educación IT
CODE YOUR FUTURE

Bootcamp Java Developer

Fase 2 - Java Web Developer

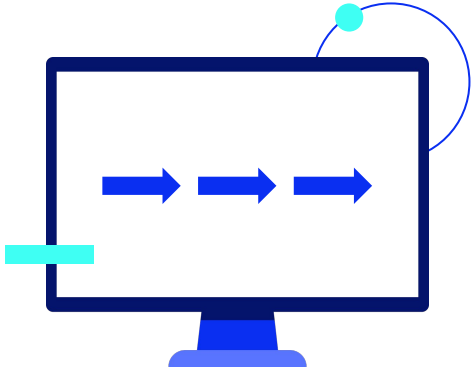
Módulo 19

Pipeline

Encadenamiento de promesas

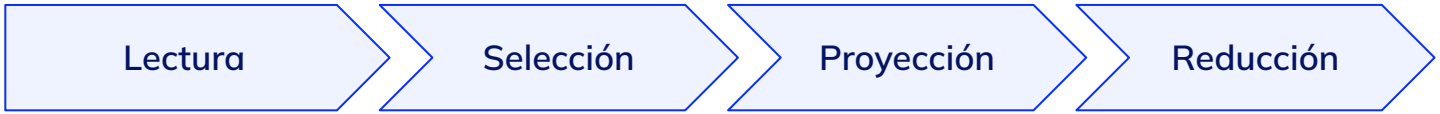
Como vimos, una promesa representa una tarea asincrónica. Ahora bien, cuando **encadenamos muchas tareas para lograr una gestión concreta tenemos un proceso asincrónico**.

Hasta ahora, vimos cómo crear una tarea. Ahora vamos a estudiar cómo encadenar tareas para generar un proceso.



Arquitectura en Pipeline

La arquitectura basada en filtros (en pipeline) consiste en **ir transformando un flujo de datos en un proceso comprendido por varias fases secuenciales**, siendo la entrada de cada una la salida de la anterior.



¿Para qué sirve esta arquitectura?

- Programación mucho más declarativa.
- Dividimos un proceso en operadores reutilizables.
- Código mucho más testeable al poder intercambiar operadores de un proceso.

Casos de uso:

- Generación de reportes complejos.
- Procesamientos intercambiables.
- División de la interfaz de un proceso.

Una división en Pipeline

```
1  const obtenerNumeros = () => ({
2      n1: Number(prompt('Ingrese el primer numero')),
3      n2: Number(prompt('Ingrese el segundo número'))
4  })
5
6  const verificarSegundoNumero = (nums) =>
7  {
8      if(nums.n2 == 0)
9          throw new Error('Division por cero');
10
11     return nums;
12 }
13
14 const dividir = (nums) =>  nums.n1 / nums.n2;
15
16 const mostrarPorConsola = (data) => console.log('Division: ', data);
17
18
19 let resultado = obtenerNumeros();
20 resultado = verificarSegundoNumero(resultado);
21 resultado = dividir(resultado);
22 resultado = mostrarPorConsola(resultado);
```

Como podemos examinar, no hay nuevo código. Se trata de una forma de organizar los conocimientos actuales.

Para implementar una arquitectura en **`pipeline`** tenemos **dos partes:**

- Por un lado los **operadores** o ***stages***. Para que sean encadenables todas las funciones que actúen como stage deben tener la misma entrada y el mismo retorno.
- Por otro lado, la **tubería** o ***pipeline*** propiamente dicha. En este caso es un *pipeline* escrito a mano en las líneas 19 a 22. Sin embargo, podemos escribir una función que reciba un array de stages y devuelva el resultado.

Función pipe()

En este caso, creamos una función sencilla que pasa el resultado inicial por una serie de etapas. **Cada etapa tiene la misma firma (mismas entradas y mismo retorno). Por eso puede ser encadenable.**

```
1  const pipe = (inicial, stages) => stages.reduce((resultado, stage) => stage(resultado), inicial);
2
3  const resultado = pipe(obtenerNumeros(), [
4      verificarSegundoNumero,
5      dividir,
6      mostrarPorConsola
7  ]);
```

Encadenamiento de promesas

La **API Promise** nos facilita el encadenamiento de tareas asincrónicas a través del `.then()`.

La clave para el encadenamiento es que una función reciba y emita siempre el mismo tipo de dato que todas las demás.

Ahora bien, podemos utilizar el tipo `Promise` como ese tipo de dato común a todos los pasos asincrónicos.

```
1   const obtenerNumeros = () => new Promise((res) => res({
2       n1: Number(prompt('Ingrese el primer numero')),
3       n2: Number(prompt('Ingrese el segundo número'))
4   }));
5
6   const verificarSegundoNumero = (nums) => new Promise((res, rej) =>
7   {
8       if(nums.n2 == 0) {
9           rej('Division por cero');
10          return;
11      }
12
13      res(nums);
14      return nums;
15  })
16
17  const dividir = (nums) =>  new Promise((res) => res(nums.n1 / nums.n2));
18
19  const mostrarPorConsola = (data) => console.log('Division: ', data);
20
```

¿Cómo se usa esto?

Ejecutamos la primera función y asignamos su **`.then()`**. Si este **`.then()`** devuelve una promesa, se puede escribir otro **`.then()`** seguido que tomará el resultado de la última promesa.

Si el proceso se ejecuta normalmente se ejecutará cada promesa y se pasará el valor a través de toda la tubería. Si hay algún error salta directamente al **`.catch()`** y termina.

Si bien este ejemplo es con procesos comunes, **tiene sentido aplicar este encadenamiento de promesas cuando tratemos con operaciones asincrónicas como consultas a API complejas.**

Ejemplo

```
1  obtenerNumeros()
2      .then(verificarSegundoNumero)
3      .then(dividir)
4      .then(mostrarPorConsola)
5      .catch((err) => console.log(err))
```

¡Sigamos trabajando!