

## Clase 2

(Final de la parte aburrida)

## Event-Loop: Repaso

- Tal como lo define su nombre, el event loop es un bucle
- El Event-Loop, tiene “pasos” que va ejecutando en un orden particular, los mismos se denominan “Tick”
- El call stack, es decir, la ejecución de las funciones en nuestro código, también es controlada por este event loop
- Dicho callstack, se agrega como otro paso del event loop
- Cuando el event loop no tiene callbacks por ejecutar, procede a continuar el Stack (de hecho, hace algunos otros pasos, pero eso ya es para nerds)
- El comienzo de ejecución de nuestro código tanto como otras partes de él, son considerados callbacks (o eventos) dentro del event loop, y son agregados a una cola de ejecución
- Corre infinitamente hasta que se termine el proceso

## Async/Await: Repaso

En resumen, una forma bonita de usar promesas. Son dos constructores de lenguaje que transforman o utilizan de forma especial una función.

Async transforma una función a una asíncrona, es decir, utilizándolo le decimos a Node que el código de la misma contiene lógica asíncrona.

Con await, le decimos a node que la función que estamos llamando puede tardar en completarse y que necesitamos ese resultado antes de continuar. Esto puede utilizarse también para promesas, en cuyo caso, obtendremos la data que el método `.then()` recibiría.

Generalmente, las funciones asíncronas nativas (las que no contienen promesas dentro, como los módulos de Node - http, fs, etc) pueden ser utilizadas con await, sin pasarles un callback como parámetro. Todo esto siempre y cuando, el prototype de las mismas haya sido actualizado para soportar esto (ver documentación para estar seguro).

Algo muy importante que no hay que olvidar es que al utilizar await, en caso de que haya un error dentro de la promesa, ese error ahora va a ser `THROWeado`. Es decir, necesitamos envolver esto dentro de un try/catch para prevenir errores de ejecución.

## Promesas: Repaso

```
1  const pepe = new Promise(function(resolve, reject) {
2    /**
3     * Esta funcion cuyos parametros son resolve y reject,
4     * se utiliza para realizar tareas asincronas.
5     * Al terminar las mismas, se deben llamar a los parametros resolve o reject
6     * los cuales "avisan" que la promesa fue finalizada
7     */
8    functionAsincrona(function(err, data) {
9      if (err) {
10        // En este caso, la funcion asincrona devolvio err
11        // Por eso llamamos a reject, para avisar que la promesa falló
12        return reject(err);
13      }
14
15      // Si la promesa no falló, llamamos a resolve para avisar que fue exitosa
16      resolve(data);
17    })
18  });
19
20  /**
21   * El código anterior se ejecuta instantáneamente, apenas se crea la promesa
22   * Al utilizar el constructor de new Promise(), obtenemos un objeto con tres métodos:
23   */
24  pepe
25    .then(data => console.log('Pepe termino!', data))
26    .catch(err => console.log('Pepe falló!', err))
27    .finally(() => console.log('Esto se muestra sin importar si hubo exito o no'));
28
29
```

```
const promises = [ pepe, pepe2, pepe3 ];

// La clase Promise tiene un metodo estatico .all() el cual recibe un array de promesas
// Y devuelve, por supuesto, una promesa
Promise
  .all(promises)
  .then(data => {
    // Esto se ejecutaria cuando todas las promesas son finalizadas
  })
  .catch(err => {
    // Y esto si alguna de las mismas falla
  })
```

## Introducción a APIs

Qué es una API? Son siglas de Application Programming Interface. Es un método de comunicación entre dos o más aplicaciones. En nuestro caso, utilizamos esta forma de trabajo generalmente para evitar exponer información sensible al usuario final. Por ejemplo, en el caso de un home banking, no podemos dar acceso completo a la base de datos ya que sería una falta de seguridad por nuestra parte. Tampoco podemos acceder a la misma desde el frontend, ya que el código fuente es visible al usuario.

En este caso, todo el manejo de datos se hacen del lado del servidor, y se da un acceso limitado a los datos y operaciones mediante una API.

## RESTful API

REST es un conjunto de estándares mayormente compartidos para desarrollar APIs. Se dice que son mayormente compartidos porque en realidad no hay una reglamentación, o algo por el estilo que determine si una API es REST o no. Puede haber algunas api que cumplan algunas reglas y otras no. Pero por lo general, la mayoría cumple con lo siguiente:

- Existen endpoints que representan una determinada entidad
- No se usan endpoints con verbos ni acciones, sino que la acción a realizar se determina según el tipo de método HTTP con el que se solicita la url
- Generalmente se utiliza GET para obtener un listado o un recurso determinado, POST para crearlos, PUT o PATCH para actualizar y DELETE para borrarlo
- La api es stateless. Esto significa, que no se guardan estados entre las diferentes requests, y cada una se trata por separado
- It's all about JSON.
- Se utilizan códigos de respuesta HTTP para informar estados, como por ejemplo 404 si no se encuentra un elemento, 201 si un elemento fue creado exitosamente, etc

## Otros tipos de API:

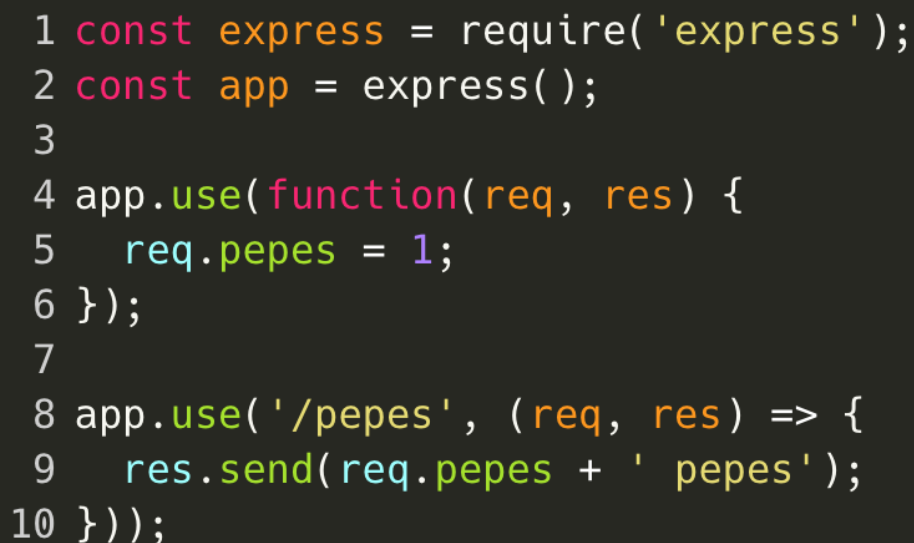
- SOAP 🤖 - Super legacy
- GraphQL ❤️ - Nuevo, trending y fancy

## ExpressJS y Middlewares

ExpressJS es la librería por excelencia para crear APIs. Es minimalista, y muy extensible. Se extiende por medio de los middlewares.

Los middlewares son funciones de ExpressJS que tienen acceso a los objetos Request y Response. Pudiendo acceder a información sobre la request y a los métodos de respuesta. Una aplicación desarrollada con esta librería es básicamente un conjunto de middlewares.

Los middleware no necesariamente son el final del procesamiento de la request, sino que pueden realizar alguna acción, pero continuar la misma para que otro middleware la procese y devuelva la respuesta final.



```
1 const express = require('express');
2 const app = express();
3
4 app.use(function(req, res) {
5   req.pepes = 1;
6 });
7
8 app.use('/pepes', (req, res) => {
9   res.send(req.pepes + ' pepes');
10 }));
```

## Actividad

- Crear una API rest para un listado de productos
- Intentar implementar algunos de los temas asíncronos que vimos previamente