

1. Write a Lambda function to CRUD over the Students DynamoDB table. **30 points.**

El acrónimo CRUD hace referencia a la serie de operaciones Create, Read, Update and Delete, que son el conjunto básico de acciones en programación para gestionar el almacenamiento de datos en Bases de Datos o File Systems.

Para esta tarea en particular, la operación de Create agregara un registro de Dynamodb en la tabla de Students. Read, extrae la información de un registro en base a su ID. Update, actualiza los valores de un ID determinado. Delete, remueve un registro indicando el ID del estudiante.

Una Función Lambda entra dentro del modelo FaaS (Function-as-a-Service), que ofrece un tipo de servicio computacional serverless proporcionado por AWS que permite a los desarrolladores ejecutar código en respuesta a eventos sin tener que administrar servidores. Este servicio es compatible entre varios lenguajes de programación y puede utilizarse para diversas tareas, como el procesamiento de datos, el procesamiento de archivos en tiempo real y para el desarrollo del Back-end en una aplicación web.

Las operaciones CRUD se implementaron en node.js v18.13, empleando el AWS SDK para cliente JavaScript DynamoDB V3. A continuación el código es el siguiente:

```
// Haciendo require del SDK de DynamoDB para Node.js
const {DynamoDBClient} = require("@aws-sdk/client-dynamodb");
// Método que sirve para convertir un objeto Javascript a un registro de DynamoDB
const {marshall} = require("@aws-sdk/util-dynamodb");
const {PutItemCommand, GetItemCommand, UpdateItemCommand, DeleteItemCommand, ScanCommand} =
require("@aws-sdk/client-dynamodb");
```

```

// Archivo JSON con el que se hicieron las pruebas
const lambdaFunctions = require("./LambdaTest.json");

// Se genera un cliente para tener acceso a los Comandos de DynamoDB
const client = new DynamoDBClient({region: 'us-east-1'});
// Este es la función Handler que va a identificar el Lambda para su ejecución
exports.handler = async (event) => {
  if(typeof(event.item) === 'undefined' || typeof(event.item.id) === 'undefined') {return
  {'statusCode': 500, 'body': "event.item was undefined"}};
  let response;
  switch (event.httpMethod){
    case "POST":
      response = await postOperation(event.item); break;

    case "GET":
      response = await getOperation(event.item); break;

    case "PATCH":
      response = await patchOperation(event.item); break;

    case "DELETE":
      response = await deleteOperation(event.item); break;

    default:
      response = {"statusCode": 405, "body": "Method Not Allowed"}; break;
  }
  return response;
}

// CREATE
const postOperation = async (item) => {
  // Se checa primero que el ID del item que se quiere crear, no exista en la tabla Students
  if (await runCheckID(item.id)){return {'statusCode': 404, body: "Item Already Exists"}};
  // Los parámetros del JSON se guardan en una variable como objeto Javascript y con
  // Marshall se convierten a como si fuera un registro de DynamoDB con el tipo de dato
  // especificado
  const putUserParams = {TableName: "Students",
    Item: marshall({
      id : item.id,
      full_name : item.full_name,
      personal_website : item.personal_website
    }),
    ReturnValues : "ALL_OLD"};

  try {
    // Aquí se ejecuta el comando PutItemCommand con los respectivos parámetros del
    // usuario. Y se hace un await para esperar que el output del comando se almacene en la
    // variable data
    const data = await client.send(new PutItemCommand(putUserParams));
    return {
      'statusCode': data.$metadata.httpStatusCode,
      'body': putUserParams.Item,
      'observation': "POST Request was Successful"
    }
  }
}

```

```

    };
  } catch (e) {
    return {'statusCode': 500, 'body': String(e)};
  }
};

// RETRIEVE
// Para la función getOperation solo se extrae el ID del item para ejecutar el
// GetItemCommand y extraer los datos del registro que coincida con dicho ID. También cabe
// destacar que al principio de la función se checa si el ID del item existe en la tabla
// Students
const getOperation = async (item) => {
  if (!(await runCheckID(item.id))) {return { statusCode: 404, body: "Item Not Found" };}
  const getUserParams = { TableName: "Students", Key: marshall({ id: item.id }) };
  try {
    const data = await client.send(new GetItemCommand(getUserParams));
    return {
      statusCode: data.$metadata.httpStatusCode,
      body: data.Item,
      observation: "GET Request was Successful",
    };
  } catch (e) {
    return { statusCode: 500, body: String(e) };
  }
};

// UPDATE
// Para realizar un update se requiere el item con los valores con los que se va a
// reemplazar las llaves del antiguo ID.
const patchOperation = async (item) => {
  if(!(await runCheckID(item.id))) {return {'statusCode': 404, 'body': 'Item Not Found'}}
  const patchUserParams = {TableName : "Students",
    Key : marshall({'id': item.id}),
    UpdateExpression : 'SET full_name = :f, personal_website = :p',
    ExpressionAttributeValues : marshall ({
      ':f' : item.full_name,
      ':p' : item.personal_website}),
    ReturnValues : "ALL_NEW"};

  try {
    const data = await client.send(new UpdateItemCommand(patchUserParams));
    return {
      'statusCode': data.$metadata.httpStatusCode,
      'body': data.Attributes,
      'observation':"PATCH Request was Successful"}
  } catch (e) {
    return {'statusCode': 500, 'body': String(e)};
  }
};

// DELETE
// Para la operación de borrado solo se requiere en ID, sin embargo para mantener la

```

```

//consistencia con las demás operaciones, todas toman como parámetro el ítem, para que
//después extraigan el ID. Aquí de la misma manera que con getOperation se checa si el ID
//previamente existe, si no es el caso, se manda un error 404, indicando que no se puede
//borrar un ID que no existe.
const deleteOperation = async (item) => {
    if(!await runCheckID(item.id)) {return {'statusCode': 404, 'body': 'Item Not Found'}}
    const deleteUserParams = {TableName : "Students",
                                Key : marshall({'id': item.id}),
                                ReturnValues : "ALL_OLD"};

    try {
        const data = await client.send(new DeleteItemCommand(deleteUserParams));
        return {
            'statusCode': data.$metadata.httpStatusCode,
            'body': data.Attributes,
            'observation':"DELETE Request was Successful"
        };
    } catch (e) {
        return {'statusCode': 500, 'body': String(e)};
    }
};

// El propósito de esta función es emplear el ScanCommand para verificar si un determinado
// ID se encuentra en la tabla.
const checkID = async (id) => {
    const data = await client.send(new ScanCommand({TableName:"Students"}));
    for(const item of data.Items){
        if(item.id.S === id) {return true};
    } return false;
};

async function runCheckID(id) {
    return await checkID(id)
}
// Esta función se utilizó para comprobar que las operaciones estuvieran funcionando
//correctamente
async function main(){
    console.log(await exports.handler(lambdaFunctions));
}

main();

```

Para crear una función Lambda se requiere de ejecutar el siguiente comando:

```
aws lambda create-function \
--function-name AgrazLambdaV5 \
--runtime nodejs18.x \
--role arn:aws:iam::292274580527:role/lambda_ice191 \
--zip-file fileb:///Users/kekaz16/Documents/Semestre6/CloudComputing/ \
scripts/LambdaFunction/AgrazCRUDV2.js.zip
--handler AgrazCRUD.handler
```

Los argumentos de **create-function** son los siguientes:

**--function-name:** es el nombre que va a tener la función Lambda

**--runtime:** es la versión del lenguaje de programación con la que se va a ejecutar la Lambda

**--role:** es el ARN del usuario IAM que tiene los permisos necesarios para ejecutar los comandos de creación de una Función Lambda. En este caso se usó el usuario que se proporcionó en clase para tener accesos al servicio de Lambda.

**--zip-file:** es la ruta al archivo Zip, donde se encuentra el código fuente que el Lambda va a ejecutar.

**--handler:** es el nombre del método dentro del código que Lambda va mandar a llamar para correr su función.

Una vez creada el Lambda se obtiene el siguiente Output:

```
{
  "FunctionName": "AgrazLambdaV5",
  "FunctionArn": "arn:aws:lambda:us-east-1:292274580527:function:AgrazLambdaV5",
  "Runtime": "nodejs18.x",
  "Role": "arn:aws:iam::292274580527:role/lambda_ice191",
  "Handler": "V2AgrazCRUD.handler",
  "CodeSize": 1359,
  "Description": "",
  "Timeout": 3,
  "MemorySize": 128,
  "LastModified": "2023-03-07T08:28:22.615+0000",
  "CodeSha256": "Y6/TMbQemr1Zb4XxRJecB6yH95wER+tM5D6nyEhqNhI=",
  "Version": "$LATEST",
  "TracingConfig": {
    "Mode": "PassThrough"
  },
  "RevisionId": "3fcc707-4c82-4b62-8b6f-0e018676d32c",
  "State": "Pending",
  "StateReason": "The function is being created.",
  "StateReasonCode": "Creating",
  "PackageType": "Zip",
  "Architectures": [
    "x86_64"
  ],
  "EphemeralStorage": {
    "Size": 512
  },
  "SnapStart": {
    "ApplyOn": "None",
    "OptimizationStatus": "Off"
  },
  "RuntimeVersionConfig": {
    "RuntimeVersionArn": "arn:aws:lambda:us-east-1::runtime:c869d752e4ae21a3945cfcb3c1ff2beb1f160d7bcec3b0a8ef7caceae73c055f"
  }
}
```

Es importante guardar el valor de la llave FunctionArn, ya que se usará más adelante para invocar junto el API Gateway.

Para verificar que el Lambda está funcionando correctamente se tiene que ejecutar con el comando **aws lambda invoke**.

```
aws lambda invoke
--function-name AgrazLambdaV5
--cli-binary-format raw-in-base64-out
--payload file://LambdaTest.json response.json
```

Dentro de la función **invoke** se tiene que cumplir los siguientes parámetros:

**--function-name:** es el nombre de la función Lambda que va a invocar.

**--cli-binary-format:** El estilo de formato que se utilizará para los blobs binarios. El formato por defecto es base64. El formato base64 espera que los blobs binarios se proporcionen como una cadena codificada en base64.

**--payload:** es la ruta al archivo JSON que tiene los datos del request

**response.json:** es nombre del archivo que se va a crear una vez que se ejecute la Lambda. Este archivo va a contener la respuesta a la solicitud de invocación de Lambda.

Así se vería al probarla con las distintas operaciones CRUD.

Con CREATE

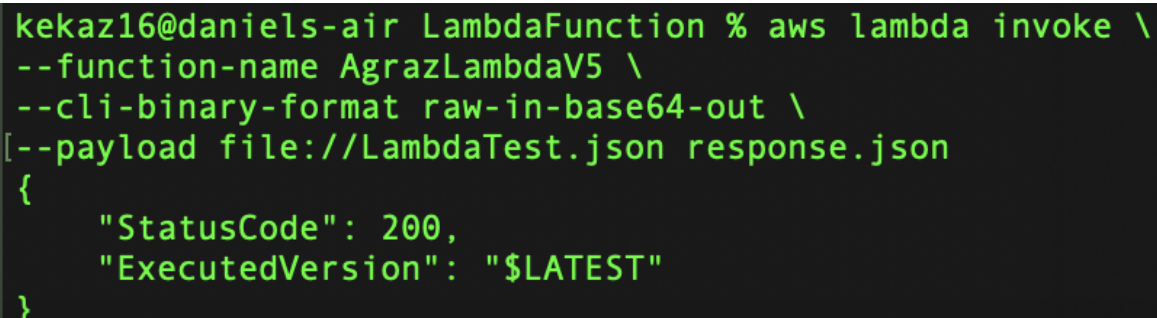
Este es el **payload:**

A screenshot of a code editor with three tabs: 'V2AgrazCRUD.js', 'LambdaTest.json', and 'response.json'. The 'LambdaTest.json' tab is active, showing a JSON object with the following structure: 

```
{  "httpMethod": "POST",  "item": {    "id": "25",    "full_name": "PepeAguilar",    "personal_website": "TaqueriaAguilar.com"  }}
```

 The code is written in a dark-themed editor with syntax highlighting.

Este es el output que se recibe al invocar el Lambda. El statusCode 200, indica que la solicitud fue resuelta con éxito.

A screenshot of a terminal window with a dark background and green text. It shows the command to invoke a Lambda function and the resulting JSON output: 

```
kekaz16@daniels-air LambdaFunction % aws lambda invoke \  --function-name AgrazLambdaV5 \  --cli-binary-format raw-in-base64-out \  [--payload file://LambdaTest.json response.json]  {    "StatusCode": 200,    "ExecutedVersion": "$LATEST"  }
```

Y al final se en el **response.json** genera el output del Lambda.

```
JS V2AgrazCRUD.js  {} LambdaTest.json  {} response.json X
{} response.json > {} body
1  {
2      "statusCode": 200,
3      "body": {
4          "id": {
5              "S": "25"
6          },
7          "full_name": {
8              "S": "PepeAguilar"
9          },
10         "personal_website": {
11             "S": "TaqueriaAguilar.com"
12         }
13     },
14     "observation": "POST Request was Successful"
15 }
```

Haciendo un **aws dynamodb scan - - table-name Students** se puede comprobar que si aparece el registro creado.

```
{
  "full_name": {
    "S": "PepeAguilar"
  },
  "id": {
    "S": "30"
  },
  "personal_website": {
    "S": "TaqueriaAguilar.com"
  }
},
```



Con la operación RETRIEVE.

Este es el **payload**:

```
JS V2AgrazCRUD.js {} LambdaTest.json X
{} LambdaTest.json > ...
1  {
2    "httpMethod": "GET",
3    "item": {
4      "id": "10"
5    }
6  }
```

Output del Invoke:

```
kekaz16@daniels-air LambdaFunction % aws lambda invoke \
--function-name AgrazLambdaV5 \
--cli-binary-format raw-in-base64-out \
[--payload file://LambdaTest.json response.json
{
  "StatusCode": 200,
  "ExecutedVersion": "$LATEST"
}
```

El **response.json** que genera el Lambda.

```
JS V2AgrazCRUD.js {} LambdaTest.json {} response.json X
{} response.json > ...
1  {
2    "statusCode": 200,
3    "body": {
4      "full_name": {
5        "S": "Jim"
6      },
7      "id": {
8        "S": "10"
9      },
10     "personal_website": {
11       "S": "Jim.com"
12     }
13   },
14   "observation": "GET Request was Successful"
15 }
```

Con la operación UPDATE.

Este es el **payload**:

```
JS V2AgrazCRUD.js  {} LambdaTest.json ×  {} resp
{} LambdaTest.json > ...
1  {
2    "httpMethod": "PATCH",
3    "item": {
4      "id": "30",
5      "full_name": "Agustin Alizondo",
6      "personal_website": "AlizaTin.com"
7    }
8  }
```

Output del Invoke:

```
kekaz16@daniels-air LambdaFunction % aws lambda invoke \
--function-name AgrazLambdaV5 \
--cli-binary-format raw-in-base64-out \
[--payload file://LambdaTest.json response.json
{
  "statusCode": 200,
  "ExecutedVersion": "$LATEST"
}
```

El **response.json** que genera el Lambda. Con los datos del nuevo registro.

```
JS V2AgrazCRUD.js  {} LambdaTest.json  {} response.json ×
{} response.json > ...
1  {
2    "statusCode": 200,
3    "body": {
4      "full_name": {
5        "S": "Agustin Alizondo"
6      },
7      "id": {
8        "S": "30"
9      },
10     "personal_website": {
11       "S": "AlizaTin.com"
12     }
13   },
14   "observation": "PATCH Request was Succesful"
15 }
```

Con la operación DELETE.

Este es el **payload**:

```
JS V2AgrazCRUD.js {} LambdaTest.json X
{} LambdaTest.json > ...
1  {
2    "httpMethod": "DELETE",
3    "item": {
4      "id": "30"
5    }
6  }
```

Output del Invoke:

```
kekaz16@daniels-air LambdaFunction % aws lambda invoke \
--function-name AgrazLambdaV5 \
--cli-binary-format raw-in-base64-out \
--payload file://LambdaTest.json response.json
{
  "statusCode": 200,
  "executedVersion": "$LATEST"
}
```

**response.json**

```
JS V2AgrazCRUD.js {} LambdaTest.json {} response.json X
{} response.json > ...
1  {
2    "statusCode": 200,
3    "body": {
4      "full_name": {
5        "S": "Agustin Alizondo"
6      },
7      "id": {
8        "S": "30"
9      },
10     "personal_website": {
11       "S": "AlizaTin.com"
12     },
13   },
14   "observation": "DELETE Request was Succesful"
15 }
```

2. Write an API Gateway API to CRUD over your Lambda function. **30 points.**

**Esta en Progreso**

3. Implement 404 HTTP response when an invalid id is passed to the Read in your API Gateway API/Lambda. **30 points.**

**Esta en Progreso**

4. Read the Test Driven Development is the best thing that has happened to software design article and write a summary and opinions about it. **10 points**

### **Summary**

TDD (Test Driven Development) es un enfoque de desarrollo de software en el que se escriben primero las pruebas antes de escribir código. TDD se originó alrededor de los principios de los 2000s por el libro de Kent Beck, "Test-Driven Development: By Example" en 2002. Sin embargo, las ideas de TDD se remontan al marco de trabajo de XP (2002).

El objetivo principal de TDD se concentra en representar el comportamiento de una funcionalidad del programa mediante la elaboración de pruebas casos de fallo. Este enfoque es iterativo y se maneja por ciclos:

1. Escribe una Prueba de Fallo
2. Haz que pase la prueba
3. Refactoriza el código

Algunas de las razones que da el artículo de la efectividad de TDD:

- Ofrece retroalimentación sobre la implementación del código y su diseño.
- Permite delimitar la funcionalidad del código, haciendo que sea más simple.
- Ayuda a conocer el "Qué", verificando el comportamiento de la función y no los detalles de la implementación.
- Te hace tener una expectativa sobre lo que el código tiene que hacer

- Apoya a que los requerimientos sean cumplidos con la aplicación del programa

Dentro de TDD existen mecanismos que posibilitan la evaluación de código que es imposible de probar en primera instancia. De los cuales el artículo menciona:

1. **Spying or Mocking:** Cuando se tiene un componente que hace uso de llamadas externas a un servicio de terceros, que igual genera cambios en el servidor.
2. **Variable Test Data:** Cuando se tienen datos que varían con el tiempo, de tal modo, que tus pruebas siempre van a fallar en dado momento. La manera en la que se soluciona esto, es generalizando la información para establecer que sea el tiempo que se consulta la información no
3. **Bloated Setup:** Es síntoma se asocia a las funciones que rompen el Principio de Responsabilidad Singular, y está fuertemente acoplado con otros componentes.
4. **Mocking Hell:** Hace referencias a las consecuencias de cuando se rompe La Ley de Demetrio, que indica que un objeto solo debe ser capaz de comunicarse con sus vecinos adyacentes sin compartir o tener conocimiento de cómo funcionan otros componentes.

## Opinions

Personalmente, Testing no ha sido una área la cual me llame mucho la atención, hasta ahora tengo una vaga percepción sobre lo que involucra hacer pruebas de Testing. Por ello, considero que este artículo me llevó a investigar para conocer un poco más sobre este proceso, así como su aplicación en TDD. Recuerdo haber escuchado el concepto de Test Driven Development en los libros de Robert C. Martin, sin embargo, nunca llegué a entenderlo en su totalidad, o pensaba que era algo muy complicado.

Después de haber leído el artículo, nunca había pensado sobre el acercamiento de Test-First. Es instintivo que un programador primero escriba código y luego realice sus pruebas, haga debugging y refactorización. Resultó un descubrimiento conocer de esta

metodología, y el concepto de que las Pruebas buscan concebir una expectativa que se tiene en el comportamiento del programa y no tanto en la parte de la implementación. También he escuchado de BDD (Behaviour Driven Development), que también tiene un enfoque en el desarrollo en base al comportamiento, sin embargo, no tengo la certeza, si posee alguna similitud o comparten algo en común TDD y BDD.

El hecho que TDD sea una forma de proveer retroalimentación sobre la implementación y diseño del código son indicadores que te permiten realizar cambios a tiempo, y tener un registro de estas modificaciones en base a las pruebas que se hicieron. Considero que el crear pruebas con el propósito de hacer fallar el sistema o componente, a largo plazo ofrece muchos beneficios.

La mayoría de ventajas que tiene TDD se pueden visualizar mejor en el Summary realizado, no obstante, me gustaría retomar el beneficio de que cuando tu código está estructurado en pruebas, tienes una percepción de cuando parar. Ya que las pruebas delimitan el alcance en funcionalidad del componente, tal que, la hora de agregar un nuevo requerimiento y tener que abrir el objeto a extensión, resulta más fácil el incorporar cambios en el código y esto se refleja en que la implementación del código se de manera más espontánea.

Dentro de los ejemplos de código difícil de Testear, los síntomas se me figuraron a lo que ocurre cuando no se llevan buenas prácticas al escribir código. El Principio de Responsabilidad Singular, La Ley de Demetrio, el acoplamiento y generación de dependencias, el uso de llamadas a servicios externos sin posibilidad de rastro, fueron las razones que me llevaron a pensar que el motivo de porque puede llegar a existir código difícil de poner a prueba depende de los hábitos del programador.

Otra cosa que me pareció interesante del artículo fue la mención de la regla YAGNI (You Ain't Gonna Need It), que en base a lo que leí, es una manera de aceptar que si el diseño de una función no está concreta y no hay razón fuerte de crear pruebas para su implementación.

## Referencias Bibliográficas

Torczuk, A (2019). *Test Driven Development is the best thing that has happened to software design*.

<https://www.thoughtworks.com/insights/blog/test-driven-development-best-thing-has-happened-software-design>.

AgileAlliance (2023). *TDD*.

~(infinite~false~filters~(postType~(~'page~'post~'aa\_book~'aa\_event\_session~'aa\_experience\_report~'aa\_glossary~'aa\_research\_paper~'aa\_video)~tags~(~'tdd))~searchTerm~'~sort~false~sortDirection~'asc~page~1)

AWS (2023). *DynamoDB Client - AWS SDK for JavaScript v3*.

<https://docs.aws.amazon.com/AWSJavaScriptSDK/v3/latest/clients/client-dynamodb/index.html>

AWS CLI Command Reference (2023). *invoke*.

<https://awscli.amazonaws.com/v2/documentation/api/latest/reference/lambda/invoke.html>

AWS CLI Command Reference (2023). *create-function*.

<https://awscli.amazonaws.com/v2/documentation/api/latest/reference/lambda/create-function.html>