

[Contextos](#)

[call](#)

[apply](#)

[Intervalles](#)

[setInterval](#)

[clearInterval](#)

[setTimeout](#)

[Ejemplo integrador.](#)

## Contextos

Ya hemos visto en capítulos anteriores el concepto de closures, scope y el ámbito de las variables.

Dentro del cuerpo de una función sabemos que contamos con "this", que representa al cuerpo de la función, lo que llamaremos a partir de ahora "contexto". El contexto puede ser "generado" gracias al método call u apply de las funciones.

### call

El método (prototype de las funciones) nos permite ejecutar una función pasando como primer argumento el contexto (this) y como los siguientes argumentos los argumentos de la función a ejecutar.

```
var f = function(){ console.log(this, arguments); }
f.call({a:1,b:2,c:3}, "hola", []);
```



### apply

Al igual que el método call, apply ejecuta la función pasando como primer argumento un nuevo contexto, pero la diferencia es que el segundo argumento es un array con todos los argumentos a pasar a la función llamada.

```
var f = function(){ console.log(this, arguments); }
f.apply({pepe:"luis", hi: function(){ console.log("hola");} }, [1,2,3,4]);
```



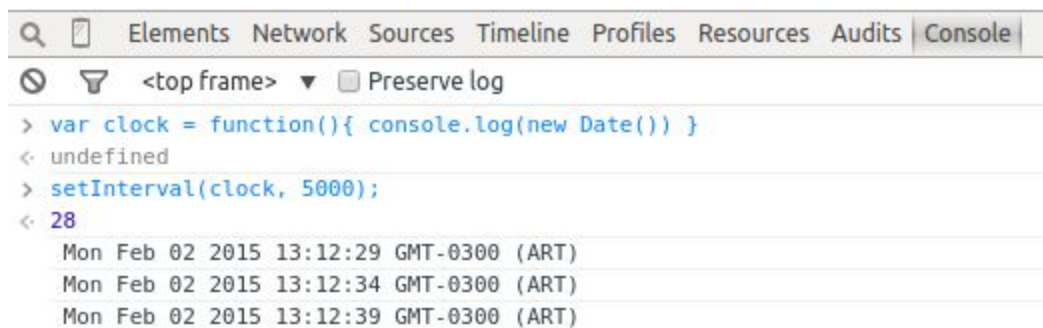
## Intervalles

### setInterval

setInterval es una función que recibe dos argumentos, el primero es una función a ejecutar, la segunda es cada cuanto tiempo (en milisegundos). Básicamente funciona como un "cron", que ejecuta una y otra vez la misma función cada N milisegundos.

La función retorna un intervalID, el cual nos servirá para por ejemplo poder parar el "cron".

```
var clock = function(){ console.log(new Date()) }
setInterval(clock, 5000);
```



Como vemos la ejecución de setInterval retorna "28", el cual es el intervalID.

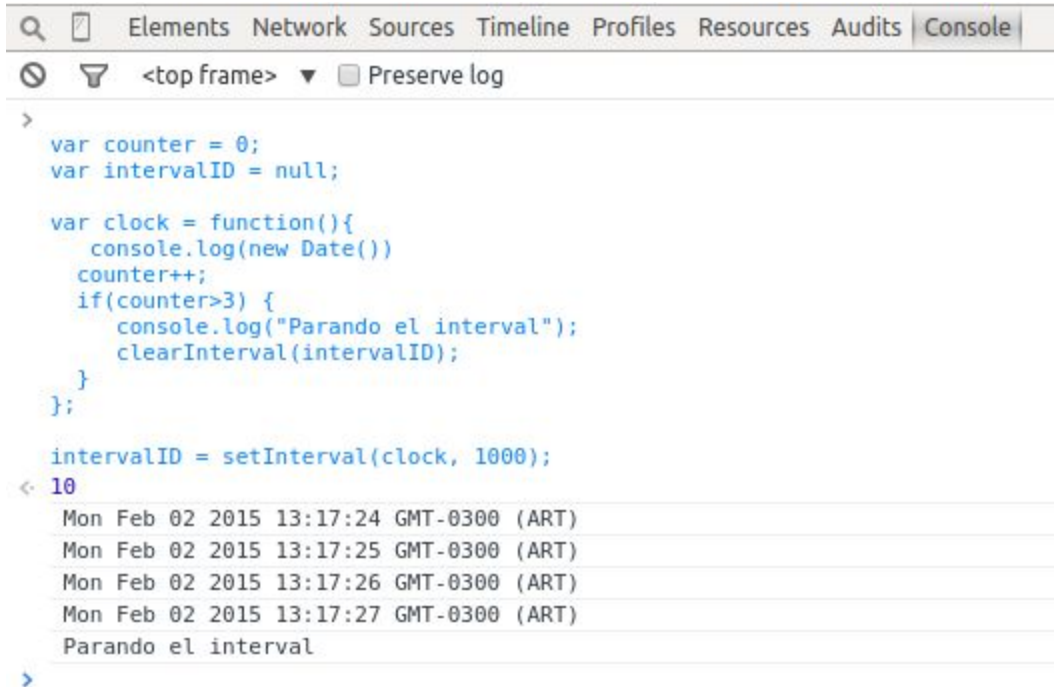
### clearInterval

La función clearInterval detiene un proceso setInterval (lo elimina). Recibe como argumento un intervalID.

```
var counter = 0;
var intervalID = null;

var clock = function(){
  console.log(new Date())
  counter++;
  if(counter>3) {
    console.log("Parando el interval");
    clearInterval(intervalID);
  }
};

intervalID = setInterval(clock, 1000);
```



```

>
var counter = 0;
var intervalID = null;

var clock = function(){
  console.log(new Date())
  counter++;
  if(counter>3) {
    console.log("Parando el interval");
    clearInterval(intervalID);
  }
};

intervalID = setInterval(clock, 1000);
10
Mon Feb 02 2015 13:17:24 GMT-0300 (ART)
Mon Feb 02 2015 13:17:25 GMT-0300 (ART)
Mon Feb 02 2015 13:17:26 GMT-0300 (ART)
Mon Feb 02 2015 13:17:27 GMT-0300 (ART)
Parando el interval
>

```

## setTimeout

setTimeout es una función que hace exactamente lo mismo que setInterval con la diferencia de que en vez de ejecutar la función pasada como primer argumento cada N milisegundos esperará el tiempo pasado como segundo argumento y ejecutará la función una única vez.

```

var clock = function(){
  console.log(new Date())
};

```

```

setTimeout(clock, 1000);

```



```

> var clock = function(){
  console.log(new Date())
};

setTimeout(clock, 1000);
54
Mon Feb 02 2015 17:09:18 GMT-0300 (ART)
>

```

## Ejemplo integrador.

En el siguiente ejemplo creamos un módulo de usuario llamado Cache, el cual nos permite guardar datos en cache en memoria RAM.

El mismo tiene un procedimiento para controlar que el valor guardado en memoria no hay expirado, si expiró lo elimina automáticamente.

```
"use strict";

module.exports = function(){
  var _this = this
  this.fs = require('fs');
  this.Caches = {};

  setInterval( function(){
    for(var c in _this.Caches){
      if(new Date().getTime() - _this.Caches[c].created.getTime() >=
        _this.Caches[c].ttl) { delete _this.Caches[c]; }
    }
  }, 500);

  /**
   * Retrieves value by key
   * @method get
   * @param {string} key
   * @return {Object} value
   */
  this.get = function(key, driver){
    var driver = driver || 'memory';
    switch(driver){
      case 'file':
        return _this.getFile(key);
        break;
      case 'memory':
      default:
        if(typeof _this.Caches[key] == "undefined") return false;
        else return _this.Caches[key].data;
        break;
    }
  };

  /**
   * Adds key value to cache by driver (memory, file ...)
   * @method add
   * @param {string} key key for access
   * @param {Object} value object to save
   * @param {Number} ttl (time to live) cache time
   */
}
```

```
* @param {string} driver is the way the data is stored (memory, file). Default memory
*/
this.add = function(key, val, ttl, driver){
  var driver = driver || 'memory';
  switch(driver){
    case 'file': return _this.writeFile(key, val, ttl); break;
    case 'memory':
    default:
      _this.Caches[key] = { data: val, created: new Date(), ttl: ttl*1000, key: key };
      break;
  }
};
/**
 * alias for add method
 * @method add
 *
 */
this.set = this.add;

this.getFile = function(key){
  console.log('exists? : ', _this.fs.existsSync(key));

  if(_this.fs.existsSync(key)) { return _this.fs.readFileSync(key).toString(); }
  else { return false; }
};

this.writeFile = function(key, val, ttl){
  _this.fs.writeFileSync(key, val, {flags:'w'});
};

};
```