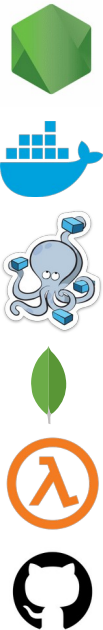


4.5



## *TypeScript – Guía Rápida*

*Martín Castagnolo – Software Developer*

Martyn C. 

# Introducción



TypeScript es un lenguaje de programación desarrollado y mantenido por Microsoft, el cual ha sido construido sobre **Javascript**. Es correcto decir que es JS pero con más características. Algunos dirían: “Javascript con esteroides”. Por lo que cualquier código escrito en JS es código TS válido.

Además, esta herramienta por debajo incluye un subconjunto de JS que facilita la realización de migraciones. Esto significa que ambos pueden convivir en un mismo proyecto.

Vale la pena utilizar TS en lugar de JS, debido a que se incorpora un sistema de tipos. O sea, nos va a permitir definir de antemano la clasificación de los diferentes valores y expresiones que se pueden usar en un lenguaje. Si hacemos un paralelismo con C++, al momento de realizar una función es necesario informar qué tipo de dato vamos a pasar. “¿Se va a recibir una cadena de caracteres? ¿un entero?”. Entonces, lo que nos brinda TS es información bastante útil cuando estamos implementando una función, ya que sabemos precisamente con lo que se está trabajando.

Un detalle no menor es que es imposible que TS se compile si en algún punto usamos mal lo anterior mencionado. Por lo tanto, si hemos declarado que un método trabajará con dos números, tendremos la seguridad de que eso será respetado siempre.



## Práctica



Veremos a continuación la sintaxis que caracteriza a TS para que te familiarices y después iremos de lleno a unos ejemplos prácticos. Algunos de ellos pueden ser aplicados tanto en el front como en el back. Depende del entorno en el que quieras realizar las pruebas. En este caso, el enfoque es back.

## 1. Tipos de Datos

```
1  const nombreDeUnPlaneta: string = "Júpiter";
2  const cantidadDeHoras: number = 24;
3  const aprendiendoTypeScript: boolean = true;
4  const sinDefinir: undefined = undefined;
5  const nulo: null = null;
```

También una variable puede contener más de un tipo de dato:

```
6  const dosOpciones: number | string = "opción válida";
7  const tresOpciones: undefined | string | number = 26;
```

En el caso de que se nos ocurriera asignar un valor cuyo tipo de dato no coincide con lo especificado, nuestro programa lanzará un error y no podremos avanzar hasta que no se realice el ajuste necesario.

## 2. Arreglos

Hay dos maneras para definir arrays. La primera es como antes pero asignando un juego de corchetes []. La segunda, es a través de una matriz genérica Array. Estamos hablando de exactamente lo mismo.

```
8  const meses: string[] = ["enero", "febrero", "marzo", "abril", "mayo", "junio", "julio", "agosto",
    "septiembre", "octubre", "noviembre", "diciembre"];
9  const cantidadDeDias: Array<number> = [1, 2, 3, 4, 5, 6, 7];
10 const finDeSemana: boolean[] = [false, false, false, false, true, true, true];
```

Para incluir distintos tipos de datos en una matriz, usamos any[]:

```
11  const datosMezclados: any[] = [false, "algo", 24];
```

### 3. Objetos

```
13  const auto { color: string, puertas: number } = {  
14      color: "blanco",  
15      puertas: 4  
16  }
```

Como se puede observar, los objetos permiten especificar los tipos de las variables.

### 4. Aliases

Si se requiere, existe la posibilidad de crear uno(a) su propios tipos usando alias de tipo.

```
18  type vehiculo = {  
19      color: string;  
20      puertas: number;  
21  }
```

```
23  const camion: vehiculo = {  
24      color: "rojo",  
25      puertas: 2  
26  }
```

```
28  const camioneta: vehiculo = {  
29      color: "blanco",  
30      puertas: 4  
31  }
```

Entrando en detalles, type es una colección de datos que admite la creación de un nuevo nombre para un tipo. Cabe resaltar que no permite el uso de un objeto ni tampoco se pueden utilizar varias declaraciones fusionadas.

## 5. Interfaces

Muy parecido al punto anterior, dentro de TS, existe la posibilidad de definir métodos y propiedades sin hacer su implementación. Esto quiere decir, que lo que se está indicando es una descripción de cómo debería verse el objeto. La diferencia con los aliases, está en que estos son responsables de verificar el tipo de dato de cualquier valor tomado antes de que se pueda utilizar como una entrada en el código. En cambio, acá no. Un ejemplo:

```
33  interface UsuarioInterface {
34      nombre: string;
35      apellido: string;
36      edad: number;
37      developer: boolean;
38  }
40  class Usuario implements UsuarioInterface {
41      nombre: string;
42      apellido: string;
43      edad: number;
44      developer: boolean;
46      constructor(
47          nombre: string,
48          apellido: string,
49          edad: number,
50          developer: boolean
51      ) {
```

```

52         this.nombre = nombre;
53         this.apellido = apellido;
54         this.edad = edad;
55         this.developer = developer;
56     }

58     retornarInfoDeUsuario(): string {
59         return `${this.nombre} - ${this.apellido} - ${this.edad} - ${this.developer}`;
60     }
61 }

62 const usuario = new Usuario("Martín", "Castagnolo", 26, true);
63 console.log(usuario.retornarInfoDeUsuario()); // más abajo veremos cómo ejecutar ficheros ts ;)

```

Si tenemos dos interfaces o más, con un mismo nombre, serán fusionadas. Aspecto que no se encuentra en el caso de type. Y la diferencia sustancial entre ambas definiciones es que type no tiene propósito de implementación y las interfaces, sí.

Notar que en el método `retornarInfoDeUsuario()` estoy señalando que se va a retornar un string.

## 6. Funciones

Si bien existen varios tipos de funciones en JS, con ver los siguientes ejemplos, vas a tener una idea para luego realizar prácticas por cuenta propia:

```

65     const retornarLongitud = (matriz: Array<string>): number => {
66         return matriz.length;
67     }

69     function retornarPlanetas(): Array<object> {

```



```

70         return [ {
71             primer_planeta: "Mercurio",
72             segundo_planeta: "Venus",
73             tercer_planeta: "Tierra",
74             cuarto_planeta: "Marte",
75             quinto_planeta: "Júpiter",
76             sexto_planeta: "Saturno",
77             septimo_planeta: "Urano",
78             octavo_planeta: "Neptuno",
79             noveno_planeta: "Plutón"
80         } ];
81     }
82
83     const procesarDatos = async (datosIngresados: string): Promise<boolean> => {
84         return new Promise((resolve) => {
85             datosIngresados.includes("typescript") ? resolve(true) : resolve(false);
86         });
87     }
88
89     console.log(retornarLongitud(["elementoA", "elementoB", "elementoC"]));
90     console.table(retornarPlanetas());
91     procesarDatos("node react typescript javascript vue").then(console.log);

```

La Arrow Function `retornarLongitud()` estará disponible para recibir únicamente arreglos cuyos valores contenidos sean cadenas de caracteres. Como resultado, va a retornar el número de elementos contados. En el caso de `retornarPlanetas()`, estamos indicando que se retornará un arreglo de objetos. Por último, `procesarDatos()` aceptará un objeto de tipo `string` y devolverá un valor

booleano. Detalle: es asíncrona porque internamente trabaja con promesas.

Yo realizaré de ahora en más las pruebas con Node.JS 16.17.0 en Linux Mint 20.3 Una. Info: <https://nodejs.org/es/> - <https://www.linuxmint.com/download.php>

Para ver los `console.log()`, es necesario tener ó nodemon que es una librería para ejecutar scripts y ver en tiempo real los cambios que apliquemos en nuestro código ó ts-node que es una dependencia la cual tiene la característica de evitar precompilar para ejecutar código TS en Node.JS. Si no tenés instaladas ninguna de las dos, con el gestor de paquetes NPM ejecutá cualquiera de los siguientes comandos (ó ambos): `npm install -g nodemon` / `npm install -g ts-node`

El flag `-g` significa que se va a instalar a nivel global. Para trabajar en un único proyecto, no hace falta indicarlo.

Información adicional:

> <https://www.npmjs.com/package/nodemon>

> <https://www.npmjs.com/package/ts-node>

Ejecución con el primer recurso: `nodemon <nombre_del_archivo>.ts`

Ejecución con el segundo recurso: `ts-node <nombre_del_archivo>.ts`

Ahora si, veamos el resultado del último código en la terminal:

```
3
```

(index)	primer_planeta	segundo_planeta	tercer_planeta	cuarto_planeta	quinto_planeta	sexto_planeta	septimo_planeta	octavo_planeta	noveno_planeta
0	'Mercurio'	'Venus'	'Tierra'	'Marte'	'Júpiter'	'Saturno'	'Urano'	'Neptuno'	'Plutón'

```
true
```

## Primer Proyecto



En cualquier sección donde puedas hacer el siguiente trabajo, creá un directorio llamado “CHAT\_TS”. Allí adentro, crearemos un package.json con `npm init -y` o podes copiarlo tal cual figura acá:

```
1  {
2      "name": "CHAT_TS",
3      "version": "1.0.0",
4      "description": "Chat TCP con TS",
5      "main": "index.js",
6      "scripts": {
7          "start": "nodemon ./src/app.ts"
8      },
9      "author": "Martín Castagnolo",
10     "license": "ISC",
11     "dependencies": {
12         "@typescript-eslint/eslint-plugin": "^5.9.0",
13         "@typescript-eslint/parser": "^5.9.0",
14         "esbuild": "^0.15.7",
15         "eslint": "^8.6.0",
16         "eslint-config-prettier": "^8.3.0",
17         "eslint-config-standard": "^14.1.1",
18         "eslint-plugin-import": "^2.22.0",
19         "eslint-plugin-node": "^11.1.0",
20         "eslint-plugin-prettier": "^4.0.0",
21         "eslint-plugin-promise": "^4.2.1",
22         "eslint-plugin-react": "^7.20.6",
23         "eslint-plugin-standard": "^4.0.1",
24         "prettier": "^2.5.1",
25         "chalk": "^4.1.2"
26     }
27 }
```

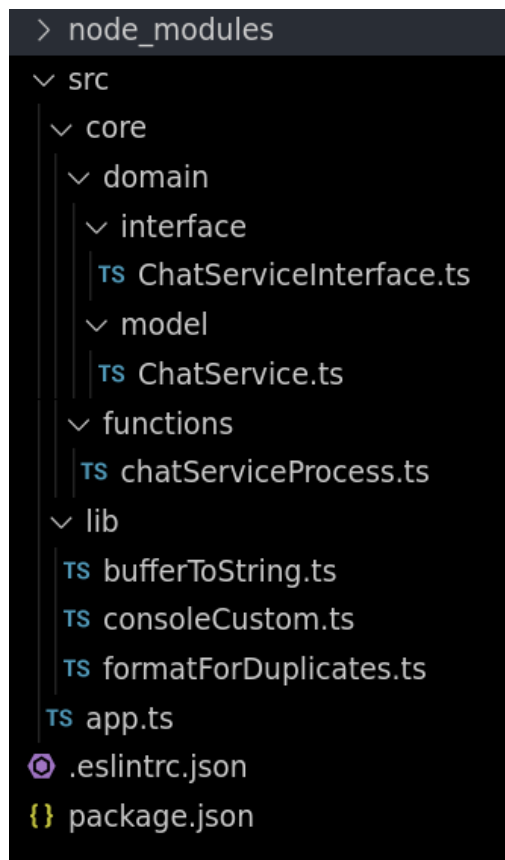
Las dependencias indicadas sirven para asegurarnos de que no existan problemas en nuestro código. Abarcando cuestiones de patrones, sintaxis, entre otras cosas.

Lo que haremos ahora, es crear un archivo oculto llamado `.eslintrc.json` a la misma altura del fichero anterior para especificar reglas. No detallaremos nada en especial, pero podés investigar más al respecto en otro momento.

```
1  {
2      "env": {
3          "browser": true,
4          "es2021": true
5      },
6      "extends": [
7          "eslint:recommended",
8          "plugin:@typescript-eslint/recommended",
9          "plugin:prettier/recommended"
10     ],
11     "parser": "@typescript-eslint/parser",
12     "parserOptions": {
13         "ecmaVersion": 13,
14         "sourceType": "module"
15     },
16     "plugins": [
17         "@typescript-eslint"
18     ],
19     "rules": {
20     }
21 }
```

Tip: si vas a trabajar con Visual Studio Code, existe una configuración para hacer que cuando guardes los cambios de tu código, se corrijan de forma automática los errores de sintaxis.

Procedemos a instalar las dependencias con `npm install` y creamos la siguiente estructura de directorios y ficheros:



El punto de entrada para este proyecto es app.ts. Colocamos allí:

```
1 import chatService from "../core/functions/chatServiceProcess";
2 chatService();
```

Definimos la interfaz para el modelo:

```
1 export default interface ChatSeviceInterface {
2     minimumNumberOfUsers: number;
3     maximumNumberOfUsers: number;
4     usernames: string[];
5     userList: string[] | object;
6     offlineUsers: string[];
7     sockets: string[] | any;
```

```

8      port: number;
9  }

```

Este servicio tendrá un mínimo y máximo de usuarios(as) permitido; se almacenarán los nombres ingresados; se mostrará quienes se hayan desconectado de la sala; se manejará un dato tipo cliente el cual contendrá sockets y especificaremos un puerto.

Ahora, el modelo:

```

1  import net from "net";
2  import ChatServiceInterface from "../interface/ChatServiceInterface";
3  import consoleCustom from "../lib/consoleCustom";
4  import bufferToString from "../lib/bufferToString";
5  import formatForDuplicates from "../lib/formatForDuplicates";

```

Lo primero que haremos es importar los recursos necesarios para que el servidor pueda trabajar correctamente. Importamos el módulo nativo NET, el cual nos permitirá crear una red TCP (Transfer Control Protocol) mediante una API. Incorporaremos ChatServiceInterface para su posterior implementación. Traemos una función que personalizará nuestros console.log(). Invocamos también otra función que convertirá los datos ingresados a través de process.stdin en strings. En instantes estaremos viendo esto con detenimiento. Por último, traemos una función que formateará aquellos nombres de usuarios(as) que estén repetidos.

```

7  export default class ChatService implements ChatServiceInterface { }

```

Creamos nuestra clase ChatService y de paso hacemos dos cosas: implementamos la interfaz correspondiente y exportamos por default para utilizarla en un controlador.

Indicamos las propiedades implementadas:

```
7   export default class ChatService implements ChatServiceInterface {  
8       minimumNumberOfUsers: number;  
9       maximumNumberOfUsers: number;  
10      usernames: string[];  
11      userList: string[] | object;  
12      offlineUsers: string[];  
13      sockets: string[] | any;  
14      port: number;  
16      constructor() {  
17          this.minimumNumberOfUsers = 2;  
18          this.maximumNumberOfUsers = 5;  
19          this.usernames = [];  
20          this.userList = [];  
21          this.offlineUsers = [];  
22          this.sockets = [];  
23          this.port = 4000;  
24      }  
25  }
```

Definimos dentro del constructor valores por default. Podrían ser declarados en un archivo de variables de entorno pero los vamos a dejar hardcodeados.

Ahora crearemos un método público el cual va a trabajar con `process.stdout` para imprimir un breve mensaje que dirá el mínimo y máximo de usuarios(as) permitidos, una instrucción de cómo añadir a alguien a la sala y la forma en la que se deben conectar. Info sobre process: <https://nodejs.org/api/process.html>



```

7      export default class ChatService implements ChatServiceInterface {
8
9          minimumNumberOfUsers: number;
10         maximumNumberOfUsers: number;
11
12         usernames: string[];
13         userList: string[] | object;
14         offlineUsers: string[];
15         sockets: string[] | any;
16         port: number;
17
18         constructor() {
19             this.minimumNumberOfUsers = 2;
20             this.maximumNumberOfUsers = 5;
21             this.usernames = [];
22             this.userList = [];
23             this.offlineUsers = [];
24             this.sockets = [];
25             this.port = 4000;
26         }
27
28         start(): void {
29             process.stdout.write(`▶ Allowed ${this.minimumNumberOfUsers} to $
30             {this.maximumNumberOfUsers} users. \n▶ Write: username + Intro. \n▶ Confirm: Press two times Intro.\n▶ Run
31             sudo nc localhost {port_number} \n\n`);
32
33             process.stdin.on("data", (data: string): void => {
34
35                 const incomingUsername: string = bufferToString(data);
36
37                 this.usernames.push(incomingUsername);
38
39                 if (incomingUsername === "") {

```

```

33         if (this.usernames.length === this.minimumNumberOfUsers) {
34             process.exit();
35         } else {
36             this.usernames.pop();
37             this.configForDuplicates();
38         }
39     }
40 });
41 }
42 }

```

Luego de imprimir el mensaje, lo siguiente que sucederá es que el servidor se quedará esperando a que a través de la terminal se ingresen los datos requeridos. Por lo que cuando reciba dichos datos, serán convertidos a cadena de caracteres. Si hacés `console.log()` de `data`, verás algo como: “<Buffer 6d 61 72 74 69 6e 20 63 61 73 74 61 67 6e 6f 6c 6f 0a>”.

Ahora bien, en el caso de que el número de usuarios(as) sea inferior al mínimo establecido, el proceso será dado de baja. Sino, se elimina el último elemento del arreglo que contiene los nombres, porque por defecto el programa deja un string vacío y se pasa a chequear con el método `configForDuplicates()` los datos repetidos.

Al ser un método que no retorna nada, especificamos que es de tipo `void`.

```

7     export default class ChatService implements ChatServiceInterface {
8         minimumNumberOfUsers: number;
9         maximumNumberOfUsers: number;
10        usernames: string[];

```

```

11     userList: string[] | object;
12
13     offlineUsers: string[];
14
15     sockets: string[] | any;
16
17     port: number;
18
19     constructor() {
20
21         this.minimumNumberOfUsers = 2;
22
23         this.maximumNumberOfUsers = 5;
24
25         this.usernames = [];
26
27         this.userList = [];
28
29         this.offlineUsers = [];
30
31         this.sockets = [];
32
33         this.port = 4000;
34     }
35
36     start(): void {
37
38         process.stdout.write(`▶ Allowed ${this.minimumNumberOfUsers} to $
39 {this.maximumNumberOfUsers} users. \n▶ Write: username + Intro. \n▶ Confirm: Press two times Intro.\n▶ Run
40 sudo nc localhost {port_number} \n\n`);
41
42         process.stdin.on("data", (data: string): void => {
43
44             const incomingUsername: string = bufferToString(data);
45
46             this.usernames.push(incomingUsername);
47
48             if (incomingUsername === "") {
49
50                 if (this.usernames.length === this.minimumNumberOfUsers) {
51
52                     process.exit();
53
54                 } else {
55
56                     this.usernames.pop();
57                 }
58             }
59         });
60     }
61 }

```

```

37         this.configForDuplicates();
38     }
39 }
40 });
41 }
42
43 private configForDuplicates(): void {
44     this.userList = formatForDuplicates(this.usernames);
45     this.chat();
46 }
47 }

```

Terminado el formateo para los datos duplicados, se invoca al método chat().

```

7     export default class ChatService implements ChatServiceInterface {
8         minimumNumberOfUsers: number;
9         maximumNumberOfUsers: number;
10        usernames: string[];
11        userList: string[] | object;
12        offlineUsers: string[];
13        sockets: string[] | any;
14        port: number;
15
16        constructor() {
17            this.minimumNumberOfUsers = 2;
18            this.maximumNumberOfUsers = 5;
19            this.usernames = [];
20            this.userList = [];
21            this.offlineUsers = [];

```

```

22         this.sockets = [];
23         this.port = 4000;
24     }

26     start(): void {
27         process.stdout.write(`▶Allowed ${this.minimumNumberOfUsers} to $
{this.maximumNumberOfUsers} users. \n▶ Write: username + Intro. \n▶ Confirm: Press two times Intro.\n▶ Run
sudo nc localhost {port_number} \n\n`);
28         process.stdin.on("data", (data: string): void => {
29             const incomingUsername: string = bufferToString(data);
30             this.usernames.push(incomingUsername);

32             if (incomingUsername === "") {
33                 if (this.usernames.length === this.minimumNumberOfUsers) {
34                     process.exit();
35                 } else {
36                     this.usernames.pop();
37                     this.configForDuplicates();
38                 }
39             }
40         });
41     }

43     private configForDuplicates(): void {
44         this.userList = formatForDuplicates(this.usernames);
45         this.chat();
46     }

48     private chat(): void {

```

```
49         const list: string[] | any = this.userList;
50
51         net
52             .createServer()
53             .on("connection", (client: any): void => {
54
55                 this.sockets.push(client);
56
57                 if (list.length === 0) {
58                     consoleCustom.error("Error: room is full");
59                     this.sockets.pop();
60                     client.destroy();
61                 }
62
63                 if (list.length !== 0) {
64                     client.username = list.shift();
65                     consoleCustom.ok(`${client.username} is online`);
66                     client.on("data", (data: any): void => {
67                         for (let i = 0; i < this.sockets.length; i++) {
68                             if (this.sockets[i] === client) continue;
69                             this.sockets[i].write(`${client.username}: ${data}`);
70                         }
71                     });
72                 }
73
74                 client.on("end", (): void => {
75                     consoleCustom.error(`${client.username} is offline`);
76                     this.offlineUsers.push(client.username);
77                     if (this.offlineUsers.length === this.sockets.length) {
78                         consoleCustom.error("\nfinished service");
79                     }
80                 });
81             });
82     }
83 }
```

```

76         process.exit();
77     }
78 });
79 })
80 .listen(this.port, () => {
81     process.stdin.pause();
82     consoleCustom.ok(`Room ${this.port} open`);
83     consoleCustom.warning("press ctrl+c to exit");
84 });
85 }
86 }

```

Vayamos despacio:

- Se define una lista de usuarios con los nombres ingresados.
- Creamos un servidor con NET.
- Se levanta la conexión para que ingresen nuevos clientes.
- Cuando haya un nuevo cliente, será cargado en memoria.
- Si la lista de usuarios(as) está vacía porque han sido procesados y están conectados(as), y alguien intenta sumarse, se indicará un error, su cliente generado será removido y eliminado.
- En el caso de que no esté vacía el listado de usuarios(as) porque por default el arreglo tiene un string vacío, cuando se conecte alguien, se capturará el nombre e indicará un mensaje de que está en línea.
- Queda en espera el ingreso de datos. Al momento de cargar un mensaje, en la interfaz de los(as) demás se verá reflejado el nombre de quien lo envió más

la conversación.

- Al cerrar sesión, se indica en el servidor que X usuario(a) se ha desconectado.
- Si la lista de usuarios(as) desconectados(as) tiene la misma longitud de clientes, o sea  $0 \equiv 0$ , se finaliza el proceso de escucha y el servidor es dado de baja.
- Establecemos la escucha en el puerto que habíamos especificado en el constructor.
- Indicamos que no se admita el ingreso de nuevos datos en la terminal del servidor una vez levantada la sala.
- Y por último señalamos cómo se sale de la sala.

Pasemos ahora al fichero chatServiceProcess.ts que será nuestro controlador:

```
1 import ChatService from "../domain/model/ChatService";
2 const service = new ChatService();
3 export default () => service.start();
```

Simplemente se invocan las funcionalidades necesarias del modelo que hemos creado.

En la función bufferToString(), que se encuentra en lib colocamos:

```
1 export default (data: string) =>
2     data.toString().replace(/\s+/g, "").toLowerCase();
```

En consoleCustom() las instrucciones serán:

```
1 import chalk from "chalk";
2 export default {
3     ok(message: string): void {
4         console.log(chalk.bold.green(message));
```



```

5      },
6      error(message: string): void {
7          console.log(chalk.bold.red(message));
8      },
9      warning(message: string): void {
10         console.log(chalk.bold.white.bgRedBright("\n", message, "\n"));
11     },
12 };

```

Primero hacemos uso de chalk. Si te fijaste en el package.json, hemos indicado también dicha dependencia. Esta librería permite añadirle colores a los mensajes que se impriman por consola. Segundo, definimos tres shorthand method definition en un objeto y las exportamos.

> Más info sobre esta librería: <https://www.npmjs.com/package/chalk>

Finalmente, formatForDuplicates():

```

1  export default (usernames: string[]) => {
2      const counter: any = {};
3      return usernames.map((name: string): string => {
4          if (!counter[name]) {
5              counter[name] = 0;
6          }
7          counter[name]++;
8          return counter[name] > 1 ? `${name}${counter[name]}` : name;
9      });
10 };

```

Acá detallamos que si existe una coincidencia entre dos ó más strings, le

va a asignar de forma autoincremental un número desde 1 al nombre repetido.

¿Y ahora? Si seguiste al pie de la letra todos los pasos, puedes ir a una nueva terminal y ejecutar `npm start` en la raíz del proyecto. Luego de levantar el servidor, en base a la cantidad de usuarios(as) ficticios que cargues, abrirás el mismo número de terminales aparte. Yo por ejemplo, voy a cargar cuatro.

```

ts/CHAT_TS → npm start
> CHAT_TS@1.0.0 start
> nodemon ./src/app.ts

[nodemon] 2.0.19
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: ts,json
[nodemon] starting `ts-node ./src/app.ts`
> Allowed 2 to 5 users.
> Write: username + Intro.
> Confirm: Press two times Intro.
> Run sudo nc localhost {port_number}

Martyn
Nadia
Luciano
Sofia

Room 4000 open

press ctrl+c to exit

martyn is online
nadia is online
luciano is online
sofia is online

[0:45:53] martyn:CHAT_TS $ sudo nc localhost 4000
Hola! 🙋
luciano: holis, cómo están??
sofia: acá estamos, luchando con una API hecha con PHP 😄😄😄
nadia: jajajajaj pasate a Node y TS, no seas amarga 😊
luciano: Nadia tiene la posta 😊

→ CHAT_TS sudo nc localhost 4000
martyn: Hola! 🙋
luciano: holis, cómo están??
sofia: acá estamos, luchando con una API hecha con PHP 😄😄😄
jajajajaj pasate a Node y TS, no seas amarga 😊
luciano: Nadia tiene la posta 😊

CHAT_TS > sudo nc localhost 4000
martyn: Hola! 🙋
holis, cómo están??
sofia: acá estamos, luchando con una API hecha con PHP 😄😄😄
nadia: jajajajaj pasate a Node y TS, no seas amarga 😊
Nadia tiene la posta 😊

10019 ts/CHAT_TS » sudo nc localhost 4000
martyn: Hola! 🙋
luciano: holis, cómo están??
acá estamos, luchando con una API hecha con PHP 😄😄😄
nadia: jajajajaj pasate a Node y TS, no seas amarga 😊
luciano: Nadia tiene la posta 😊

```

Si desconecto a cualquier cliente, veremos:

```

press ctrl+c to exit

martyn is online
nadia is online
luciano is online
sofia is online
luciano is offline

```

En el caso de que se den de baja todos(as):

```
luciano is offline  
martyn is offline  
sofia is offline  
nadia is offline  
  
finished service
```

Detalle: al estar ejecutándose con nodemon, si el servicio es finalizado puedes escribir rs para reiniciarlo.

¡Genial! Ahora con este gran ejemplo, sentite libre de modificarlo, de añadirle o quitarle cosas, de hacerle una mejor configuración, lo que te parezca. La finalidad de este primer trabajo es que veas cómo se aplica TS en conjunto con JS.

Link del repositorio: [https://github.com/MartynLCD3/CHAT\\_TS](https://github.com/MartynLCD3/CHAT_TS)

---

¿Se puede aplicar TS en un proyecto React? Sí, por supuesto. ¿Vue? Obvio. ¿Y en Angular? Pues TypeScript es el lenguaje principal de dicha herramienta. ¿Puedo aplicar esto en frameworks de Node? Sep. De hecho, Nest.JS trabaja 100% con TS. Conclusión hasta el momento: si hay JS, puede haber TS. Como se dijo al comienzo de todo.

¿Dónde puedo averiguar más sobre este tema? ¡Me encanta!

> <https://www.typescriptlang.org/docs/>

## Continuemos



Hemos visto diferentes tipos de datos y sus implementaciones. Pero... ¿es todo? Nop. Podemos hacer que nuestro código sea más complejo al definir datos genéricos.

En términos amigables, un tipo de dato genérico es la forma de inyectar parámetros dentro de definiciones de tipos. Lo que nos va a permitir esto, es tener un mayor control de los datos que se manejen según el escenario planteado.

Supongamos que creamos un server el cual va a responder un mensaje u otra cosa de acuerdo a los resultados que obtuvo tras consultar si hay datos o no en una base. Por lo que definimos las siguientes interfaces:

```
1  interface RespuestaInterface<MensajeGenerico> {  
2      titulo: string;  
3      codigo: number;  
4      operacion: string;  
5      mensaje: MensajeGenerico;  
6  }  
8  interface DatosInterface {  
9      nombre: string;  
10     edad: number;  
11     cliente: boolean;  
12 }
```

Ese mensaje puede ser cualquier cosa. Podría ser un número, una cadena de caracteres, un objeto personalizado, lo que fuere. Acá es donde MensajeGenerico toma el rol protagonista.

Imaginemos ahora que la primer respuesta va a ser un intento fallido de la obtención de datos:

```
15  const primerRespuesta: RespuestaInterface<string> = {
16      titulo: "obtención de datos",
17      codigo: 404,
18      operacion: "GET",
19      mensaje: "no hay datos registrados"
20  };
```

Como se puede observar, por parámetro indicamos que el mensaje para este caso en particular va a ser de tipo string. Pero en otro, podría ser un arreglo de objetos:

```
22  const segundaRespuesta: RespuestaInterface<DatosInterface[]> = {
23      titulo: "obtención de datos",
24      codigo: 200,
25      operacion: "GET",
26      mensaje: [{
27          nombre: "Martín",
28          edad: 26,
29          cliente: false
30      }, {
31          nombre: "Daniela",
32          edad: 34,
33          cliente: true
34      }]
35  };
```

Notar que ahora se ha indicado que los objetos respetan la estructura de

DatosInterface y que a su vez estarán dentro de un arreglo.

Podemos declarar que una propiedad de una interfaz sea opcional con el signo ? y aplicar otro argumento genérico:

```
1  interface RespuestaInterface<MensajeGenerico, FechaGenerica> {  
2      titulo: string;  
3      codigo: number;  
4      operacion: string;  
5      mensaje: MensajeGenerico;  
6      fecha?: FechaGenerica;  
7  }
```

De esta forma, nos veremos con el compromiso de si o si especificar también qué tipo de dato será el que defina al segundo:

```
16  const primerRespuesta: RespuestaInterface<string, null> = {  
17      titulo: "obtención de datos",  
18      codigo: 404,  
19      operacion: "GET",  
20      mensaje: "no hay datos registrados"  
21  };  
23  const segundaRespuesta: RespuestaInterface<DatosInterface[], string> = {  
24      titulo: "obtención de datos",  
25      codigo: 200,  
26      operacion: "GET",  
27      mensaje: [{  
28          nombre: "Martín",  
29          edad: 26,
```

```
30         cliente: false
31     }, {
32         nombre: "Daniela",
33         edad: 34,
34         cliente: true
35     }],
36     fecha: `${new Date()}`
37 };
```

Tener en cuenta que se ha señalado en el primer caso un null porque no asignaremos nada ya que no hemos incluido la propiedad fecha. Y en segundo ejemplo, fecha tomará como valor un dato de tipo string.

Lo ideal es que cuando usemos datos de genéricos, empleemos nombres descriptivos.

Ahora veamos, algo más específico: Genéricos en funciones.

Se puede definir distintos tipos, extenderlos y hacer uso de ellos de forma genérica dentro de una función. Por ejemplo, estamos desarrollando un mánager de publicaciones para un foro que tendrá las siguientes características:

1. Entrarán notas.
2. Entrarán notas con atributos.
3. Entrarán fotos (sin la nota).
4. Entrarán videos (sin la nota).
5. Entrarán notas con todo lo anterior dicho más la fecha de publicación.

Podemos encararlo así:



```

1    type Nota = { cuerpo: string };
2    type NotaColorida = Nota & { color: string };
3    type Foto = { urlFoto: string };
4    type Video = { urlVideo: string, duracion: number };
5    type NotaCompleta = Nota & NotaColorida & Foto & Video & { fecha: string };

```

Entonces, una función que denominaremos `postearPublicacion()` podrá recibir cualquiera de los casos anteriores y lo asociaremos a una definición genérica que la denominaremos `Publicacion`. Esta a su vez, retornará lo ingresado.

```

7    const postearPublicacion = <Publicacion>(publicacionEntrante: Publicacion): Publicacion => {
8        return publicacionEntrante;
9    };

```

Hagamos las pruebas:

```

11   const primerPosteo: NotaColorida = { cuerpo: "Hey! ¿aprendiendo TS?", color: "orange" };
12   const segundoPosteo: Foto = { urlFoto: "http://foto_de_perfil.jpg" };
13   const tercerPosteo: Nota = { cuerpo: "En el día de la fecha estamos viendo TS" };
14   const cuartoPosteo: Video = { urlVideo: "http://video_de_perfil.mp4", duracion: 5 };
15   const quintoPosteo: NotaCompleta = {
16       cuerpo: "Nada más lindo que programar",
17       color: "green",
18       urlFoto: "http://alguna_foto.jpg",
19       urlVideo: "http://algun_video.mp4",
20       duracion: 16,
21       fecha: `${new Date()}`
22   };
23   console.log(postearPublicacion(primerPosteo));

```

```

24 console.log(postearPublicacion(segundoPosteo));
25 console.log(postearPublicacion(tercerPosteo));
26 console.log(postearPublicacion(cuartoPosteo));
27 console.log(postearPublicacion(quintoPosteo));

```

```

{ cuerpo: 'Hey! ¿aprendiendo TS?', color: 'orange' }
{ urlFoto: 'http://foto_de_perfil.jpg' }
{ cuerpo: 'En el día de la fecha estamos viendo TS' }
{ urlVideo: 'http://video_de_perfil.mp4', duracion: 5 }
{
  cuerpo: 'Nada más lindo que programar',
  color: 'green',
  urlFoto: 'http://alguna_foto.jpg',
  urlVideo: 'http://algun_video.mp4',
  duracion: 16,
  fecha: 'Sun Oct 02 2022 20:11:35 GMT-0300 (hora estándar de Argentina)'
}

```

Y ahora habría que preguntarse algo: ¿cómo podemos limitar los tipos que se pueden especificar en un genérico? Porque de cierta manera podemos pasarle a `postearPublicacion()` cualquier cosa. Fijate: `console.log(postearPublicacion(null));`

La solución para esto podría ser la definición de una interfaz principal de la cual `Nota`, `NotaColorida`, `Foto`, `Video`, `NotaCompleta` extiendan. Por lo que estas también tranquilamente las podríamos definir como interfaces. Entonces, crearemos una restricción que indique que el objeto entrante tiene que tener un ID de tipo numérico.

```

1 interface Post {
2     id: number;
3 }
4
5 interface Nota extends Post {
6     cuerpo: string;
7 }

```

```

9    interface NotaColorida extends Nota {
10        color: string;
11    }
12
13    interface Foto extends Post {
14        urlFoto: string;
15    }
16
17    interface Video extends Post {
18        urlVideo: string;
19        duracion: number;
20    }
21
22    interface NotaCompleta extends Post, Nota, NotaColorida, Foto, Video {
23        fecha: string;
24    }
25
26    const postearPublicacion = <Publicacion extends Post>(publicacionEntrante: Publicacion): Publicacion => {
27        return publicacionEntrante;
28    };
29
30    const primerPosteo: Nota = { id: 1, cuerpo: "Hey! ¿sigues aquí?" };
31    const segundoPosteo: Nota = { cuerpo: "esto va a explotar" };
32    console.log(postearPublicacion(segundoPosteo));

```

```

/usr/lib/node_modules/ts-node/src/index.ts:859
    return new TSError(diagnosticText, diagnosticCodes, diagnostics);
           ^
TSError: x Unable to compile TypeScript:
application.ts:31:7 - error TS2741: Property 'id' is missing in type '{ cuerpo: string; }' but required in type 'Nota'.

31 const segundoPosteo: Nota = { cuerpo: "esto va a explotar" };
   ~~~~~

application.ts:2:2
   2 id: number;
     ~~~
   'id' is declared here.

```

## Segundo Proyecto



Este segundo y último proyecto será un tanto más complejo. Desafiante te diría... Vamos a enfocarnos en realizar un microservicio; un sistema que se encargará de almacenar y brindar información de los productos que se carguen desde un mánager.

Tecnologías adicionales que utilizaremos:

- > MongoDB
- > Docker
- > docker-compose
- > AWS Sam CLI

Requisitos:

Instalar Docker en tu equipo > <https://www.docker.com/>

Instalar docker-compose > <https://docs.docker.com/compose/install/>

Instalar AWS Sam CLI > <https://docs.aws.amazon.com/serverless-application-model/latest/developerguide/serverless-sam-cli-install.html>

¿Para qué Docker? Para virtualizar nuestro servicio y hacerlo correr, comunicarse y desplegarse dentro de un contenedor. A su vez, se comunicará con su propia base de datos la cual sera vinculada mediante una configuración. Dicha base, o sea MongoDB, no hará falta que la instalemos en nuestro equipo.

¿Por qué SAM CLI? Porque haremos uso de un entorno que emplea funciones lambda. Detalle de color.

Te dejo a mano lo que deberías ver en tu terminal para poder continuar (las versiones pueden variar según cuando estés haciendo esta guía):

```

~ > docker --version
Docker version 20.10.12, build 20.10.12-0ubuntu2~20.04.1
~ > docker-compose --version
docker-compose version 1.26.0, build d4451659
~ > sam --version
SAM CLI, version 1.55.0

```

¿Arrancamos? ¿Ya instalaste todo lo necesario? ¡Vamos!

Lo primero que haremos es crear un nuevo directorio llamado “Stock\_Manager” e ingresaremos para cargar nuestro archivo de configuración llamado `template.yaml` para que las funciones lambda puedan ser levantadas. ¡Ojo con las tabulaciones!

```

1  AWSTemplateFormatVersion: "2010-09-09"
2  Transform: AWS::Serverless-2016-10-31
3  Description: >
4      stock_manager
6      Sample SAM Template for stock_manager

```

Estas instrucciones son una breve descripción.

Definimos nuestras variables de entorno:

```

8  Globals:
9      Function:
10         Environment:
11             Variables:
12                 SAM_CLI_BETA_ESBUILD: 1
13                 MONGODB_HOST: 192.168.43.122
14                 MONGODB_PORT: "27017"
15                 MONGODB_USER: "admin"

```

```

16             MONGODB_PASSWORD: "P4zsW0rD"
17             MONGODB_DBNAME: "stock_manager"
18             MONGODB_COLLNAME: "products"
19             MONGODB_RESOURCE: "mongodb"
20         Timeout: 180
21         Runtime: nodejs16.x
22         MemorySize: 2048

```

En la propiedad MONGODB\_HOST irá la dirección IP de tú equipo. En la terminal si ponés el comando ifconfig te va a salir. Esto porque estaremos trabajando en un entorno virtualizado.

Especificaremos los recursos. Es decir, definiremos nuestros puntos de entrada, los métodos y el tipo de lambda. Adicional, señalamos que trabajaremos con TS :D

```

24     Resources:
25         SaveProductFunction:
26             Type: AWS::Serverless::Function
27             Properties:
28                 CodeUri: src/save/
29                 Handler: app.saveHandler
30                 Runtime: nodejs16.x
31                 Architectures:
32                     - x86_64
33                 Events:
34                     Api:
35                         Type: Api
36                 Properties:

```

```

37                                     Path: /api/stock-manager
38                                     Method: POST
39     Metadata:
40         BuildMethod: esbuild
41     BuildProperties:
42         Minify: false
43         Target: 'es2020'
44         Sourcemap: true
45         UseNpmCi: true
47     ReadProductsFunction:
48         Type: AWS::Serverless::Function
49     Properties:
50         CodeUri: src/read/
51         Handler: app.readHandler
52         Runtime: nodejs16.x
53     Architectures:
54         - x86_64
55     Events:
56         Api:
57             Type: Api
58             Properties:
59                 Path: /api/stock-manager
60                 Method: GET
61     Metadata:
62         BuildMethod: esbuild

```



```

63         BuildProperties:
64             Minify: false
65             Target: 'es2020'
66             Sourcemap: true
67             UseNpmCi: true
68
69     DeleteProductFunction:
70         Type: AWS::Serverless::Function
71         Properties:
72             CodeUri: src/delete/
73             Handler: app.deleteHandler
74             Runtime: nodejs16.x
75             Architectures:
76                 - x86_64
77             Events:
78                 Api:
79                     Type: Api
80                     Properties:
81                         Path: /api/stock-manager/{id}
82                         Method: DELETE
83             Metadata:
84                 BuildMethod: esbuild
85                 BuildProperties:
86                     Minify: false
87                     Target: 'es2020'
88                     Sourcemap: true

```

```

89         UseNpmCi: true

91     UpdateProductFunction:

92         Type: AWS::Serverless::Function

93         Properties:

94             CodeUri: src/update/

95             Handler: app.updateHandler

96             Runtime: nodejs16.x

97             Architectures:

98                 - x86_64

99             Events:

100                 Api:

101                     Type: Api

102                     Properties:

103                         Path: /api/stock-manager/{id}

104                         Method: PUT

105         Metadata:

106             BuildMethod: esbuild

107             BuildProperties:

108                 Minify: false

109                 Target: 'es2020'

110                 Sourcemap: true

111             UseNpmCi: true

```

Procedemos a establecer el package.json

```

1  {
2      "name": "Stock_Manager",
3      "version": "1.0.0",

```

```

4      "description": "Mánager para gestionar productos",
5      "main": "index.js",
6      "scripts": {
7          "start": "clear && sam build --cached --beta-features && sam local start-api --log-file
access.log --port 8080"
8      },
9      "author": "Martín Castagnolo",
10     "license": "ISC",
11     "dependencies": {
12         "@typescript-eslint/eslint-plugin": "^5.9.0",
13         "@typescript-eslint/parser": "^5.9.0",
14         "esbuild": "^0.15.7",
15         "eslint": "^8.6.0",
16         "eslint-config-prettier": "^8.3.0",
17         "eslint-config-standard": "^14.1.1",
18         "eslint-plugin-import": "^2.22.0",
19         "eslint-plugin-node": "^11.1.0",
20         "eslint-plugin-prettier": "^4.0.0",
21         "eslint-plugin-promise": "^4.2.1",
22         "eslint-plugin-react": "^7.20.6",
23         "eslint-plugin-standard": "^4.0.1",
24         "prettier": "^2.5.1",
25         "mongodb": "^4.10.0"
26     },
27     "devDependencies": {
28         "@types/aws-lambda": "^8.10.102"
29     }
30 }

```

Creamos un fichero docker-compose.yaml para levantar una imagen de MongoDB:

```

1  version: '2'
2  services:
3      mongodb:
4          image: 'mongo:5.0'
5          restart: always
6          volumes:

```

```

7             - './mongo_data:/data/db'
8         ports:
9             - '27017:27017'

```

Si ya tenés MongoDB en tu equipo o tenés una imagen de dicha base para utilizar, no hace falta que emplees esto. Podés obviar lo siguiente si es así.

Creamos un directorio a la misma altura del package.json, template.yaml y docker-compose.yaml que se va a llamar mongo\_data. Este dir contendrá la información generada por el contenedor.

Ahora, procedemos a descargar la imagen de MongoDB y simultáneamente la pondremos en marcha: `docker-compose up -d`

```

stock_manager > docker-compose up -d
Pulling mongodb (mongo:5.0)...
5.0: Pulling from library/mongo
675920708c8b: Downloading [=====] 12.9
MB/28.57MB0f: Download complete
73738965c4ce: Download complete
7fd6635b9ddf: Download complete
MB/6.506MBad: Download complete
5abd0e4db1c7: Download complete
90013e6cca25: Download complete
d88b6355fefb: Downloading [=] 4.31
c5f0e33dbefc: Download complete

```

Ejecutamos `docker ps` ver el ID del contenedor de MongoDB:

```

stock_manager > docker ps

```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
33aaa2a73f6c	mongo:5.0	"docker-entrypoint.s..."	39 seconds ago	Up 32 seconds	0.0.0.0:27017->27017/tcp, :::27017->27017/tcp	stock_manager_mongodb_1

Luego ponemos el container ID que nos figura en el comando `docker exec`:

```
> docker exec -it <container_id> sh
```

En mi caso sería:

```
stock_manager > docker exec -it 33aaa2a73f6c sh
#
```

Ponemos mongo en la línea de comandos del contenedor y tendrías que visualizar esto:

```
stock_manager > docker exec -it 33aaa2a73f6c sh
# mongo
MongoDB shell version v5.0.13
connecting to: mongodb://127.0.0.1:27017/?compressors=disabled&gssapiServiceName=mongodb
Implicit session: session { "id" : UUID("cb87a7d4-f855-4bf3-a609-17aca6fc6ae7") }
MongoDB server version: 5.0.13
=====
Warning: the "mongo" shell has been superseded by "mongosh",
which delivers improved usability and compatibility. The "mongo" shell has been deprecated and will be removed in
an upcoming release.
For installation instructions, see
https://docs.mongodb.com/mongodb-shell/install/
=====
---
The server generated these startup warnings when booting:
  2022-10-05T01:15:10.490+00:00: Using the XFS filesystem is strongly recommended with the WiredTiger storage engine
system
  2022-10-05T01:15:11.357+00:00: Access control is not enabled for the database. Read and write access to data and
---
---
  Enable MongoDB's free cloud-based monitoring service, which will then receive and display
  metrics about your deployment (disk utilization, CPU, operation statistics, etc).

  The monitoring data will be available on a MongoDB website with a unique URL accessible to you
  and anyone you share the URL with. MongoDB may use this information to make product
  improvements and to suggest MongoDB products and deployment options to you.

  To enable free monitoring, run the following command: db.enableFreeMonitoring()
  To permanently disable this reminder, run the following command: db.disableFreeMonitoring()
---
>
```

Estando ya en la línea de comandos de Mongo, procedemos a crear/acceder a nuestra base de datos la cual indicamos en el template.yaml → "stock\_manager":

```
> use stock_manager
switched to db stock_manager
>
```

Y creamos a nuestro usuario, con su contraseña y permisos:

```
> use stock_manager
switched to db stock_manager
> db.createUser({ user: "admin", pwd: "P4zsW0rD", roles: [] });
Successfully added user: { "user" : "admin", "roles" : [ ] }
> █
```

Listo. ¡Ya tenemos nuestra base de datos preparada para recibir consultas!

Como la terminal quedó bloqueada por lo que acabamos de hacer, abrimos una nueva y nos aseguramos de estar de nuevo en la raíz del proyecto, que de momento se verá así:

```
stock_manager > l
docker-compose.yaml  mongo_data  package.json  template.yaml
```

Creamos dos ficheros ocultos; el `.eslinttrc.json` que habíamos hecho en el primer proyecto:

```
1  {
2      "env": {
3          "browser": true,
4          "es2021": true
5      },
6      "extends": [
7          "eslint:recommended",
8          "plugin:@typescript-eslint/recommended",
9          "plugin:prettier/recommended"
10     ],
11     "parser": "@typescript-eslint/parser",
12     "parserOptions": {
13         "ecmaVersion": 13,
14         "sourceType": "module"
15     },
16     "plugins": [
17         "@typescript-eslint"
18     ],
19     "rules": {
20     }
21 }
```

Y un .gitignore el cual contendrá:

→ node\_modules/

→ .aws-sam

Perfecto. Ahora creamos un directorio llamado src el cual contendrá toooodo el código y pasamos a visualizar cómo debes tener el arbol de directorios hasta acá:

```
> mongo_data
> src
.eslintrc.json
.gitignore
docker-compose.yaml
package.json
template.yaml
```

Empezaremos con la primer lambda que tendrá la responsabilidad de guardar nuevos productos. Entonces, lo que haremos es crear dentro de src un directorio llamado save. Tal cual como lo definimos en la propiedad CodeURI del template.yaml. A su vez, dentro de dicho dir ponemos un nuevo fichero llamado app.ts:

```
> mongo_data
  > src / save
    TS app.ts
.eslintrc.json
.gitignore
docker-compose.yaml
package.json
template.yaml
```

En dicho fichero, procedemos a poner:

```
1 import { APIGatewayProxyEvent, APIGatewayProxyResult } from "aws-lambda";
2 let response: APIGatewayProxyResult | PromiseLike<APIGatewayProxyResult>;
3 import processProduct from "../src/core/functions/processProduct";
```

```
5   export async function saveHandler(  
6       event: APIGatewayProxyEvent  
7   ): Promise<APIGatewayProxyResult> {  
8       try {  
9           let body: null;  
  
11          if (typeof event.body === "string") {  
12              body = JSON.parse(event.body);  
13          } else {  
14              body = event.body;  
15          }  
  
17          const { code, data } = await processProduct(body);  
18          response = {  
19              statusCode: code,  
20              body: JSON.stringify(data),  
21          };  
22      } catch (err) {  
23          console.log(err);  
24      }  
25      return response;  
26  }
```

Explicación:

→ Traemos los recursos necesarios para trabajar con lambdas.

→ Definimos el tipo de respuesta señalando la configuración de las integraciones de proxy de Lambda. Esto con el fin de que se llame a una única función



del servidor.

→ Importamos un controlador.

→ Exportamos nuestra lambda la cual recibirá un objeto, será parseado y posteriormente procesado para retornar una respuesta y un status code según los resultados obtenidos.

A la misma altura de nuestro app.ts creamos otro directorio llamado src. Este va a ser único y solamente accedido por saveHandler(). Una vez hecho, dejamos la siguiente estructura definida:



Vamos en el siguiente orden de implementaciones: de core hasta lib.

Dentro de domain/interface creamos dos archivos. El primero se llamará ProductInterface.ts y el segundo ResponseInterface.ts. Procedemos a editar

ProductInterface.ts:

```
1  export default interface ProductInterface {  
2      name: string;  
3      price: string;  
4      stock: boolean;  
5      created: Date;  
6  }
```

ResponseInterface.ts:

```
1  export default interface ResponseInterface {  
2      code: number;  
3      data: string;  
4  }
```

Continuamos ahora con nuestro modelo, el cual se llamará product.ts y tendrá lo siguiente:

```
1  import ProductInterface from "../interface/ProductInterface";  
  
3  export default class Product implements ProductInterface {  
4      name: string;  
5      price: string;  
6      stock: boolean;  
7      created: Date;  
  
9      constructor(  
10         name: string | undefined = "",  
11         price: string | undefined = "",  
12         stock: boolean,
```

```
13         created: Date
14     ) {
15         this.assignValues(name, price, stock, created);
16     }
17
18     private assignValues(
19         name: string,
20         price: string,
21         stock: boolean,
22         created: Date
23     ) {
24         this.name = name;
25         this.price = price;
26         this.stock = stock;
27         this.created = created;
28     }
29
30     dataToObject(data: any) {
31         this.name = data.name;
32         this.price = data.price;
33         this.stock = data.stock;
34         this.created = data.created;
35     }
36
37     validateName() {
38         return this.name ? true : false;
39     }
```

```
41     validatePrice() {
42         return this.price ? true : false;
43     }
44
45     validateCreated() {
46         return this.created ? true : false;
47     }
48
49     validationProcess() {
50         const validation = [
51             this.validateName(),
52             this.validatePrice(),
53             this.validateCreated(),
54         ];
55         return validation.every((condition: boolean) => condition === true);
56     }
57
58     generateNewProduct() {
59         return {
60             name: this.name,
61             price: this.price,
62             stock: this.stock,
63             created: this.created,
64         };
65     }
66 }
```

Explicación:

→ Estamos haciendo una copia de lo que recibiremos desde afuera especificando qué datos ingresados y lo que se espera.

→ Validamos cada uno de los campos que nos interesa y en el caso de que alguno falle, `validation.every()` se lo avisará mediante un booleano al controlador.

Y ya que hablamos del controlador, pasamos a functions para crear `processProduct.ts`:

```
1   import { _saveProductOnMongoDB } from "../interactors";
2   import Product from "../domain/model/product";
3   import ResponseInterface from "../domain/interface/ResponseInterface";
4
5   export default async (data: any): Promise<ResponseInterface> => {
6       let response = { code: 404, data: "error to save" };
7       const product = new Product();
8       product.dataToObject(data);
9       const validData = product.validationProcess();
10      if (validData) {
11          const productSaved = await _saveProductOnMongoDB(product);
12          if (productSaved) {
13              response = { code: 201, data: `${productSaved}` };
14          }
15      }
16      return response;
17  };
```

Explicación:

→ Nuestro controlador importa un recurso para grabar el nuevo producto, el

modelo de lo que se espera para hacer la copia/validación y trae la interfaz de lo que va a ser la respuesta.

→ La variable `data` está definida con `any` porque realmente no sabemos lo ingresará desde afuera. Podemos ser estrictos, pero no es necesario en este caso.

→ En aspectos generales, validamos y si está todo OK pasamos a guardar el nuevo producto e indicar un mensaje. Líneas más adelante estaremos viendo qué es lo que retorna `productSaved`.

Pasamos a interactores y creamos dos archivos: `index.ts` `saveProduct.interactor.ts`

`index.ts`:

```
1 import saveProduct from "../saveProduct.interactor";
2 import MongoDB from "../datasource/mongo.datasource";
4 const mongoRepository = new MongoDB();
6 const _saveProductOnMongoDB = saveProduct(mongoRepository);
8 export { _saveProductOnMongoDB };
```

Explicación:

→ Importamos los recursos que vendrían a ser nuestro puente, por decirlo de forma sencilla, entre lo que ha sido procesado y las instrucciones a implementar para el guardado.

→ Exportamos la función que tendrá la responsabilidad de acceder a la implementación anteriormente mencionada.

`saveProduct.interactor.ts`:

```
1 import Product from "../domain/model/product";
```

```

2    import MongoRepository from "../repositories/mongo.repository";

4    export default (mongoRepository: MongoRepository) => {

5        return async (_product: Product): Promise<string | boolean> => {

6            return await mongoRepository.saveProductOnMongoDB(_product);

7        };

8    };

```

Explicación:

→ A través de esta inyección de dependencias estamos haciendo uso de un repositorio el cual se comunicará con un adaptador.

El repositorio, dentro del directorio repositories es mongo.repository.ts:

```

1    import Product from "../domain/model/product";

3    export default interface MongoRepository {

4        saveProductOnMongoDB(provider: Product): Promise<string | boolean>;

5    }

```

Con eso ya completado, pasamos a nuestro datasource el cual contendrá dos cosas: un directorio llamado manager y un fichero llamado mongo.datasource.ts

Primero editaremos mongo.datasource.ts:

```

1    import Product from "../core/domain/model/product";

2    import MongoRepository from "../core/repositories/mongo.repository";

3    import MongoDBManager from "../manager/products.mongodb";

5    export default class MongoDS implements MongoRepository {

6        async saveProductOnMongoDB(_newProduct: Product): Promise<string | boolean> {

7            try {

```

```

8         const process = new MongoDBManager();
9         const productsList = await process.save(_newProduct);
10        return productsList;
11    } catch {
12        return false;
13    }
14 }
15 }

```

Explicación:

→ A través de este datasource de Mongo, expondremos los métodos encargados de comunicarse con el manager.

→ En este caso, estamos señalando que un nuevo producto ingresará, que tendrá las características del modelo especificado y que se requiere su almacenamiento.

Entendiendo lo anterior, seguimos con el archivo que estará adentro de manager el cual se llamara: products.mongodb.ts:

```

1    import Database from "../lib/database";
2
3    export default class MongoDBManager {
4        private collectionName: any = process.env.MONGODB_COLLNAME;
5        private db = Database.getInstance();
6        private resource: any = process.env.MONGODB_RESOURCE;
7
8        public async save(product: any) {
9            try {
10                const client = this.db(this.resource);

```



```

11         const _id: any = await client.insertOne(this.collectionName, product);
12         return _id;
13     } catch {
14         return [];
15     }
16 }
17 }

```

Esta clase trabajará en conjunto con una librería diseñada para realizar queries hacia MongoDB. La definimos de la siguiente forma dentro de lib:

```

1  import { MongoClient } from "mongodb";

3  class MongoDB {

4      resourceType = "mongodb";

5      client: MongoClient;

6      dbName: any;

7      private static instance: { [k: string]: MongoDB } = {};

9      private constructor() {

10         try {

11             this.client = new MongoClient(

12                 `mongodb://${process.env.MONGODB_USER}:${process.env.MONGODB_PASSWORD}@${process.env.MONGODB_HOST}:${process.env.MONGODB_PORT}/${process.env.MONGODB_DBNAME}`

13             );

14         } catch (err) {

15             throw new Error(err);

16         }

```

```
17     }  
  
19     public static getInstance(_resName: string) {  
20         if (!MongoDB.instance[_resName]) {  
21             MongoDB.instance[_resName] = new MongoDB();  
22         }  
23         return MongoDB.instance[_resName];  
24     }  
  
26     async insertOne(collection: string, data: any) {  
27         const client = await this.client.connect();  
28         const response = await client  
29             .db(this.dbName)  
30             .collection(collection)  
31             .insertOne(data);  
33         return response.insertedId;  
34     }  
35 }  
  
36 export default MongoDB;
```

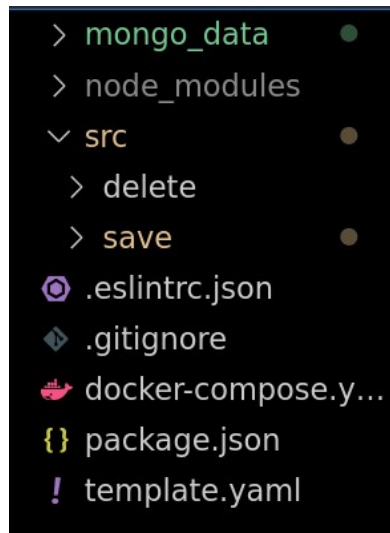
Explicación:

→ Creamos un nuevo cliente, le pasamos las credenciales definidas en el `template.yaml` y creamos un método el cual se encargará de insertar un nuevo elemento a la base de datos.

¡Fantástico! Ya tenemos nuestra primera lambda desarrollada con sus propios recursos. Si intentamos en este punto levantar el proyecto, tendremos un error al

buildear porque nos faltan definir 3 más. Por lo que seguimos con la responsable de borrar productos.

Creamos a la misma altura del directorio read un dir llamado delete.



Dentro de delete creamos la siguiente estructura:



Nuestra lambda recibirá por parámetro el ID del producto que sea indicado en la URI cuando hagamos un DELETE, entonces en app.ts:

```
1   import { APIGatewayProxyEvent, APIGatewayProxyResult } from "aws-lambda";
2   let response: APIGatewayProxyResult | PromiseLike<APIGatewayProxyResult>;
3   import processID from "../src/core/functions/processID";
4
5   export async function deleteHandler(
6       event: APIGatewayProxyEvent
7   ): Promise<APIGatewayProxyResult> {
8       try {
9           const param: any = event.pathParameters;
10          const { code, data } = await processID({ _id: param.id });
11          response = {
12              statusCode: code,
13              body: JSON.stringify(data),
14          };
15      } catch (err) {
16          console.log(err);
17      }
18      return response;
19  }
```

En el mismo orden como hicimos en el dir save, para detallar las instrucciones, arrancamos por nuestras interfaces:

```
1   export default interface ProdIdInterface {
2       _id: string | undefined;
3   }
```

ResponseInterface.ts:

```
1 export default interface ResponseInterface {  
2     code: number;  
3     data: string;  
4 }
```

El modelo, productID.ts será:

```
1 import ProdIdInterface from "../interface/ProdIdInterface";  
  
3 export default class ProductID implements ProdIdInterface {  
4     _id: string;  
  
6     constructor(_id: string | undefined = " ") {  
7         this.assignValues(_id);  
8     }  
  
10    private assignValues(_id: string) {  
11        this._id = _id;  
12    }  
  
14    dataToObject(data: any) {  
15        this._id = data._id;  
16    }  
  
18    validateID() {  
19        return this._id ? true : false;  
20    }  
  
22    generateID() {
```

```

23         return this._id;
24     }
25 }

```

El controlador, processID:

```

1     import {
2         _readProductFromMongoDB,
3         _deleteProductOnMongoDB,
4     } from "../interactors";
5     import ProductID from "../domain/model/productID";
6     import ResponseInterface from "../domain/interface/ResponseInterface";
7
8     export default async (data: any): Promise<ResponseInterface> => {
9         let response = { code: 404, data: "error to delete" };
10        const process = new ProductID();
11        process.dataToObject(data);
12        const _id: any = process.generateID();
13        const existingProduct = await _readProductFromMongoDB(_id);
14        if (existingProduct) {
15            const deleteProduct = await _deleteProductOnMongoDB(_id);
16            if (deleteProduct) {
17                response = { code: 200, data: "successful delete" };
18            }
19        }
20        return response;
21    };

```

Explicación:

→ La obligación que tiene processID() es la de recibir datos de afuera; evaluar sus propiedades y posteriormente comprobar si existe o no en la base de datos. En caso de ser afirmativo, procede a borrarlo. Sino, mensaje de error definido por defecto.

Pasamos a los interactores, comenzando por el index.ts:

```
1 import readProduct from "../readProduct.interactor";
2 import deleteProduct from "../deleteProduct.interactor";
3 import MongoDS from "../../datasource/mongo.datasource";
4
5 const mongoRepository = new MongoDS();
6
7 const _readProductFromMongoDB = readProduct(mongoRepository);
8 const _deleteProductOnMongoDB = deleteProduct(mongoRepository);
9
10 export { _readProductFromMongoDB, _deleteProductOnMongoDB };
```

Casi similar a lo realizado en el index.ts de save, pero la diferencia acá es que se definen las funcionalidades de lectura y borrado implementando el repositorio de Mongo.

Continuamos con readProduct.interactor.ts, a la misma altura del index.ts:

```
1 import MongoRepository from "../repositories/mongo.repository";
2
3 export default (mongoRepository: MongoRepository) => {
4     return async (_id: string): Promise<boolean> =>
5         await mongoRepository.readProductFromMongoDB(_id);
6     };
7 }
```

deleteProduct.interactor.ts:

```

1   import MongoRepository from "../repositories/mongo.repository";

3   export default (mongoRepository: MongoRepository) => {

4       return async (_id: string): Promise<boolean> =>

5           await mongoRepository.deleteProductOnMongoDB(_id);

6   };

```

El repositorio, en repositories llamado mongo.repository.ts:

```

1   export default interface MongoRepository {

2       readProductFromMongoDB(provider: string): Promise<boolean>;

3       deleteProductOnMongoDB(provider: string): Promise<boolean>;

4   }

```

Ahora es momento de hacer el datasource. Entonces ponemos al igual que save, un nuevo directorio llamado manager y a la misma altura mongo.datasource.ts:

```

1   import MongoRepository from "../core/repositories/mongo.repository";

2   import MongoDBManager from "../manager/products.mongodb";

4   export default class MongoDS implements MongoRepository {

5       async readProductFromMongoDB(_id: string): Promise<boolean> {

6           try {

7               const process = new MongoDBManager();

8               const product = await process.read(_id);

9               return product ? true : false;

10          } catch {

11              return false;

12          }

13      }

```



```

15     async deleteProductOnMongoDB(_id: string): Promise<boolean> {
16         try {
17             const process = new MongoDBManager();
18             const response = await process.delete(_id);
19             return response ? true : false;
20         } catch {
21             return false;
22         }
23     }
24 }

```

Este datasource lo que provee a diferencia del save, son justamente métodos expuestos para comunicarse con lo que viene de afuera con la finalidad de leer/borrar y el manager:

```

1     import Database from "../lib/database";
2
3     export default class MongoDBManager {
4         private collectionName: any = process.env.MONGODB_COLLNAME;
5         private db = Database.getInstance();
6         private resource: any = process.env.MONGODB_RESOURCE;
7
8         public async read(_id: string) {
9             try {
10                 const client = this.db(this.resource);
11                 const response = await client.readOne(this.collectionName, _id);
12                 return response.length ? true : false;
13             } catch {
14                 return false;
15             }
16         }
17     }

```

```

15         }
16     }

18     public async delete(_id: any) {
19         try {
20             const client = this.db(this.resource);
21             await client.deleteOne(this.collectionName, _id);
22             return true;
23         } catch {
24             return false;
25         }
26     }
27 }

```

Notar que siempre estamos interactuando con el ID del producto. Todo manejado por database.ts en lib:

```

1     import { MongoClient, ObjectId } from "mongodb";

3     export default class DataBase {
4         client: MongoClient;
5         dbName: any;
6         private static instance: { [k: string]: DataBase } = {};

8         private constructor() {
9             try {
10                 this.client = new MongoClient(
11                     `mongodb://${process.env.MONGODB_USER}:${
12 {process.env.MONGODB_PASSWORD}@${process.env.MONGODB_HOST}:${process.env.MONGODB_PORT}/${
13 {process.env.MONGODB_DBNAME}`

```

```
12         );
13     } catch (err) {
14         throw new Error(err);
15     }
16 }

18 static getInstance(_resName: string) {
19     if (!DataBase.instance[_resName]) {
20         DataBase.instance[_resName] = new DataBase();
21     }
22     return DataBase.instance[_resName];
23 }

25 async readOne(collection: string, id: string) {
26     const client = await this.client.connect();
27     const search = client
28         .db(this.dbName)
29         .collection(collection)
30         .find({ _id: new ObjectId(id) });
31     return await search.toArray();
32 }

34 async deleteOne(collection: string, id: string) {
35     const client = await this.client.connect();
36     return await client
37         .db(this.dbName)
38         .collection(collection)
```

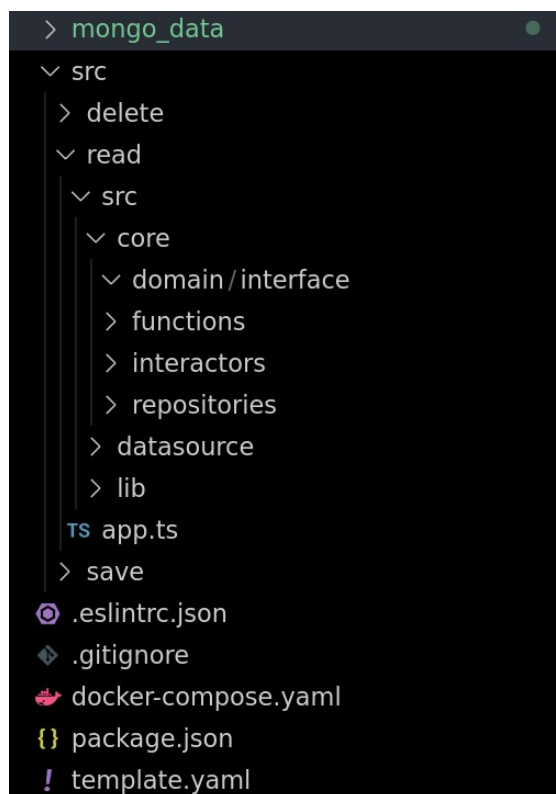
```

39         .deleteOne({ _id: new ObjectId(id) });
40     }
41 }

```

Perfecto. Ahora tenemos la segunda lambda configurada y lista para borrar productos!

Crearemos a la misma altura de save y delete, el directorio read y su correspondiente estructura:



Empezamos por app.ts:

```

1  import { APIGatewayProxyEvent, APIGatewayProxyResult } from "aws-lambda";
2  let response: APIGatewayProxyResult | PromiseLike<APIGatewayProxyResult>;
3  import getProducts from "../src/core/functions/getProducts";
5  export async function readHandler(

```

```
6      event: APIGatewayProxyEvent
7    ): Promise<APIGatewayProxyResult> {
8      try {
9        const { code, data } = await getProducts();
10       response = {
11         statusCode: code,
12         body: JSON.stringify(data),
13       };
14     } catch (err) {
15       console.log(event, err);
16     }
17     return response;
18   }
```

Pasamos a definir la interfaz de Product y Response:

```
1  export default interface ProductInterface {
2    _id: string;
3    name: string;
4    price: string;
5    stock: boolean;
6    created: Date;
7  }
```

```
1  import ProductInterface from "./ProductInterface";
3  export default interface ResponseInterface {
4    code: number;
5    data: ProductInterface[] | [];
6  }
```

Definimos directamente nuestro controlador, `getProducts.ts`:

```

1    import ProductInterface from "../domain/interface/ProductInterface";
2    import { _readProductsFromMongoDB } from "../interactors";
3    import ResponseInterface from "../domain/interface/ResponseInterface";

5    export default async (): Promise<ResponseInterface> => {
6        const response: ProductInterface[] | [] = await _readProductsFromMongoDB();
7        return response.length > 0
8            ? { code: 200, data: response }
9            : { code: 404, data: response };
10   };

```

Prácticamente estamos señalando que se retornará ó un conjunto de elementos con las propiedades de `ProductInterface` ó un arreglo vacío. Para ambos casos, personalizamos un tipo de respuesta.

Veamos ahora dentro de `interactors` lo que va en `index.ts`:

```

1    import readProducts from "../readProducts.interactor";
2    import MongoDS from "../datasource/mongo.datasource";

4    const mongoRepository = new MongoDS();
5    const _readProductsFromMongoDB = readProducts(mongoRepository);

7    export { _readProductsFromMongoDB };

```

`readProducts.interactor`:

```

1    import ProductInterface from "../domain/interface/ProductInterface";
2    import MongoRepository from "../repositories/mongo.repository";

4    export default (mongoRepository: MongoRepository) => {

```

```

5      return async (): Promise<ProductInterface[] | []> => {
6          return await mongoRepository.readProductsFromMongoDB();
7      };
8  };

```

En repositories, mongo.repository.ts:

```

1  import ProductInterface from "../domain/Interface/ProductInterface";

3  export default interface MongoRepository {
4      readProductsFromMongoDB(): Promise<ProductInterface[] | []>;
5  }

```

La estructura dentro de datasource es exactamente similar a las de save y delete.

Con la diferencia que mongo.datasource.ts tendrá:

```

1  import ProductInterface from "../core/domain/interface/ProductInterface";
2  import MongoRepository from "../core/repositories/mongo.repository";
3  import MongoDBManager from "../manager/products.mongodb";

5  export default class MongoDS implements MongoRepository {
6      public async readProductsFromMongoDB(): Promise<ProductInterface[] | []> {
7          try {
8              const process = new MongoDBManager();
9              const productsList = await process.read();
10             return productsList;
11         } catch {
12             return [];
13         }
14     }

```

```
15 }
```

Dentro de manager, products.mongodb.ts:

```
1  import Database from "../lib/database";

3  export default class MongoDBManager {

4      private collectionName: any = process.env.MONGODB_COLLNAME;

5      private db = Database.getInstance;

6      private resource: any = process.env.MONGODB_RESOURCE;

8      public async read() {

9          try {

10             const client = this.db(this.resource);

11             const productsList: any = await client.read(this.collectionName);

12             return productsList;

13         } catch {

14             return [];

15         }

16     }

17 }
```

Ya terminando la configuración para read, definimos nuestro archivo database.ts ubicado en lib:

```
1  import { MongoClient } from "mongodb";

3  export default class DataBase {

4      client: MongoClient;

5      dbName: any;

6      private static instance: { [k: string]: DataBase } = {};
```

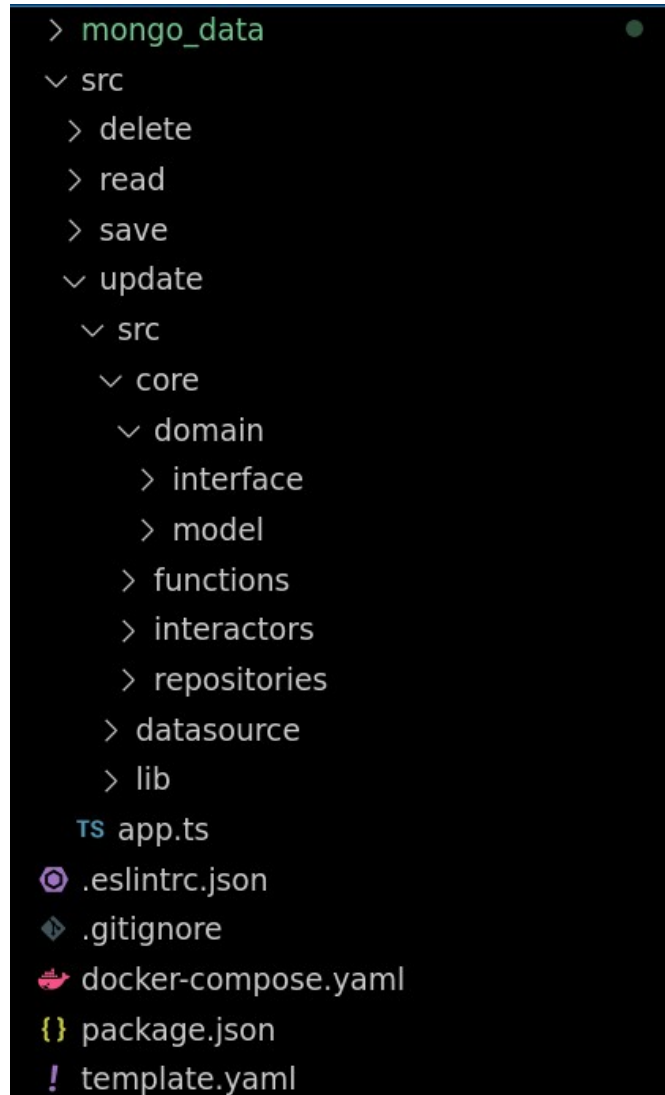


```

8      private constructor() {
9          try {
10             this.dbName = process.env.MONGODB_DBNAME;
11             this.client = new MongoClient(
12                 `mongodb://${process.env.MONGODB_USER}:${
13 {process.env.MONGODB_PASSWORD}@${process.env.MONGODB_HOST}:${process.env.MONGODB_PORT}/${
14 {process.env.MONGODB_DBNAME}`
15             );
16         } catch (err) {
17             throw new Error(err);
18         }
19     }
20
21     static getInstance(_resName: string) {
22         if (!DataBase.instance[_resName]) {
23             DataBase.instance[_resName] = new DataBase();
24         }
25         return DataBase.instance[_resName];
26     }
27
28     async read(collection: string, filters: any = {}) {
29         const client = await this.client.connect();
30         const find = client.db(this.dbName).collection(collection).find(filters);
31         return await find.toArray();
32     }
33 }

```

Finalmente pasamos a la lambda update:



Siguiendo los mismos procedimientos de antes, arrancamos por app.ts:

```

1  import { APIGatewayProxyEvent, APIGatewayProxyResult } from "aws-lambda";
2  let response: APIGatewayProxyResult | PromiseLike<APIGatewayProxyResult>;
3  import processProduct from "../src/core/functions/processProduct";

5  export async function updateHandler(
6      event: APIGatewayProxyEvent
7  ): Promise<APIGatewayProxyResult> {

```

```
8      try {
9          let body: null;

11         if (typeof event.body === "string") {
12             body = JSON.parse(event.body);
13         } else {
14             body = event.body;
15         }

17         const param: any = event.pathParameters;
18         const { code, data } = await processProduct(body, param.id);
19         response = {
20             statusCode: code,
21             body: JSON.stringify(data),
22         };
23     } catch (err) {
24         console.log(err);
25     }
26     return response;
27 }
```

ProductInterface.ts y ResponseInterface.ts, dentro del directorio interface:

```
1  export default interface ProductInterface {
2      _id: string;
3      name: string;
4      price: string;
5      stock: boolean;
6      created: Date;
```

```
7    }
```

```
1    export default interface ResponseInterface {  
2        code: number;  
3        data: string;  
4    }
```

Definimos el modelo, product.ts:

```
1    import ProductInterface from "../interface/ProductInterface";  
  
3    export default class Product implements ProductInterface {  
4        _id: string;  
5        name: string;  
6        price: string;  
7        stock: boolean;  
8        created: Date;  
  
10       constructor(  
11           _id: string | undefined = "",  
12           name: string | undefined = "",  
13           price: string | undefined = "",  
14           stock: boolean,  
15           created: Date  
16       ) {  
17           this.assignValues(_id, name, price, stock, created);  
18       }  
  
20       private assignValues(  

```

```
21         _id: string,
22         name: string,
23         price: string,
24         stock: boolean,
25         created: Date
26     ) {
27         this._id = _id;
28         this.name = name;
29         this.price = price;
30         this.stock = stock;
31         this.created = created;
32     }
33
34     dataToObject(data: any) {
35         this._id = data._id;
36         this.name = data.name;
37         this.price = data.price;
38         this.stock = data.stock;
39         this.created = data.created;
40     }
41
42     validateId() {
43         return this._id ? true : false;
44     }
45
46     validateName() {
47         return this.name ? true : false;
```

```
48     }

50     validatePrice() {
51         return this.price ? true : false;
52     }

54     validateCreated() {
55         return this.created ? true : false;
56     }

58     validationProcess() {
59         const validation = [
60             this.validateId(),
61             this.validateName(),
62             this.validatePrice(),
63             this.validateCreated(),
64         ];
65         return validation.every((condition: boolean) => condition === true);
66     }

68     generateProduct() {
69         return {
70             _id: this._id,
71             name: this.name,
72             price: this.price,
73             stock: this.stock,
74             created: this.created,
75         };
```

```

76         }
77     }

```

Muy parecido al modelo de save, pero con la diferencia de que acá se comprueba el ID del producto ingresado.

Pasamos a realizar el controlador, processProduct.ts:

```

1     import {
2         _readProductFromMongoDB,
3         _updateProductOnMongoDB,
4     } from "../interactors";
5     import Product from "../domain/model/product";
6     import ResponseInterface from "../domain/interface/ResponseInterface";
7
8     export default async (data: any, id: any): Promise<ResponseInterface> => {
9         let response = { code: 404, data: "error to update" };
10        const process = new Product();
11        process.dataToObject(data);
12        process._id = id;
13        const validatedData: boolean = process.validationProcess();
14        const product: any = process.generateProduct();
15        if (validatedData) {
16            const existingProduct = await _readProductFromMongoDB(id);
17            if (existingProduct) {
18                const update = await _updateProductOnMongoDB(product);
19                if (update) {
20                    response = { code: 200, data: "successful update" };
21                }

```

```

22         }
23     }
24     return response;
25 };

```

Explicación:

→ Entra un objeto el cual pasa a ser procesado/verificado, si el producto existe en la base de datos es actualizado. Sino, mensaje de error definido por default.

Pasamos a los interactores, comenzando por el index.ts:

```

1  import readProduct from "../readProduct.interactor";
2  import updateProduct from "../updateProduct.interactor";
3  import MongoDS from "../../datasource/mongo.datasource";

5  const mongoRepository = new MongoDS();

7  const _readProductFromMongoDB = readProduct(mongoRepository);
8  const _updateProductOnMongoDB = updateProduct(mongoRepository);

10 export { _readProductFromMongoDB, _updateProductOnMongoDB };

```

A la misma altura, readProduct.interactor.ts:

```

1  import MongoRepository from "../repositories/mongo.repository";

3  export default (mongoRespository: MongoRepository) => {
4      return async (_id: string): Promise<boolean> => {
5          return await mongoRespository.readProductFromMongoDB(_id);
6      };
7  };

```

Y también updateProduct.interactor.ts:



```

1   import Product from "../domain/model/product";
2   import MongoRepository from "../repositories/mongo.repository";

4   export default (mongoRepository: MongoRepository) => {
5       return async (_product: Product): Promise<boolean> =>
6           await mongoRepository.updateProductOnMongoDB(_product);
7   };

```

Definimos en repositories, mongo.repository.ts:

```

1   import Product from "../domain/model/product";

3   export default interface MongoRepository {
4       readProductFromMongoDB(provider: string): Promise<boolean>;
5       updateProductOnMongoDB(provider: Product): Promise<boolean>;
6   }

```

En datasource creamos la estructura que venimos aplicando desde el inicio: un directorio llamado manager y un fichero llamado mongo.datasource.ts:

```

1   import Product from "../core/domain/model/product";
2   import MongoRepository from "../core/repositories/mongo.repository";
3   import MongoDBManager from "../manager/products.mongodb";

5   export default class MongoDS implements MongoRepository {
6       async readProductFromMongoDB(_id: string): Promise<boolean> {
7           try {
8               const manager = new MongoDBManager();
9               const product = await manager.read(_id);
10              return product ? true : false;
11          } catch {

```

```
12         return false;
13     }
14 }

16 async updateProductOnMongoDB(_productToUpdate: Product): Promise<boolean> {
17     try {
18         const manager = new MongoDBManager();
19         const response = await manager.update(_productToUpdate);
20         return response ? true : false;
21     } catch {
22         return false;
23     }
24 }
25 }
```

```
1 import Database from "../lib/database";

3 export default class MongoDBManager {
4     private collectionName: any = process.env.MONGODB_COLLNAME;
5     private db = Database.getInstance();
6     private resource: any = process.env.MONGODB_RESOURCE;

8     public async read(_id: string) {
9         try {
10             const client = this.db(this.resource);
11             const response = await client.readOne(this.collectionName, _id);
12             return response ? true : false;
```

```

13         } catch {
14             return false;
15         }
16     }

18     public async update(product: any) {
19         try {
20             const client = this.db(this.resource);
21             const { _id, ...data } = product;
22             const result = await client.updateOne(this.collectionName, _id, data);
23             return result ? true : false;
24         } catch {
25             return false;
26         }
27     }
28 }

```

Finalmente, en lib especificamos dentro del archivo database.ts:

```

1     import { MongoClient, ObjectId } from "mongodb";

3     export default class DataBase {
4         client: MongoClient;
5         dbName: any;
6         private static instance: { [k: string]: DataBase } = {};

8         private constructor() {
9             try {
10                 this.dbName = process.env.MONGODB_DBNAME;

```

```

11         this.client = new MongoClient(
12             `mongodb://${process.env.MONGODB_USER}:${
13 {process.env.MONGODB_PASSWORD}@${process.env.MONGODB_HOST}:${process.env.MONGODB_PORT}/${
14 {process.env.MONGODB_DBNAME}`
15         );
16     } catch (err) {
17         throw new Error(err);
18     }
19
20     static getInstance(_resName: string) {
21         if (!DataBase.instance[_resName]) {
22             DataBase.instance[_resName] = new DataBase();
23         }
24         return DataBase.instance[_resName];
25     }
26
27     async readOne(collection: string, id: string) {
28         const client = await this.client.connect();
29         const search = client
30             .db(this.dbName)
31             .collection(collection)
32             .find({ _id: new ObjectId(id) });
33         return await search.toArray();
34     }
35
36     async updateOne(collection: string, id: string, data: any) {
37         const client = await this.client.connect();

```

```

37         const result = await client
38             .db(this.dbName)
39             .collection(collection)
40             .updateOne(
41                 {
42                     _id: new ObjectId(id),
43                 },
44                 { $set: data }
45             );
46         return result.matchedCount !== 0 ? true : false;
47     }
48 }

```

Con todo el código que hemos hecho, ahora instalamos las dependencias y ejecutamos el comando definido en el package.json:

```

stock_manager > npm install && npm start
( [redacted] ) : idealTree:stock_manager: sill idealTree bui

```

Si todo salió correctamente, tendrías que poder visualizar lo siguiente:

```

package.json file not found. Bundling source without dependencies.
Running NodejsNpmEsbuildBuilder:EsbuildBundle
Building codeuri: /home/martyn/Pruebas/segundo_proyecto/stock_manager/src/update runtime: nodejs16.x metadata: {'BuildMethod': 'esbuild', 'BuildProperties': {'Minify': False, 'Target': 'es2020', 'SourceMap': True, 'UseNpmCi': True}} architecture: x86_64 functions: UpdateProductFunction
package.json file not found. Bundling source without dependencies.
Running NodejsNpmEsbuildBuilder:EsbuildBundle

Build Succeeded

Built Artifacts : .aws-sam/build
Built Template : .aws-sam/build/template.yaml

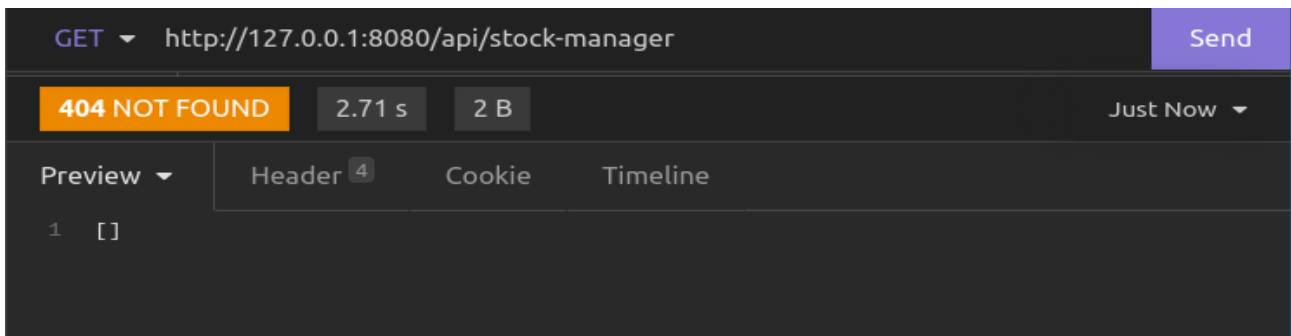
Commands you can use next
=====
[*] Validate SAM template: sam validate
[*] Invoke Function: sam local invoke
[*] Test Function in the Cloud: sam sync --stack-name {stack-name} --watch
[*] Deploy: sam deploy --guided

Mounting UpdateProductFunction at http://127.0.0.1:8080/api/stock-manager/{id} [PUT]
Mounting DeleteProductFunction at http://127.0.0.1:8080/api/stock-manager/{id} [DELETE]
Mounting SaveProductFunction at http://127.0.0.1:8080/api/stock-manager [POST]
Mounting ReadProductsFunction at http://127.0.0.1:8080/api/stock-manager [GET]
You can now browse to the above endpoints to invoke your functions. You do not need to restart/reload SAM CLI while working on your functions, changes will be reflected instantly/automatically. You only need to restart SAM CLI if you update your AWS SAM template
2022-10-08 13:55:39 * Running on http://127.0.0.1:8080/ (Press CTRL+C to quit)

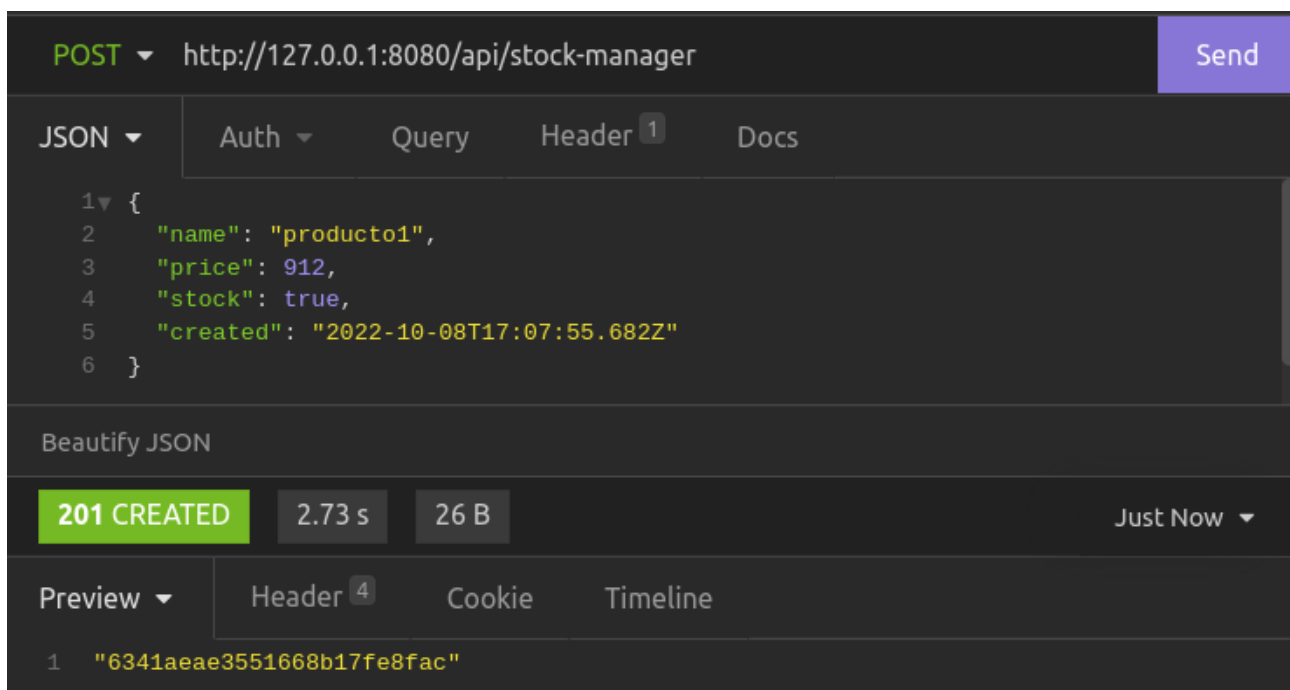
```

Si te fijas en la terminal, se están indicando los puntos de entrada que indicamos en el `template.yaml`. ¡Hagamos las pruebas!

Si hacemos un GET al punto `http://127.0.0.1:8080/api/stock-manager` obtendremos un arreglo vacío porque aún no se ha cargado nada en la base de datos, pero para asegurarnos de que todo está bien, hacemos la petición:

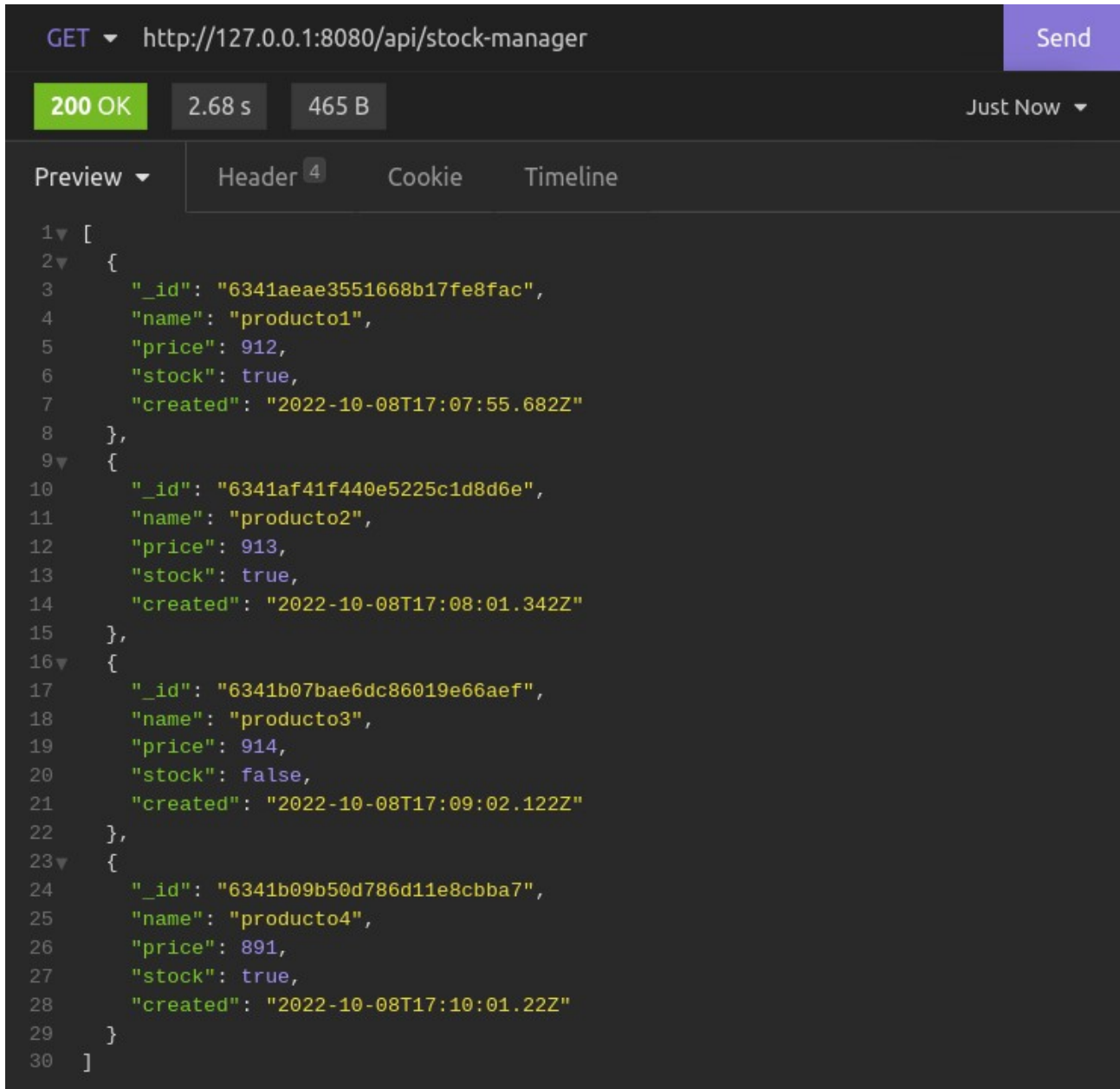


Procedemos a cargar un nuevo producto pegándole con POST:



Vemos que efectivamente se está cumpliendo lo que indicamos. Es decir, que se retorne el ID generado por MongoDB una vez que se haya cargado un producto.

Carguemos más productos y visualicemos nuevamente el GET para ver cómo ha quedado nuestro arreglo:



```
GET http://127.0.0.1:8080/api/stock-manager Send
200 OK 2.68 s 465 B Just Now
Preview Header 4 Cookie Timeline
1▼ [
2▼ {
3    "_id": "6341aeae3551668b17fe8fac",
4    "name": "producto1",
5    "price": 912,
6    "stock": true,
7    "created": "2022-10-08T17:07:55.682Z"
8  },
9▼ {
10   "_id": "6341af41f440e5225c1d8d6e",
11   "name": "producto2",
12   "price": 913,
13   "stock": true,
14   "created": "2022-10-08T17:08:01.342Z"
15  },
16▼ {
17   "_id": "6341b07bae6dc86019e66aef",
18   "name": "producto3",
19   "price": 914,
20   "stock": false,
21   "created": "2022-10-08T17:09:02.122Z"
22  },
23▼ {
24   "_id": "6341b09b50d786d11e8cbba7",
25   "name": "producto4",
26   "price": 891,
27   "stock": true,
28   "created": "2022-10-08T17:10:01.22Z"
29  }
30 ]
```

Intentamos actualizar un producto, en mi caso por ejemplo elegiré el que tiene ID 6341b07bae6dc86019e66aef:

The screenshot shows a REST client interface with a PUT request to `http://127.0.0.1:8080/api/stock-manager/6341b07bae6dc86019e66aef`. The request body is a JSON object representing a product update. The response is a `200 OK` status with a response time of `2.67 s` and a body size of `19 B`. The response body is `"successful update"`.

```
PUT http://127.0.0.1:8080/api/stock-manager/6341b07bae6dc86019e66aef Send
```

JSON ▾ Auth ▾ Query Header 1 Docs

```
1 {
2   "name": "productoActualizado",
3   "price": 988,
4   "stock": true,
5   "created": "2022-10-08T17:10:02.122Z"
6 }
```

Beautify JSON

200 OK 2.67 s 19 B 2 Minutes Ago ▾

Preview ▾ Header 4 Cookie Timeline

```
1 "successful update"
```

Chequeamos nuestro arreglo de productos:

The screenshot shows a REST client interface with a GET request to `http://127.0.0.1:8080/api/stock-manager`. The response is a `200 OK` status with a response time of `2.77 s` and a body size of `474 B`. The response body is a JSON array containing four product objects.

```
GET http://127.0.0.1:8080/api/stock-manager Send
```

200 OK 2.77 s 474 B Just Now ▾

Preview ▾ Header 4 Cookie Timeline

```
1 [
2   {
3     "_id": "6341aeae3551668b17fe8fac",
4     "name": "producto1",
5     "price": 912,
6     "stock": true,
7     "created": "2022-10-08T17:07:55.682Z"
8   },
9   {
10    "_id": "6341af41f440e5225c1d8d6e",
11    "name": "producto2",
12    "price": 913,
13    "stock": true,
14    "created": "2022-10-08T17:08:01.342Z"
15  },
16  {
17    "_id": "6341b07bae6dc86019e66aef",
18    "name": "productoActualizado",
19    "price": 988,
20    "stock": true,
21    "created": "2022-10-08T17:10:02.122Z"
22  },
23  {
24    "_id": "6341b09b50d786d11e8cbba7",
25    "name": "producto4",
26    "price": 891,
27    "stock": true,
28    "created": "2022-10-08T17:10:01.22Z"
29  }
30 ]
```



Efectivamente se ha actualizado con éxito. Por último borremos dos productos al azar y verifiquemos que se haya borrado realizando un GET de nuevo:

A screenshot of a REST client interface. The top bar shows a **DELETE** method and the URL `http://127.0.0.1:8080/api/stock-manager/6341b09b50d786d11e8cbba7|`. A **Send** button is on the right. Below the bar, a status bar shows **200 OK**, **2.75 s**, and **19 B**, with a **Just Now** dropdown. The main area has tabs for **Preview**, **Header** (with a count of 4), **Cookie**, and **Timeline**. The **Preview** tab is active, showing a single line of JSON: `1 "successful delete"`.

A screenshot of a REST client interface. The top bar shows a **DELETE** method and the URL `http://127.0.0.1:8080/api/stock-manager/6341aeae3551668b17fe8fac`. A **Send** button is on the right. Below the bar, a status bar shows **200 OK**, **2.74 s**, and **19 B**, with a **Just Now** dropdown. The main area has tabs for **Preview**, **Header** (with a count of 4), **Cookie**, and **Timeline**. The **Preview** tab is active, showing a single line of JSON: `1 "successful delete"`.

A screenshot of a REST client interface. The top bar shows a **GET** method and the URL `http://127.0.0.1:8080/api/stock-manager`. A **Send** button is on the right. Below the bar, a status bar shows **200 OK**, **2.77 s**, and **243 B**, with a **Just Now** dropdown. The main area has tabs for **Preview**, **Header** (with a count of 4), **Cookie**, and **Timeline**. The **Preview** tab is active, showing a JSON array with two objects. The first object has `"_id": "6341af41f440e5225c1d8d6e"`, `"name": "producto2"`, `"price": 913`, `"stock": true`, and `"created": "2022-10-08T17:08:01.342Z"`. The second object has `"_id": "6341b07bae6dc86019e66aef"`, `"name": "productoActualizado"`, `"price": 988`, `"stock": true`, and `"created": "2022-10-08T17:10:02.122Z"`. The JSON is formatted with line numbers 1 through 16.

Genial, ya hemos realizado un pequeño CRUD con Node, TS, SAM dentro de un entorno virtualizado.

Te dejo a mano el repositorio de este proyecto por si hay algo que no pudiste hacer, no levanta o quizá no quedó del todo claro:

> [https://github.com/MartynLCD3/Stock\\_Manager](https://github.com/MartynLCD3/Stock_Manager)

---

## Conclusiones



Luego de haber visto qué es TypeScript, ver sus principales características, implementarlas en dos proyectos y poner a prueba cómo funcionan, podemos decir que ya tenés las bases para seguir investigando más sobre el tema.

También quiero añadir que esta guía tiene como finalidad adentrarte en materia para que el día de mañana puedas jugar con este lenguaje y enriquecer tus trabajos.

Por último, TS en definitiva te va a permitir eliminar la mayoría de bugs que puede tener un proyecto. Cosa que con JS no es posible a la primera por la gran flexibilidad que tiene. Por eso como recomendación final, es que si vas a trabajar en proyectos grandes y querés ahorrarte con tu equipo muchos dolores de cabeza, TypeScript es la mejor opción.

De momento ¡esto ha sido todo!

Cualquier consulta podés escribirme en LinkedIn o a mi casilla de mail:  
[martincastagno@protonmail.com](mailto:martincastagno@protonmail.com)

Te deseo muchos éxitos en tu carrera! Hasta la próxima.