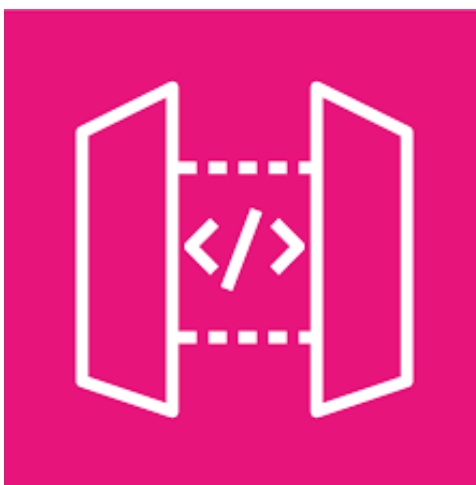


# Práctica 8

## 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:

### Chat AWS

Usuario:

Mensaje:

[22:36:04] alice: hola

[15:45:30] jorge: hola

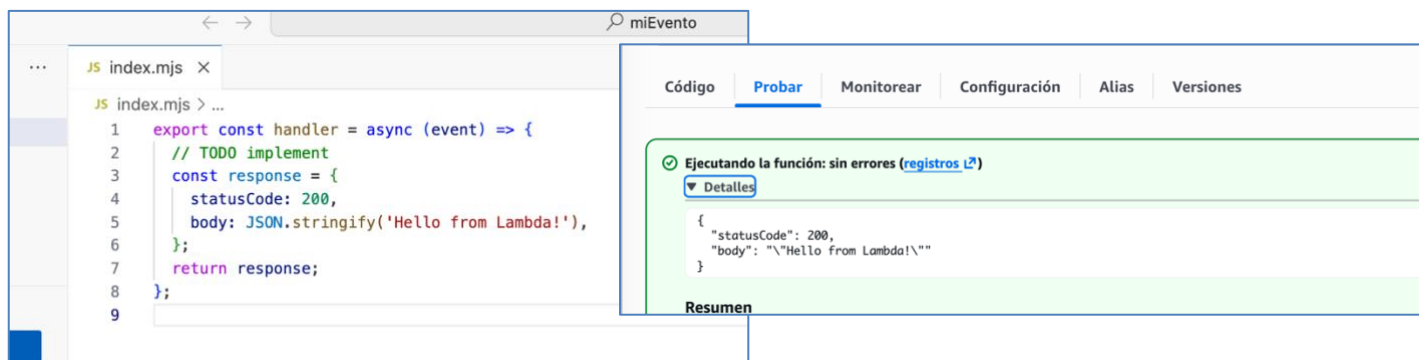
[15:45:41] pepe: hola

[15:52:10] manolo: hola

[15:52:21] bea: hola

## 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.



The screenshot shows the AWS Lambda console interface. On the left, the code for the function is displayed in a text editor:

```
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
```

On the right, the 'Probar' (Test) tab is selected, showing a successful execution result:

✓ Ejecutando la función: sin errores (registros 12)

Detalles

```
{  
  "statusCode": 200,  
  "body": "\"Hello from Lambda!\""  
}
```

Resumen

## 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" }),
      };
    }
  }

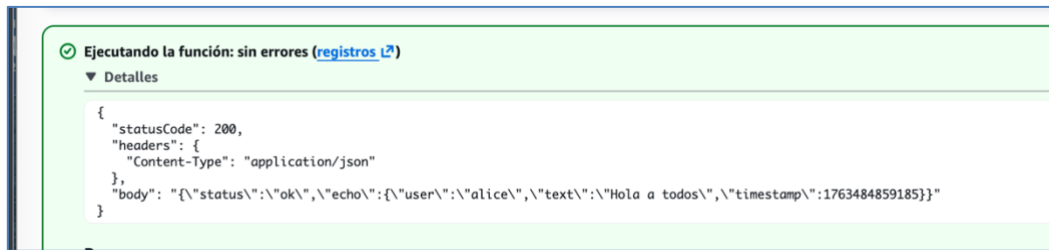
  const { user, text } = body;

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

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

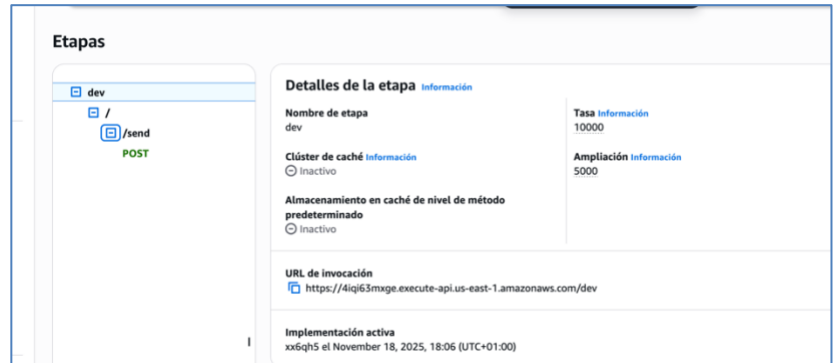
  return {
    statusCode: 200,
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ status: "ok", echo: message }),
  };
};
```

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



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:



```

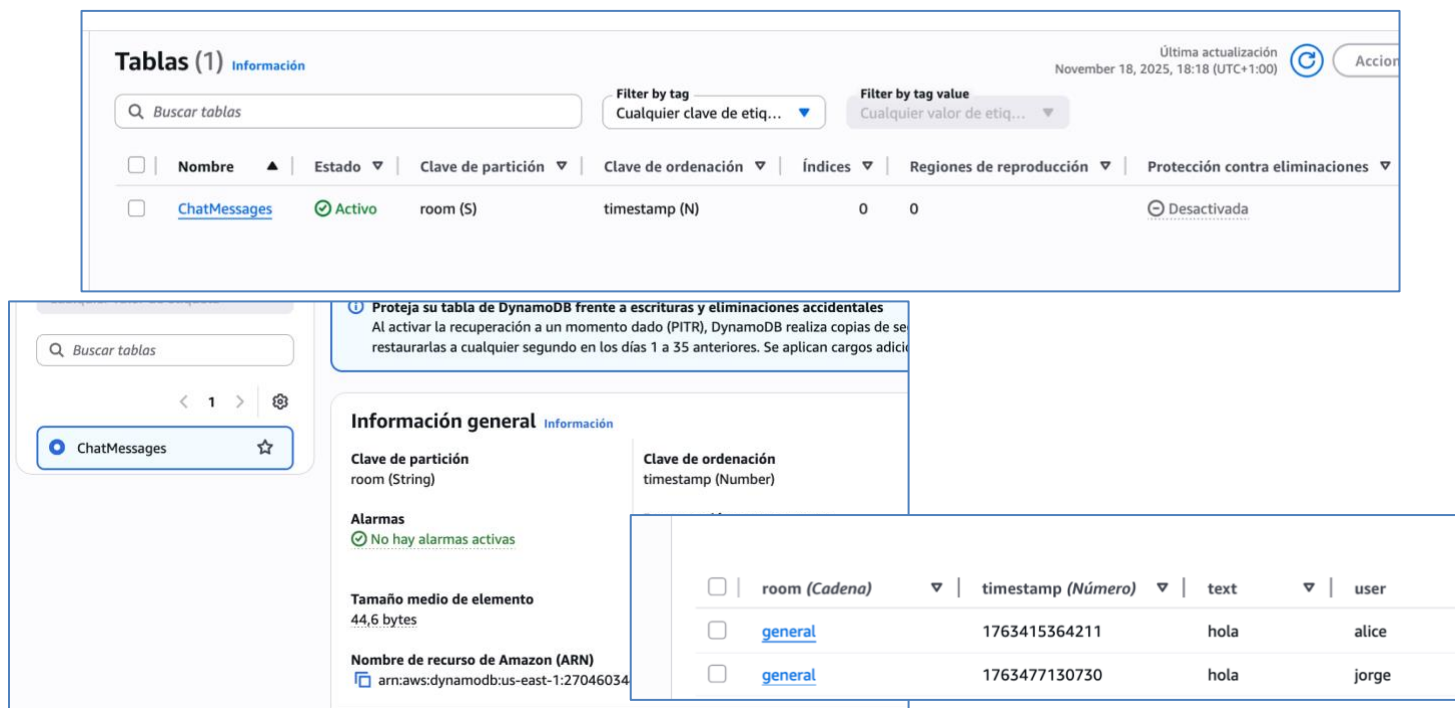
1  export const handler = async (event) => {
2      console.log("EVENT:", JSON.stringify(event, null, 2));
3
4      let body = event.body;
5
6      // Caso: API Gateway pasa user/text directamente
7      if (!body && (event.user || event.text)) {
8          body = event;
9      }
10
11     // Si body 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:



**Tablas (1) Información**

Última actualización: November 18, 2025, 18:18 (UTC+1:00)

Buscar tablas

Filter by tag: Cualquier clave de eti... Filter by tag value: Cualquier valor de eti...

	Nombre	Estado	Clave de partición	Clave de ordenación	Índices	Regiones de reproducción	Protección contra eliminaciones
<input type="checkbox"/>	<a href="#">ChatMessages</a>	Activo	room (S)	timestamp (N)	0	0	Desactivada

**Información general**

Clave de partición: room (String)

Clave de ordenación: timestamp (Number)

Alarmas: No hay alarmas activas

Tamaño medio de elemento: 44,6 bytes

Nombre de recurso de Amazon (ARN): arn:aws:dynamodb:us-east-1:27046034

	room (Cadena)	timestamp (Número)	text	user
<input type="checkbox"/>	<a href="#">general</a>	1763415364211	hola	alice
<input type="checkbox"/>	<a href="#">general</a>	1763477130730	hola	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://4iqi63mxge.execute-api.us-east-1.amazonaws.com/dev/send

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

<input type="checkbox"/>	<a href="#">general</a>	1763477130730	hola	manolo
<input type="checkbox"/>	<a href="#">general</a>	1763477541238	hola	bea
<input type="checkbox"/>	<a href="#">general</a>	1763491006530	mensaje guardado en Dynamo	alice

```
Vista Previa README.md  JS mainfuncion.js  main.html  chat-messages.ts  JS imp
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 /messages

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 2)

▼ Detalles

```
{
  "statusCode": 200,
  "body": "[{"messages":[{"room":"general","timestamp":176349186530,"user":"alice","text":"mensaje guardado en Dynamo"}, {"room":"general","timestamp":1763477541238,"user":"bea","text":"hola"}, {"room":"general","timestamp":1763477538421,"user":"manolola","text":"hola"}, {"room":"general","timestamp":1763477141489,"user":"pepe","text":"hola"}, {"room":"general","timestamp":1763477130730,"user":"jorge","text":"hola"}, {"room":"general","timestamp":1763415364211,"user":"alice","text":"hola"}]]"
```

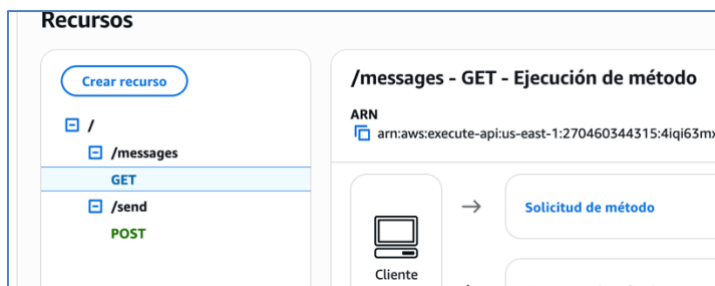
Resumen

Código SHA-256  
q8E7Nex5xhKT9/d4bGpAYOXJYFAUJJOUDJ80ivK8E=

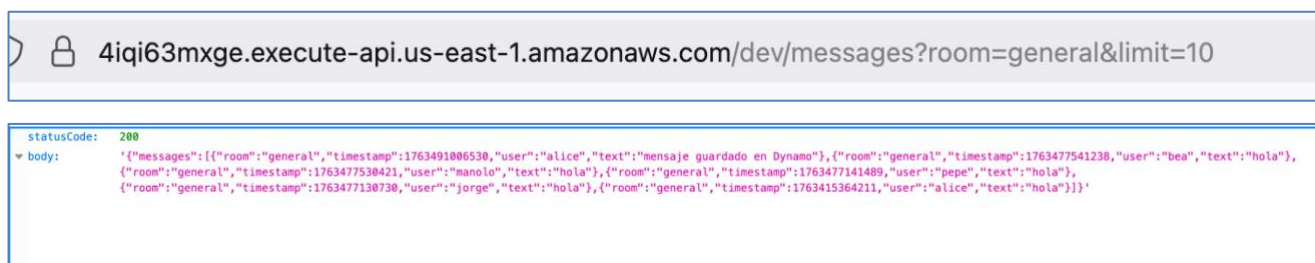
Tiempo de ejecución  
hace 2 segundos

Ahora solo falta integrarlo en el api gateway.





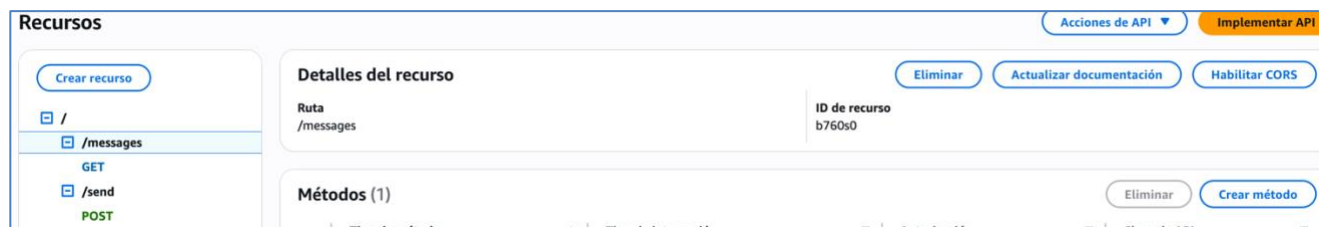
Y desplegarlo para obtener un endpoint. Para ello podremos probar en el navegador:



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.

### Crear método

**Detalles del método**

Tipo de método  
GET

Tipo de integración

☒ **Función de Lambda**  
Integre su API con una función de Lambda.

☐ **HTTP**  
Lleve a cabo la integración con un punto de conexión HTTP.

☐ **Servicio de AWS**  
Lleve a cabo la integración con un servicio de AWS.

☐ **Enlace de VPC**  
Lleve a cabo la integración con un recurso al que red pública de Internet.

☒ **Integración de proxy de Lambda**  
Envíe la solicitud a la función de Lambda como un evento estructurado.

**Función de Lambda**  
Proporcione el nombre de la función de Lambda o un alias. También puede proporcionar un ARN de otra cuenta.

us-east-1

**Otorgue permiso a API Gateway para invocar la función de Lambda**  
Al guardar los cambios, API Gateway actualiza la política basada en recursos de la función de Lambda para permitir que esta A

Ahora la UI quedará como se muestra en la imagen.

```
</head>
<body>
  <h1>Chat AWS <img alt="AWS logo" data-bbox="545 825 565 840"/></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>
</body>
```

Y el script que llamará a las lambdas:

```
3
4
5 <script>
6   const API_BASE = "https://4iqi63mxge.execute-api.us-east-1.amazonaws.com/dev"; // CAMBIAR
7
8   async function sendMessage() {
9     const user = document.getElementById("user").value;
10    const text = document.getElementById("text").value;
11
12    if (!user || !text) return alert("Usuario y mensaje son obligatorios");
13
14    const res = await fetch(`${API_BASE}/send`, {
15      method: "POST",
16      headers: { "Content-Type": "application/json" },
17      body: JSON.stringify({ user, text, room: "general" }),
18    });
19
20    const data = await res.json();
21    console.log("Enviado:", data);
22    document.getElementById("text").value = ""; // limpiar input
23    loadMessages();
24  }
25
26  async function loadMessages() {
27    try {
28      const res = await fetch(`${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
51 </script>
52 </body>
```

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