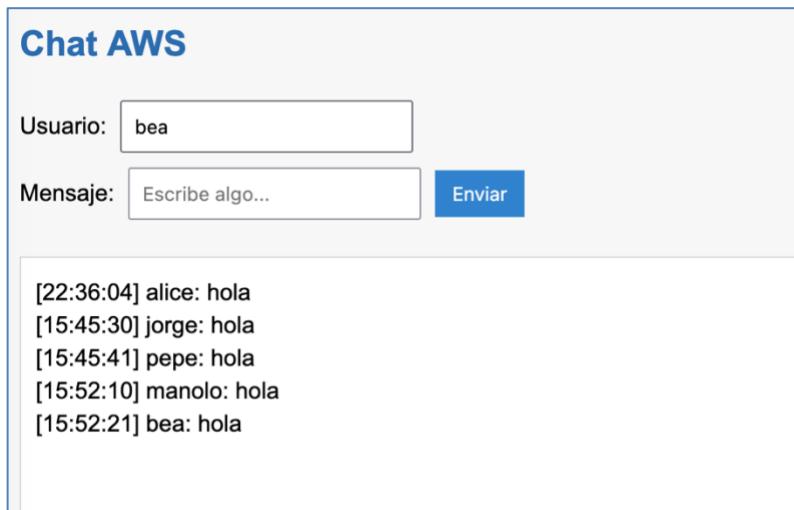


Introducción .....	3
Creación de SNS.....	3
Creación de la Función Lambda.....	4
Configuración de API Gateway.....	5
Actividades .....	7

## Introducción

Vamos a crear un chat donde una API envía un mensaje, éste se guarda en una BBDD DynamoDB y luego recuperaremos los mensajes, tanto la lectura como el almacenamiento usaremos el servicio Lambda, y para la parte del front integrando estas llamadas una página web alojada en un S3.

La visión final debería ser algo muy parecido a la imagen siguiente:



## Lambda "Hola mundo" + API Gateway REST

En primer lugar, crearemos una lambda hola, donde haremos las pruebas para ver si funciona, de hecho, es código nos viene prácticamente dado y sólo hay que hacer la prueba.

```

JS index.mjs X
JS index.mjs > ...
1  export const handler = async (event) => {
2    // TODO implement
3    const response = {
4      statusCode: 200,
5      body: JSON.stringify('Hello from Lambda!'),
6    };
7    return response;
8  };
9

```

## Endpoint /send

A continuación, crearemos otra lambda chat-send, la cual debería responder al siguiente contrato:

**Evento JSON**

```

1 * [ {
2 *   "body": {
3 *     "user": "alice",
4 *     "text": "Hola a todos"
5 *   }
6 }]
```

**json**

```
{
  "status": "ok",
  "echo": {
    "user": "alice",
    "text": "Hola a todos",
    "timestamp": 1234567890
}
```

La primera versión de código para poder probarlo:

```

export const handler = async (event) => {
  let body;

  // Si viene como objeto, úsalo directo
  if (typeof event.body === "object") {
    body = event.body;
  } else {
    // Si viene como string (caso API Gateway o Lambda Test), parsear
    try {
      body = JSON.parse(event.body || "{}");
    } catch {
      return {
        statusCode: 400,
        body: JSON.stringify({ error: "JSON inválido" }),
      };
    }
  }

  const { user, text } = body;

  if (!user || !text) {
    return {
      statusCode: 400,
      body: JSON.stringify({ error: "user y text son obligatorios" }),
    };
  }

  const message = {
    user,
    text,
    timestamp: Date.now(),
  };

  return {
    statusCode: 200,
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ status: "ok", echo: message }),
  };
}
```

Cuando pruebo desde lambda con el JSON correcto debería salir algo como lo siguiente:

✓ Ejecutando la función: sin errores ([registros](#))

▼ Detalles

```
{
  "statusCode": 200,
  "headers": {
    "Content-Type": "application/json"
  },
  "body": "{\"status\":\"ok\", \"echo\":{\"user\":\"alice\", \"text\":\"Hola a todos\", \"timestamp\":1763484859185}}"
}
```

En el API Gateway crearemos algo con la misma estructura para poder probar lambda. (El post usa la función lambda chat-send), y la desplegamos.

Y para probar el endpoint con curl, nos devolverá un mensaje de error que podremos consultar en los eventos de Cloudwatch, una vez consultado retocaremos la lambda:

**Etapas**

<pre>└─ dev     └─ /         └─ send             POST</pre>	<p><b>Detalles de la etapa</b> <a href="#">Información</a></p> <table border="1" style="width: 100%; border-collapse: collapse;"> <tr> <td style="width: 50%;">Nombre de etapa</td> <td style="width: 50%;">Tasa <a href="#">información</a> 10000</td> </tr> <tr> <td>Clúster de caché <a href="#">información</a></td> <td><input checked="" type="radio"/> Inactivo</td> </tr> <tr> <td>Almacenamiento en caché de nivel de método predeterminado</td> <td><input checked="" type="radio"/> Inactivo</td> </tr> <tr> <td>URL de invocación</td> <td><a href="https://4iqi63mxge.execute-api.us-east-1.amazonaws.com/dev">https://4iqi63mxge.execute-api.us-east-1.amazonaws.com/dev</a></td> </tr> </table> <p>Implementación activa xx5qh5 el November 18, 2025, 18:06 (UTC+01:00)</p>	Nombre de etapa	Tasa <a href="#">información</a> 10000	Clúster de caché <a href="#">información</a>	<input checked="" type="radio"/> Inactivo	Almacenamiento en caché de nivel de método predeterminado	<input checked="" type="radio"/> Inactivo	URL de invocación	<a href="https://4iqi63mxge.execute-api.us-east-1.amazonaws.com/dev">https://4iqi63mxge.execute-api.us-east-1.amazonaws.com/dev</a>
Nombre de etapa	Tasa <a href="#">información</a> 10000								
Clúster de caché <a href="#">información</a>	<input checked="" type="radio"/> Inactivo								
Almacenamiento en caché de nivel de método predeterminado	<input checked="" type="radio"/> Inactivo								
URL de invocación	<a href="https://4iqi63mxge.execute-api.us-east-1.amazonaws.com/dev">https://4iqi63mxge.execute-api.us-east-1.amazonaws.com/dev</a>								

```

1  export const handler = async (event) => {
2    console.log("EVENT:", JSON.stringify(event, null, 2));
3
4    let body = event.body;
5
6    // Caso: API Gateway pasa user/text directamente
7    if (!body && (event.user || event.text)) {
8      body = event;
9    }
10
11   // Si body es string → parsear
12   if (typeof body === "string") {
13     try {
14       body = JSON.parse(body);
15     } catch {
16       return {
17         statusCode: 400,
18         body: JSON.stringify({ error: "JSON inválido" }),
19       };
20     }
21   }
22
23   const { user, text } = body || {};
24
25   if (!user || !text) {
26     return {
27       statusCode: 400,
28       body: JSON.stringify({ error: "user y text son obligatorios" }),
29     };
30   }
31
32   const message = {
33     user,
34     text,
35     timestamp: Date.now(),
36   };
37
38   return {
39     statusCode: 200,
40     headers: { "Content-Type": "application/json" },
41     body: JSON.stringify({ status: "ok", echo: message }),
42   };
43 };
44

```

Con esto ya podemos “enviar” mensajes, estos mensajes para hacerlos persistentes los guardaremos en una BBDD DynamoDB.

## Base de datos DynamoDB

Se adjunta la configuración de la tabla creada:

The screenshot shows the AWS DynamoDB console. At the top, it displays the table count (1) and provides search and filter options. Below this, the table details for 'ChatMessages' are shown, including its primary key ('room' of type String) and sort key ('timestamp' of type Number). The table has 0 items and 0 streams. Protection against deletions is disabled.

In the main area, there's a tooltip about protecting the table from accidental writes and deletions. The 'Información general' section shows the partition key ('room') and sort key ('timestamp'). The data section displays two items: one for user 'general' at timestamp 1763415364211 with text 'holo' and user 'alice', and another for user 'general' at timestamp 1763477130730 with text 'holo' and user 'jorge'.

Ahora pasaremos a modificar la lambda para que guarde estos mensajes en la tabla cada vez que hagamos una llamada curl.

```
jorge@mjolnir chat-aws % curl -X POST \
-H "Content-Type: application/json" \
-d '{"user":"alice","text":"mensaje guardado en Dynamo"}' \
https://4iqiqi63mxge.execute-api.us-east-1.amazonaws.com/dev/send

{"statusCode":200,"headers":{"Content-Type":"application/json"},"body":"{\\"status\\":\\"saved\\",\\"message\\":{\\"room\\":\\"gen\\",\\"timestamp\\":1763491006530,\\"user\\":\\"alice\\",\\"text\\":\\"mensaje guardado en Dynamo\\\"}}}"%
jorge@mjolnir chat-aws %
```

	general	1763477130730	nota	mensaje
<input type="checkbox"/>	general	1763477541238	holo	bea
<input type="checkbox"/>	general	1763491006530	mensaje guardado en Dynamo	alice

```

1  import { DynamoDBClient } from "@aws-sdk/client-dynamodb";
2  import { PutCommand, DynamoDBDocumentClient } from "@aws-sdk/lib-dynamodb";
3
4  const client = new DynamoDBClient({});
5  const docClient = DynamoDBDocumentClient.from(client);
6  const TABLE_NAME = process.env.TABLE_NAME || "ChatMessages";
7
8  export const handler = async (event) => {
9    console.log("EVENT:", JSON.stringify(event, null, 2));
10
11   let { user, text, room } = event;
12
13  if (!user || !text) && event.body) {
14    try {
15      const body =
16        typeof event.body === "string"
17          ? JSON.parse(event.body)
18          : event.body;
19      user = body.user;
20      text = body.text;
21      room = body.room;
22    } catch {
23      return {
24        statusCode: 400,
25        body: JSON.stringify({ error: "JSON inválido" }),
26      };
27    }
28  }
29
30  if (!user || !text) {
31    return {
32      statusCode: 400,
33      body: JSON.stringify({ error: "user y text son obligatorios" }),
34    };
35  }
36
37  if (!room) room = "general"; // ➔ agregado
38
39  const timestamp = Date.now();
40  const message = { room, timestamp, user, text };
41
42  await docClient.send(
43    new PutCommand({
44      TableName: TABLE_NAME,
45      Item: message,
46    })
47  );
48
49  return {
50    statusCode: 200,
51    headers: { "Content-Type": "application/json" },
52    body: JSON.stringify({ status: "saved", message }),
53  };
54};
55

```

Ahora llegados a este punto ya somos capaces de llamar a un endpoint de nuestra API y guardar datos, el siguiente paso será mostrar los mensajes.

## Endpoint /mesages

En este apartado primero crearemos una lambda para recuperar los mensajes y luego un endpoint en el AGW para poder llamarlo. Tendrá la forma “**GET /messages?room=general&limit=20**”.

Al probar el siguiente código en la lambda:

```

1 import { DynamoDBClient, QueryCommand } from "@aws-sdk/client-dynamodb";
2
3 const client = new DynamoDBClient({});
4
5 export const handler = async (event) => {
6   const room = event.queryStringParameters?.room || "general";
7   const limit = parseInt(event.queryStringParameters?.limit || "20", 10);
8
9   try {
10     const result = await client.send(
11       new QueryCommand({
12         TableName: "ChatMessages",
13         KeyConditionExpression: "room = :room",
14         ExpressionAttributeValues: {
15           ":room": { S: room },
16         },
17         ScanIndexForward: false, // false = orden descendente (más recientes primero)
18         Limit: limit,
19       })
20     );
21
22     const messages = (result.Items || []).map((item) => ({
23       room: item.room.S,
24       timestamp: Number(item.timestamp.N),
25       user: item.user.S,
26       text: item.text.S,
27     }));
28
29     return {
30       statusCode: 200,
31       body: JSON.stringify({ messages }),
32     };
33   } catch (err) {
34     console.error(err);
35     return {
36       statusCode: 500,
37       body: JSON.stringify({ error: "Error leyendo mensajes" }),
38     };
39   }
40 };
41 
```

Nos devolverá los registros:

Ejecutando la función: sin errores ([registros](#))

Detalles

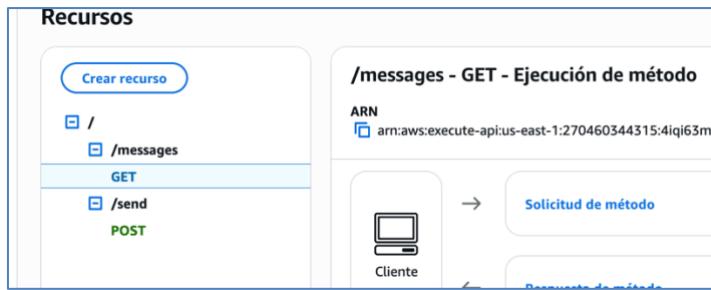
Resumen

Código SHA-256: q8E7Nexf5xxKT9/d4bGpAYOJYFAUjJ0UDjB0ivK8E

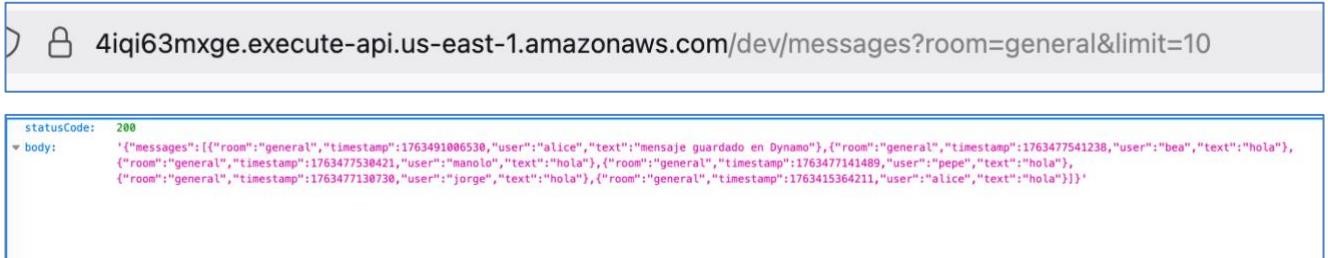
Tiempo de ejecución: hace 2 segundos

```
{
  "statusCode": 200,
  "body": "[{"room": "general", "timestamp": 1763491006530, "user": "alice", "text": "mensaje guardado en Dynamo"}, {"room": "general", "timestamp": 1763477541238, "user": "bed", "text": "holo"}, {"room": "general", "timestamp": 1763477530421, "user": "monolo", "text": "holo"}, {"room": "general", "timestamp": 1763477141489, "user": "pepe", "text": "holo"}, {"room": "general", "timestamp": 1763477130730, "user": "jorge", "text": "holo"}, {"room": "general", "timestamp": 1763415364211, "user": "alice", "text": "holo"}]"
}
```

Ahora solo falta integrarlo en el api gateway.



Y desplegarlo para obtener un endpoint. Para ello podremos probar en el navegador:



A screenshot of a browser window showing the URL: [4iqi63mxge.execute-api.us-east-1.amazonaws.com/dev/messages?room=general&limit=10](https://4iqi63mxge.execute-api.us-east-1.amazonaws.com/dev/messages?room=general&limit=10). Below the URL, the browser's developer tools show the response body:

```

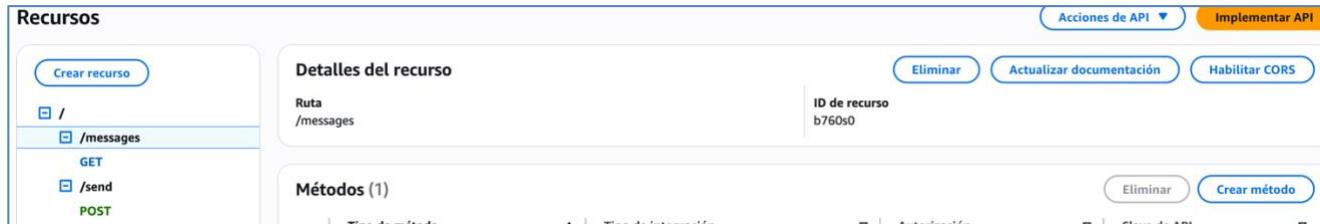
statusCode: 200
body:
  [
    {
      "room": "general",
      "timestamp": 1763491006530,
      "user": "alice",
      "text": "mensaje guardado en Dynamo"
    },
    {
      "room": "general",
      "timestamp": 1763477530421,
      "user": "manolo",
      "text": "holá"
    },
    {
      "room": "general",
      "timestamp": 1763477141489,
      "user": "pepe",
      "text": "holá"
    },
    {
      "room": "general",
      "timestamp": 1763477130730,
      "user": "jorge",
      "text": "holá"
    },
    {
      "room": "general",
      "timestamp": 1763415364211,
      "user": "alice",
      "text": "holá"
    }
  ]

```

Llegados a este punto donde listamos y guardamos mensajes, ahora sólo nos faltaría el frontend.

## Frontend

Cuando probemos a usar el navegador nos encontraremos con que nos da error, para ello tanto en las lambdas como en los recursos de nuestra API deberemos habilitar CROSS.



Aun así, veremos que la página SI que envía información y la guarda en la tabla, pero NO muestra los mensajes, para ello deberemos borrar el método GET y volverlo a crear con Lambda Proxy Integration, para esto analizaremos como recibe la información.



Ahora la UI quedará como se muestra en la imagen.

```
</head>
<body>
<h1>Chat AWS <img alt="chat icon" style="vertical-align: middle;"></h1>

<div id="messages"></div>

<label>Usuario:</label><br />
<input id="user" type="text" placeholder="Tu nombre" /><br />

<label>Mensaje:</label><br />
<input id="text" type="text" placeholder="Escribe un mensaje..." /><br />

<button onclick="sendMessage()">Enviar</button>
```

Y el script que llamará a las lambdas:

```

3
4 <script>
5   const API_BASE = "https://4iqi63mxge.execute-api.us-east-1.amazonaws.com/dev"; // CAMBIAR
6
7   async function sendMessage() {
8     const user = document.getElementById("user").value;
9     const text = document.getElementById("text").value;
10
11     if (!user || !text) return alert("Usuario y mensaje son obligatorios");
12
13     const res = await fetch(`${API_BASE}/send`, {
14       method: "POST",
15       headers: { "Content-Type": "application/json" },
16       body: JSON.stringify({ user, text, room: "general" })
17     );
18
19     const data = await res.json();
20     console.log("Enviado:", data);
21     document.getElementById("text").value = ""; // limpiar input
22     loadMessages();
23   }
24
25
26   async function loadMessages() {
27     try {
28       const res = await fetch(`${API_BASE}/messages?room=general&limit=20`);
29       const data = await res.json();
30
31       const container = document.getElementById("messages");
32       container.innerHTML = ""; // limpiar
33
34       (data.messages || [])
35         .sort((a, b) => a.timestamp - b.timestamp)
36         .forEach(msg => {
37           const p = document.createElement("p");
38           const date = new Date(msg.timestamp).toLocaleTimeString();
39           p.textContent = `[${date}] ${msg.user}: ${msg.text}`;
40           container.appendChild(p);
41         });
42     } catch (e) {
43       console.error("Error cargando mensajes:", e);
44     }
45   }
46
47   // refrescar cada 5 segundos
48   setInterval(loadMessages, 5000);
49   loadMessages(); // cargar al abrir
50 </script>
51 </body>
```

<https://github.com/jorloque/chat-aws>