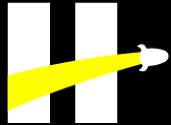
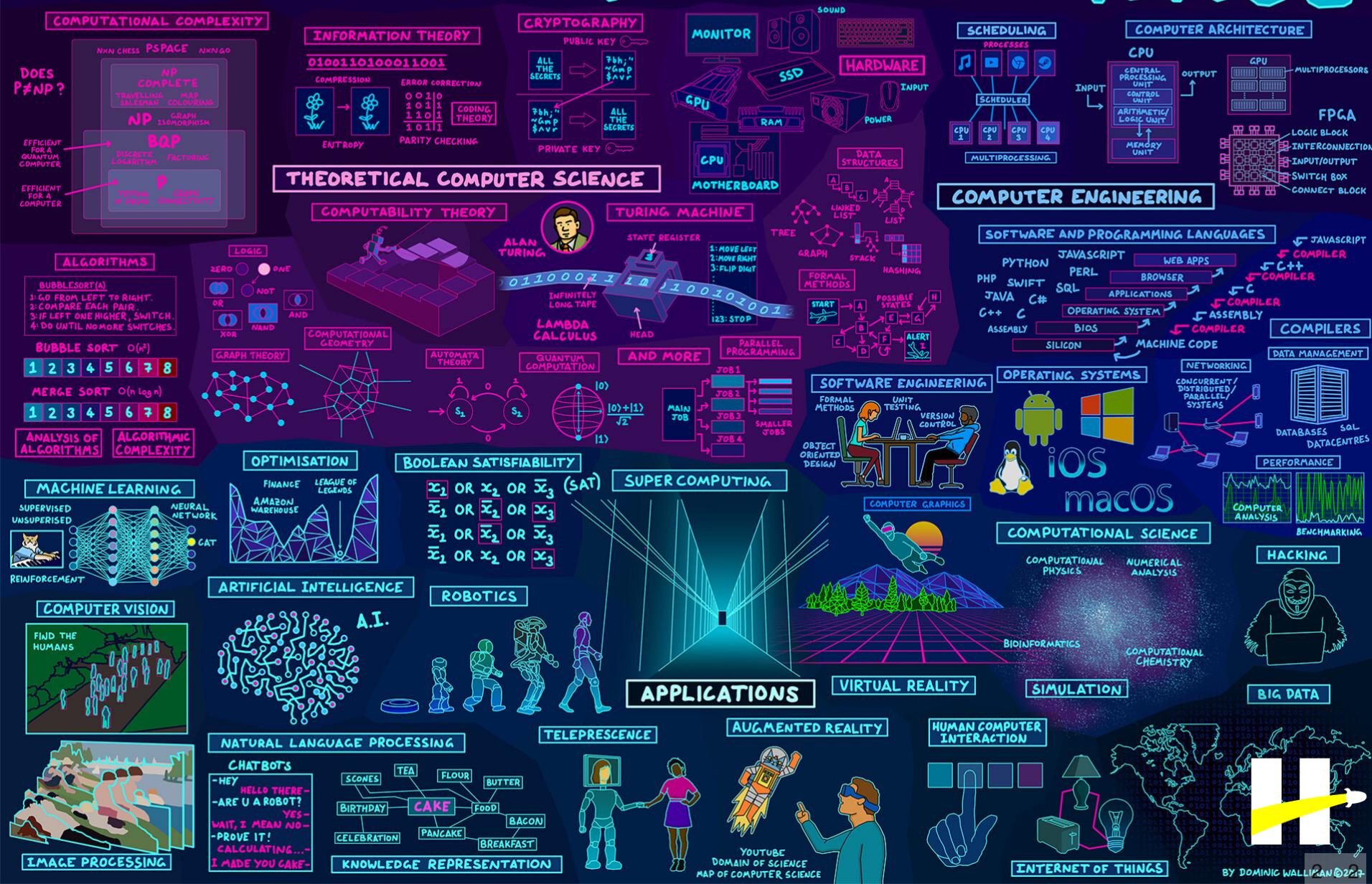


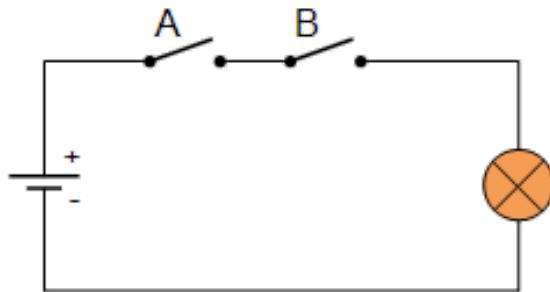
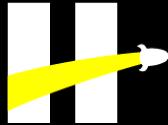
HENRY



Intro to CS

MAP OF COMPUTER SCIENCE

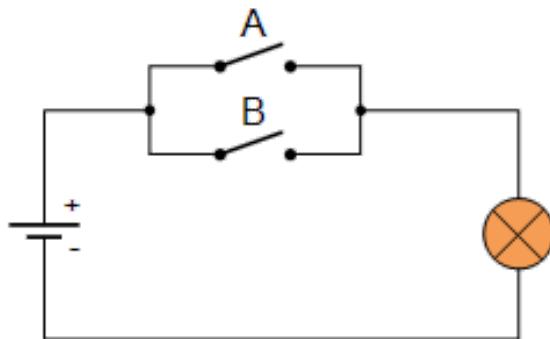




Lamp - ON = "1"
Lamp - OFF = "0"

Switch A - Open = "0", Closed = "1"

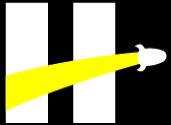
Switch B - Open = "0", Closed = "1"



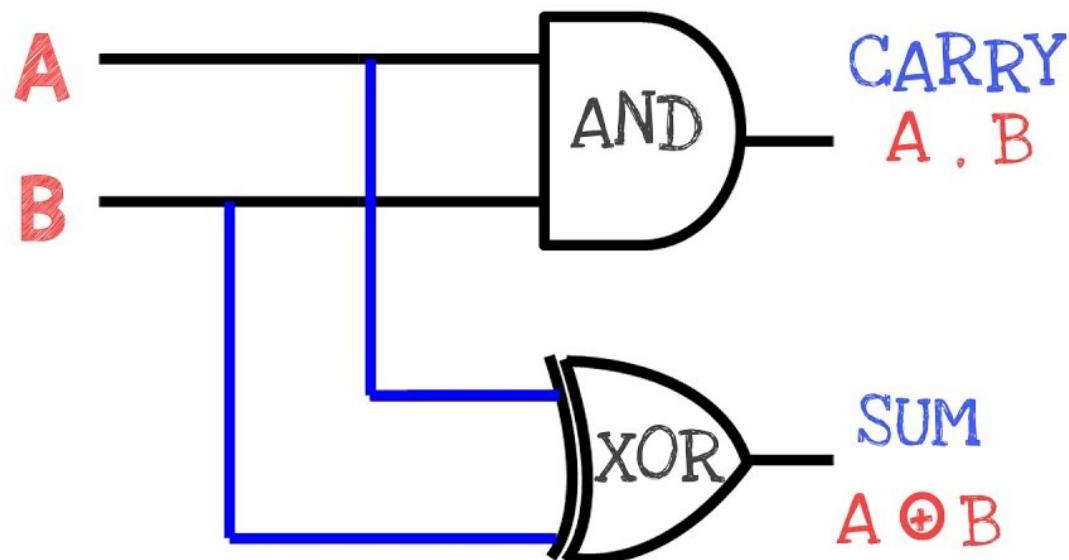
Lamp - ON = "1"
Lamp - OFF = "0"

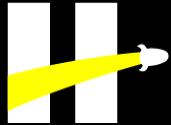
Switch A - Open = "0", Closed = "1"

Switch B - Open = "0", Closed = "1"

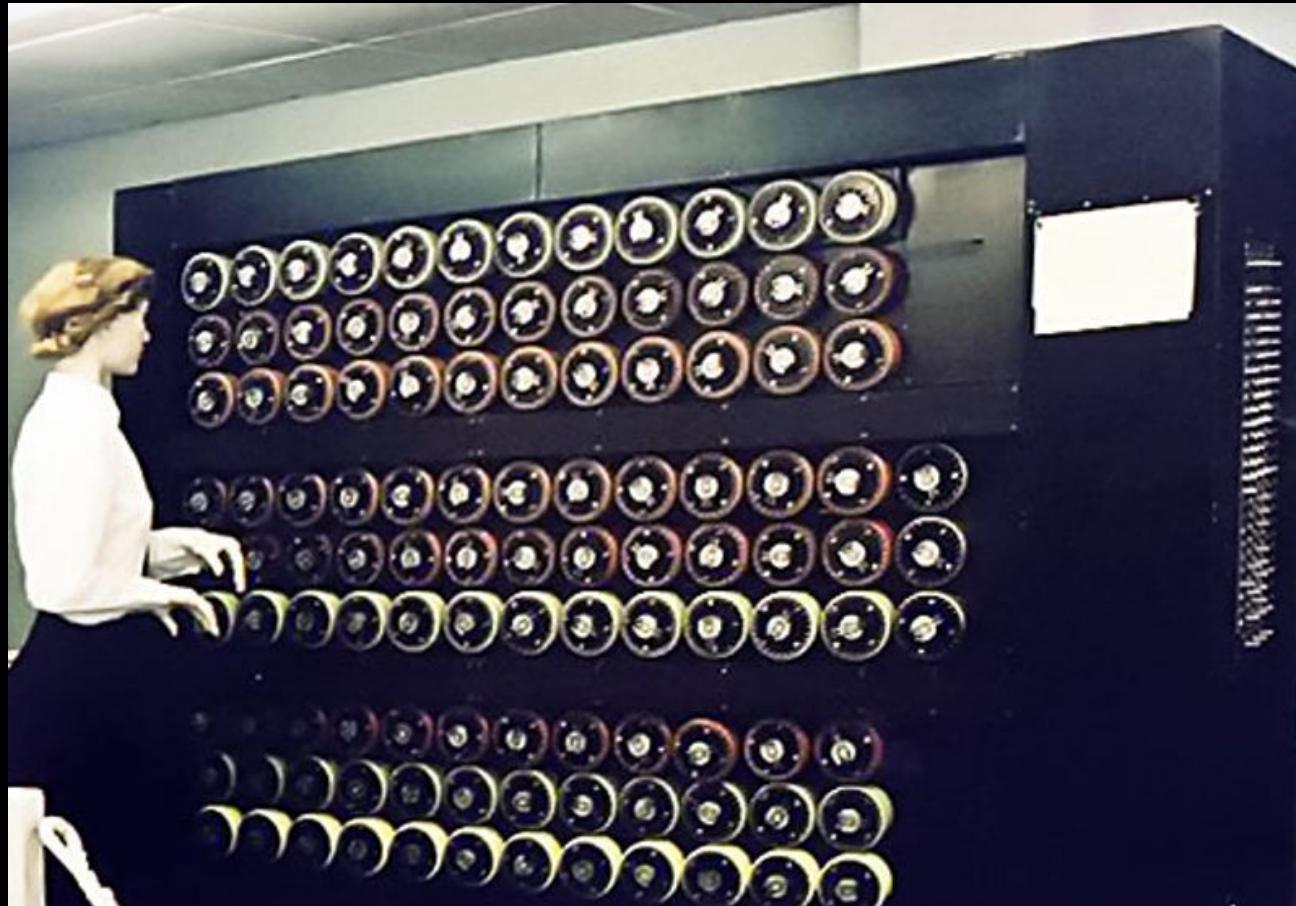
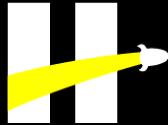


Half Adder

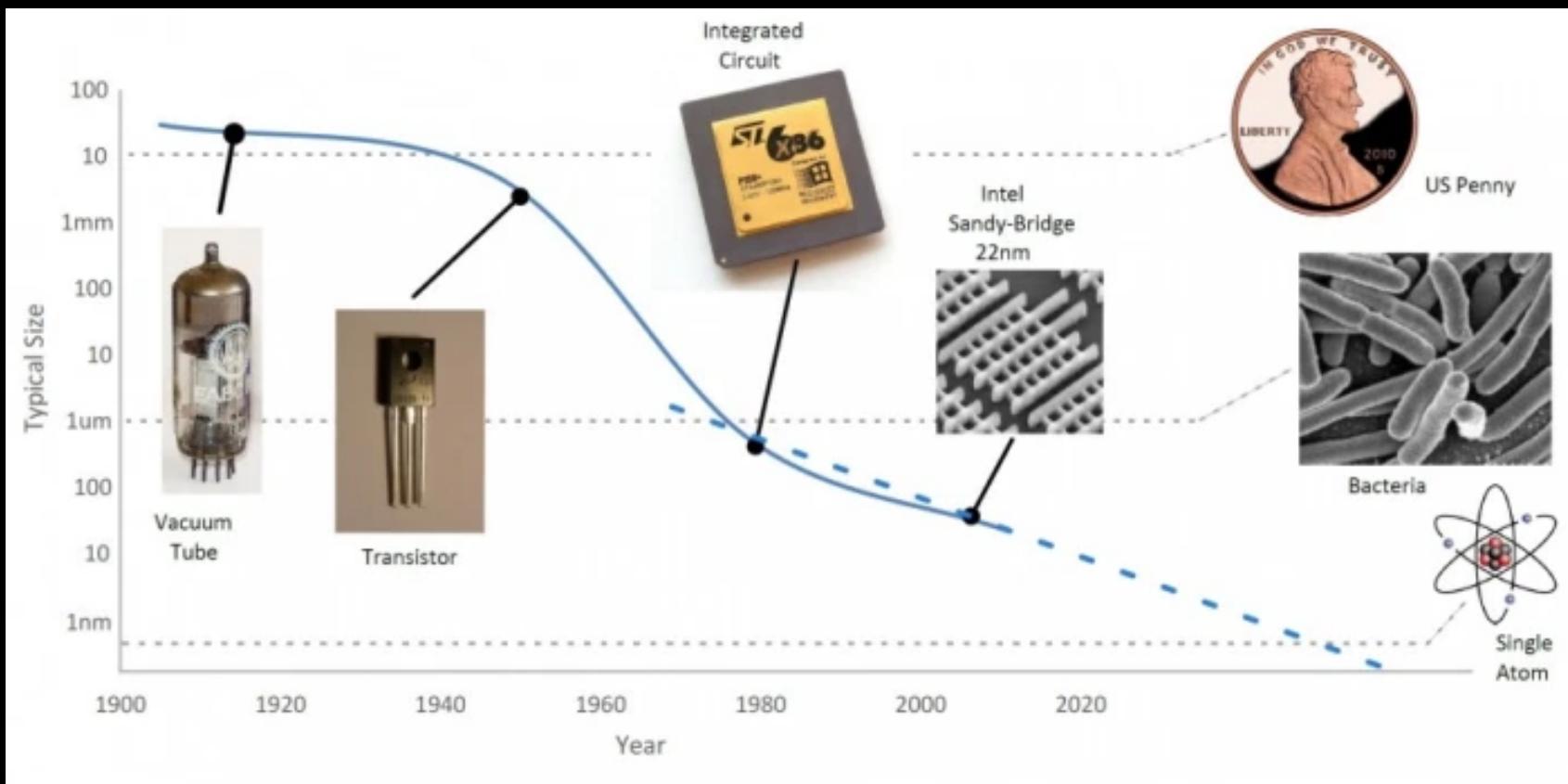
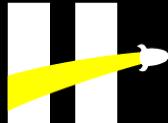




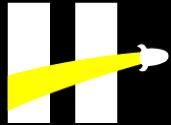
1937 - Adder



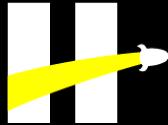
1937 - British Bombe
Alan Turing



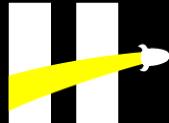
Ley de Moore.



Números Binarios



```
1 // Sistema UNARIO
2
3 I I I I I I I I
4
5 // utiliza un sólo tipo de símbolo. Su desventaja es que no permite simbolizar
6 // cómoda y rápidamente conjuntos con muchos elementos.
7
8 // Numeros ROMANOS
9
10 // Parecido al Unario, pero disponemos de más simbolos
11 // y operaciones implicitas
12
13 CXVII = cien + diez + cinco + uno + uno
14
15 MCMV = mil + (mil - cien) + cinco
16
17 // SISTEMAS POSICIONALES
18
19 // En estos sistemas, cada símbolo, además del número de unidades que representa
20 // considerado en forma aislada, tiene un significado o peso distinto
21 // según la posición que ocupa en el grupo de caracteres del que forma parte.
```



POSICIÓN

7^a

6^a

5^a

4^a

3^a

2^a

1^a

Unidad

Decena

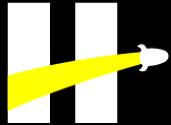
Centena

Unidad de Millar

Decena de Millar

Centena de Millar

Unidad de Millón



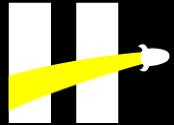
Símbolos del sistema

Cada sistema utiliza sus propios símbolos:

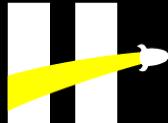
- Decimal: 0, 1, 2, 3, 4, 5, 6, 7, 8 y 9.
- Binario: 0 y 1.
- Quinario: 0, 1, 2, 3 y 4.
- Octal: 0, 1, 2, 3, 4, 5, 6 y 7.
- Hexadecimal: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E y F.

$$N = \sum_{i=-k}^{n-1} d_i \cdot 10^i$$

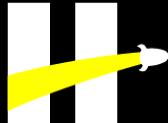
$$N = \sum_{i=-k}^n d_i \cdot 2^i$$



Representaciones



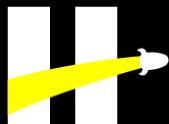
| Physical State | OFF | ON | ON | OFF | OFF | ON | ON | OFF |
|---------------------|-------------------------|-------|-------|-------|-------|-------|-------|-------|
| Binary Notation | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| Order of Magnitude | 2^7 | 2^6 | 2^5 | 2^4 | 2^3 | 2^2 | 2^1 | 2^0 |
| Decimal Value | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| Applicable Value | 0 | 64 | 32 | 0 | 0 | 4 | 2 | 0 |
| Total Decimal Value | $102 = 64 + 32 + 4 + 2$ | | | | | | | |



ASCII

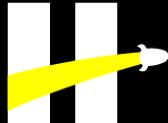
↗

| Binary | Decimal | Glyph | Binary | Decimal | Glyph |
|-----------|---------|-------|-----------|---------|-------|
| 0010 1110 | 46 | . | 0011 1010 | 58 | : |
| 0010 1111 | 47 | / | 0011 1011 | 59 | ; |
| 0011 0000 | 48 | o | 0011 1100 | 60 | < |
| 0011 0001 | 49 | I | 0011 1101 | 61 | = |
| 0011 0010 | 50 | z | 0011 1110 | 62 | > |
| 0011 0011 | 51 | 3 | 0011 1111 | 63 | ? |
| 0011 0100 | 52 | 4 | 0100 0000 | 64 | @ |
| 0011 0101 | 53 | 5 | 0100 0001 | 65 | A |
| 0011 0110 | 54 | 6 | 0100 0010 | 66 | B |
| 0011 0111 | 55 | 7 | 0100 0011 | 67 | C |
| 0011 1000 | 56 | 8 | 0100 0100 | 68 | D |
| 0011 1001 | 57 | 9 | 0100 0101 | 69 | E |

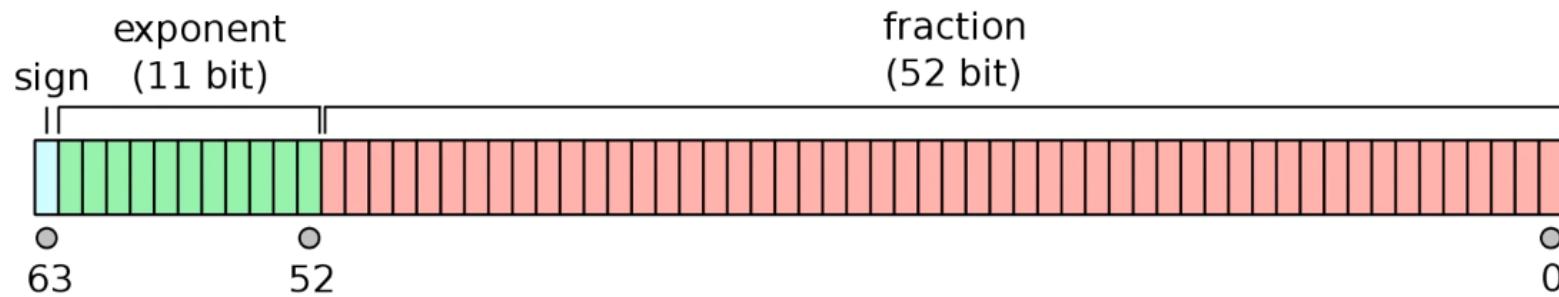


Unicode

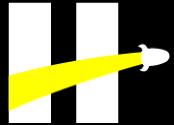
| Number of bytes | Bits for code point | First code point | Last code point | Byte 1 | Byte 2 | Byte 3 | Byte 4 |
|-----------------|---------------------|------------------|-----------------|-----------|------------|------------|------------|
| 1 | 7 | U+0000 | U+007F | 0xxxxxxxx | | | |
| 2 | 11 | U+0080 | U+07FF | 110xxxxx | 10xxxxxxxx | | |
| 3 | 16 | U+0800 | U+FFFF | 1110xxxx | 10xxxxxxxx | 10xxxxxxxx | |
| 4 | 21 | U+10000 | U+10FFFF | 11110xxx | 10xxxxxxxx | 10xxxxxxxx | 10xxxxxxxx |



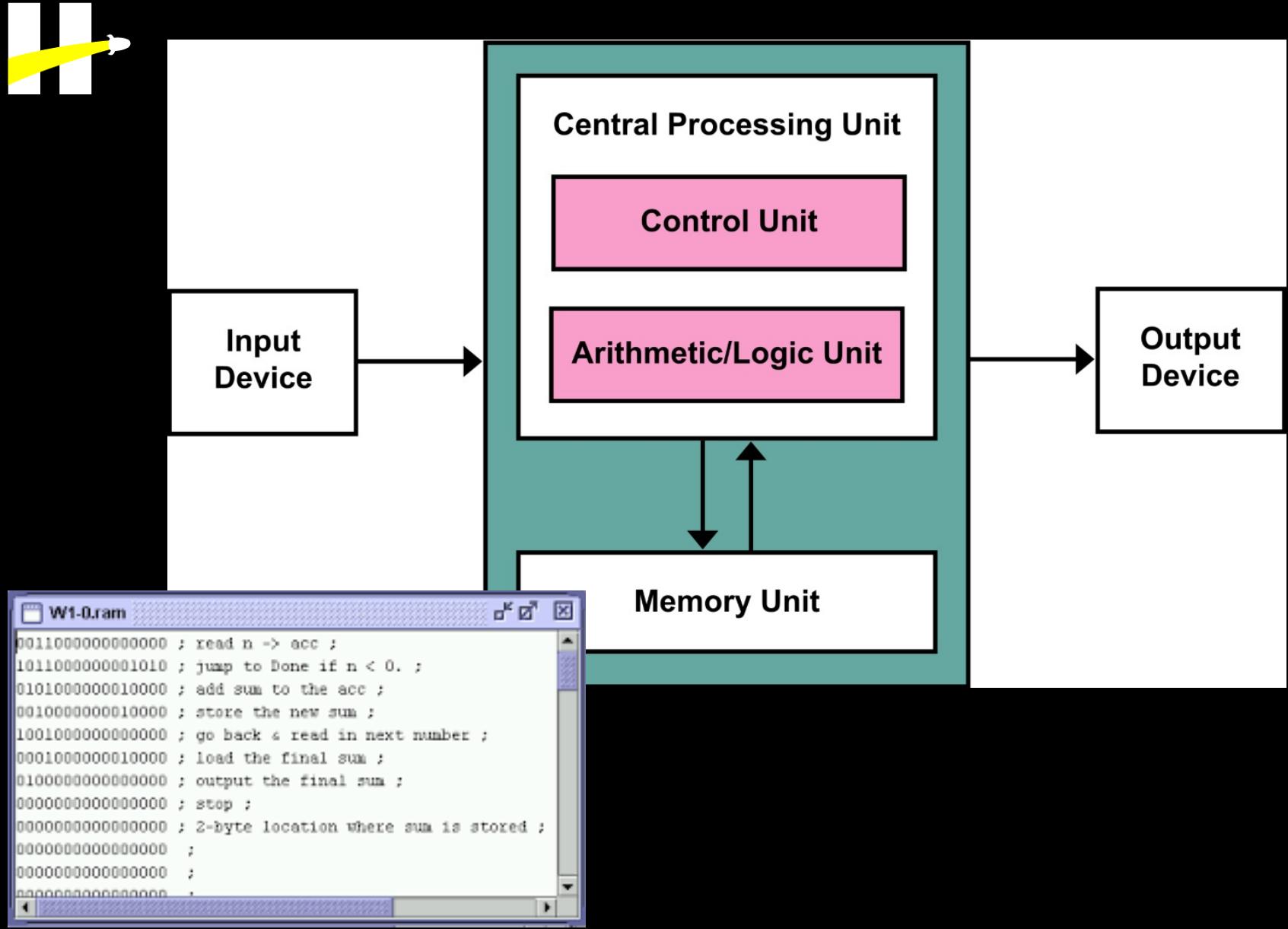
IEEE 754: Floating-Point Signed Double

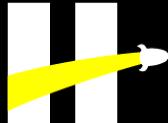


[https://www.h-
schmidt.net/FloatConverter/IEEE754.html](https://www.h-schmidt.net/FloatConverter/IEEE754.html)

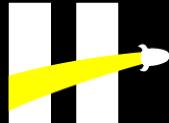


Lenguaje de Máquina

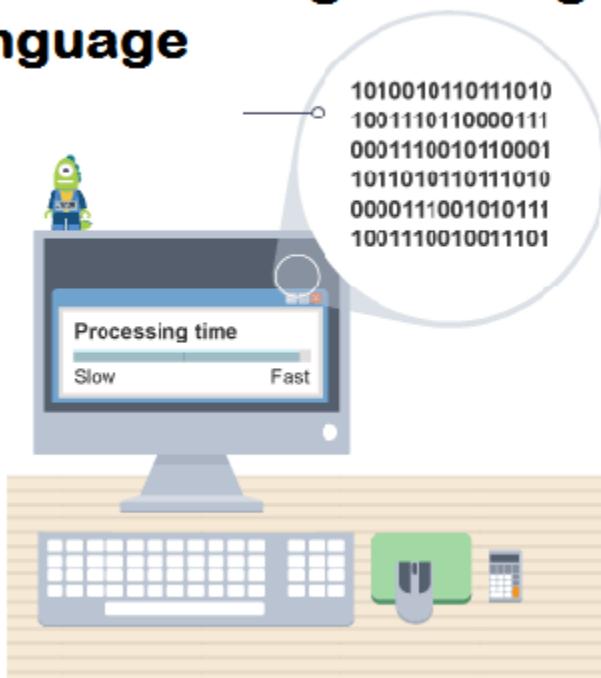




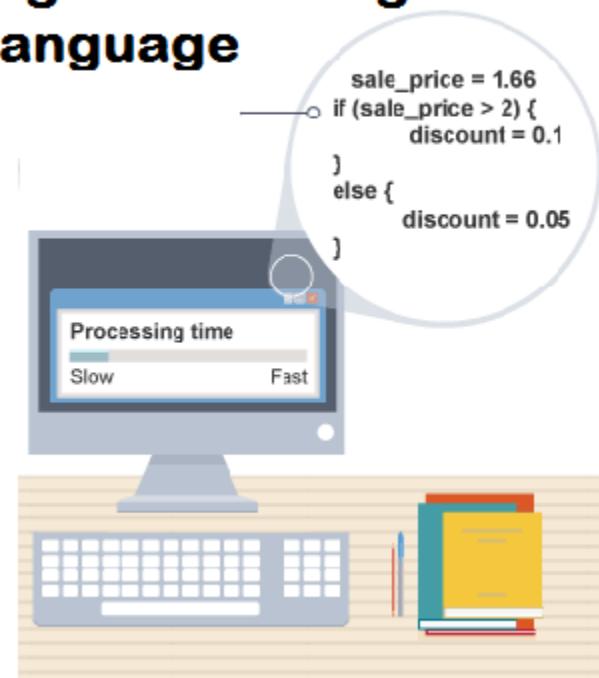
| | op. code | source register | target reg. | destination reg. | shift amount | function type |
|-------------|---|-----------------|-------------|------------------|--------------|---------------|
| instruction | 000000 | 00001 | 00010 | 00110 | 00000 | 100000 |
| hex | 0x0 | 0x1 | 0x2 | 0x6 | 0x0 | 0x20 |
| decimal | 0 | 1 | 2 | 6 | 0 | 32 |
| meaning | arithmetic | register 1 | register 2 | register 6 | no offset | addition |
| English | "take the value in register 1, add the value in register 2, place the result in register 6" | | | | | |
| assembly | add \$t6, \$t1, \$t2 | | | | | |

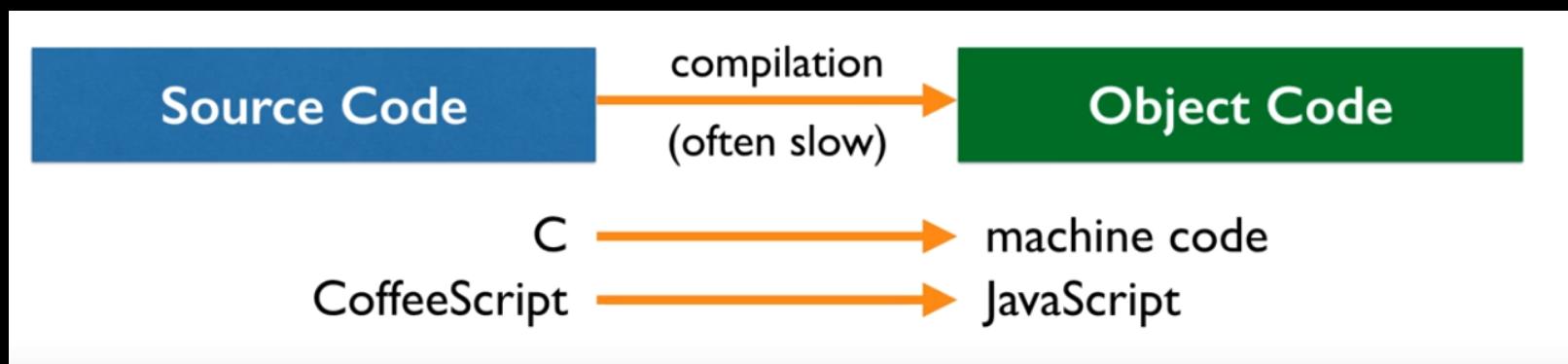


Low Level Programming Language

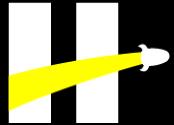


High Level Programming Language

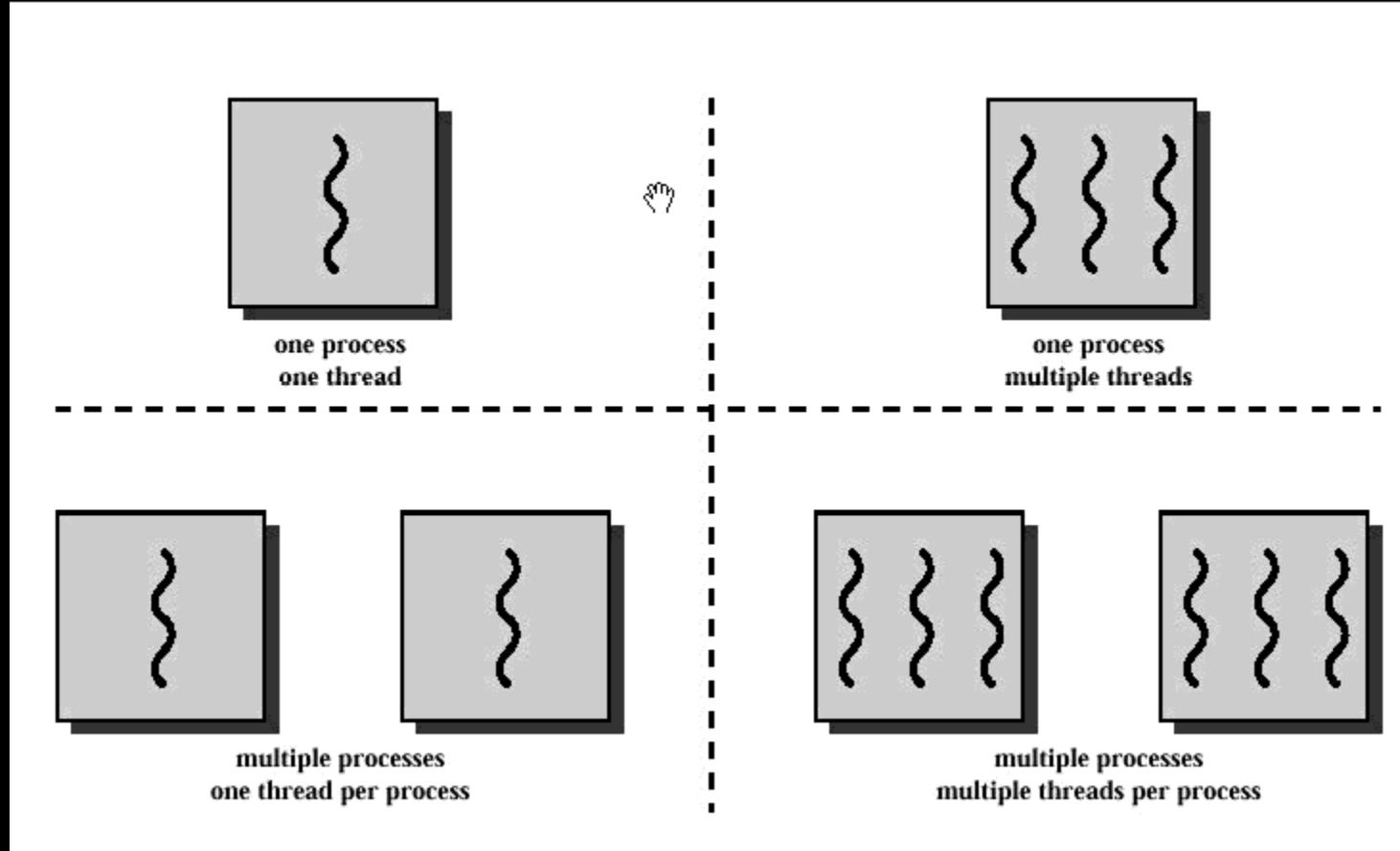
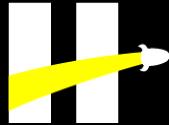




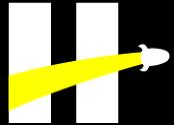
HENRY



Parte I: Entiendiendo JS



Single Threaded y Sincrónico

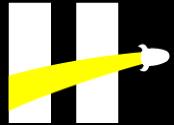


Syntax Parser

Lexical Environment

```
1 function hola(){
2     var foo = 'Hola!';
3 }
4
5 var bar = 'Chao';
```

Por ejemplo, para el interprete las dos declaraciones de variable
del arriba tendrán significados muy distintos. Si bien la
operación es igual en los dos (asignación) al estar en lugares
distintos (una dentro de una función y la otra no) el interprete
las parseará de forma distinta



Execution Context

```
// global context

var sayHello = 'Hello';

function person() {          // execution context

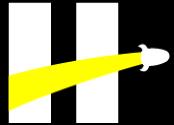
    var first = 'David',
        last = 'Shariff';

    function firstName() { // execution context
        return first;
    }

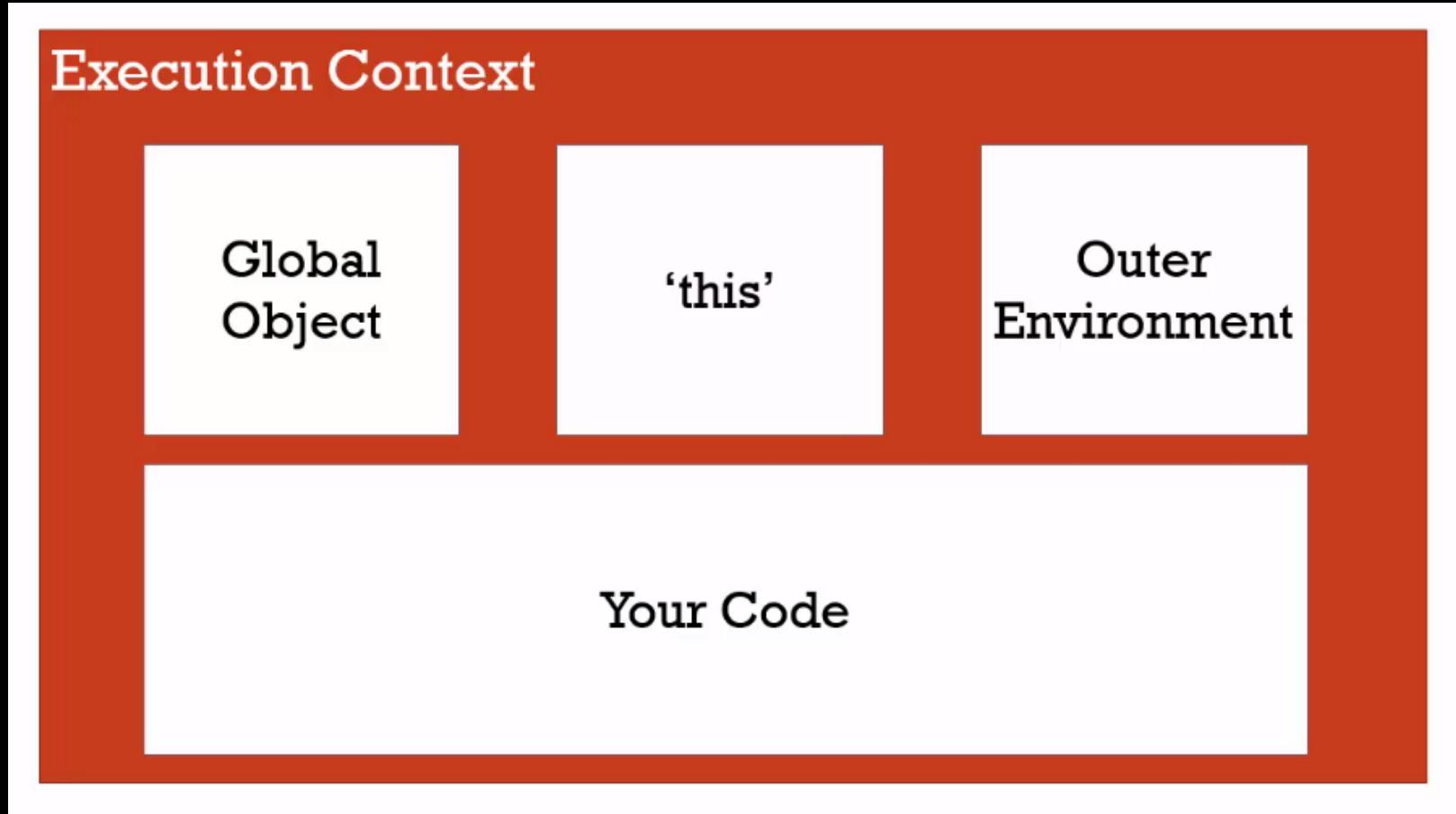
    function lastName() { // execution context
        return last;
    }

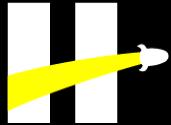
    alert(sayHello + firstName() + ' ' + lastName());
}
```

El contexto de ejecución contiene información sobre qué código se está ejecutando en cada momento. Además de mantener el código que tiene que ejecutar, también mantiene más información sobre de donde se invocó ese código, en qué lexical enviroment está, etc...



Execution Context

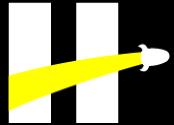




Hoisting

```
● ● ●  
1 bar();  
2 console.log(foo);  
3  
4 var foo = 'Hola, me declaro';  
5 function bar() {  
6     console.log('Soy una función');  
7 }
```

El hosting es el primer ejemplo de las *cosas extras* que hace el interprete sin que nosotros se lo pidamos. Si no las conocemos, nos puede pasar que veamos comportamientos extraños y no sepamos de donde vienen (como que podamos usar funciones que no hemos declarado antes de invocarlas!!)



Execution Stack



```
1 function b() {  
2   console.log('B!')  
3 };  
4  
5 function a() {  
6   // invoca a la función b  
7   b();  
8 }  
9  
10 //invocamos a  
11 a();
```

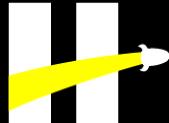
b()

Execution Context
(create and execute)

a()

Execution Context
(create and execute)

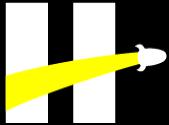
Global Execution Context
(created and code is executed)



Scope

```
1 var global = 'Hola!';
2
3 function a() {
4     // como no hay una variable llamada global en este contexto,
5     // busca en el outer que es el global
6     console.log(global);
7     global = 'Hello!'; // cambia la variable del contexto global
8 }
9
10 function b(){
11     // declaramos una variable global en nuestro contexto
12     // esta es independiente
13     var global = 'Chao';
14     console.log(global);
15 }
16
17 a(); // 'Hola!'
18 b(); // 'Chao'
19 console.log(global); // 'Hello'
```

Para esto vamos a introducir el término scope, este es el set de variable, objeto y funciones al que tenemos acceso en determinado contexto.

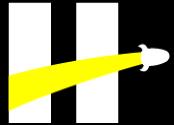


Scope

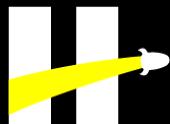


```
1 var global = 'Hola!';
2
3 function b(){
4     var global = 'Chao';
5     console.log(global); // Chao
6     function a() {
7         // como no hay una variable llamada global en este contexto,
8         // busca en el outer que es scope de b;
9         console.log(global); //Chao
10        global = 'Hello!'; // cambia la variable del contexto de b()
11    }
12    a();
13 }
14
15 //a(); Ya no puedo llamar a a desde el scope global, acá no existe.
16 b();
17 console.log(global); // 'Hola!'
```

Cada contexto maneja sus propias variables,
y son independientes de los demás



Tipos de Datos



Static vs Dynamic Typing

Java

Static typing:

| | |
|----------------|------------------------------|
| String name; | Variables have types |
| name = "John"; | Values have types |
| name = 34; | Variables cannot change type |

JavaScript

Dynamic typing:

| | |
|----------------|-----------------------------------|
| var name; | Variables have no types |
| name = "John"; | Values have types |
| name = 34; | Variables change type dynamically |

©johannsander



Operadores



```
1 var a = 2 + 3; // 5
2
3 function suma(a,b){
4     return a + b;
5     // usamos el mismo operador como ejemplo
6     // Si no deberíamos hacer sumas binarias!
7 }
8 var a = suma(2,3) // 5
```

| Infix Expression | Prefix Expression | Postfix Expression |
|------------------|-------------------|--------------------|
| A + B | + A B | A B + |
| A + B * C | + A * B C | A B C * + |

Un operador no es otra cosa que una función

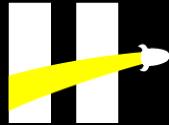


Precedencia de Operadores y Asociatividad

La *precedencia de operadores* es básicamente el orden en que se van a llamar las funciones de los operadores.

La *Asociatividad de operadores* es el orden en el que se ejecutan los operadores cuando tienen la misma precedencia, es decir, de izquierda a derecha o de derecha a izquierda.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precidence#Table

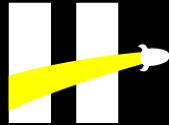


Coerción de Datos

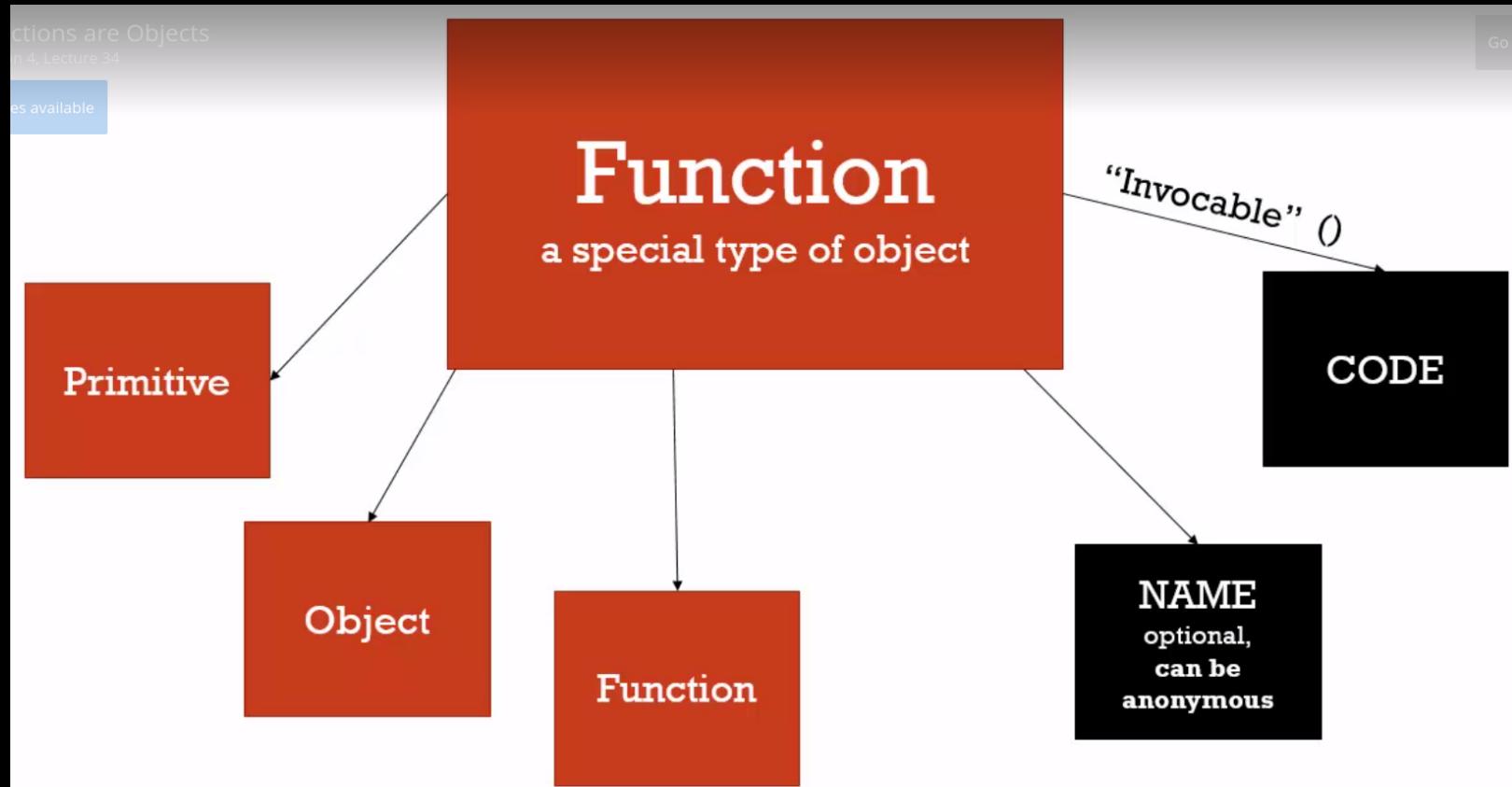


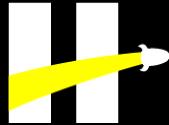
```
1
2
3 Number('3') // devuelve el número 3. Obvio!
4 Number(false) // devuelve el número 0. mini Obvio.
5 Number(true) // devuelve el número 1. menos mini Obvio.
6 Number(undefined) // devuelve `NaN`. No era obvio, pero tiene sentido.
7 Number(null) // devuelve el número 0.
8 // WTFFFF!!! porqueEE no debería ser `NaN` ??
```

Tabla



First Class Functions

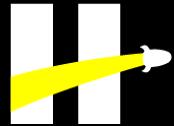




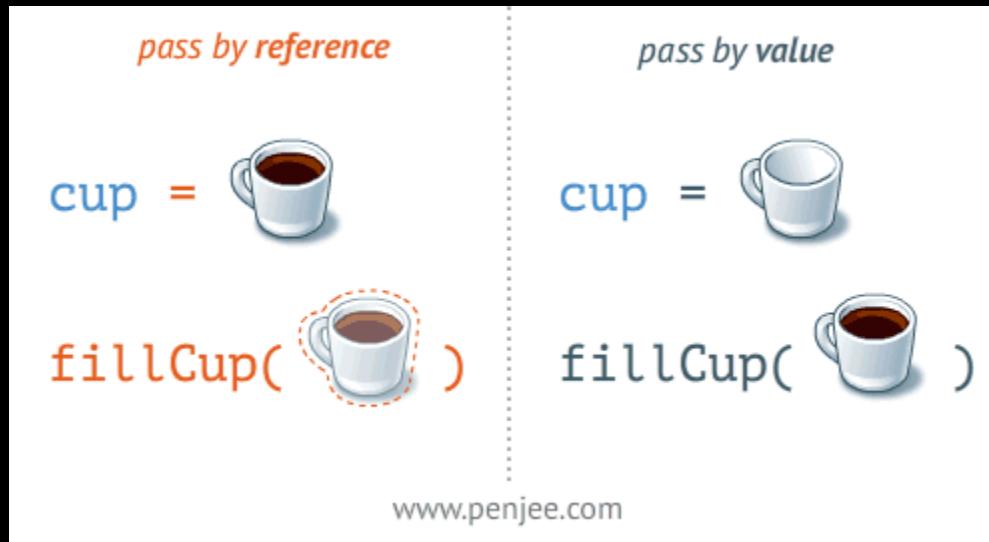
Expresiones y Statements

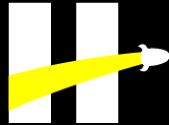
```
● ● ●

1 // Expresion
2
3 1 + 1;
4 a = 3;
5
6 //Statement
7
8 if (condicion) {
9     // bloque de código
10 }
11
12 // function statement
13
14 function saludo(){
15     console.log('hola');
16 }
17
18 // function expression
19
20 var saludo = function(){
21     console.log('Hola!');
22 }
23
24 console.log(function(){
25     //hola;
26 })
```

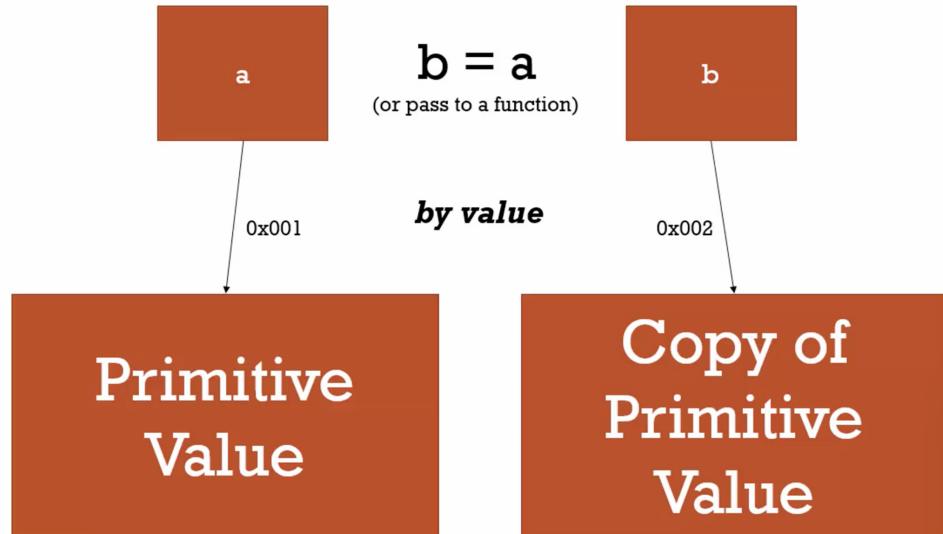


Valor y Referencia

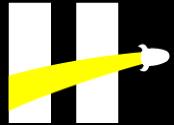




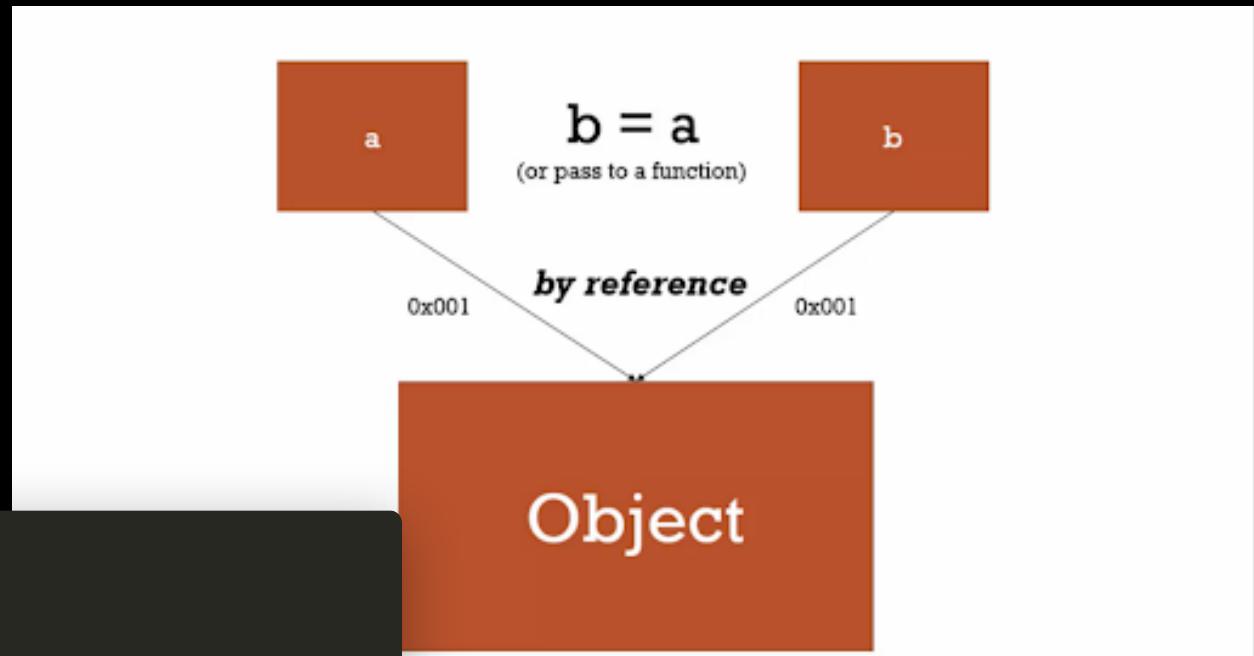
Valor y Referencia



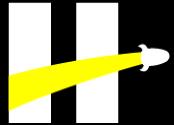
```
1 var a = 1;
2 var b = 2;
3
4 a = b;
5
6 b = 3;
7
8 console.log(a) // 2
9 console.log(b) // 3
```



Valor y Referencia



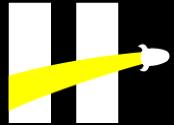
```
1 var a;
2 var b = { nombre : 'hola'};
3
4 a = b ;
5
6 b.nombre = 'Chao';
7
8 console.log(a.nombre); // 'Chao'
9 // Cuando se hizo la asignación se pasó
10 // la referencia de b, por lo tanto
11 // cuando cambiamos la propiedad nombre
12 // de b, se ve reflejado en a
13 // porque ambas variables "apuntan"
14 // al mismo objeto en memoria
```



This

Contexto global inicial

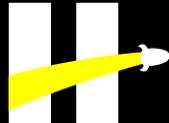
```
1 // En el browser esto es verdad:  
2 console.log(this === window); // true  
3  
4 this.a = 37;  
5 console.log(window.a); // 37
```



This

En el contexto de una función

```
1 function f1(){
2   return this;
3 }
4
5 f1() === window; // global object
```



This

Cómo un método de un objeto



```
1 var o = {  
2   prop: 37,  
3   f: function() {  
4     return this.prop;  
5   }  
6 };  
7  
8 console.log(o.f()); // logs 37  
9 // this hace referencia a `o`  
10  
11  
12 var o = {prop: 37};  
13  
14 // declaramos la función  
15 function loguea() {  
16   return this.prop;  
17 }  
18  
19 //agregamos la función como método del objeto `o`  
20 o.f = loguea;  
21  
22 console.log(o.f()); // logs 37  
23 // el resultado es le mismo!
```



This

Cómo un método de un objeto

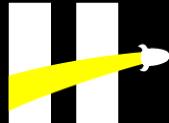


```
1 var obj = {  
2   nombre: 'Objeto',  
3   log: function(){  
4     this.nombre = 'Cambiado'; // this se refiere a este objeto, a `obj`  
5     console.log(this) // obj  
6   }  
7   var cambia = function( str ){  
8     this.nombre = str; // Uno esperaria que this sea `obj`  
9   }  
10  cambia('Hoola!!');  
11  console.log(this);  
12}  
13 }  
14 }
```

Prácticamente, no podemos saber a ciencia cierta qué valor va a tomar el keyword hasta

el momento de ejecución de una función. Porque depende fuertemente de cómo haya

sido ejecutada.

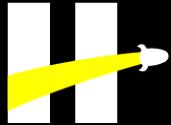


This

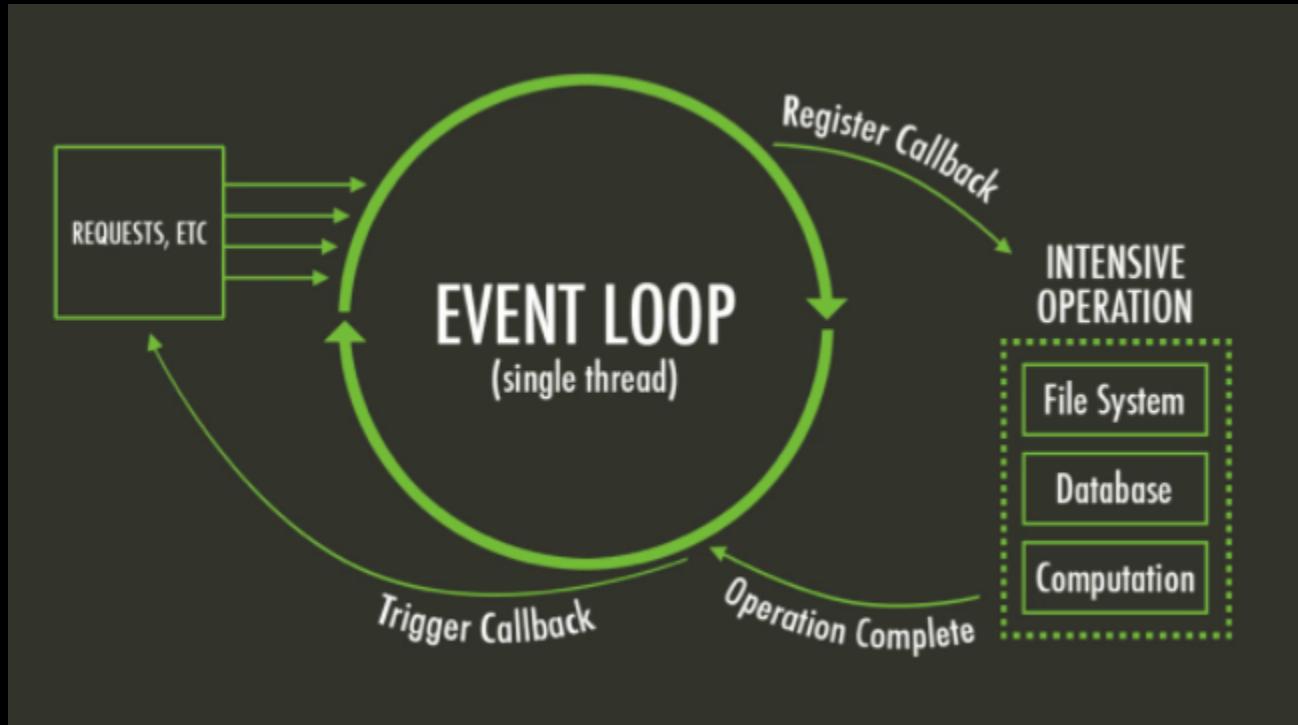
Cómo un método de un objeto

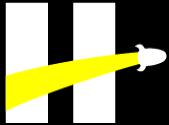


```
1 var obj = {  
2   nombre: 'Objeto',  
3   log : function(){  
4     this.nombre = 'Cambiado'; // this se refiere a este objeto, a `obj`  
5     console.log(this) // obj  
6  
7     var that = this; // Guardo la referencia a this  
8  
9     var cambia = function( str ){  
10       that.nombre = str; // Uso la referencia dentro de esta funcion  
11     }  
12  
13     cambia('Hoola!!');  
14     console.log(this);  
15   }  
16 }
```



Event Loop



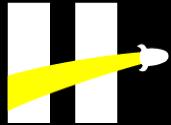


Event Loop



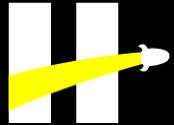
```
1
2
3 function saludarMasTarde(){
4     var saludo = 'Hola';
5
6     setTimeout( function(){
7         console.log(saludo);
8     },3000)
9 }
10
11 saludarMasTarde();
```

< DEMO />

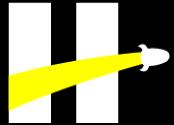


Event Loop

< Ejemplo />



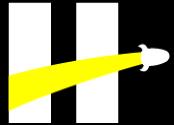
Parte II: Closures



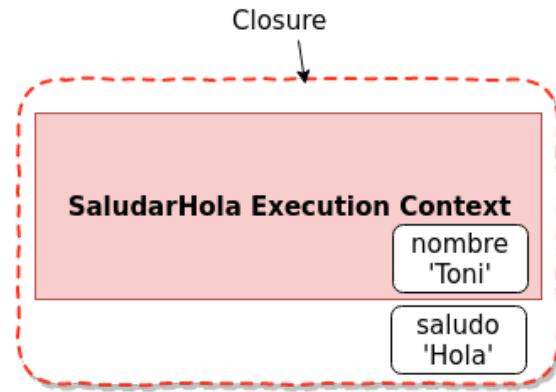
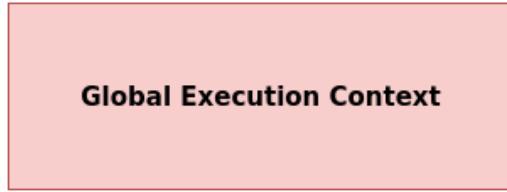
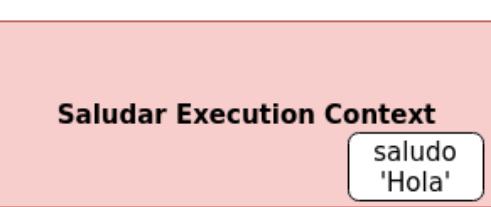
Closures

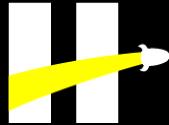


```
1 function saludar( saludo ){
2     return function( nombre ){
3         console.log(saludo + ' ' + nombre);
4     }
5 }
6
7 var saludarHola = saludar('Hola'); // Esto devuelve una función
8
9 saludarHola('Toni'); // 'Hola Toni'
```



Closures

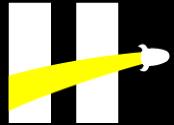




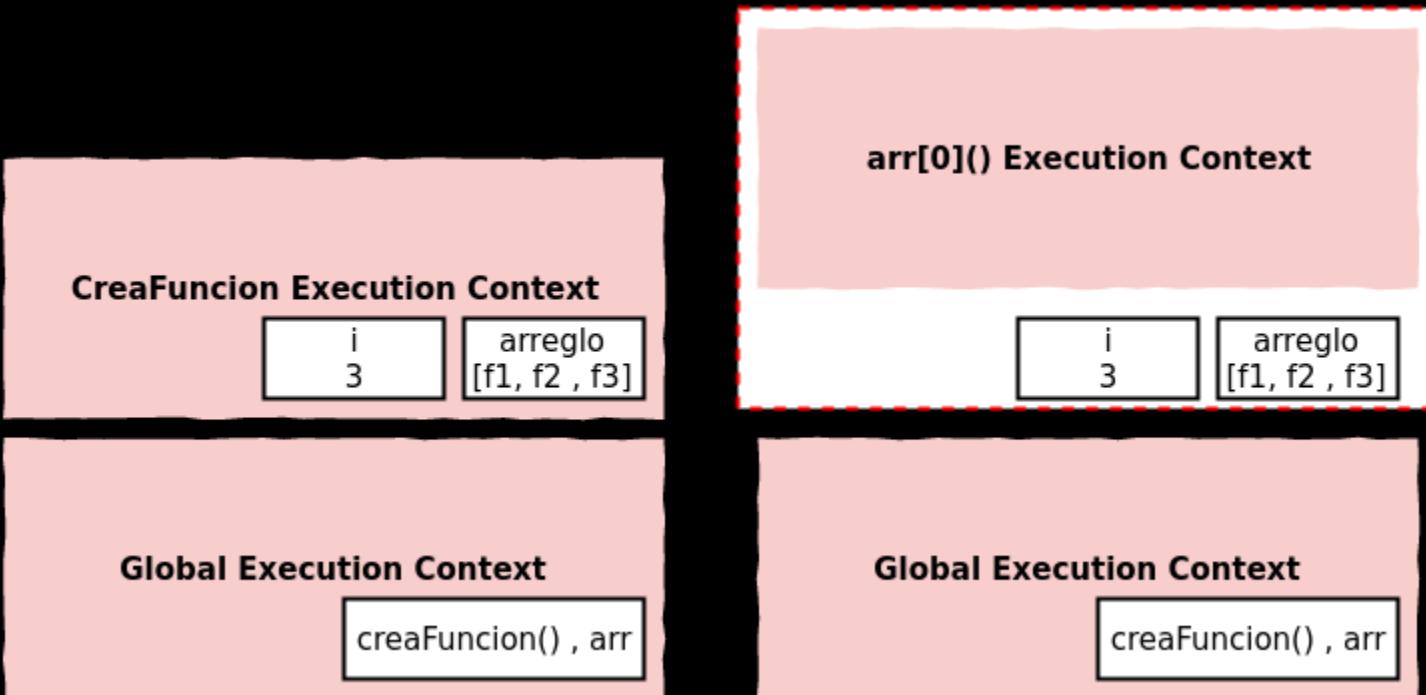
Closures

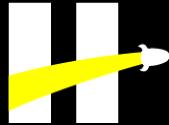


```
1 var creaFuncion = function(){
2     var arreglo = [ ];
3
4     for ( var i=0; i < 3; i++){
5         arreglo.push(
6             function(){
7                 console.log(i);
8             }
9         )
10    }
11    return arreglo;
12 }
13
14 var arr = creaFuncion();
15
16 arr[0]() // 3 sale un 3, qué esperaban ustedes??
17 arr[1]() // 3
18 arr[2]() // 3
```



Closures

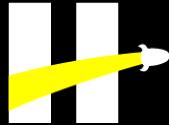




Closures



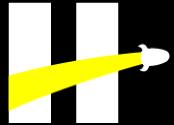
```
1 var creaFuncion = function(){
2     var arreglo = [];
3     for ( var i=0; i < 3; i++){
4         // IIFE
5             arreglo.push(
6                 (function(j){
7                     return function() {console.log(j);}
8                 }(i)))
9             )
10        }
11    return arreglo;
12 }
13
14 var arr = creaFuncion();
15
16 arr[0]() // 0
17 arr[1]() // 1
18 arr[2]() // 2
```



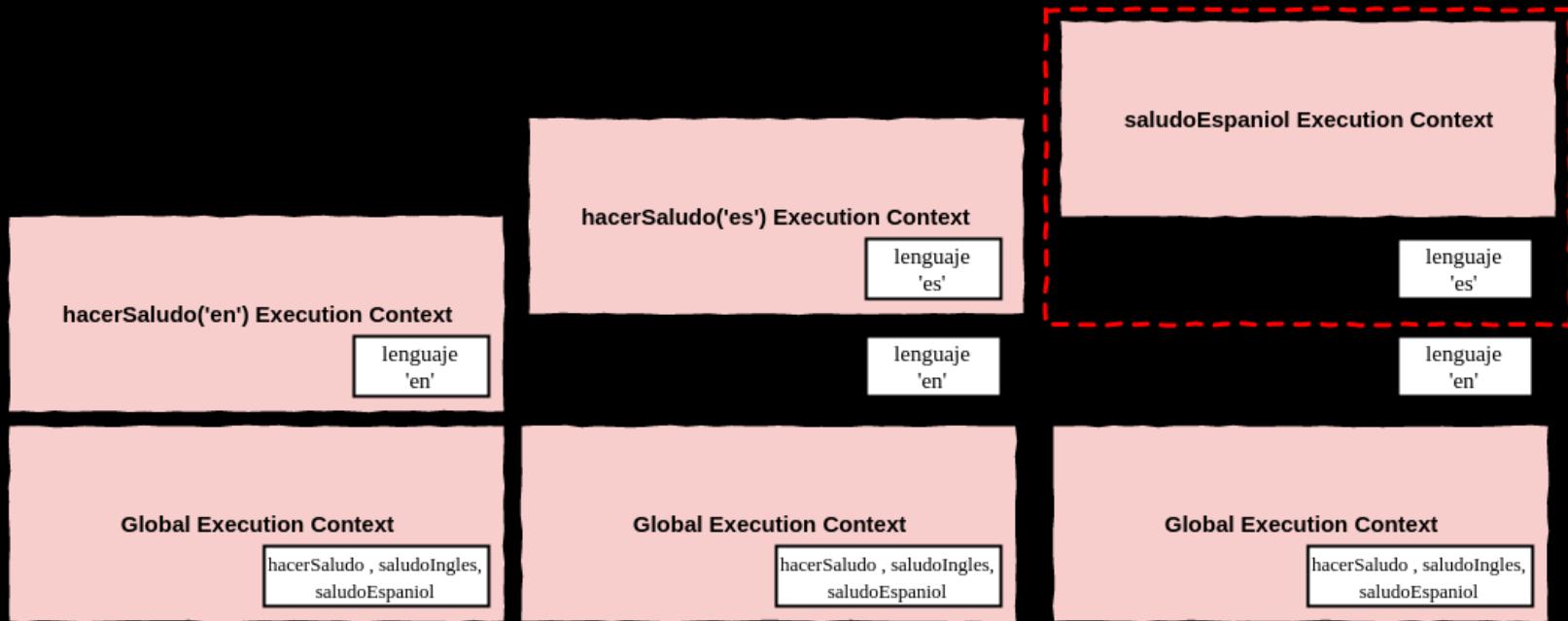
Closures

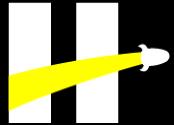


```
1 function hacerSaludo( lenguaje ){
2     if ( lenguaje === 'en'){
3         return function(){
4             console.log('Hi!');
5         }
6     }
7
8     if ( lenguaje === 'es'){
9         return function(){
10            console.log('Hola!');
11        }
12    }
13 }
14
15 var saludoIngles = hacerSaludo('en');
16 var saludoEspaniol = hacerSaludo('es');
```

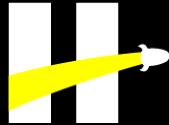


Closures





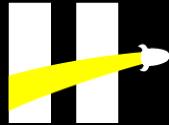
Bind, Call & Apply



Bind, Call & Apply

```
 1 var persona = {  
 2     nombre: 'Guille',  
 3     apellido: 'Aszyn',  
 4 }  
 5  
 6 var logNombre = function(){  
 7     console.log(this.nombre);  
 8 }  
 9  
10 var logNombrePersona = logNombre.bind(persona);  
11 // el primer parametro de bind es el this!  
12 logNombrePersona();  
13  
14 // BIND DEVUELVE UNA FUNCION!
```

Cuando vimos el keyword this, dijimos que el interprete era el que manejaba el valor de este. Bueno, esto no es del todo cierto, hay una serie de funciones que nos van a permitir poder setear nosotros el keyword this.



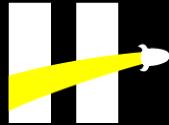
Bind, Call & Apply



```
1 function multiplica(a, b){  
2     return a * b;  
3 }  
4  
5 var multiplicaPorDos = multiplica.bind(this, 2);  
6 // el Bind le `bindeó` el 2 al argumento a.  
7 // y devolvió una función nueva con ese parámetro bindeado.
```

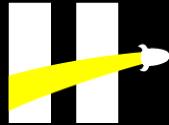
Bind acepta más parámetros, el primero siempre es el `this`, los siguientes sirven para bindear parámetros de una función.

Esto se conoce como function currying.



Bind, Call & Apply

```
● ● ●  
1 var persona = {  
2     nombre: 'Guille',  
3     apellido: 'Aszyn',  
4 }  
5  
6 var logNombre = function(){  
7     console.log(this.nombre);  
8 }  
9  
10 // el primer parametro de call es el this!  
11 logNombre.call(persona);  
12  
13 // Call hace lo mismo que Bind, solo que invoca la función,  
14 // no devuelve una nueva.  
15 // tambien bindea argumentos!  
16 //  
17 var logNombre = function(arg1, arg2){  
18     console.log(arg1 + ' ' + this.nombre + ' ' + arg2);  
19 }  
20  
21 logNombre.call(persona, 'Hola', ', Cómo estas?');  
22  
23 ////Hola Guille, Cómo estas?
```

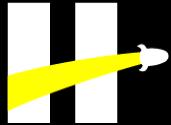


Bind, Call & Apply

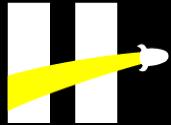


```
1 // Apply es igual a call, solo que el segundo argumento es un
2 // arreglo.
3
4 var logNombre = function(arg1, arg2){
5     console.log(arg1 + ' ' + this.nombre + ' ' + arg2);
6 }
7
8 logNombre.apply(persona, ['Hola', ' ', Cómo estas?]);
9 //Hola Guille , Cómo estas?
```

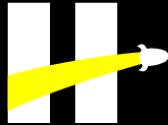
HENRY



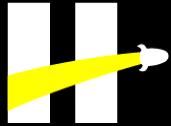
Recursion



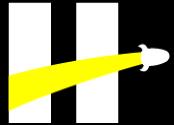
Para aprender la recursión, primero hay que aprender la recursión.



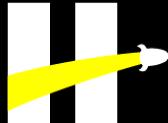
```
1 function factorial(x)
2 {
3     if (x > -1 && x < 2) return 1;    // Cuando -1 < x < 2
4         // devolvemos 1 puesto que 0! = 1 y 1! = 1
5     else if (x < 0) return 0;        // Error no existe factorial de números negativos
6     return x * factorial(x - 1);    // Si x >= 2 devolvemos el producto de x por el factorial de x - 1
7 }
```



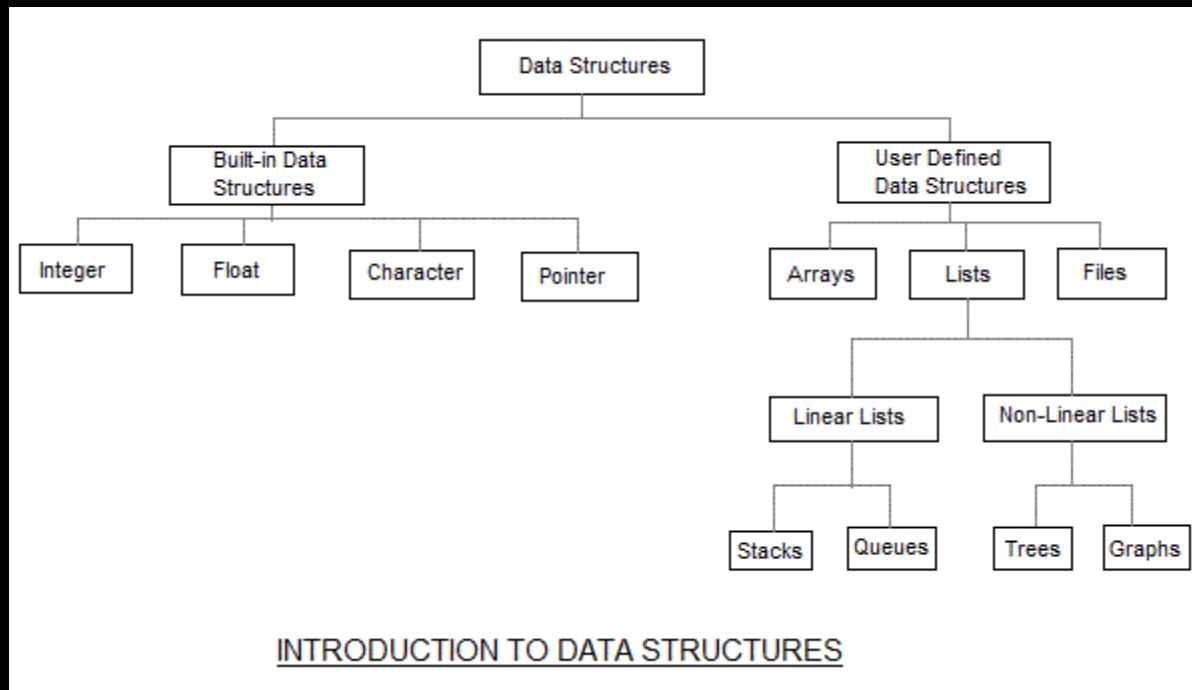
< Demo />

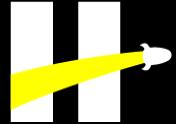


Estructuras de Datos - Parte I



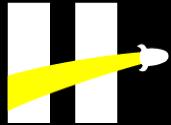
Cuando hablamos a estructura de Datos nos referimos a cómo organizamos los datos cuando programamos. Básicamente, este tema trata de encontrar formas particulares de organizar datos de tal manera que puedan ser utilizados de manera eficiente.





Arreglos

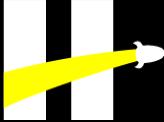
| Index | 0 | 1 | 2 | 3 | 4 |
|---------|---|---|---|---|---|
| | H | e | l | l | o |
| Address | 0x23451 0x23452 0x23453 0x23454 0x23455 | | | | |



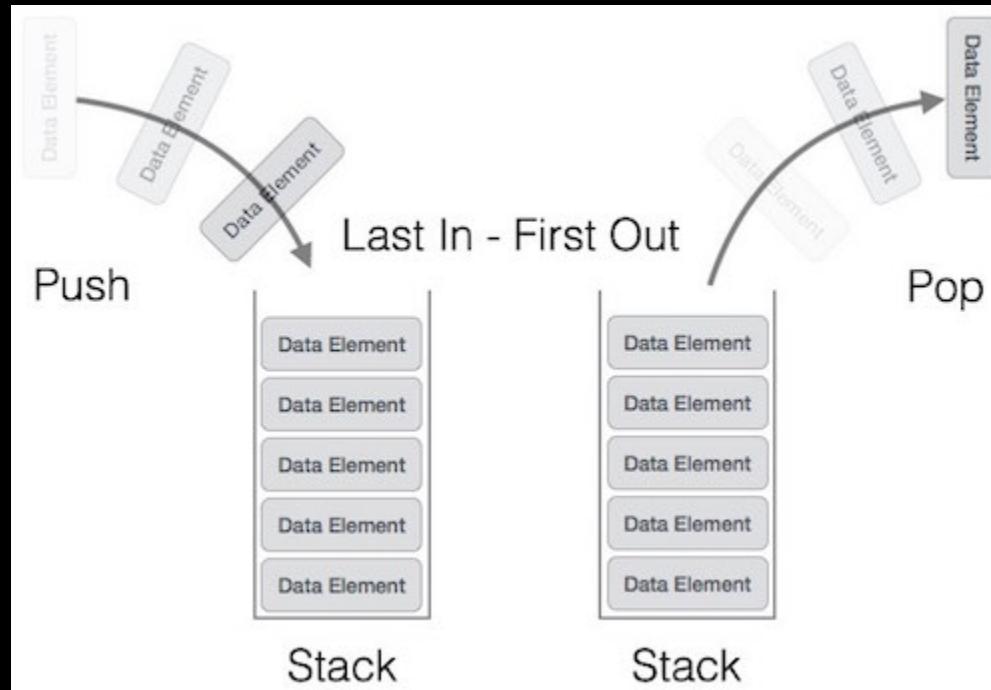
Sets

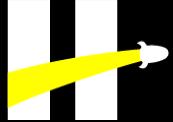


```
1 var arreglo = [1,2,3,4,4,5,5,1,2]
2 var set1    = new Set(arreglo)
3 console.log(arreglo) // [ 1, 2, 3, 4, 4, 5, 5, 1, 2 ]
4 console.log(set1)   // Set { 1, 2, 3, 4, 5 }
```



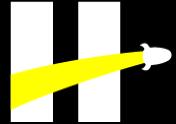
Pilas (Stacks)



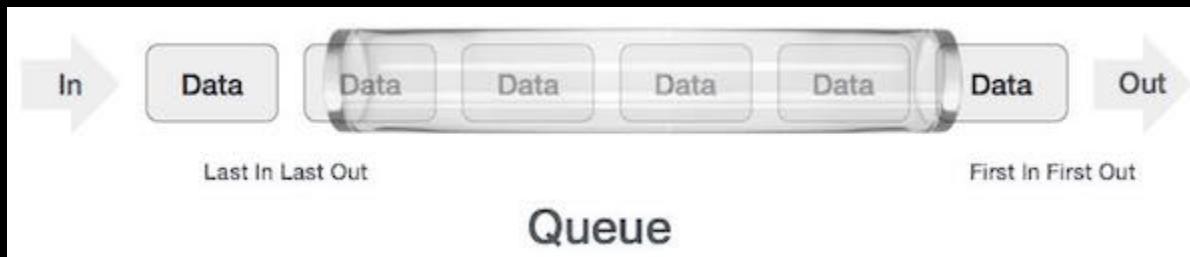


Pilas (Stacks)

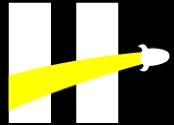
```
1 var stack = [ ];  
2 stack.push(1);           // la pila es [1]  
3 stack.push(10);          // la pila es ahora [1, 10]  
4 var i = stack.pop();    // la pila [1]  
5 console.log(i);         // muestra 10
```



Colas (Queue)



```
1 var queue = [];
2 queue.push(1);           // la cola es [1]
3 queue.push(2);           // la cola es [1, 2]
4 var i = queue.shift();  // la cola es [2]
5 console.log(i);         // muestra 1
```



Estructuras de Datos - Parte II



Listas Enlazadas



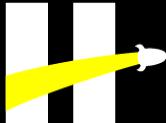
● ● ●

```
1 function Node(data) {
2     this.data = data;
3     this.next = null;
4 }
5
6 function List() {
7     this._length = 0;
8     this.head = null;
9 }
```



Listas Enlazadas

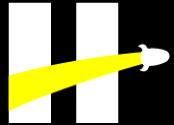
- *Iterar sobre la lista:* Recorrer la lista viendo sus elementos o hasta que econtremos el elemento deseado.
- *Insertar un nodo:* La operación va a cambiar según el lugar donde querramos insertar el nodo nuevo:
 - Al principio de la lista.
 - En el medio de la lista.
 - Al final de la lista.
- *Sacar un nodo:*
 - Del principio de la lista.
 - Del medio de la lista.



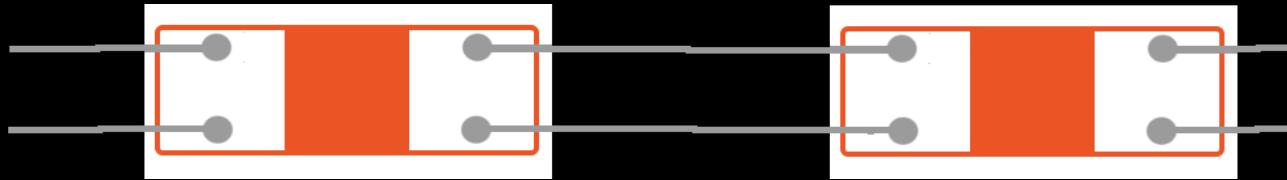
Listas Enlazadas



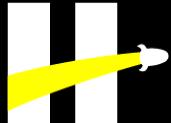
```
1 List.prototype.add = function(data) {
2     var node = new Node(data),
3     current = this.head;
4     // Si está vacía
5     if (!current) {
6         this.head = node;
7         this._length++;
8         return node;
9     }
10    // Si no está vacía, recorro hasta encontrar el último
11    while (current.next) {
12        current = current.next;
13    }
14    current.next = node;
15    this._length++;
16    return node;
17 };
18
19 List.prototype.getAll = function(){
20     current = this.head //empezamos en la cabeza
21     if (!current){
22         console.log('La lista está vacía!')
23         return
24     }
25     while(current){
26         console.log(current.data);
27         current = current.next;
28     }
29     return
30 };
```



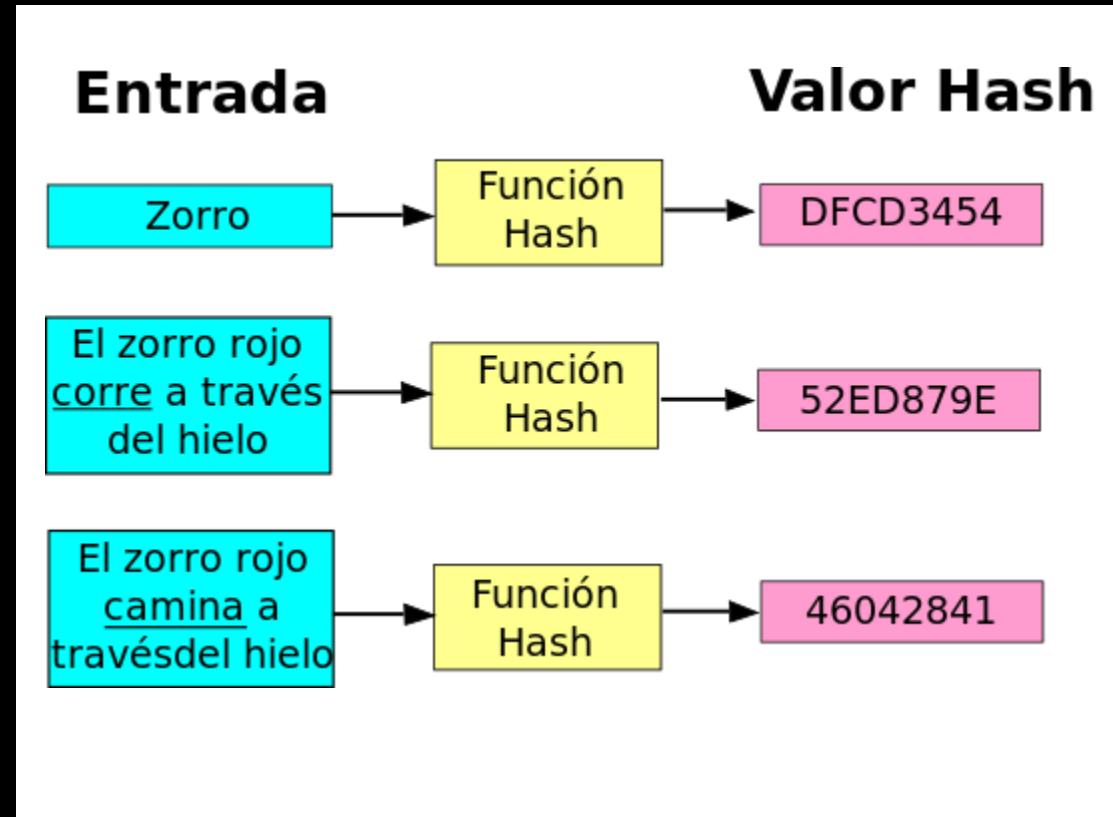
Listas Dblemente Enlazadas

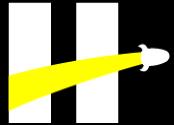


En la lista que vimos antes, sólo podemos recorrer la lista en un solo sentido. En algunos casos nos puede servir recorrer la lista en los dos sentidos, para tales casos lo que vamos a usar es una lista doblemente enlazada

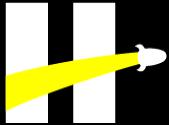


Hash Table

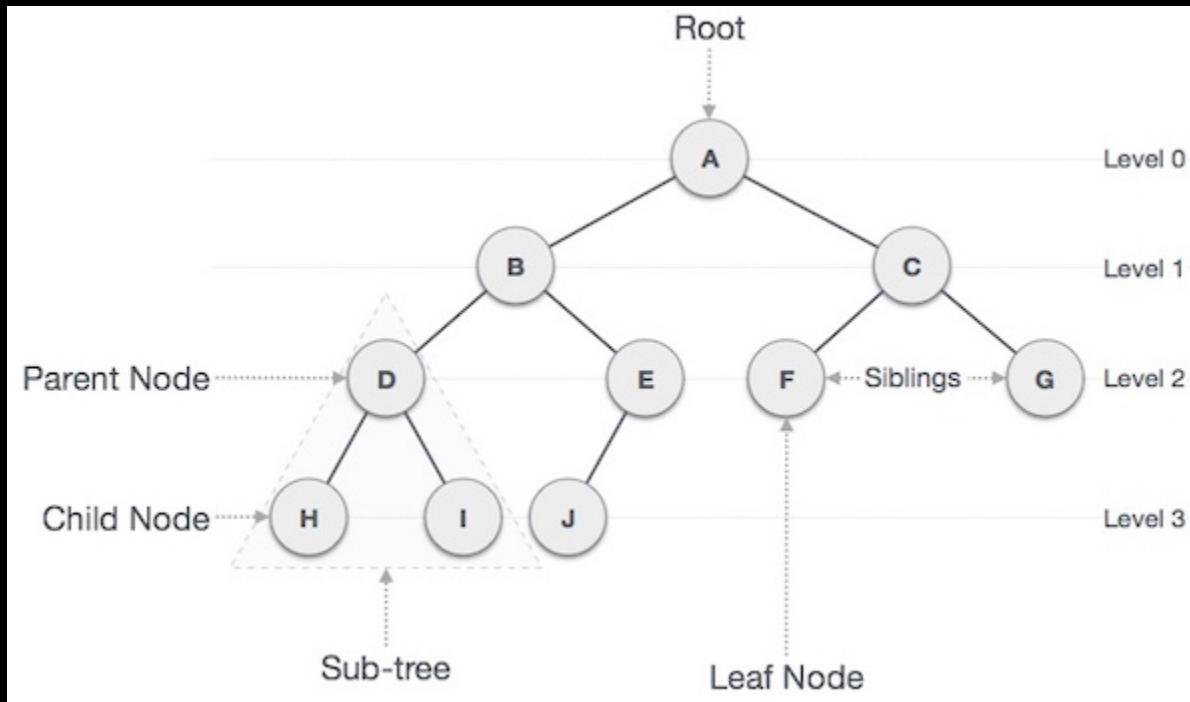


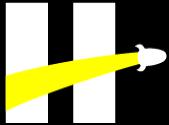


Estructuras de Datos - Parte III

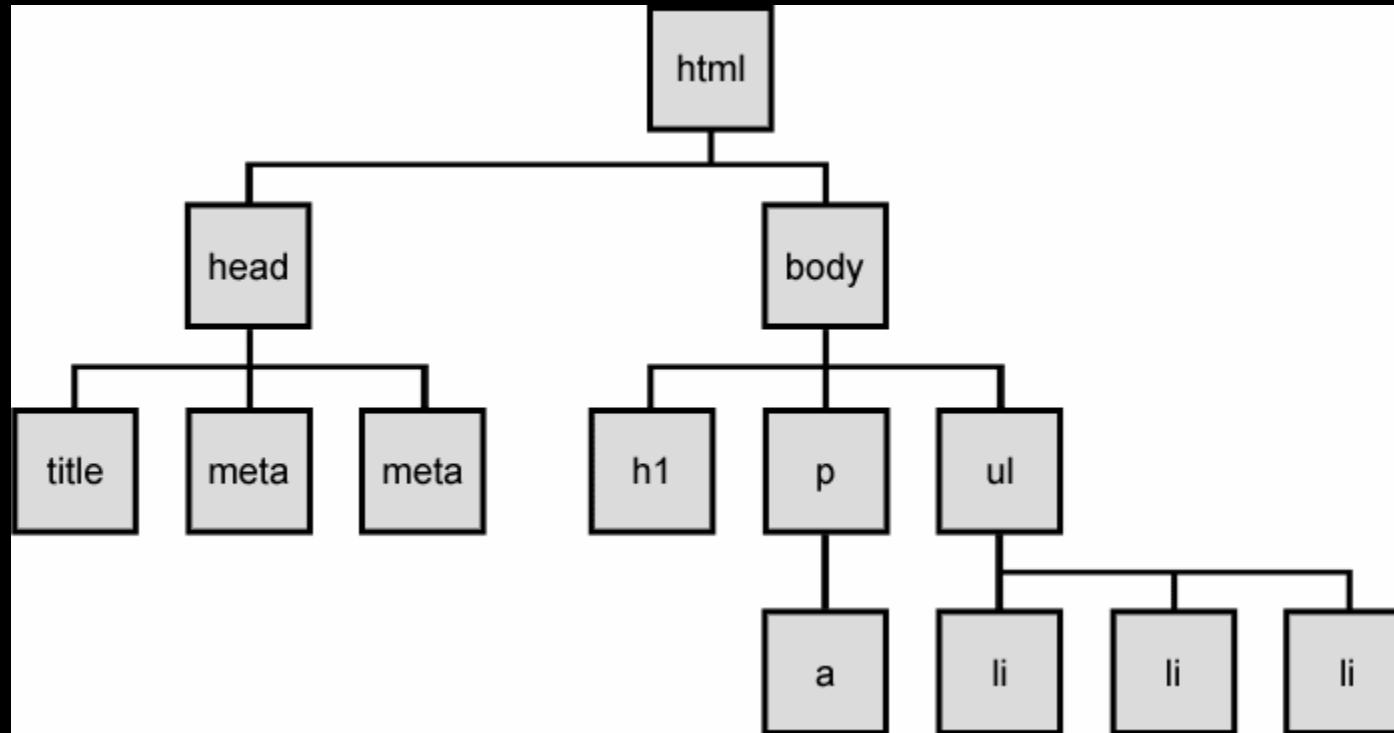


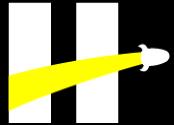
Árboles (trees)



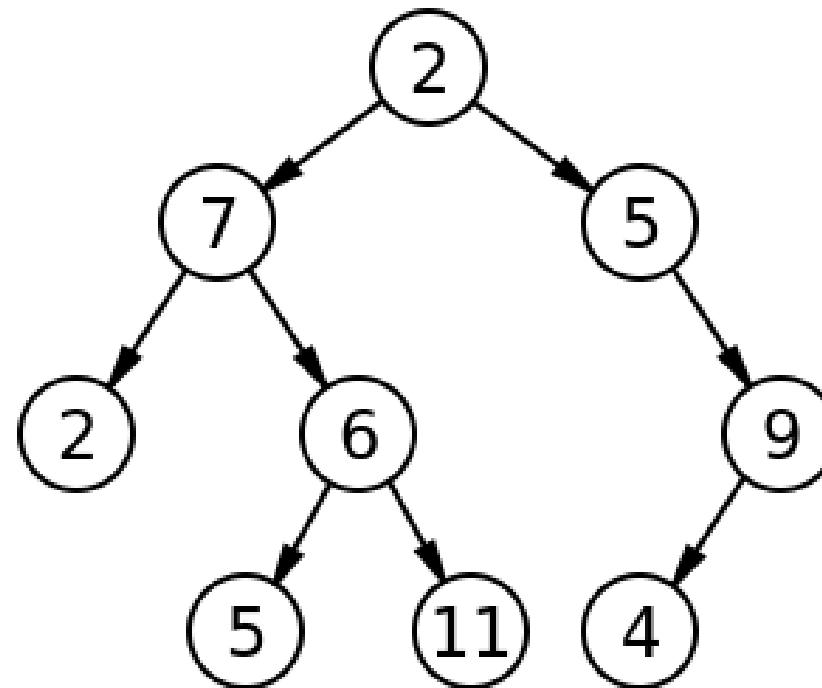


Árboles (trees)

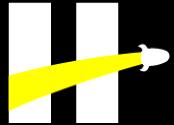




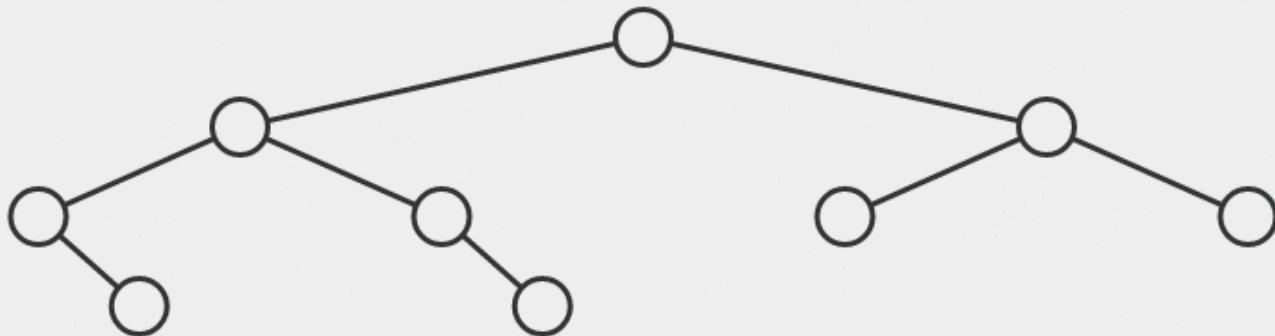
Tipos de árboles



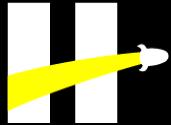
Binary Tree



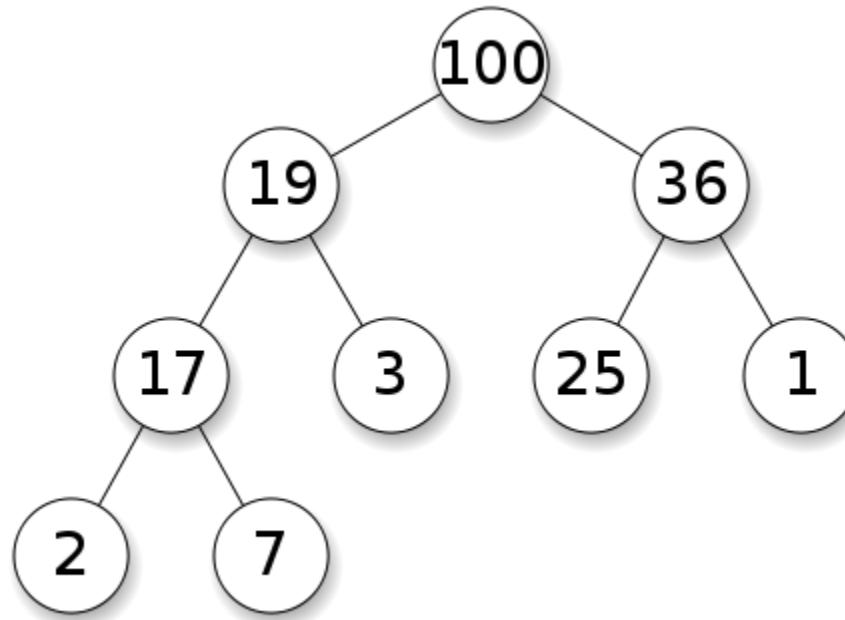
Tipos de árboles



AVL Tree

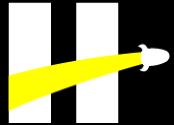


Tipos de árboles

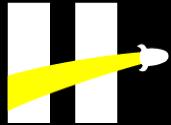


Heap

HENRY

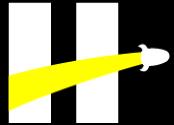


Algoritmos



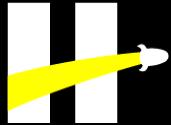
Algoritmos

Un algoritmo es un conjunto prescrito de instrucciones o reglas bien definidas, ordenadas y finitas que permite realizar una actividad mediante pasos sucesivos que no generen dudas a quien deba realizar dicha actividad. O sea, una serie de pasos a seguir para completar una tarea.



Algoritmos

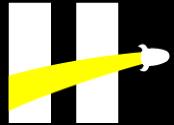
- 1. Resuelve un problema:** Este es el objetivo principal del algoritmo, fue diseñado para eso. Si no cumple el objetivo, no sirve para nada :S.
- 2. Debe ser comprensible:** El mejor algoritmo del mundo no te va a servir si es demasiado complicado de implementar.
- 3. Hacerlo eficientemente:** No sólo queremos tener la respuesta perfecta (o la más cercana), si no que también queremos que lo haga usando la menor cantidad de recursos posibles.



Algoritmos

¿Cómo medimos la eficiencia de un algoritmo?

- Tiempo
- Espacio
- Otros recursos:
 - Red
 - Gráficos
 - Hardware (Impresoras, Cpus, Sensores, etc...)



Algoritmos

Ejemplo juego Adivinar un número:

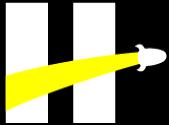
$$N = 1/2^x$$

$$\log^2(2^x) = \log^2 N$$

$$x * \log^2(2) = \log^2 N$$

$$x * 1 = \log^2 N$$

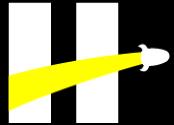
$$x = \log^2 N$$



Algoritmos

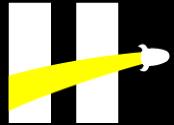
Ahora... ¿Por qué nos importa medir la complejidad de los algoritmos? Básicamente nos va a servir a:

1. Predecir el comportamiento: hay casos donde algo puede tardar tanto que no tenemos el lujo del prueba y error, tenemos que conocer de antemano si un algoritmo va a terminar en un tiempo adecuado para el problema.
2. Compararlos: Según el problema vamos a tener que decidir cuál es el mejor algoritmo para usar, tampoco podemos ponernos a probar uno por uno.



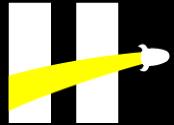
Cota superior asintótica (Big O Notation / Notación O grande)

$$O(g(x)) = \left\{ \begin{array}{l} f(x) : \text{existen } x_0, c > 0 \text{ tales que} \\ \forall x \geq x_0 > 0 : 0 \leq |f(x)| \leq c|g(x)| \end{array} \right\}$$

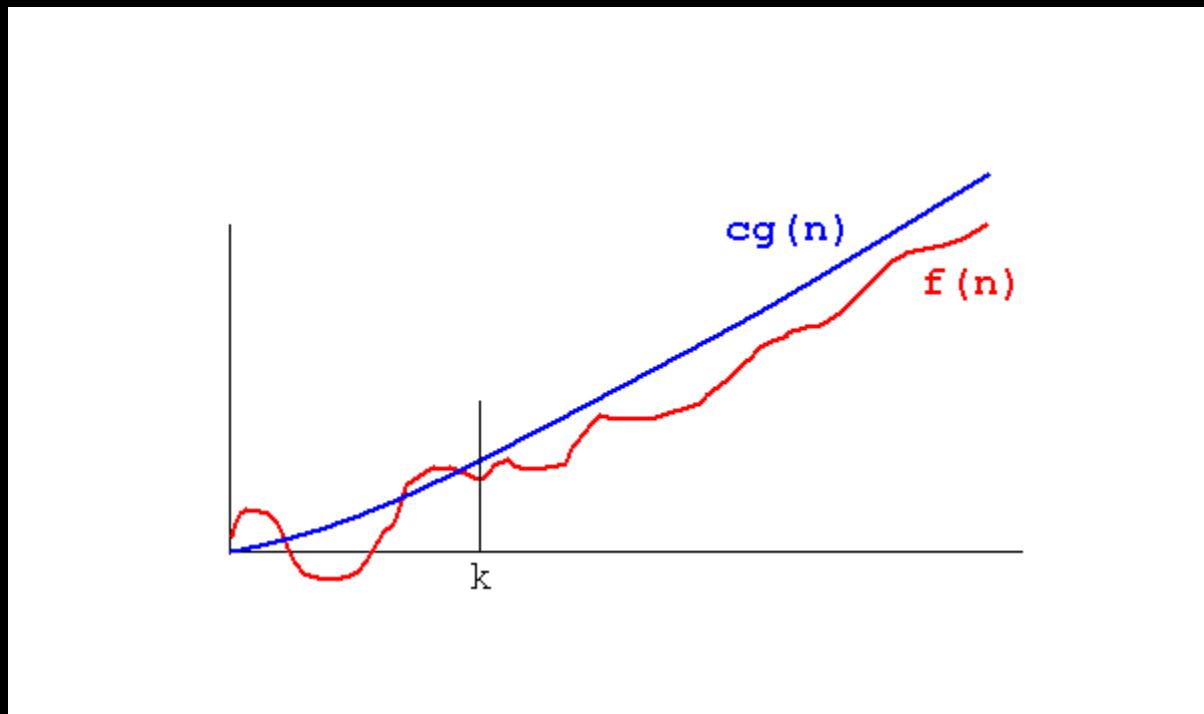


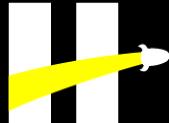
Cota superior asintótica (Big O Notation / Notación O grande)





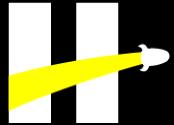
Cota superior asintótica (Big O Notation / Notación O grande)





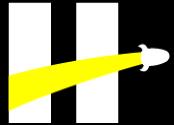
Cota superior asintótica (Big O Notation / Notación O grande)

| N | N^2 | N^2+N | %N |
|---------|----------------|----------------|--------|
| 10 | 100 | 110 | 10% |
| 100 | 10,000 | 10,100 | 1% |
| 1,000 | 1,000,000 | 1,001,000 | 0.1% |
| 10,000 | 100,000,000 | 100,010,000 | 0.01% |
| 100,000 | 10,000,000,000 | 10,000,100,000 | 0.001% |



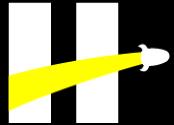
Ejemplos

```
1 //Encontrar el máximo
2 var max = array[0];
3 for( var i = 0; i <= array.length; i++){
4     if( array[i] > max) {
5         max = array[i];
6     }
7 }
8 console.log(max);
9 // O ( N )
```



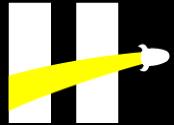
Ejemplos

```
1 for( var i = 0, i <= array.length; i++){  
2   for( var j = 0, j <= array.length; j++){  
3     if(array[i] === array[j]){  
4       return true;  
5     }  
6   }  
7 };  
8 // O( N x N) = O (n2)
```

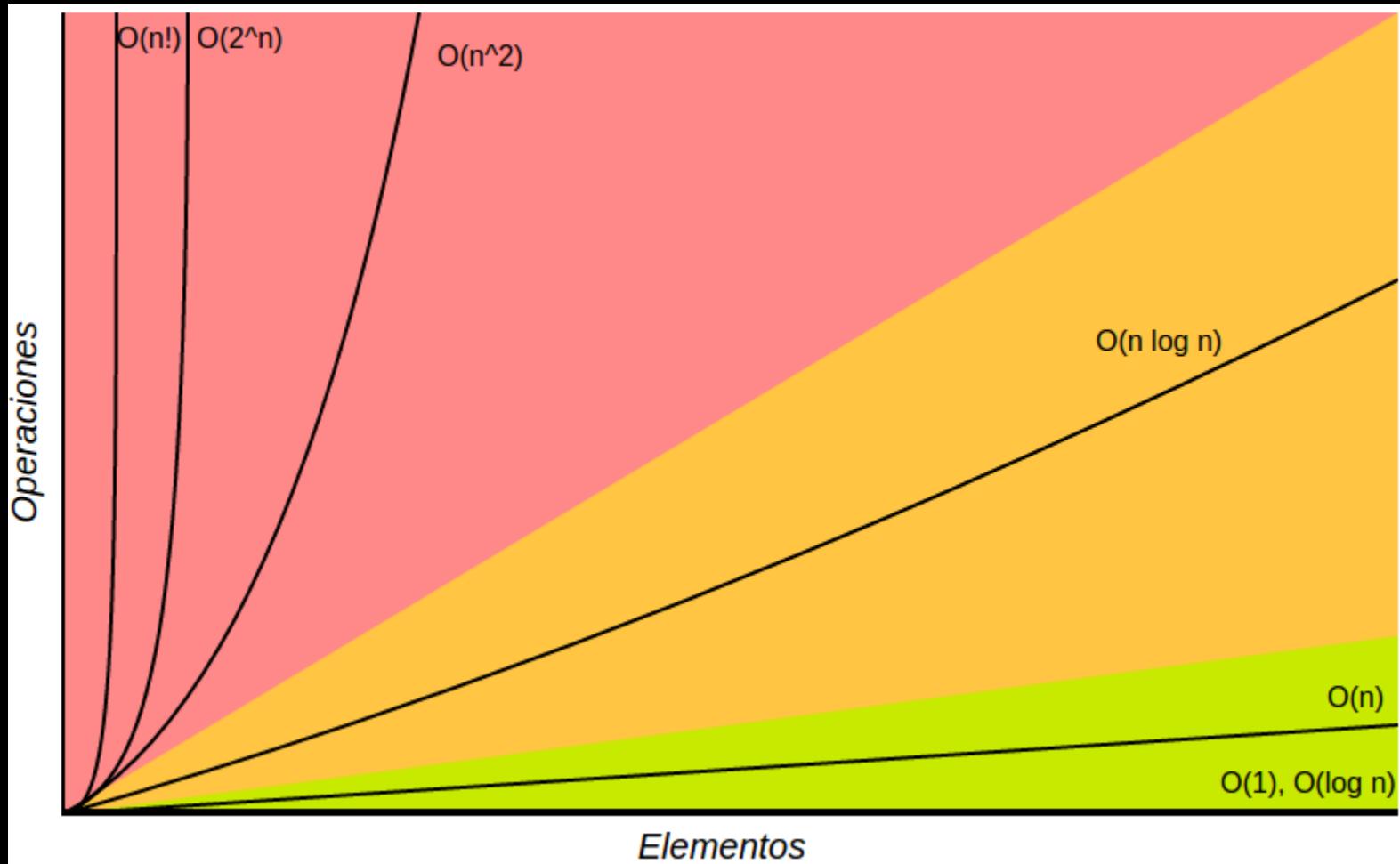


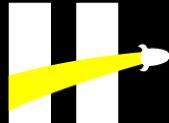
Ejemplos

```
1 function sumArray(array, n) {
2     var fin = array.length - 1;
3     var ini = 0;
4     while (ini < fin) {
5         var suma = array[ini] + array[fin];
6         if( suma === n) return true;
7         if( suma > n) fin = fin - 1;
8         if( suma < n) ini = ini + 1;
9     }
10    return false;
11};
```



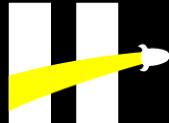
Medidas Comunes





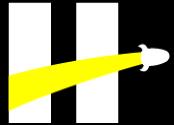
Medidas Comunes

| Runtime | F(1,000) | Time |
|---------------|-------------------------|-------------------------------|
| $O(\log N)$ | 10 | 0.00001 sec |
| $O(\sqrt{N})$ | 32 | 0.00003 sec |
| N | 1,000 | 0.001 sec |
| N^2 | 1,000,000 | 1 sec |
| 2^N | 1.07×10^{301} | 3.40×10^{287} years |
| $N!$ | 4.02×10^{2567} | 1.28×10^{2554} years |

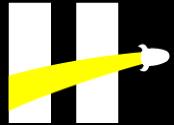


¿Qué cantidad de datos podría procesar cada algoritmo en un segundo?

| Runtime | N |
|---------------|---------------------------|
| $O(\log N)$ | $> 1 \times 10^{300,000}$ |
| $O(\sqrt{N})$ | 1 trillion |
| N | 1 million |
| N^2 | 1 thousand |
| 2^N | 20 |
| $N!$ | 10 |

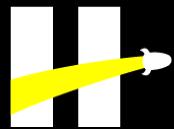


Algoritmos de Ordenamiento I

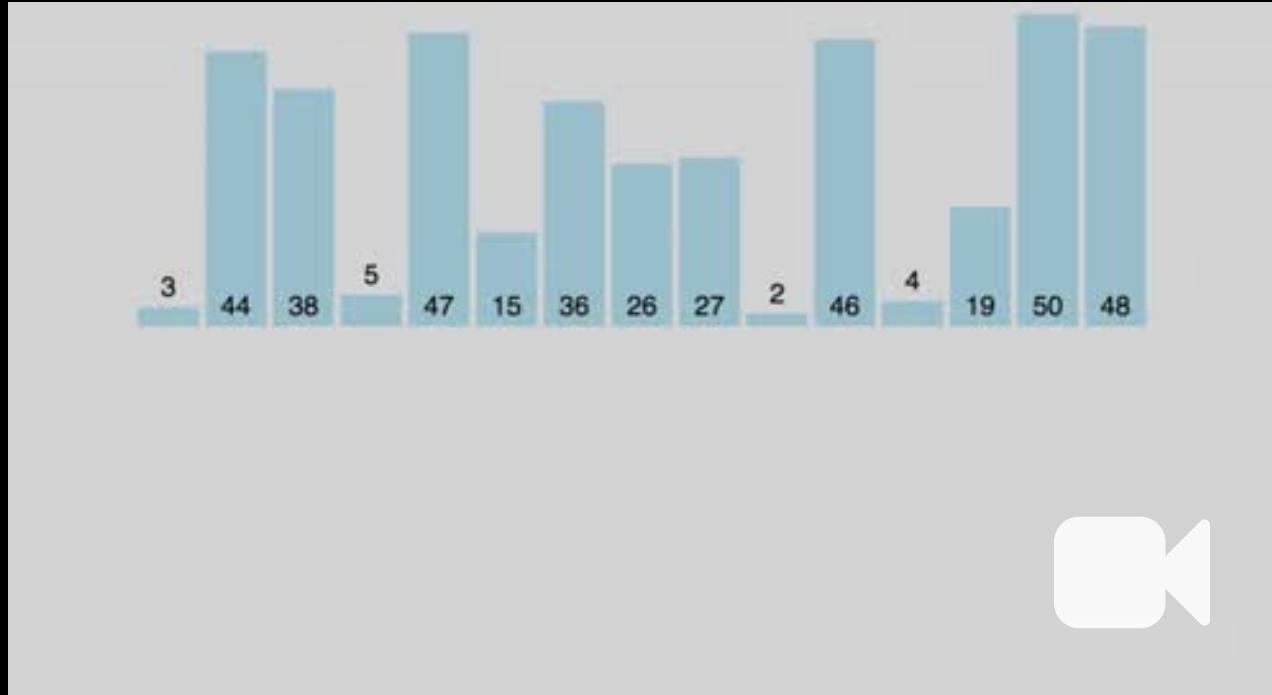


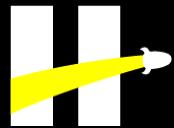
Bubble Sort



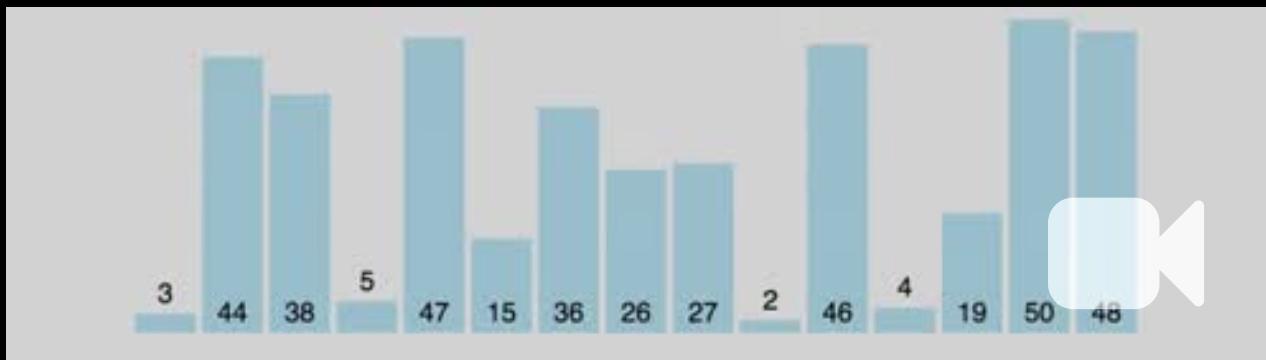


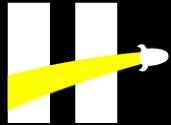
Insertion Sort



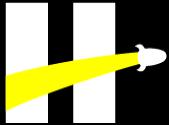


Selection Sort



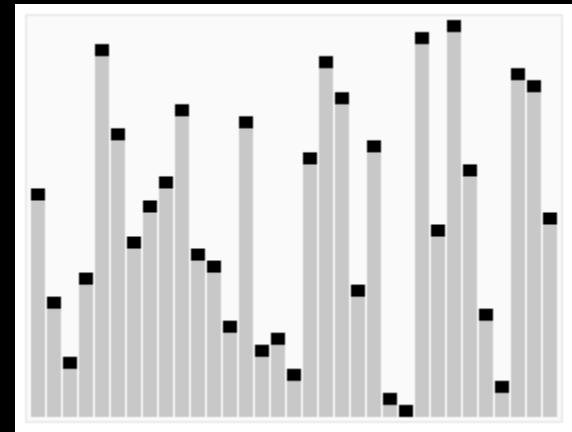


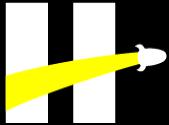
Algoritmos de Ordenamiento II



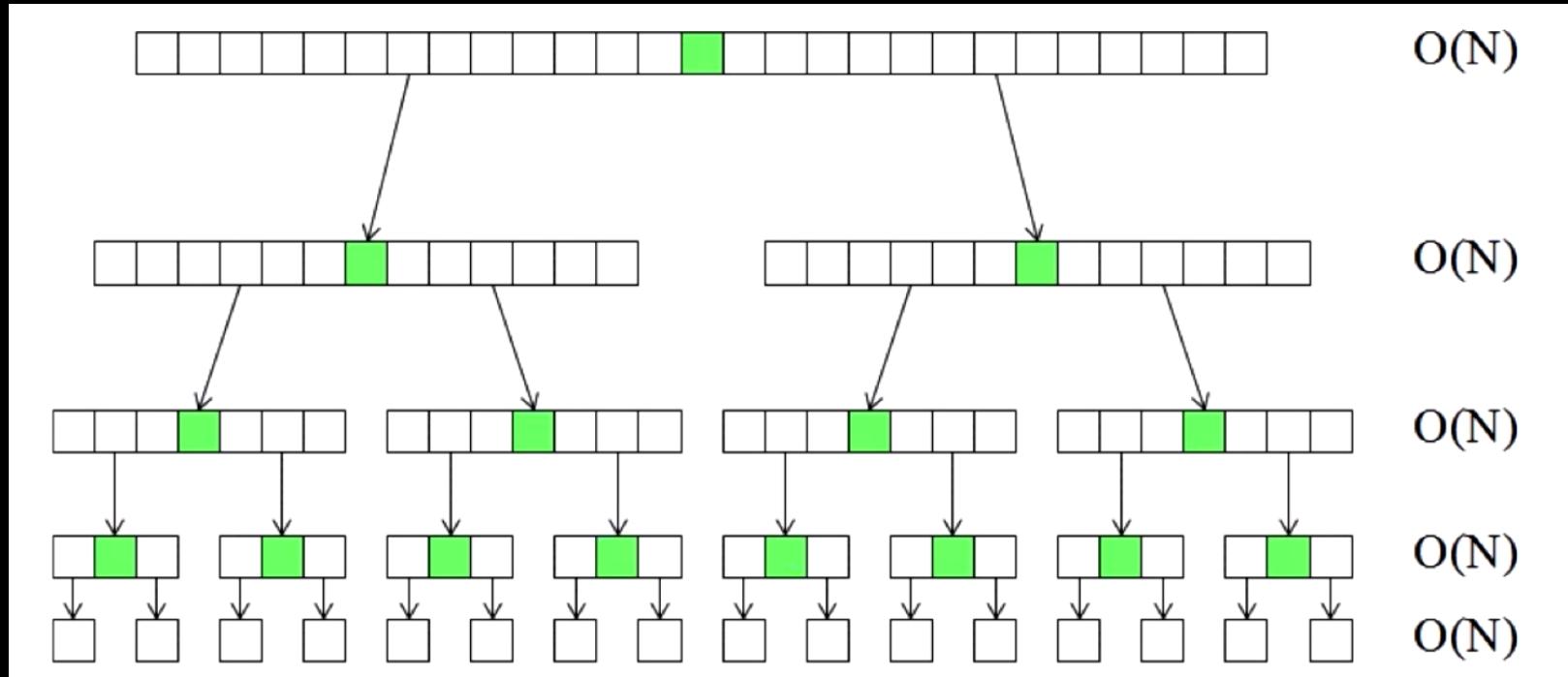
Quick Sort

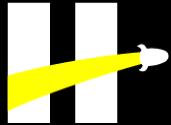
- Elegir un elemento de la lista de elementos a ordenar, al que llamaremos pivote.
- Mover los demás elementos de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores. Los elementos iguales al pivote pueden ser colocados tanto a su derecha como a su izquierda, dependiendo de la implementación deseada. En este momento, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada.
- La lista queda separada en dos sublistas, una formada por los elementos a la izquierda del pivote, y otra por los elementos a su derecha.
- Repetir este proceso de forma recursiva para cada sublista mientras éstas contengan más de un elemento. Una vez terminado este proceso todos los elementos estarán ordenados.



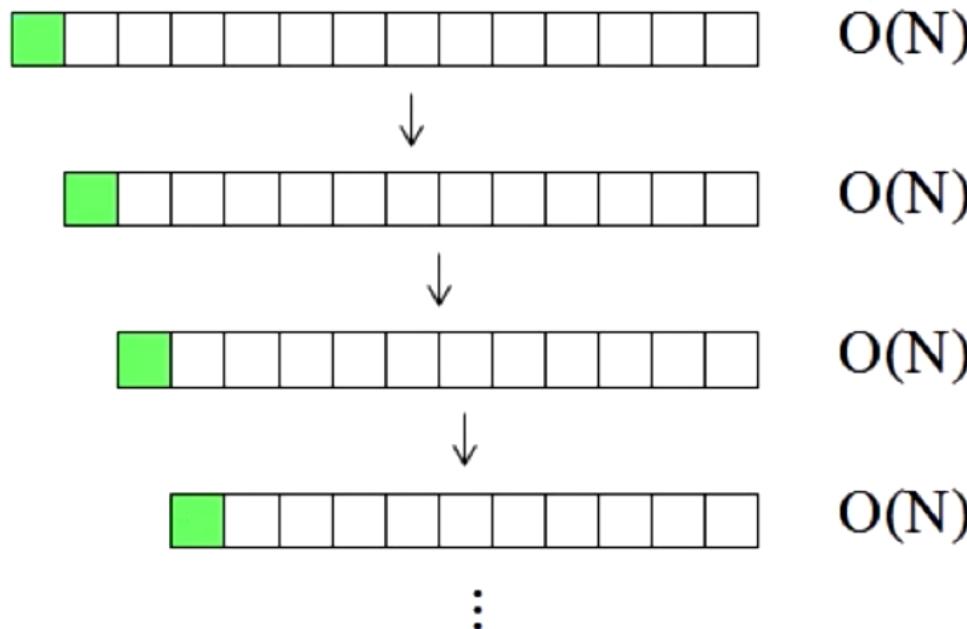


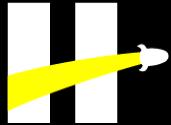
Quick Sort





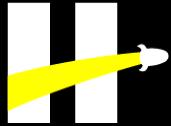
Quick Sort





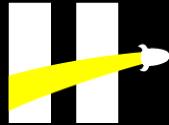
Merge Sort

- 1 - Divide el conjunto en dos grupos iguales.
- 2 - Ordena recursivamente los dos grupos
- 3 - Junta (o mergea) los grupos ordenados.



Merge Sort

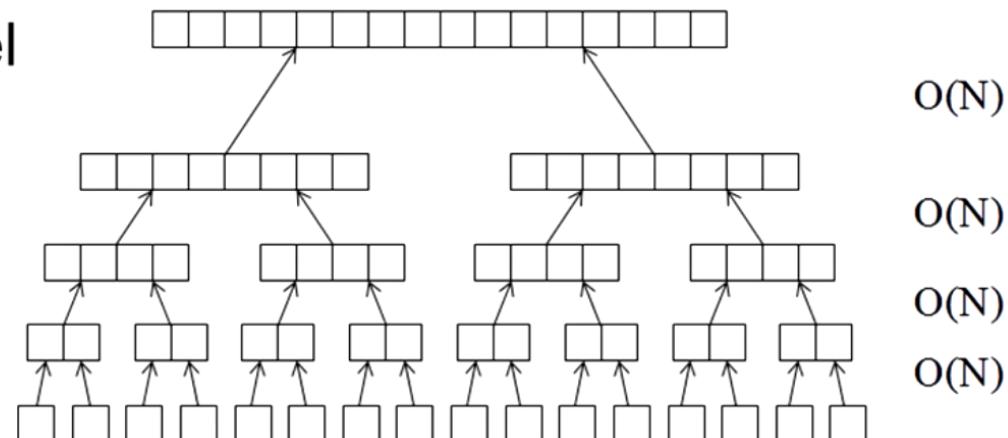
| | | | | | |
|---|---|---|---|---|---|
| 9 | 2 | 4 | 5 | 1 | 8 |
| 0 | 1 | 2 | 3 | 4 | 5 |



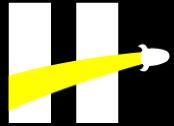
Merge Sort

Complejidad:

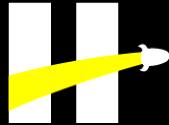
- $O(N)$ steps per level
 - $O(\log N)$ Levels
 - Total: $O(N \log N)$



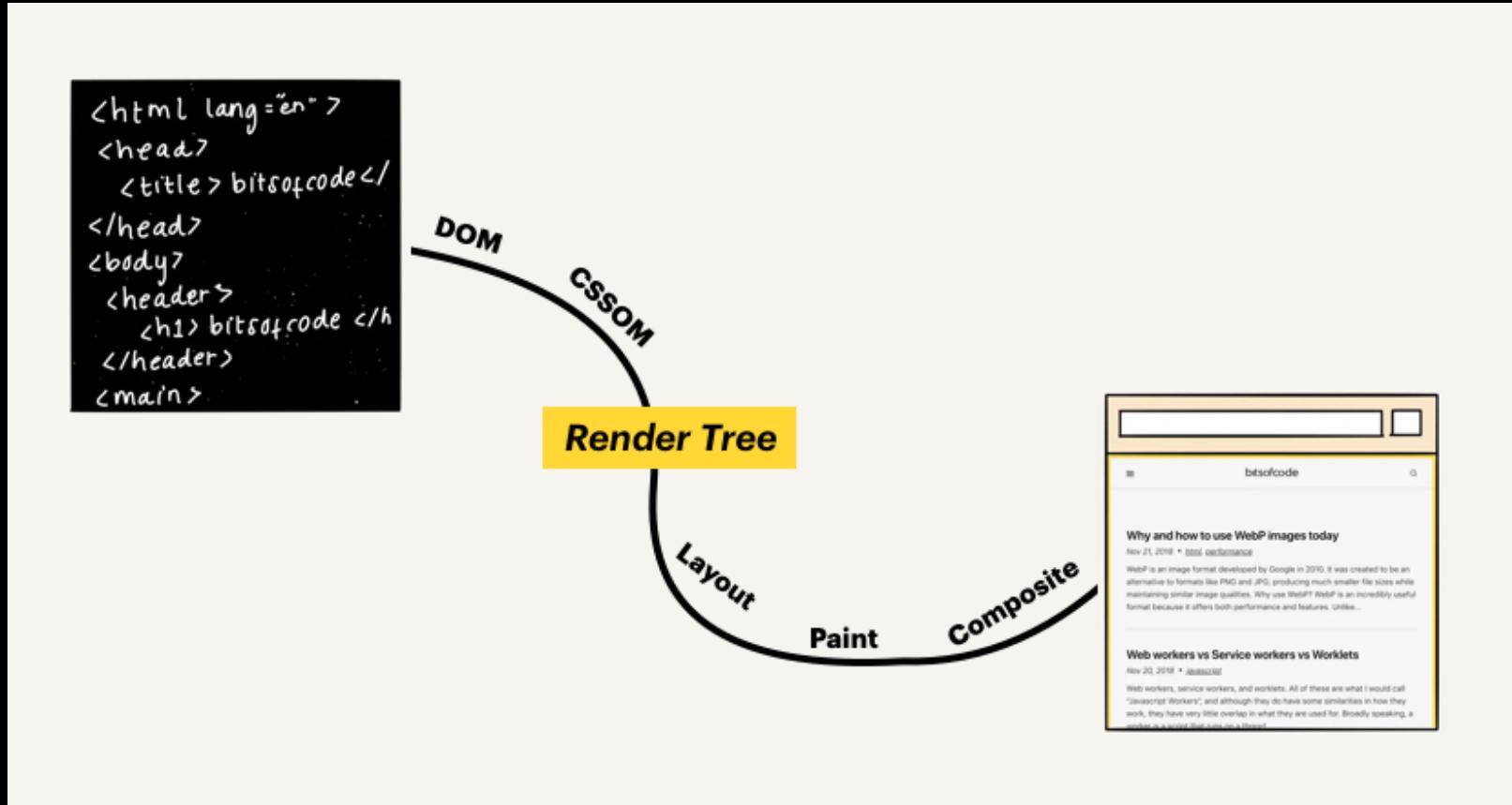
HENRY

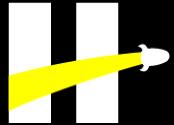


DOM



¿Cómo se construye una página?





¿Cómo se construye una página?

1. Construcción del DOM
2. Construcción del CSSOM
3. Ejecuta JavaScript
4. Creación del Render Tree
5. Generación del Layout
6. Painting

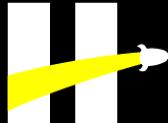


Construcción del DOM

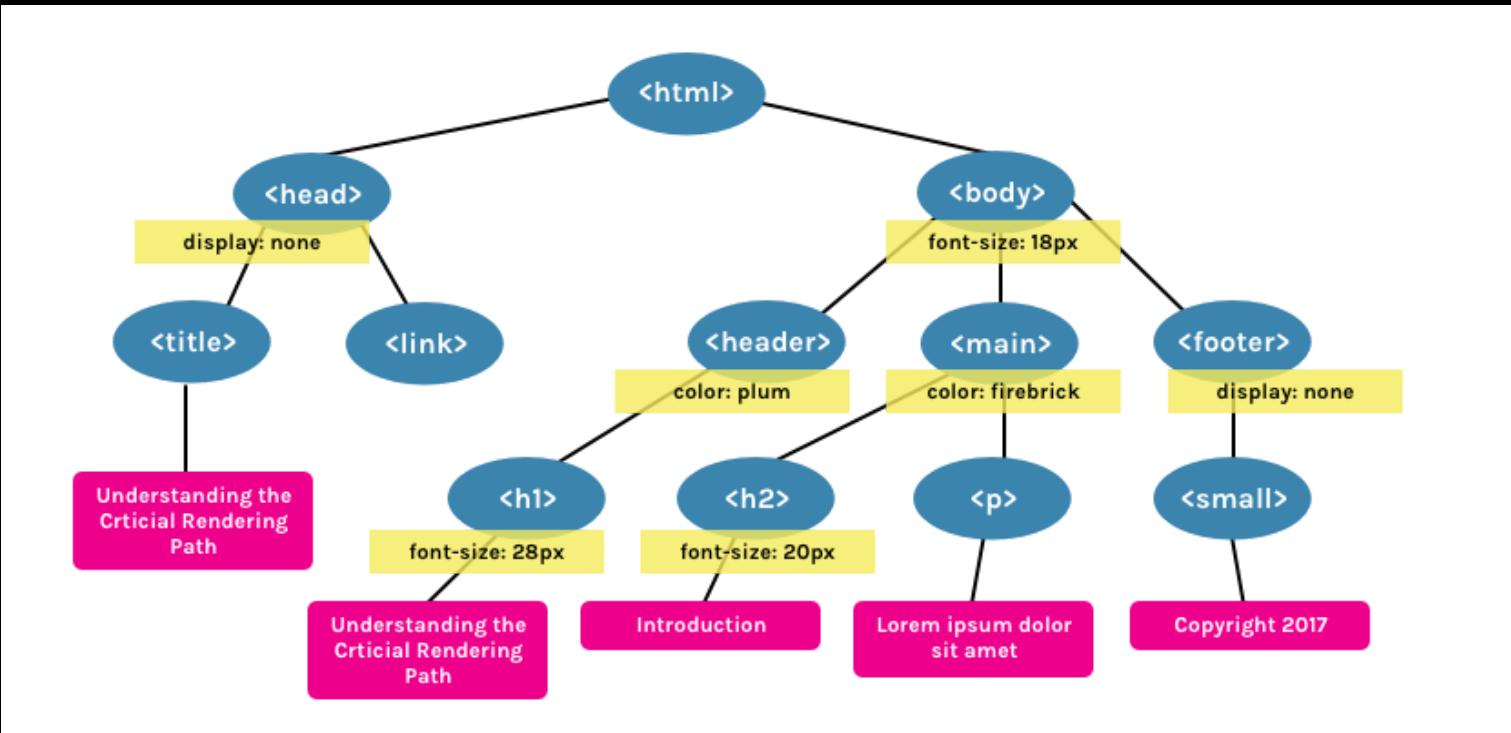
```
1 <!doctype html>
2 <html lang="en">
3   <head>
4     <title>My first web page</title>
5   </head>
6   <body>
7     <h1>Hello, world!</h1>
8     <p>How are you?</p>
9   </body>
10 </html>
```



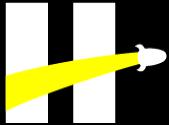
El DOM (**Document Object Model**) es una representación en un Objeto de la página HTML parseada.



Construcción del CSSOM



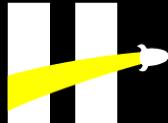
EL CSSOM (CSS Object Model) es un Objeto representando los estilos asociados al DOM.



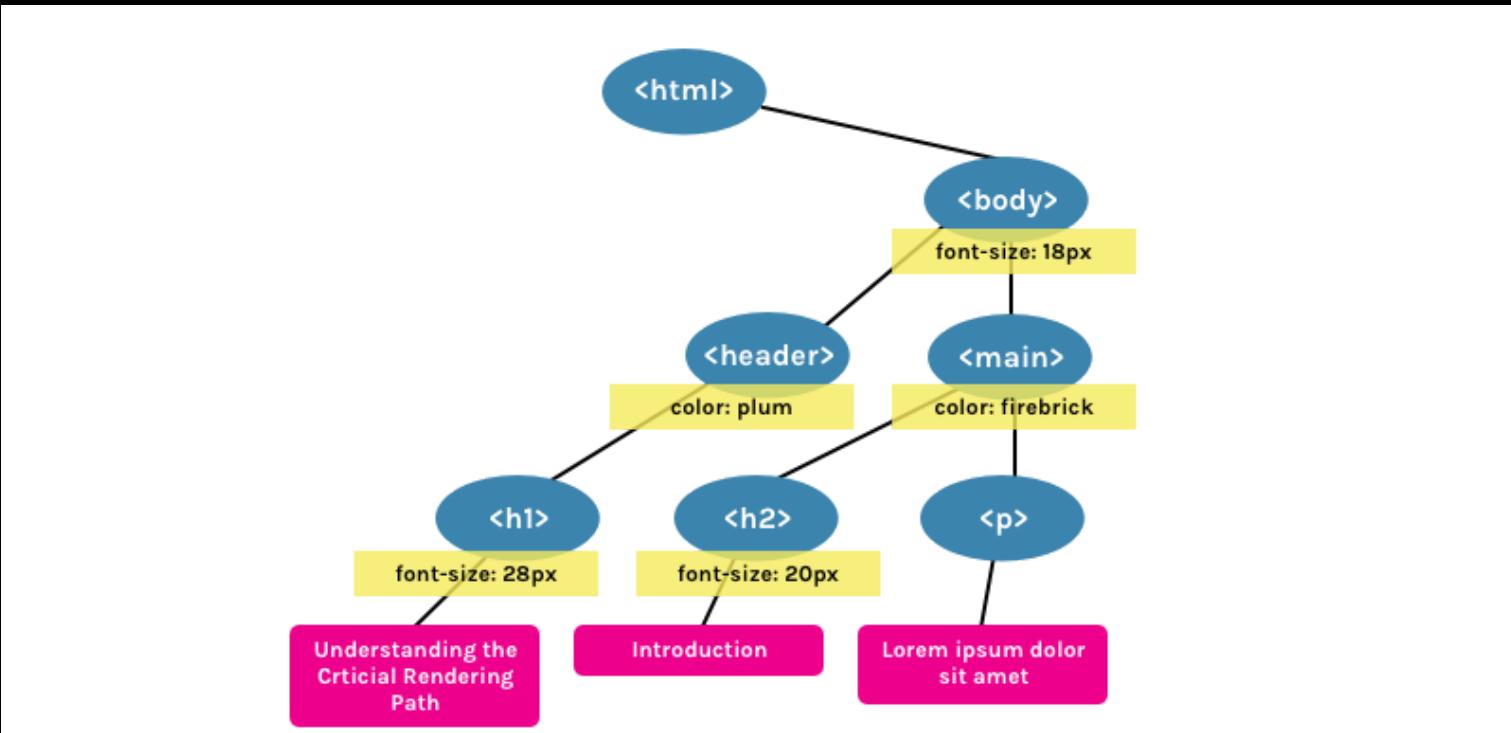
Ejecutando JS

JavaScript bloquea el parseo del DOM.
Cuando el parser llega a un tag <script />
frena para poder traer ese recurso y
ejecutarlo.

Por eso, cuando agregamos scripts de JS, lo
hacemos al final del documento HTML



Creando el render Tree



El render Tree es una combinación del DOM y el CSSOM, en donde se deja sólo los elementos visibles.

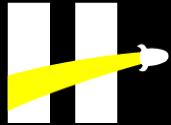


Generando el Layout

El Layout es lo que determina el tamaño del viewport, crea da el contexto necesario para calcular los estilos que depende de él, por ejemplo: % o vw units.

El View port se puede configurar a traves del tag meta viewport

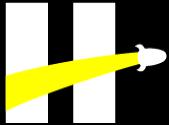
```
1 <meta name="viewport" content="width=device-width,initial-scale=1">
```



Painting

Finalmente, la parte visible de la página se convierte en pixels que se muestran en la pantalla.

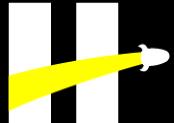
El tiempo de pintado depende del tamaño del DOM, como los estilos que se le aplican.



<Script />

```
1
2 <html>
3   <head>
4     <script>
5       alert('Inyectando código Javascript');
6     </script>
7   </head>
8 </html>
```

```
1
2 <html>
3   <head>
4     <script type="text/javascript" src="./index.js" async></script>
5   </head>
6 </html>
```

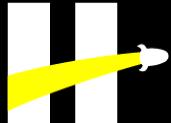


DOM API

El *browser* nos da una *API* para interactuar con el DOM usando JavaScript

Esta *API* nos permite

- inspeccionar los elementos y la estructura del documento
- modificar la *estructura*: agregar, modificar o eliminar elementos HTML ó atributos
- modificar el *contenido* del documento
- modificar los *estilos* (CSS)
- agregar o eliminar eventos
- reaccionar a determinados eventos
- etc...



Objeto *document*

Mediante la ejecución de Javascript tenemos la posibilidad de acceder a un objeto global denominado **document** que contiene las propiedades del DOM y métodos de su prototipo que nos permiten acceder a los elementos del DOM y manipularlos.

```
>> document
<- ▼ HTMLDocument https://github.com/atralice/caronte/tree/M2-
  ajax/M2/00-DOM
    URL: "https://github.com/atralice/caronte/tree/M2-ajax/M2/00-
  DOM"
    ▶ activeElement: <body class="logged-in env-production min-
  width-lg"> ◻
      alinkColor: ""
    ▶ all: HTMLAllCollection { 0: html ◻ , 1: head ◻ , 2: meta ◻ ,
    ... }
    ▶ anchors: HTMLCollection { length: 0 }
    ▶ applets: HTMLCollection { length: 0 }
      baseURI: "https://github.com/atralice/caronte/tree/M2-ajax/M2
  /00-DOM"
      bgColor: ""
    ▶ body: <body class="logged-in env-production min-width-lg"> ◻
      characterSet: "UTF-8"
      charset: "UTF-8"
      childElementCount: 1
    ▶ childNodes: NodeList [ <!DOCTYPE html>, html ◻ ]
    ▶ children: HTMLCollection { 0: html ◻ , length: 1 }
      compatMode: "CSS1Compat"
      contentType: "text/html"
      cookie: "_ga=GA1.2.899944563.1515683265;
```



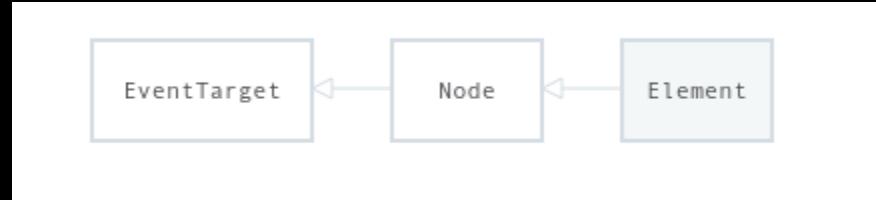
Selectores

```
>> document.getElementById('user-content-document-selectors')
<- ▼ a#user-content-document-selectors.anchor
  accessKey: ""
  accessKeyLabel: ""
  assignedSlot: null
  ▶ attributes: NamedNodeMap(4) [ id="user-content-document-selectors", class="anchor", aria-hidden="true", ... ]
  baseURI: "https://github.com/stralice/caronte/tree/M2-ajax/M2/00-DOM"
  charset: ""
  childElementCount: 1
  ▶ childNodes: NodeList [ svg.octicon.octicon-link ]
  ▶ children: HTMLCollection { 0: svg.octicon.octicon-link }
  length: 1
  ▶ classList: DOMTokenList [ "anchor" ]
  className: "anchor"
  clientHeight: 24
  clientLeft: 0
  clientTop: 0
  clientWidth: 20
  contentEditable: "inherit"
  contextMenu: null
  coords: ""
  ▶ dataset: DOMStringMap(0)
```

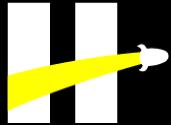
Los **selectores** nos permitirán buscar y recuperar un elemento del DOM. (como cuando buscábamos un elemento en un árbol de búsqueda), sólo que ahora el elemento retornado es un objeto JS que representa una entidad HTML.



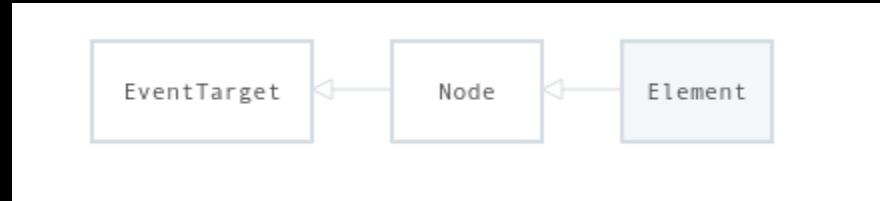
Elements



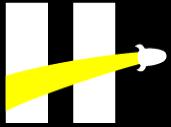
- **EventTarget**: es una interfaz implementada por los objetos que pueden recibir eventos y pueden tener escuchadores para los objetos.
- **Node**: es una interfaz en la cuál un número de objetos de tipo DOM API heredan. Esta interfaz permite que esos objetos sean tratados similarmente.
- **Element**: Representa un elemento HTML.



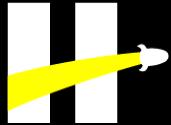
API



```
1
2
3 const divs = document.getElementsByClassName('divClass');
4
5 const div = document.getElementById('divId');
6
7 const div = document.querySelector('.divId');
8
9 const divs = document.querySelectorAll('.divId');
```



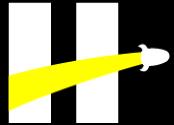
Eventos



Eventos

Un *evento* es una señal que algo sucedió.
Todos los nodos del DOM pueden generar
estas señales.

Un *Event Listener* es el encargado de
escuchar por esas señales y hacer *algo*.



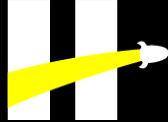
Eventos

The screenshot shows a browser's developer tools open to the 'Console' tab. On the left, there is a visual representation of three light blue rectangular boxes labeled '1', '2', and '3' from left to right. On the right, the 'Console' panel displays the following output:

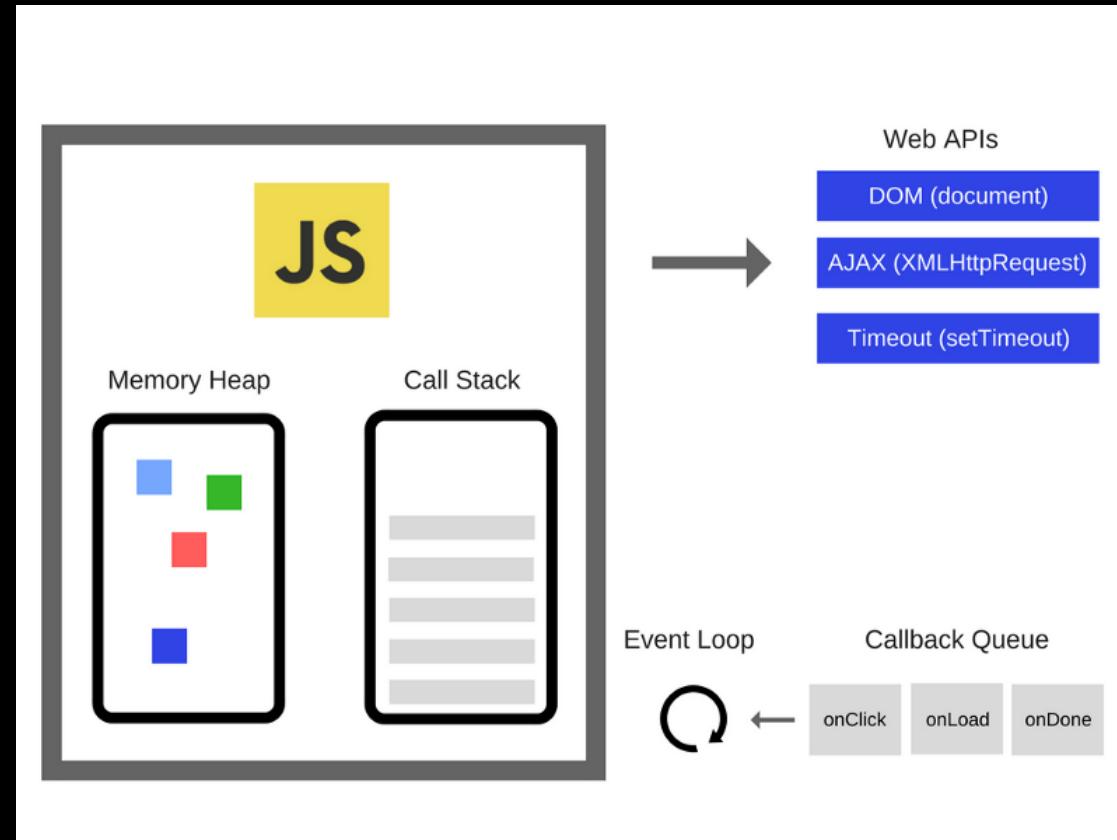
```
MouseEvent {isTrusted: true, screenX: 635, screenY: 358, clientX: 82, clientY: 51, ...} ⓘ
  altKey: false
  bubbles: true
  button: 0
  buttons: 0
  cancelBubble: false
  cancelable: true
  clientX: 82
  clientY: 51
  composed: true
  ctrlKey: false
  currentTarget: null
  defaultPrevented: false
  detail: 1
  eventPhase: 0
  fromElement: null
```

The output shows a MouseEvent object with various properties like screenX, clientX, button, and detail. The 'bubbles' property is explicitly mentioned as being true.

Propiedades de los eventos



Event Loop



Henry



Hacé click acá para dejar tu feedback sobre esta clase.



Hacé click acá completar el quizz teórico de esta lecture.

Lesson 00 - DOM

En esta Lesson se verán los siguientes temas:

- DOM
- script
- document
- document Selectors
- Element Methods
- Event Handlers

Introduction al DOM

El nombre **DOM** proviene de sus siglas en inglés de 'Document Object Model'. Cuando un navegador carga una página web, toma todo el contenido HTML y genera un modelo para dicho contenido. Utilizando código Javascript podemos acceder y manipular ese modelo ya sea agregando o quitando elementos, cambiando sus atributos y también modificando sus estilos.

El elemento **script**

Es posible injectar código Javascript dentro de una página HTML utilizando el elemento **script**. Para ello existen dos formas distintas de realizarlo:

1. Insertando la etiqueta **<script>** en el **<head>** de la página HTML:

```
<html>
  <head>
    <script>
      // Acá es donde irá el código Javascript
      alert('Inyectando código Javascript');
    </script>
  </head>
</html>
```

- Utilizar la etiqueta `<script>` para injectar código Javascript contenido en un archivo externo en nuestra página HTML:

```
<html>
  <head>
    <script type="text/javascript" src=".index.js" async></script>
  </head>
</html>
```

Nota: el atributo `type` debe colocarse como "text/javascript" y en el `src` se debe indicar la ubicación del archivo con código Javascript que queremos injectar. Es posible también incluir la palabra `async` al final de la etiqueta para indicarle al navegador que debe cargar dicho script de forma asincrónica

document

Mediante la ejecución de Javascript tenemos la posibilidad de acceder a un objeto global denominado `document` que contiene las propiedades del DOM y métodos de su prototipo que nos permiten acceder a los elementos del DOM y manipularlos.

En el motor de JS que se ejecuta en el browser, el objeto global es `document`, es decir que `this` apunta a `document` cuando lo usamos en el contexto global.

document Selectors

Cuando el Browser parsea el documento HTML, crea una estructura de árbol (el DOM), pensemos que el DOM es un modelo de la página en forma de objetos. JavaScript no sabe cómo trabajar con HTML, pero sí con objetos. Por lo tanto, cada elemento html que esté en el dom, podremos usarlo como un objeto, que tendrá sus propiedades y métodos. Dentro de todos los métodos que contiene `document` en su prototipo los más útiles son los **selectores**. Los **selectores** nos permitirán buscar y recuperar un elemento del DOM. (como cuando buscábamos un elemento en un árbol de búsqueda), sólo que ahora el elemento returned es un objeto JS que representa una entidad HTML.

Los principales 5 selectores son los siguientes:

`document.getElementsByClassName`

`getElementsByClassName` se encarga de encontrar elementos en función del nombre de su clase. Devuelve un array conteniendo los objetos que coincidieron con la búsqueda que puede ser iterado. Debemos brindarle como parámetro un string con el nombre de la clase que deseamos buscar. Ejemplo:

```
const divs = document.getElementsByClassName('divClass');
```

En este ejemplo estamos buscando los elementos que contengan 'divClass' como su clase definida

`document.getElementById`

`getElementById` se encarga de encontrar un único elemento en función de su id, por lo que devolverá dicho elemento. Debemos brindarle como parámetro un string con el id del elemento que deseamos buscar. Ejemplo:

```
const div = document.getElementById('divId');
```

En este ejemplo estamos buscando el elemento cuyo id es igual a 'divId'

`document.querySelector`

`querySelector` es un método que busca los elementos basándose en uno o más selectores CSS. Recordemos que es posible hacer referencia a clases utilizando un `.`, a ids con `#` y a elementos usando el nombre de su etiqueta directamente. Es recomendable utilizar sólo ids con `querySelector` ya que sólo retornará el primer elemento que coincide con el selector indicado. Ejemplo:

```
const div = document.querySelector('.divId');
```

En este ejemplo obtendremos el primer elemento con la clase 'divId' pero si hay más elementos con dicha clase no los tendrá en cuenta

`document.querySelectorAll`

`querySelectorAll` funciona de la misma forma que `querySelector` pero en vez de devolver únicamente el primer elemento que coincide con el selector devolverá un array con todos los elementos que coincidan it returns an array like object containing all elements that match the selector. Ejemplo:

```
const divs = document.querySelectorAll('.divId');
```

En este ejemplo obtendremos un array con todos los elementos que contengan la clase 'divId'

`document.createElement`

En el caso de que queramos crear un elemento para agregarlo al DOM podemos utilizar `document.createElement`. Este método recibe como parámetro un string indicando el tipo de elemento que deseamos crear y devuelve un elemento vacío de dicho tipo. Ejemplo:

```
const newDiv = document.createElement('div');
```

En este ejemplo estamos creando un nuevo elemento 'div' vacío

Element Methods

Una vez seleccionado el elemento podemos utilizar distintos métodos y propiedades para modificarlo como por ejemplo cambiar su estilo, cambiar de atributos, agregar/eliminar elementos anidados, agregar/eliminar `event listeners`. Algunos de los métodos más comunes son:

.innerHTML

Con el método `innerHTML` podemos acceder a la información que se encuentra entre las etiquetas de apertura y cierre de un elemento HTML tanto para simplemente lectura o para su modificación. Ejemplo:

```
const p = document.querySelector('#pID');
console.log(p.innerHTML) // Va a imprimir el texto dentro del párrafo con el id 'pID'

p.innerHTML = 'Nuevo texto'; // Acá estamos modificando el texto del párrafo mencionado anteriormente

console.log(p.innerHTML); // Va a imprimir el nuevo texto que le seteamos, es decir: "Nuevo texto"
```

.[attribute] y .setAttribute

Podemos llamar al método `.setAttribute` para agregar un atributo a un elemento o sobreescribirlo en el caso de que ya se encuentre definido. Otra forma equivalente de realizarlo pero más corta sería llamando a `[nombre del atributo] = [nuevo valor]`. Ejemplo:

```
const a = document.querySelector('#linkHenry'); // Obtengo el elemento a cuyo id es 'linkHenry'

a.setAttribute('href', 'https://www.soyhenry.com/'); // Seteo el atributo href del elemento a para que redireccione a la página principal de Henry

a.href = 'https://www.soyhenry.com/'; // Equivalente al anterior pero más corto
```

.style

Podemos modificar el estilo de un elemento utilizando `.style`. Cabe mencionar que con esto no estamos accediendo al estilo CSS sino que lo que estamos haciendo es agregarle la propiedad `style` dentro de la etiqueta HTML. Ejemplo:

```
const div = document.querySelector('#divId');

div.style.height = '300px'; // Le damos una altura de 300 pixeles al div cuyo id es 'divId'
div.style.background = 'red'; // Le seteamos el color de fondo en rojo a dicho div
```

.className y .id

Podemos utilizar `.className` y `.id` para acceder y modificar las clases o ids de los elementos. Esto es útil cuando ya tenemos definido en los estilos CSS un estilo en particular asociado a una clase o id y queremos simplemente modificando la clase o id del elemento cambiar su estilo sin tener que modificar propiedad por propiedad. Ejemplo:

```
const div = document.querySelector('#divId');

console.log(div.id); // Utilizando ',id' accedemos al nombre de su id, en este caso 'divId'
div.className = 'nuevaClase'; // Le seteamos la clase 'nuevaClase'
div.id = 'nuevoId'; // Le seteamos el id 'nuevoId'
```

.appendChild

Es posible agregar elementos directamente al `DOM` utilizando `.appendChild` sobre el elemento que queremos que sea su padre. Ejemplo:

```
const body = document.querySelector('body');
const newDiv = document.createElement('div'); // Creamos un nuevo elemento div

body.appendChild(newDiv); // Agregarmos el div recién creado dentro del body de la página
```

Event Listeners

Un `Event Listener` es una función que se ejecuta luego de que ocurra un determinado evento. Existen diferentes tipos de eventos, entre ellos se encuentran: un click, un desplazamiento del mouse por encima del elemento, el apretado de una tecla, etc. Para ver la lista completa pueden consultar el siguiente [link](#)

Click

El evento más común es el de 'click' y en particular es el único que posee la propiedad `.onclick` para asignarle una función que será ejecutada al clickear el componente indicado. Ejemplo:

```
const div = document.querySelector('#divId');
div.onclick = function() {
    console.log('clickeado');
};
```

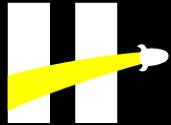
En este ejemplo lo que estamos haciendo es indicarle que cuando se clique el div cuyo id es 'divId' se ejecute la función ahí definida que lo único que hará en este caso es escribir por consola "clickeado"

addEventListener y otros eventos

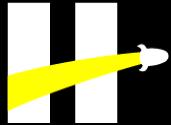
.`addEventListener` es un método que recibe como primer parámetro el tipo de evento que va a estar esperando y como segundo parámetro una función callback que es la que va a ejecutarse cuando ocurra dicho evento. Nota: es mejor utilizar `addEventListener` en todos los casos incluyendo los clicks. Ejemplo:

```
const div = document.querySelector('#divId');
div.addEventListener('mouseenter', function() {
    console.log('El mouse entró!');
});
```

En este ejemplo lo que estamos haciendo es indicarle que cuando el mouse ingrese al div cuyo id es 'divId' se ejecute la función ahí definida que lo único que hará en este caso es escribir por consola "El mouse entró!"



CSS



Framework

En primer lugar un 'Framework' es un marco de referencia o marco de trabajo que nos provee distintas herramientas que se puede utilizar para facilitar el desarrollo de aplicaciones, ofreciéndonos una forma estándar y por lo general más simple para programar.



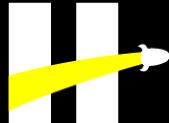
Framework CSS

En particular para el caso de un 'Framework CSS', se refiere a un conjunto de estilos predefinidos que pueden utilizarse para elaborar una interfaz de usuario atractiva sin necesidad de tener que definir a mano todas y cada una de las propiedades CSS de nuestros elementos HTML.



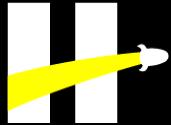
Bootstrap

```
1 <!DOCTYPE html>
2 <html>
3 <head>
4   <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4
5 </head>
6 <style media="screen">
7   .buttonComun {
8     color: white;
9     background-color: red;
10    border-radius: 5px;
11  }
12 </style>
13 <body>
14 <button type="button" class="buttonComun">Boton Común</button>
15 <button class="btn btn-danger">Boton Bootstrap</button>
16 </body>
17 </html>
```



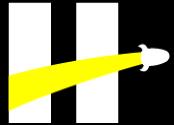
CSS Media Queries

```
1 body {  
2   background-color: red;  
3 }  
4  
5 /* Pantallas de menos de 992px de ancho */  
6 @media screen and (max-width: 992px) {  
7   body {  
8     background-color: blue;  
9   }  
10 }  
11  
12 /* Pantallas de menos de 600px de ancho */  
13 @media screen and (max-width: 600px) {  
14   body {  
15     background-color: black;  
16   }  
17 }
```



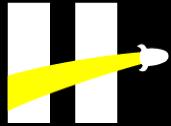
Grid System

- col- (extra small devices - menos de 576px)
- .col-sm- (small devices - mayor o igual a 576px)
- .col-md- (medium devices - mayor o igual a 768px)
- .col-lg- (large devices - mayor o igual a 992px)
- .col-xl- (xlarge devices - mayor o igual a 1200px)

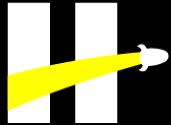


Grid System

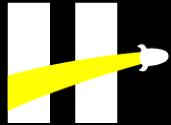
| | | | | | | | | | | | | |
|---------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|
| span 1 | span 1 | span 1 | span 1 | span 1 | span 1 | span 1 | span 1 | span 1 | span 1 | span 1 | span 1 | span 1 |
| span 4 | | | | span 4 | | | | span 4 | | | | |
| span 4 | | | | span 8 | | | | | | | | |
| span 6 | | | | | | | span 6 | | | | | |
| span 12 | | | | | | | | | | | | |



< DEMO />

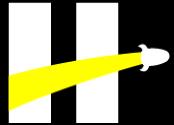


CSS Preprocessors



CSS Preprocessors

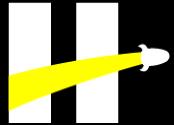
Un preprocesador CSS es un programa que te permite generar CSS a partir de la syntax única del preprocesador. Existen varios preprocesadores CSS de los cuales escoger, sin embargo la mayoría de preprocesadores CSS añadiran algunas características que no existen en CSS puro, como variable, mixins, selectores anidados, entre otros. Estas características hacen la estructura de CSS más legible y fácil de mantener.



{less}

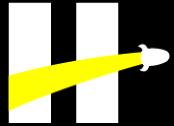
```
1 // Variables
2
3 @color-fondo: #F55;
4 @width: 10px;
5 @height: @width + 10px;
6 /* También es posible realizar operaciones sobre las variables */
7
8 h1 {
9   background-color: @color-fondo;
10  width: @width;
11  height: @height;
12 }
```

<http://lesscss.org/less-preview/>



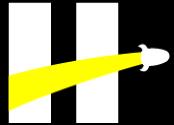
{less}

```
1 // Funciones
2
3 @base: #f04615;
4 @width: 0.5;
5
6 .class {
7   width: percentage(@width);
8   color: saturate(@base, 5%);
9   background-color: spin(lighten(@base, 25%), 8);
10 }
```



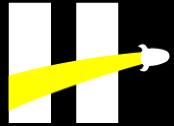
{less}

```
1 // Anidado
2
3 nav {
4   ul {
5     margin: 0;
6     padding: 0;
7     list-style: none;
8   }
9   li {
10     display: inline-block;
11   }
12   a {
13     display: block;
14     padding: 6px 12px;
15     text-decoration: none;
16   }
17 }
```



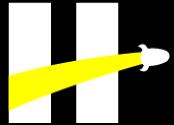
{less}

```
1 // Importaciones
2
3 @import "general";
4
5 body {
6   font-family: Helvetica, sans-serif;
7   font-size: 18px;
8   color: red;
9 }
```



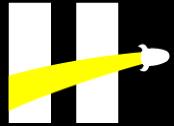
{less}

```
1 // Mixins
2
3 .important-text {
4   color: black;
5   font-size: 25px;
6   font-weight: bold;
7 }
8
9 .danger {
10   .important-text();
11   background-color: red;
12 }
13
14 .success {
15   .important-text();
16   background-color: green;
17 }
```



{less}

```
1 // Mixins con parametros
2
3 .bordered(@color; @width) {
4   border: @width solid @color;
5 }
6
7 .myArticle {
8   .bordered(blue; 1px);
9 }
10
11 // Es posible indicar el nombre del parámetro al invocar el mixin
12 // para evitar tener que respetar un orden en particular
13 .myArticle-2 {
14   .bordered(@width: 20px; @color: #33acfe);
15 }
```



{less}

```
1 // Herencia
2
3 .button-basic {
4   border: none;
5   padding: 15px 30px;
6   text-align: center;
7   font-size: 16px;
8   cursor: pointer;
9 }
10
11 .button-report {
12   &:extend(.button-basic);
13   background-color: red;
14 }
15
16 .button-submit {
17   &:extend(.button-basic);
18   background-color: green;
19   color: white;
20 }
```

Henry



Hacé click acá para dejar tu feedback sobre esta clase.

Hacé click acá completar el quiz teórico de esta lecture.

02 - CSS Avanzado

En esta Lesson se verán los siguientes temas:

- Frameworks CSS
- CSS Preprocessors

Frameworks CSS

En primer lugar un 'Framework' es un marco de referencia o marco de trabajo que nos provee distintas herramientas que se puede utilizar para facilitar el desarrollo de aplicaciones, ofreciéndonos una forma estándar y por lo general más simple para programar. En particular para el caso de un 'Framework CSS', se refiere a un conjunto de estilos predefinidos que pueden utilizarse para elaborar una interfaz de usuario atractiva sin necesidad de tener que definir a mano todas y cada una de las propiedades CSS de nuestros elementos HTML.

Existen una gran variedad de Frameworks CSS pero entre los más utilizados en la actualidad se encuentran:

- Bootstrap
- Foundation
- Bulma
- Ulkit
- Semantic UI

Bootstrap

En esta lesson nos centraremos en Bootstrap que es el más utilizado de ellos. Las ventajas que nos ofrece este Framework es que ya tiene componentes con estilos predefinidos que podemos reutilizar para ganar tiempo.

Por ejemplo supongamos que quisiéramos crear en nuestra página web un botón rojo con bordes redondeados y texto blanco:

```
<!DOCTYPE html>
<html>
<style media="screen">
button {
```

```
color: white;
background-color: red;
border-radius: 5px;
}
</style>
<body>
<button type="button">Click</button>
</body>
</html>
```

Con ese código lo que obtendríamos es lo siguiente:



Ahora bien, vamos a intentar lo mismo utilizando Bootstrap. Para ello necesitaremos agregar a nuestro HTML una referencia a la librería de Bootstrap para poder utilizar todos sus beneficios.

```
<!DOCTYPE html>
<html>
<head>
  <link rel="stylesheet"
    href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css"
    integrity="sha384-Vkoo8x4CGs03+Hhxv8T/Q5PaXtkKtu6ug5T0eNV6gBiFeWPGFN9MuhOf23Q9Ifjh"
    crossorigin="anonymous">
</head>
<style media="screen">
  .buttonComun {
    color: white;
    background-color: red;
    border-radius: 5px;
  }
</style>
<body>
<button type="button" class="buttonComun">Boton Común</button>
<button class="btn btn-danger">Boton Bootstrap</button>
</body>
</html>
```

Veamos que en el header se agrego un link hacia Bootstrap

Boton Común

Boton Bootstrap

Veamos ahora como quedó nuestra página:

No tuvimos que definir ninguna propiedad CSS para nuestro nuevo botón sino que simplemente le asignamos las clases `btn` y `btn-danger` y Bootstrap se encargó del resto.

Para ver que esto no es magia, lo que está pasando por detrás es que existe un archivo CSS 'enorme' con muchísimas clases definidas que ya contienen propiedades de estilos asignadas entonces cuando

nosotros le asignamos la clase `btn-danger` es como si estuviéramos escribiendo el siguiente código en nuestro archivo CSS o en la etiqueta `<style>` de nuestro HTML:

```
.btn-danger {  
    color: #fff;  
    background-color: #dc3545;  
    border-color: #dc3545;  
}
```

Lo mismo sucede con la clase `btn`, le aporta a nuestro elemento más propiedades CSS

En la página de [Bootstrap](#) podrán encontrar muchos componentes que pueden reutilizar en sus páginas web.

Responsive Design

Cuando queremos que nuestra página se vea 'linda' en cualquier dispositivo o cambie algunas características ya sea en una computadora, en un teléfono celular, en una tablet o incluso en un televisor smart, necesitamos hacer algunos ajustes a las propiedades de los elementos en función del dispositivo.

CSS Media Queries

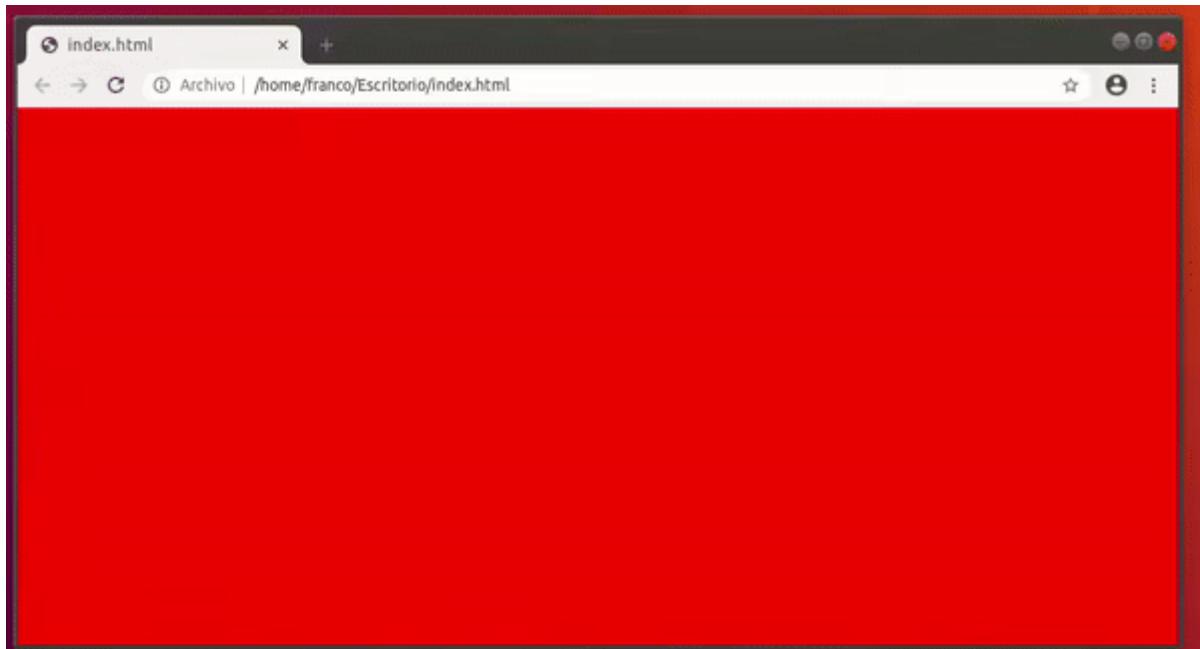
Para poder determinar que una propiedad solo se aplique en función del tamaño de la pantalla del dispositivo tenemos la posibilidad de usar [CSS Media Queries](#).

Supongamos que queremos modificar el color de fondo de la página web:

- Negro para una pantalla de 600px o menos de ancho
- Azul para una pantalla de entre 600px a 900px de ancho
- Rojo para una pantalla de más de 900px de ancho

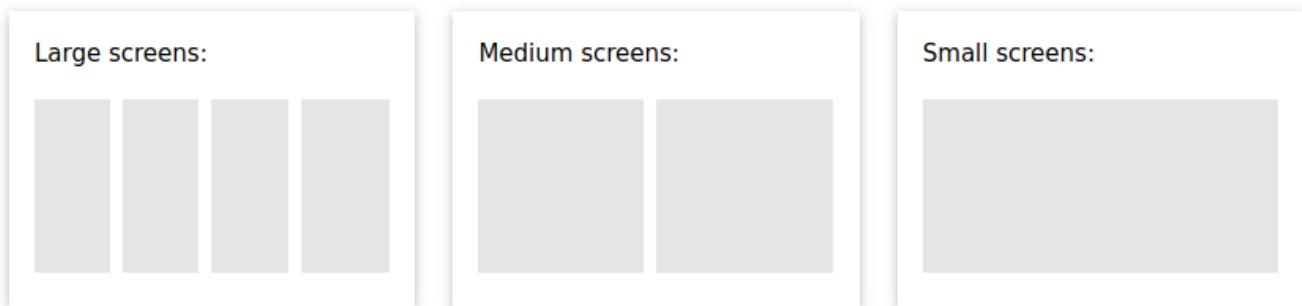
```
body {  
    background-color: red;  
}  
  
/* Pantallas de menos de 992px de ancho */  
@media screen and (max-width: 992px) {  
    body {  
        background-color: blue;  
    }  
}  
  
/* Pantallas de menos de 600px de ancho */  
@media screen and (max-width: 600px) {  
    body {  
        background-color: black;  
    }  
}
```

El resultado obtenido sería el siguiente:



Bootstrap

Supongamos ahora que queremos cambiar la cantidad de columnas que se muestren en función de la pantalla para que nos queden cuatro columnas en pantallas grandes, dos en medianas y una en pequeñas:



Podríamos realizarlo con CSS Media Queries similar al ejemplo anterior. Así que si quieren pueden intentarlo (Es un buen ejercicio para practicar lo que ya saben de CSS con esta nueva herramienta).

Pero ahora vamos a ver como solucionar esto utilizando el Framework que explicamos más arriba [Bootstrap](#).

Grid System

Bootstrap ya tiene integrado un sistema de grillas implementado a partir de flexbox que nos va a facilitar la tarea. Para ello utiliza cinco clases ya definidas:

- .col- (extra small devices - menos de 576px)
- .col-sm- (small devices - mayor o igual a 576px)
- .col-md- (medium devices - mayor o igual a 768px)
- .col-lg- (large devices - mayor o igual a 992px)
- .col-xl- (xlarge devices - mayor o igual a 1200px)

El sistema de grilla de Bootstrap permite colocar hasta una suma de 12 'espacios' por fila distribuyéndolos de la forma que se quiera, ya sea colocando 12 columnas de 1 'espacio', 2 columnas de 6 'espacios' o cualquier variante de combinaciones:

| | | | | | | | | | | | | | | | |
|---------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--------|--|--|--|--|
| span 1 | span 1 | span 1 | span 1 | span 1 | span 1 | span 1 | span 1 | span 1 | span 1 | span 1 | span 1 | | | | |
| span 4 | | | | span 4 | | | | span 4 | | | | | | | |
| span 4 | | | | span 8 | | | | | | | | | | | |
| span 6 | | | | | | span 6 | | | | | | | | | |
| span 12 | | | | | | | | | | | | | | | |

También existe la opción de dejar que Bootstrap identifique la cantidad de columnas que hay y a partir de ello le asigne el mismo ancho a cada una hasta completar la totalidad de la fila (Siempre recordando que el máximo es de 12). Para ello se utiliza simplemente la clase `.col` en cada columna

Utilizando simplemente esas clases podemos crear múltiples tipos de grillas que se adapten a nuestras pantallas.



En el gif de arriba podemos ver como en función del ancho de la pantalla va cambiando la cantidad de columnas

```
<div class="row">
  <div class="col-12 col-sm-6 col-md-3" style="background-color:black;
color:black;">.</div>
  <div class="col-12 col-sm-6 col-md-3" style="background-color:orange;
color:orange;">.</div>
  <div class="col-12 col-sm-6 col-md-3" style="background-color:yellow;
color:yellow;">.</div>
  <div class="col-12 col-sm-6 col-md-3" style="background-color:green;
color:green;">.</div>
</div>
```

La documentación completa la pueden encontrar [acá](#)

CSS Preprocessors

Un preprocesador CSS es un programa que te permite generar CSS a partir de la syntax única del preprocesador. Existen varios preprocesadores CSS de los cuales escoger, sin embargo la mayoría de preprocesadores CSS añadirán algunas características que no existen en CSS puro, como variable, mixins, selectores anidados, entre otros. Estas características hacen la estructura de CSS más legible y fácil de mantener.

Estos son algunos de los preprocesadores CSS más populares:

- SASS
- LESS
- Stylus
- PostCSS

Si quieren jugar un poco con distintos preprocesadores [codepen](#) nos brinda un entorno de fácil configuración. En la configuración del panel de CSS podemos seleccionar el que queramos

LESS (Leaner Style Sheets)

En esta lección nos centraremos en LESS ya que es uno de los más utilizados y tiene un tipo de sintaxis muy similar al código CSS pero con ciertos agregados por lo que va a ser más sencillo entenderlo.

Variables

LESS nos permite utilizar variables dentro de nuestro archivo de estilos para evitar la repetición innecesaria de definiciones de propiedades, por ejemplo no nos resulta tan fácil recordar un código de un color en formato hexadecimal en cambio si pudiéramos definirlo una única vez y asignárselo a una variable con un nombre representativo, nos sería mucho más sencillo.

Ejemplo de código SCSS:

```
@color-fondo: #F55;
@width: 10px;
@height: @width + 10px; /* También es posible realizar operaciones sobre las
variables */

h1 {
  background-color: @color-fondo;
  width: @width;
  height: @height;
}
```

En este ejemplo estamos creando variables con un color y medidas determinadas que van a poder ser reutilizadas en distintos componentes y clases las veces que queramos

Luego este código va a ser compilado en a un archivo CSS para que pueda ser interpretado por los navegadores por lo que el ejemplo anterior quedaría así:

```
h1 {
  background-color: #F55;
```

```
    width: 10px;  
    height: 20px;  
}
```

Al igual que en otros lenguajes de programación, las variables tienen un scope determinado, primero se analiza si en el contexto actual se encuentra definida dicha variable y si no la encuentra la buscará en el scope padre.

```
@var: red;  
  
#page {  
    @var: white;  
    color: @var; // white  
}
```

La variable de la primer línea podrá utilizarse dentro del resto de las definiciones pero la que se encuentra dentro de `#page` sólo será válida allí. Por otra parte, el valor correspondiente a la propiedad `color` va a ser `white` ya que en dicho contexto si se encuentra definido el valor de `@var`. En cambio si tuvieramos algo como lo siguiente:

```
@var: red;  
  
#page {  
    color: @var; // red  
}
```

En este caso el valor de `@var` sería `red` ya que en su contexto no está definida la variable pero en el contexto global sí por lo que toma su valor de allí.

También es posible utilizar variables dentro de los nombres de los selectores, de las propiedades e incluso en URL's.

Selectores

```
@my-selector: banner;  
  
.{@{my-selector} {  
    font-weight: bold;  
    line-height: 40px;  
    margin: 0 auto;  
}
```

Propiedades

```
@property: color;

.widget {
  @property: #0ee;
  background-@{property}: #999;
}
```

URLs

```
@images: "../img";

body {
  color: #444;
  background: url("@{images}/white-sand.png");
}
```

Lazy evaluation

No es necesario declarar las variables antes de usarlas, por lo que el siguiente código sería válido:

```
.lazy-eval {
  width: @var;
}

@var: 200px;
```

Funciones

LESS nos provee de ciertas funciones que nos permiten transformar colores, manipular strings y realizar cálculos matemáticos.

Ejemplo de utilización:

```
@base: #f04615;
@width: 0.5;

.class {
  width: percentage(@width);
  color: saturate(@base, 5%);
  background-color: spin(lighten(@base, 25%), 8);
}
```

En este caso por un lado con la función `percentage` estamos convirtiendo el valor `0.5` en `5%` y por otro lado, con la función `saturate` estamos incrementando la saturación del color base en un `5%`.

Para ver la documentación completa de las funciones disponibles ingresar [aquí](#)

Anidado

LESS también nos permite anidar definiciones de estilos CSS similar a como es una estructura HTML:

```
nav {  
  ul {  
    margin: 0;  
    padding: 0;  
    list-style: none;  
  }  
  li {  
    display: inline-block;  
  }  
  a {  
    display: block;  
    padding: 6px 12px;  
    text-decoration: none;  
  }  
}
```

En este caso estamos asignándole propiedades a los elementos `ul`, `li` y `a` que se encuentren dentro de un `nav`

Lo mismo puede realizarse con la directiva `@media`:

El siguiente código:

```
.component {  
  width: 300px;  
  @media (min-width: 768px) {  
    width: 600px;  
    @media (min-resolution: 192dpi) {  
      background-image: url(/img/retina2x.png);  
    }  
  }  
  @media (min-width: 1280px) {  
    width: 800px;  
  }  
}
```

Se traduciría en:

```
.component {  
  width: 300px;  
}  
@media (min-width: 768px) {  
  .component {
```

```
    width: 600px;
}
}

@media (min-width: 768px) and (min-resolution: 192dpi) {
  .component {
    background-image: url(/img/retina2x.png);
  }
}

@media (min-width: 1280px) {
  .component {
    width: 800px;
  }
}
```

Importación

La directiva `@import` nos permite incluir el contenido de otros archivos en el actual. Supongamos que tenemos un archivo less llamado "general.less" como el siguiente:

```
html,
body,
ul,
ol {
  margin: 0;
  padding: 0;
}
```

Podríamos importar dichas definiciones de atributos en otro less de la siguiente forma:

```
@import "general";

body {
  font-family: Helvetica, sans-serif;
  font-size: 18px;
  color: red;
}
```

De esta forma en nuestro último archivo de estilos también vamos a poder contener las definiciones de "general.less".

Observen que no es necesario aclarar la extensión del archivo `general`, LESS automáticamente asume que es un archivo de estilos válido

Mixins

Los mixins nos permiten incluir un set de propiedades ya definido dentro de otro.

```
.important-text {  
    color: black;  
    font-size: 25px;  
    font-weight: bold;  
}
```

Ahí estamos creando el mixin llamado `important-text` que luego podemos utilizar de la siguiente forma:

```
.danger {  
    .important-text();  
    background-color: red;  
}  
  
.success {  
    .important-text();  
    background-color: green;  
}
```

Esto se va a traducir a código CSS equivalente a:

```
.danger {  
    color: red;  
    font-size: 25px;  
    font-weight: bold;  
    border: 1px solid blue;  
    background-color: green;  
}
```

Es decir lo que sucedió es que se inyectaron todas las propiedades definidas en el mixin dentro de la clase `danger` y `success`. También es posible utilizar ids como mixins (`#b()`);

Parametros

Los mixin pueden recibir parámetros:

```
.bordered(@color; @width) {  
    border: @width solid @color;  
}  
  
.myArticle {  
    .bordered(blue; 1px);  
}  
  
// Es posible indicar el nombre del parámetro al invocar el mixin  
// para evitar tener que respetar un orden en particular  
.myArticle-2 {
```

```
.bordered(@width: 20px; @color: #33acf);  
}
```

Aquí lo que estamos haciendo es definir un mixin que recibe dos parámetros (*color* y *width*) que luego van a ser utilizados para definir el borde del elemento. Con ello podemos reutilizar el mixin simplemente llamándolo con diferentes colores o anchos como en el ejemplo que se le está dando un color azul y un borde de un pixel a los elementos con la clase *myArticle*

Valores por defecto

Adicionalmente se puede setear un valor por defecto para dichos parámetros para que, en el caso de que no se les indique un valor, tomen el por defecto:

```
.bordered(@color: blue; @width: 1px) {  
    border: @width solid @color;  
}  
  
.myArticle-default {  
    .bordered();  
}
```

Variable @arguments

La variable *@arguments* dentro de un mixin contiene todos los argumentos que le fueron suministrados a dicho mixin.

```
.box-shadow(@x: 0; @y: 0; @blur: 1px; @color: #000) {  
    box-shadow: @arguments;  
}  
.big-block {  
    .box-shadow(2px; 5px);  
}
```

Esto resultaría en:

```
.big-block {  
    box-shadow: 2px 5px 1px #000;  
}
```

Herencia

Por último también es posible, heredar/compartir las propiedades de un selector en otro. Esto es útil para aquellos casos en los que entre dos selectores comparten la mayor parte de los atributos pero tienen una o algunas pequeñas diferencias.

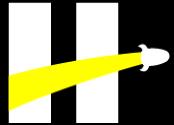
```
.button-basic {
  border: none;
  padding: 15px 30px;
  text-align: center;
  font-size: 16px;
  cursor: pointer;
}

.button-report {
  &:extend(.button-basic);
  background-color: red;
}

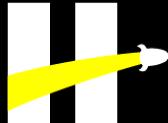
.button-submit {
  &:extend(.button-basic);
  background-color: green;
  color: white;
}
```

En este caso el botón de report y de submit extienden las propiedades del botón básico manteniendo todas sus propiedades pero agregándole algunas más que son propias de ellas

Con esto cubrimos la mayor parte de las funcionalidades agregadas por LESS pero existen otras que para aquel que le interese indagar aun más sobre este tema puede acceder a la documentación oficial [aquí](#)



ES6

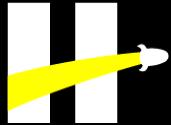


ECMA



Ecma International es una organización internacional basada en membresías de estándares para la comunicación y la información.

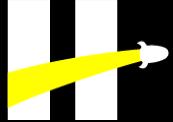
La organización fue fundada en 1961 para estandarizar los sistemas computarizados en Europa. La membresía está abierta a las empresas que producen, comercializan o desarrollan sistemas computacionales o de comunicación en Europa.



Nuevas Features

- Keywords *Let* y *Const*
- Funciones Flecha (Arrow Functions)
- Desestructuración (Destructuring)
- El operador Spread
- Literales de Template (Template Literals)

[Lista completa de features](#)



Let y Const

```
1 function f() {
2   {
3     let x;
4     {
5       // okay, block scoped name
6       const x = "sneaky";
7       // error, const
8       x = "foo";
9     }
10    // error, already declared in block
11    let x = "inner";
12  }
13}
```

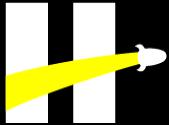


Let y var en Ciclo For

¿Qué pasaría si usamos let en vez de var?

```
1  var creaFuncion = function(){
2      var arreglo = [];
3      for ( var i=0; i < 3; i++){
4          arreglo.push(
5              function(){
6                  console.log(i);
7              }
8          )
9      }
10     return arreglo;
11 }
12
13 var arr = creaFuncion();
14 arr[0]() // 3
15 arr[1]() // 3
16 arr[2]() // 3
```

<Demo />



Arrow Functions

```
1 // Cuerpos con Expresiones
2
3 var pares = impares.map(v => v + 1);
4 var nums = pares.map((v, i) => v + i);
5
6 // Cuerpos con Statements
7
8 nums.forEach(v => {
9   if (v % 5 === 0)
10     cincos.push(v);
11 });
12
13 // this
14 var bob = {
15   _name: "Bob",
16   _friends: [],
17   printFriends() {
18     this._friends.forEach(f =>
19       console.log(this._name + " knows " + f));
20   }
21 }
```



Class



```
1 class Persona {  
2     constructor (nombre = 'Franco', apellido = "Etcheverri"){  
3         this.nombre = nombre,  
4             this.apellido = apellido  
5     }  
6  
7     getNombre() {  
8         return this.nombre + " " + this.apellido;  
9     }  
10 }  
11  
12 class Empleado extends Persona {  
13     constructor (nombre, apellido, empleo, sueldo){  
14         super(nombre, apellido);  
15         this.empleo = empleo;  
16         this.sueldo = sueldo;  
17     }  
18  
19     getEmpleo() {  
20         return this.empleo + "($" + this.sueldo + ")";  
21     }  
22 }  
23
```



Object Literals

```
1 var obj = {  
2   // __proto__  
3   __proto__: theProtoObj, //extiende el prototipo  
4   propiedad, // atajo para propiedad:propiedad  
5   // Methods  
6   toString() {  
7     // Super calls  
8     return "d " + super.toString();  
9   },  
10  // Computed (dynamic) property names  
11  [ 'prop_' + (() => 42)(): ]: 42  
12 };
```



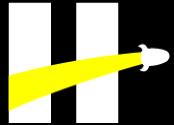
Template Strings

```
1 // Basic literal string creation
2 `In JavaScript '\n' is a line-feed.`
3
4 // Multiline strings
5 `In JavaScript this is
6 not legal.`
7
8 // String interpolation
9 var name = "Bob", time = "today";
10 `Hello ${name}, how are you ${time}?`
11
12 // Construct an HTTP request prefix is used to interpret
13 // the replacements and construction
14 POST`http://foo.org/bar?a=${a}&b=${b}
15     Content-Type: application/json
16     X-Credentials: ${credentials}
17     { "foo": ${foo},
18       "bar": ${bar}}`(myOnReadyStateChangeHandler);
```



Destructuring

```
1 // list matching
2 var [a, , b] = [1,2,3];
3
4 // object matching
5 var { op: a, lhs: { op: b }, rhs: c }
6     = getASTNode()
7
8 // object matching shorthand
9 // binds `op`, `lhs` and `rhs` in scope
10 var {op, lhs, rhs} = getASTNode()
11
12 // Can be used in parameter position
13 function g({name: x}) {
14   console.log(x);
15 }
16 g({name: 5})
17
18 // Fail-soft destructuring
19 var [a] = [];
20 a === undefined;
21
22 // Fail-soft destructuring with defaults
23 var [a = 1] = [];
24 a === 1;
```



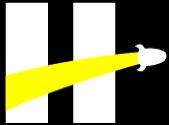
Default + Rest + Spread

```
1 function f(x, y=12) {  
2     // y is 12 if not passed (or passed as undefined)  
3     return x + y;  
4 }  
5 f(3) == 15  
6  
7 function f(x, ...y) {  
8  
9     // y is an Array  
10    return x * y.length;  
11 }  
12 f(3, "hello", true) == 6  
13  
14 function f(x, y, z) {  
15     return x + y + z;  
16 }  
17 // Pass each elem of array as argument  
18 f(...[1,2,3]) == 6
```



Promises

```
1 function timeout(duration = 0) {
2     return new Promise((resolve, reject) => {
3         setTimeout(resolve, duration);
4     })
5 }
6
7 var p = timeout(1000).then(() => {
8     return timeout(2000);
9 }).then(() => {
10     throw new Error("hmm");
11 }).catch(err => {
12     return Promise.all([timeout(100), timeout(200)]);
13 })
```



Compatibilidad

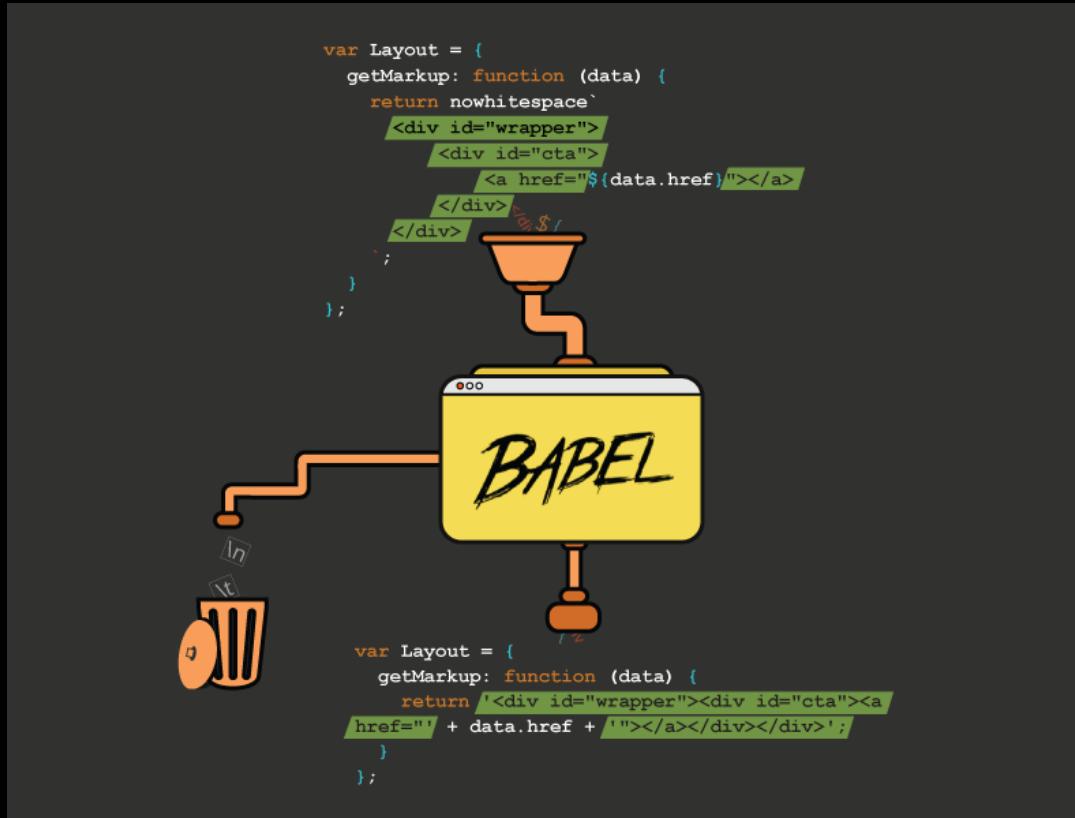
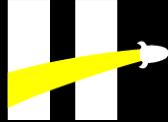
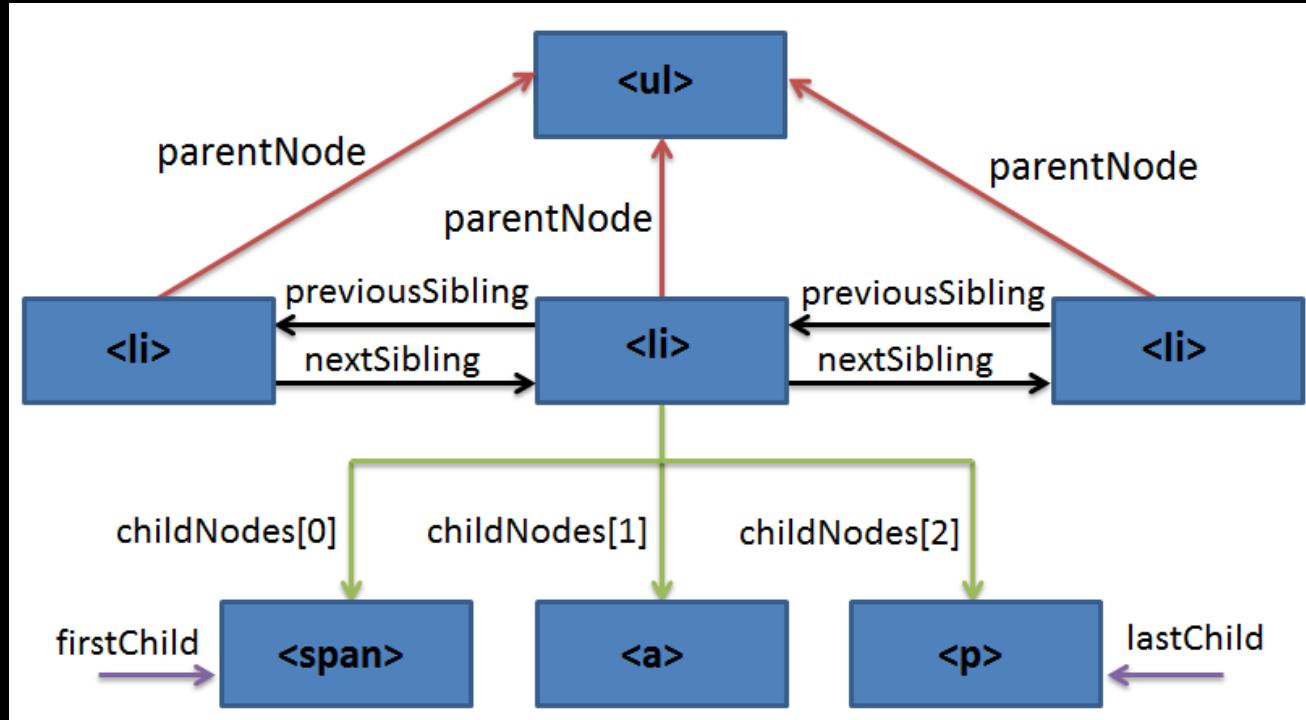


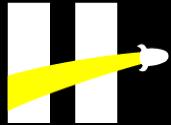
Tabla de compatibilidad



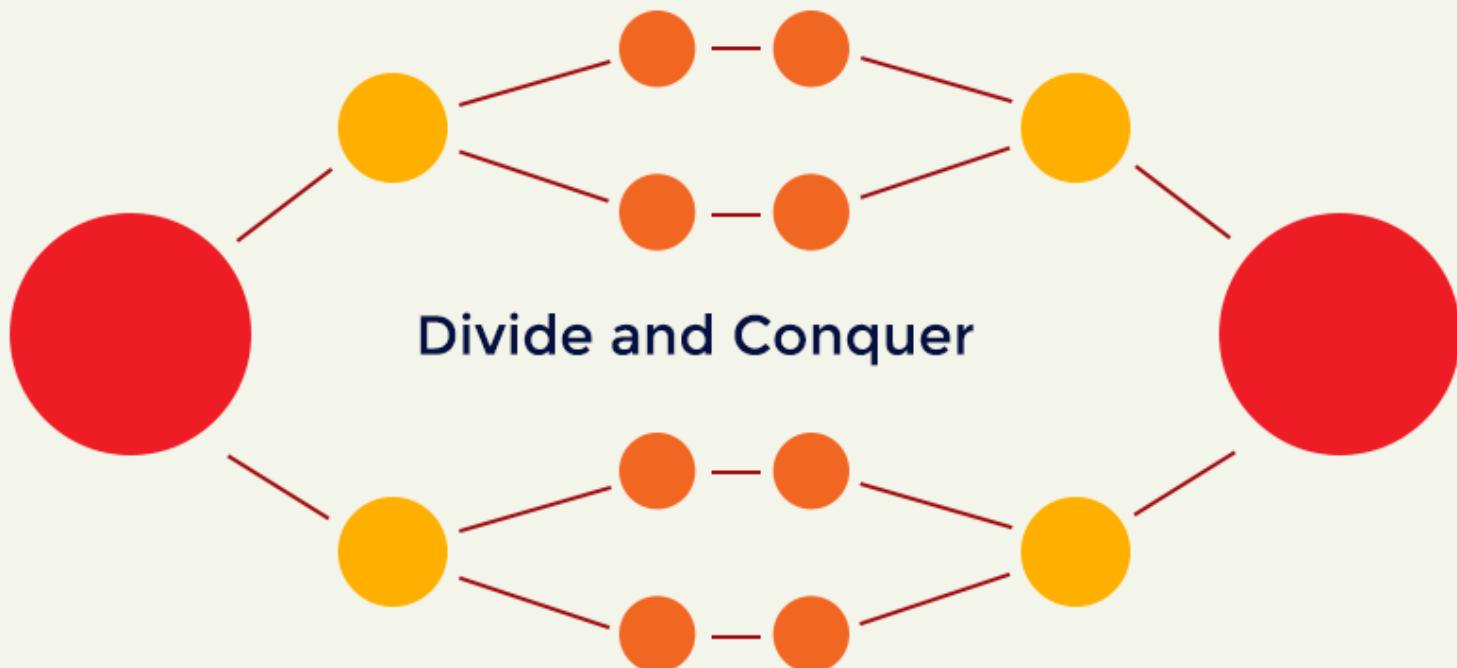
Homework



Github



Homework



Henry



Hacé click acá para dejar tu feedback sobre esta clase.



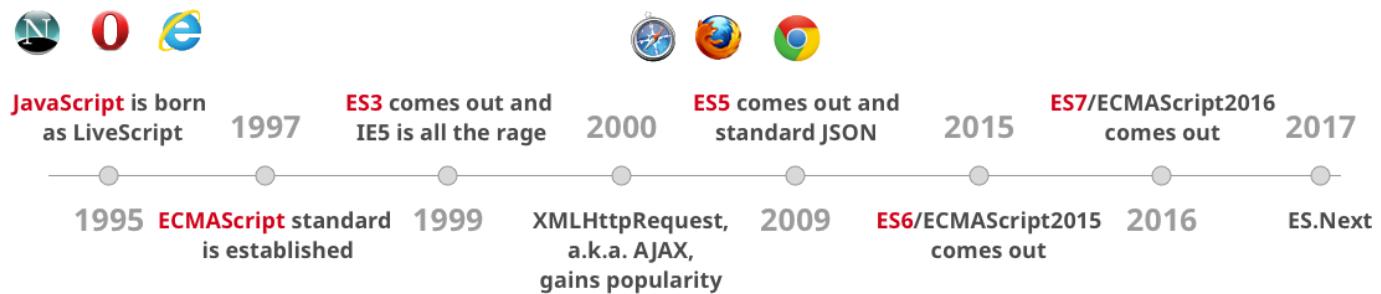
Hacé click acá completar el quiz teórico de esta lecture.

ECMAScript 6

El ES6 ,también conocido como ECMAScript 2015 o ES2015+, es la última versión del standart. Este nuevo update es el primero desde que se estandarizó el lenguaje en 2009. Las implementaciones del nuevo standart todavía se están haciendo para varios motores de JavaScript.

Historia

Desde su aparición en 1995, JS fue evolucionando lentamente. ECMAScript apareció como standart en 1997, y desde ahí viene lanzando nuevas versiones como ES3, ES5, ES6, etc..



Como ven entre ES3 y ES5 pasaron 10 años y entre ES5 y ES6 pasaron 6 años. Ahora la idea es lanzar nuevas versiones con cambios pequeños cada año.

Podemos ver un *mapa* de las compatibilidades actuales de varios engines con respecto al nuevo standart [acá](#)

| Feature name | Current browser | 97% | 100% | 100% | 97% | 97% | 96% | 93% | 92% | 86% | 83% | 71% | 59% | 59% | Type-Script + core-js | T | |
|--|-----------------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-------|-----------------------|-----------------------|--|
| Syntax | | | | | | | | | | | | | | | | | |
| • default function parameters | ► | 7/7 | 7/7 | 7/7 | 7/7 | 7/7 | 7/7 | 4/7 | 4/7 | 0/7 | 4/7 | 0/7 | 5/7 | 5/7 | Type-Script + core-js | | |
| • rest parameters | ► | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 3/5 | 0/5 | 4/5 | 4/5 | Type-Script + core-js | | |
| • spread (...) operator | ► | 15/15 | 15/15 | 15/15 | 15/15 | 15/15 | 15/15 | 15/15 | 15/15 | 15/15 | 13/15 | 11/15 | 4/15 | 4/15 | Type-Script + core-js | | |
| • object literal extensions | ► | 6/6 | 6/6 | 6/6 | 6/6 | 6/6 | 6/6 | 6/6 | 6/6 | 6/6 | 6/6 | 5/6 | 6/6 | 6/6 | Type-Script + core-js | | |
| • for..of loops | ► | 9/9 | 9/9 | 9/9 | 9/9 | 9/9 | 7/9 | 7/9 | 7/9 | 7/9 | 9/9 | 8/9 | 3/9 | 3/9 | Type-Script + core-js | | |
| • octal and binary literals | ► | 4/4 | 4/4 | 4/4 | 4/4 | 4/4 | 4/4 | 4/4 | 4/4 | 4/4 | 4/4 | 4/4 | 4/4 | 4/4 | 4/4 | Type-Script + core-js | |
| • template literals | ► | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 5/5 | 4/5 | 5/5 | 3/5 | 3/5 | Type-Script + core-js | | |
| • RegExp "y" and "u" flags | ► | 5/5 | 5/5 | 5/5 | 5/5 | 2/5 | 5/5 | 5/5 | 2/5 | 5/5 | 3/5 | 0/5 | 0/5 | 0/5 | Type-Script + core-js | | |
| • destructuring declarations | ► | 22/22 | 22/22 | 22/22 | 22/22 | 21/22 | 21/22 | 21/22 | 19/22 | 0/22 | 21/22 | 19/22 | 15/22 | 15/22 | Type-Script + core-js | | |
| • destructuring assignment | ► | 24/24 | 24/24 | 24/24 | 24/24 | 24/24 | 23/24 | 23/24 | 21/24 | 0/24 | 24/24 | 21/24 | 19/24 | 19/24 | Type-Script + core-js | | |
| • destructuring parameters | ► | 23/23 | 23/23 | 23/23 | 23/23 | 23/23 | 22/23 | 19/23 | 18/23 | 0/23 | 20/23 | 18/23 | 15/23 | 15/23 | Type-Script + core-js | | |
| • Unicode code point escapes | ► | 2/2 | 2/2 | 2/2 | 2/2 | 2/2 | 2/2 | 1/2 | 1/2 | 2/2 | 1/2 | 2/2 | 1/2 | 1/2 | Type-Script + core-js | | |
| • new.target | ► | 2/2 | 2/2 | 2/2 | 2/2 | 2/2 | 2/2 | 2/2 | 2/2 | 1/2 | 0/2 | 0/2 | 0/2 | 0/2 | Type-Script + core-js | | |
| Bindings | | | | | | | | | | | | | | | | | |
| • const | ► | 16/16 | 16/16 | 16/16 | 16/16 | 16/16 | 16/16 | 12/16 | 12/16 | 12/16 | 14/16 | 10/16 | 14/16 | 14/16 | Type-Script + core-js | | |
| • let | ► | 12/12 | 12/12 | 12/12 | 12/12 | 12/12 | 12/12 | 10/12 | 10/12 | 10/12 | 10/12 | 0/12 | 10/12 | 10/12 | Type-Script + core-js | | |
| • block-level function declaration ^[13] | ► | Yes | No | Yes | Yes | No | No | No | Type-Script + core-js | | |
| Functions | | | | | | | | | | | | | | | | | |
| • arrow functions | ► | 13/13 | 13/13 | 13/13 | 13/13 | 13/13 | 12/13 | 13/13 | 13/13 | 13/13 | 9/13 | 0/13 | 9/13 | 9/13 | Type-Script + core-js | | |
| • class | ► | 24/24 | 24/24 | 24/24 | 24/24 | 24/24 | 24/24 | 24/24 | 24/24 | 24/24 | 19/24 | 18/24 | 19/24 | 19/24 | Type-Script + core-js | | |
| • super | ► | 8/8 | 8/8 | 8/8 | 8/8 | 8/8 | 8/8 | 8/8 | 8/8 | 8/8 | 4/8 | 7/8 | 7/8 | 7/8 | Type-Script + core-js | | |
| • generators | ► | 27/27 | 27/27 | 27/27 | 27/27 | 27/27 | 27/27 | 25/27 | 25/27 | 27/27 | 24/27 | 0/27 | 0/27 | 0/27 | Type-Script + core-js | | |

Como todavía no es compatible con muchos browsers, para poder utilizarlo vamos a utilizar una librería llamada [babel.js](#) que nos servirá para traducir código ES6 a la versión actual para así mantener compatibilidad con los engines actuales.

Primero veamos algunas de las cosas que cambiaron en el nuevo standart.

Nuevas Features

ES6 incluye las siguientes features:

- [let + const \(Block Scoping\)](#)
- [arrows =>](#)
- [classes](#)
- [object literals mejorados](#)
- [template strings](#)
- [destructuring](#)
- [default + rest + spread](#)
- [iterators + for..of](#)
- [generators](#)
- [unicode](#)
- [modules](#)
- [module loaders](#)
- [map + set + weakmap + weakset](#)
- [proxies](#)
- [symbols](#)
- [subclassable built-ins](#)

- promises
- math + number + string + array + object APIs
- binary and octal literals
- reflect api
- tail calls
- Optional Chaining

ECMAScript 6 Features

Let + Const

`let` es el nuevo `var`. Sólo que `let` tiene un scope distinto, este está declarado sólo dentro del `bloque` donde aparece y no dentro del scope (por ejemplo si uso `let` dentro de un `for` no voy a poder ver esa variable afuera del mismo, está bien claro `acá`). `const` es para declarar variables inmutables o sea que no pueden cambiar. Si queremos asignar un nuevo valor a una variable declara con `const` vamos a obtener un Error.

```
function f() {
{
  let x;
  {
    // okay, block scoped name
    const x = "sneaky";
    // error, const
    x = "foo";
  }
  // error, already declared in block
  let x = "inner";
}
}
```

Más Info: [let statement](#), [const statement](#)

Arrows

Arrows (o flechas) son una forma de abreviar la declaración de una función y utiliza la sintaxis `=>` (está inspirada en sintaxis similares de C#, Java 8 y CoffeeScript). Estas soportan cuerpos que sean statements (ifs, fors, etc..) y también cuerpos que retornen el resultado de una expresión. A diferencia de las funciones normales, las funciones **arrows comparten el mismo `this` que el código que las rodea**.

```
// Cuerpos con Expresiones
var pares = impares.map(v => v + 1);
var nums = pares.map((v, i) => v + i);

// Cuerpos con Statements
nums.forEach(v => {
  if (v % 5 === 0)
    cincos.push(v);
});
```

```
// this
var bob = {
  _name: "Bob",
  _friends: [],
  printFriends() {
    this._friends.forEach(f =>
      console.log(this._name + " knows " + f));
  }
}
```

Más información: [MDN Arrow Functions](#)

Classes

Las clases en ES6 son simplemente syntax sugar sobre el patrón de prototipado de objetos. Tener una forma particular de declarar clases, hace que sea más fácil de usar y mejora la interoperabilidad. Las clases soportan la herencia basada en prototipos, llamadas a super, instance y métodos estáticos y constructores.

```
class SkinnedMesh extends THREE.Mesh {
  constructor(geometry, materials) {
    super(geometry, materials);

    this.idMatrix = SkinnedMesh.defaultMatrix();
    this.bones = [];
    this.boneMatrices = [];
    //...
  }
  update(camera) {
    //...
    super.update();
  }
  get boneCount() {
    return this.bones.length;
  }
  set matrixType(matrixType) {
    this.idMatrix = SkinnedMesh[matrixType]();
  }
  static defaultMatrix() {
    return new THREE.Matrix4();
  }
}
```

Más info: [MDN Classes](#)

Object Literals Mejorados

Ahora los Object Literals se extendieron para setear el prototipo durante su construcción, atajos para cuando hacemos una asignación de este tipo: **propiedad: propiedad**, definimos métodos, cuando hacemos llamadas a super, y al computar nombres de propiedades en una expresión. Todo esto junto mejora el

proceso de diseño orientado a objetos ya que trabaja en sinergía con las mejoras en las clases antes mencionadas.

```
var obj = {
  // __proto__
  __proto__: theProtoObj, //extiende el prototipo
  propiedad, // atajo para propiedad:propiedad
  // Methods
  toString() {
    // Super calls
    return "d " + super.toString();
  },
  // Computed (dynamic) property names
  [ 'prop_' + ((() => 42)()) ]: 42
};
```

Más Info: [MDN Grammar and types: Object literals](#)

Template Strings

Los Template Strings son una syntax sugar para la construcción de strings. Es parecido a la interpolación de strings de Perl, Python y otros lenguajes.

```
// Basic literal string creation
`In JavaScript '\n' is a line-feed.`

// Multiline strings
`In JavaScript this is
not legal.`

// String interpolation
var name = "Bob", time = "today";
`Hello ${name}, how are you ${time}?`

// Construct an HTTP request prefix is used to interpret the replacements and
construction
POST`http://foo.org/bar?a=${a}&b=${b}
Content-Type: application/json
X-Credentials: ${credentials}
{ "foo": ${foo},
  "bar": ${bar}}` (myOnReadyStateChangeHandler);
```

Más Info: [MDN Template Strings](#)

Destructuring

Las estructuras se pueden desestructurar para poder seleccionar valores usando patrones de matcheo, con soporte para arreglos y objetos. Esto funciona igual que buscar una propiedad de un objeto `foo['bar']` en

el sentido que ambos son **fail-soft**, es decir, que producen un **`undefined`** cuando algo no se encuentra.

```
// list matching
var [a, , b] = [1,2,3];

// object matching
var { op: a, lhs: { op: b }, rhs: c }
    = getASTNode()

// object matching shorthand
// binds `op`, `lhs` and `rhs` in scope
var {op, lhs, rhs} = getASTNode()

// Can be used in parameter position
function g({name: x}) {
  console.log(x);
}
g({name: 5})

// Fail-soft destructuring
var [a] = [];
a === undefined;

// Fail-soft destructuring with defaults
var [a = 1] = [];
a === 1;
```

Más Info: [MDN Destructuring assignment](#)

Default + Rest + Spread

Las declaraciones de funciones pueden tener argumentos que defaultean a un valor si no son declarados. También se puede transformar un arreglo en argumentos consecutivos. Y al revés, o sea podemos bindear los últimos elementos de los argumentos como si fuera un sólo arreglo.

```
function f(x, y=12) {
  // y is 12 if not passed (or passed as undefined)
  return x + y;
}
f(3) == 15
```

```
function f(x, ...y) {

  // y is an Array
  return x * y.length;
}
f(3, "hello", true) == 6
```

```
function f(x, y, z) {
  return x + y + z;
}
// Pass each elem of array as argument
f(...[1,2,3]) == 6
```

Más Info: [Default parameters](#), [Rest parameters](#), [Spread Operator](#)

Iterators + For..Of

Un objeto es un iterador cuando sabe como acceder a una colección de items de a uno, mientras mantiene su posición dentro de esa secuencia. En ES6 un iterador es cualquier cosa que provea un método `next()` que retorna el próximo item en la secuencia. También se generalizó el `for...in` para poder usar `for...of` en iteradores.

```
let fibonacci = {
  [Symbol.iterator]() {
    let pre = 0, cur = 1;
    return {
      next() {
        [pre, cur] = [cur, pre + cur];
        return { done: false, value: cur }
      }
    }
  }

  for (var n of fibonacci) {
    // truncate the sequence at 1000
    if (n > 1000)
      break;
    console.log(n);
  }
}
```

Más info: [MDN for...of](#)

Generators

Los iteradores son herramientas muy útiles, hay que tener mucho cuidado cuando los programamos porque necesitan mantener internamente su estado todo el tiempo. Los generadores nos proveen una alternativa poderosa: te dejan definir un algoritmo iterativo escribiendo una función que mantenga su propio estado.

```
function* idMaker(){
  var index = 0;
  while(true)
    yield index++;
```

```
}

var gen = idMaker();

console.log(gen.next().value); // 0
console.log(gen.next().value); // 1
console.log(gen.next().value); // 2
// ...
```

Más info: [MDN Iteration protocols](#)

Unicode

Hicieron adiciones para mejorar el soporte con Unicode.

```
// same as ES5.1
"𠮷".length == 2

// new RegExp behaviour, opt-in 'u'
"𠮷".match(/./u)[0].length == 2

// new form
"\u{20BB7}"=="𠮷"=="\uD842\uDFB7"

// new String ops
"𠮷".codePointAt(0) == 0x20BB7

// for-of iterates code points
for(var c of "𠮷") {
  console.log(c);
}
```

Más Info: [MDN RegExp.prototype.unicode](#)

Modules

Ahora Javascript soporta módulos de forma nativa (antes era mediante [CommonJS](#)), es muy parecido ya que están basados en la filosofía de lo que ya veníamos usando.

```
// lib/math.js
export function sum(x, y) {
  return x + y;
}
export var pi = 3.141593;
```

```
// app.js
import * as math from "lib/math";
```

```
alert("2π = " + math.sum(math.pi, math.pi));
```

```
// otherApp.js
import {sum, pi} from "lib/math";
alert("2π = " + sum(pi, pi));
```

Ahora podemos distinguir los **named exports**, que son básicamente todos los que exporten una variable con un nombre, por ejemplo: `export var pi = 3.14;`. Podemos tener muchos de estos por archivo. También existe el **Default Export**: Sólo puede haber *uno* por archivo, este es más parecido a la vieja forma de exportar. Generalmente lo usamos cuando, al importar, queremos que esté *todo* dentro de un objeto y no importar *varios* objetos separados.

```
// lib/mathplusplus.js
export * from "lib/math";
export var e = 2.71828182846;
export default function(x) {
    return Math.log(x);
}
```

```
// app.js
import ln, {pi, e} from "lib/mathplusplus";
alert("2π = " + ln(e)*pi*2);
```

Más info: [import statement](#), [export statement](#)

Map + Set + WeakMap + WeakSet

Nuevas Estructuras de datos. En general basadas en estructuras muy usadas en otros lenguajes.

```
// Sets
var s = new Set();
s.add("hello").add("goodbye").add("hello");
s.size === 2;
s.has("hello") === true;

// Maps
var m = new Map();
m.set("hello", 42);
m.set(s, 34);
m.get(s) == 34;

// Weak Maps
var wm = new WeakMap();
wm.set(s, { extra: 42 });
```

```

wm.size === undefined

// Weak Sets
var ws = new WeakSet();
ws.add({ data: 42 });
// Because the added object has no other references, it will not be held in the
set

```

Más Info: [Map](#), [Set](#), [WeakMap](#), [WeakSet](#)

Subclassable Built-ins

En ES6, objetos nativos como [Array](#), [Date](#) y elementos del [DOM](#) pueden ser heredados por una subclase.

Object construction for a function named [Ctor](#) now uses two-phases (both virtually dispatched):

- Call [Ctor\[@@create\]](#) to allocate the object, installing any special behavior
- Invoke constructor on new instance to initialize

The known [@@create](#) symbol is available via [Symbol.create](#). Built-ins now expose their [@@create](#) explicitly.

```

// Pseudo-code of Array
class Array {
    constructor(...args) { /* ... */ }
    static [Symbol.create]() {
        // Install special [[DefineOwnProperty]]
        // to magically update 'length'
    }
}

// User code of Array subclass
class MyArray extends Array {
    constructor(...args) { super(...args); }
}

// Two-phase 'new':
// 1) Call @@create to allocate object
// 2) Invoke constructor on new instance
var arr = new MyArray();
arr[1] = 12;
arr.length == 2

```

Math + Number + String + Array + Object APIs

Se agregaron cosas nuevas en las librerías nativas, incluyendo librerías matemáticas, funciones para convertir arreglos, funciones para trabajar Strings, y `Object.assign` para copiar objetos.

```

Number.EPSILON
Number.isInteger(Infinity) // false

```

```

Number.isNaN("NaN") // false

Math.acosh(3) // 1.762747174039086
Math.hypot(3, 4) // 5
Math.imul(Math.pow(2, 32) - 1, Math.pow(2, 32) - 2) // 2

"abcde".includes("cd") // true
"abc".repeat(3) // "abcababc"

Array.from(document.querySelectorAll('*')) // Returns a real Array
Array.of(1, 2, 3) // Similar to new Array(...), but without special one-arg
behavior
[0, 0, 0].fill(7, 1) // [0,7,7]
[1, 2, 3].find(x => x == 3) // 3
[1, 2, 3].findIndex(x => x == 2) // 1
[1, 2, 3, 4, 5].copyWithin(3, 0) // [1, 2, 3, 1, 2]
["a", "b", "c"].entries() // iterator [0, "a"], [1,"b"], [2,"c"]
["a", "b", "c"].keys() // iterator 0, 1, 2
["a", "b", "c"].values() // iterator "a", "b", "c"

Object.assign(Point, { origin: new Point(0,0) })

```

Más info: [Number](#), [Math](#), [Array.from](#), [Array.of](#), [Array.prototype.copyWithin](#), [Object.assign](#)

Binary y Octal Literals

Dos nuevas formas literales agregadas para binario (`b`) y octal (`o`).

```

0b111110111 === 503 // true
0o767 === 503 // true

```

Promises

Promises es una librería para mejorar la programación asíncrona. Las Promises son una representación de tipo first-class de un valor que va a estar disponible en el futuro. Esto también ya existia con otras librerías de terceros.

```

function timeout(duration = 0) {
  return new Promise((resolve, reject) => {
    setTimeout(resolve, duration);
  })
}

var p = timeout(1000).then(() => {
  return timeout(2000);
}).then(() => {
  throw new Error("hmm");
}).catch(err => {

```

```
    return Promise.all([timeout(100), timeout(200)]);
})
```

Más info: [MDN Promise](#)

Tail Calls

Se puede implementar llamadas recursivas sin tener que agregar un frame al `call stack` haciendo que sea segura la ejecución de una función recursiva (sin temer por el stack overflow).

```
function factorial(n, acc = 1) {
  'use strict';
  if (n <= 1) return acc;
  return factorial(n - 1, n * acc);
}

// Stack overflow in most implementations today,
// but safe on arbitrary inputs in ES6
factorial(100000)
```

Podemos ver una lista de features más detallada y comparada con la versión anterior [aquí](#)

Optional Chaining

El operador de encadenamiento opcional `?.` permite leer el valor de una propiedad ubicada dentro de una cadena de objetos conectados sin tener que validar expresamente que cada referencia en la cadena sea válida. El operador `?.` funciona de manera similar a el operador de encadenamiento `.`, excepto que en lugar de causar un error si una referencia es `null` o `undefined`, la expresión hace una short-circuit evaluation (`la evaluacion del segundo termino solo se efectua si el primero no permite definir un resultado ej: (true || false)---> da verdadero sin siquiera pasar por false`) con un valor de retorno de `undefined`. Cuando se usa con llamadas a funciones, devuelve `undefined` si la función dada no existe. Esto da como resultado expresiones más cortas y simples cuando se accede a propiedades encadenadas dónde existe la posibilidad de que falte una referencia. También puede ser útil al explorar el contenido de un objeto cuando no hay una garantía conocida de qué propiedades se requieren.

```
const adventurer = {
  name: 'Alice',
  cat: {
    name: 'Dinah'
  }
};

const dogName = adventurer.dog?.name;
console.log(dogName);
// expected output: undefined

console.log(adventurer.someNonExistentMethod?().());
```

```
// expected output: undefined
```

Compatibilidad

Para poder utilizar la sintaxis y las nuevas funcionalidad de ES6 en los motores no compatibles, vamos a utilizar [Babel.js](#)

Para instalarlo:

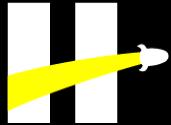
```
# install the cli and this preset
npm install --save-dev @babel/core @babel/cli

# make a .babelrc (config file) with the preset
npm install @babel/preset-env
echo '{ "presets": ["@babel/preset-env"] }' > .babelrc

# create a file to run on
echo 'console.log([1, 2, 3].map(n => n + 1));' > index.js

# run it
./node_modules/.bin/babel-node index.js
```

De esa forma vamos a poder transformar un archivo y lo tenemos que hacer manualmente. Si queremos que sea automático tenemos muchísimas opciones. Veamos algunas en la página de [babel](#) y elijamos el setup que más nos guste.

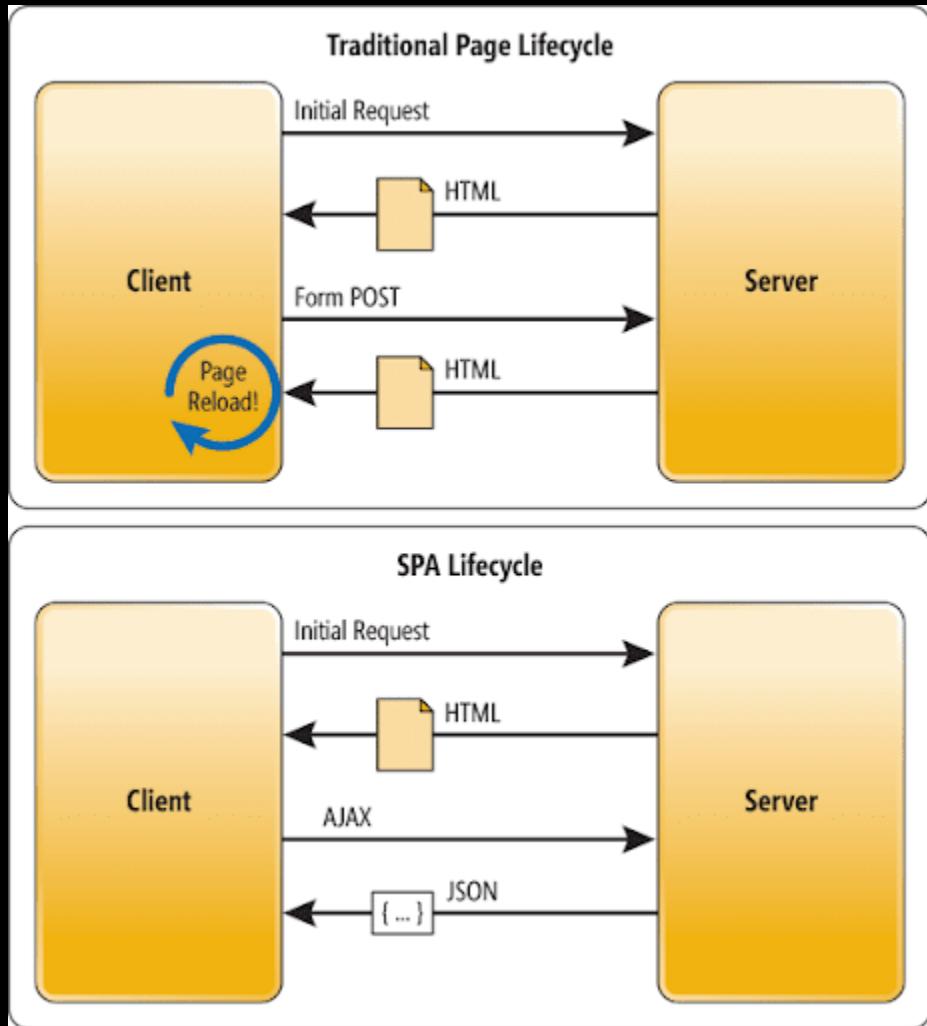


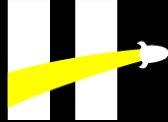
AJAX



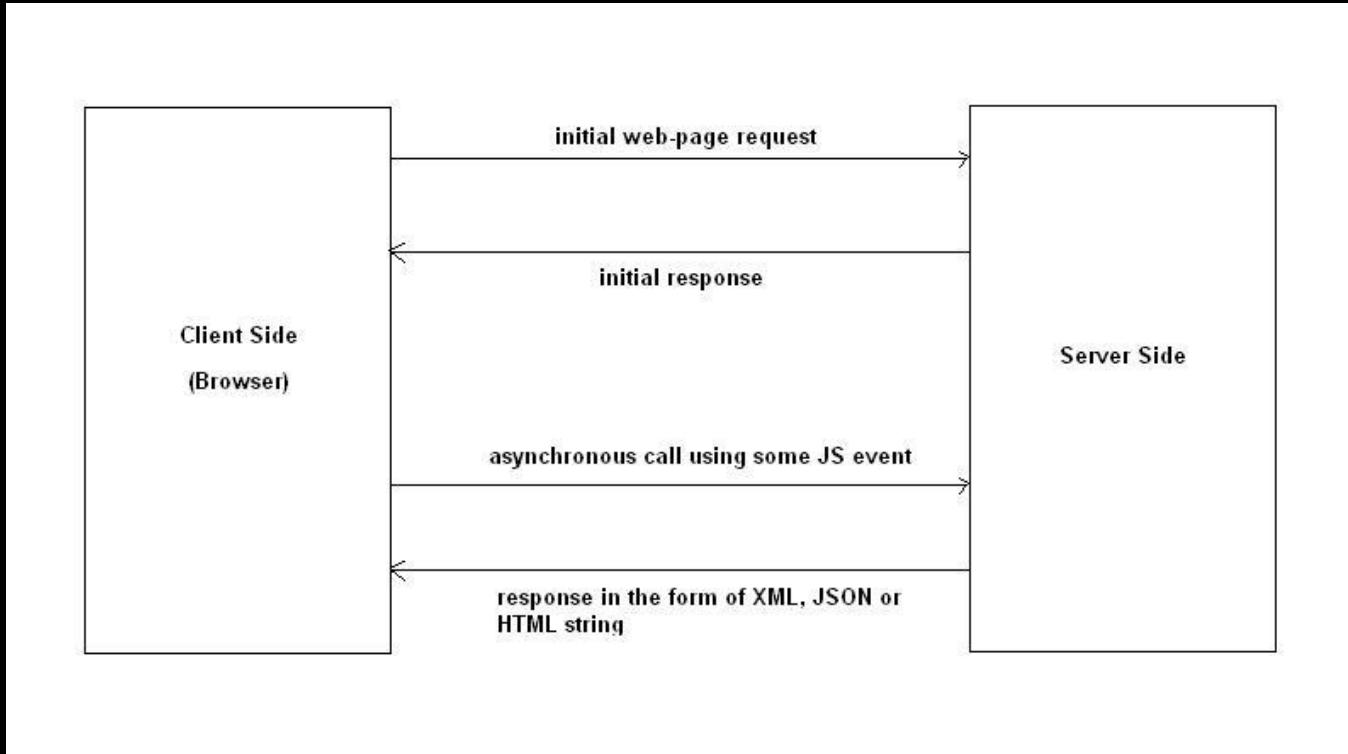
Historia

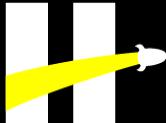
1. JavaScript - 1995
2. ECMA Script - 1997
3. AJAX by Google - 2004.
4. Chrome browser y V8 engine.
5. Aparición de Frameworks de Front-end





AJAX





XML VS JSON

XML

```
<empinfo>
  <employees>
    <employee>
      <name>James Kirk</name>
      <age>40</age>
    </employee>
    <employee>
      <name>Jean-Luc Picard</name>
      <age>45</age>
    </employee>
    <employee>
      <name>Wesley Crusher</name>
      <age>27</age>
    </employee>
  </employees>
</empinfo>
```

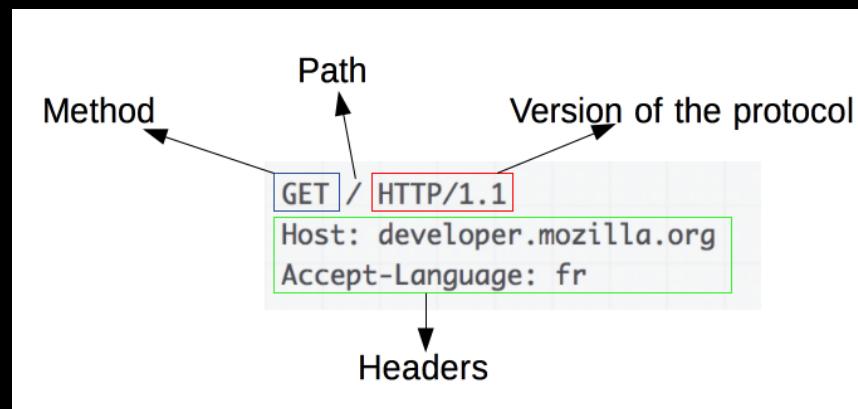
JSON

```
{  "empinfo" :
  {
    "employees": [
      {
        "name" : "James Kirk",
        "age" : 40,
      },
      {
        "name" : "Jean-Luc Picard",
        "age" : 45,
      },
      {
        "name" : "Wesley Crusher",
        "age" : 27,
      }
    ]
  }
}
```



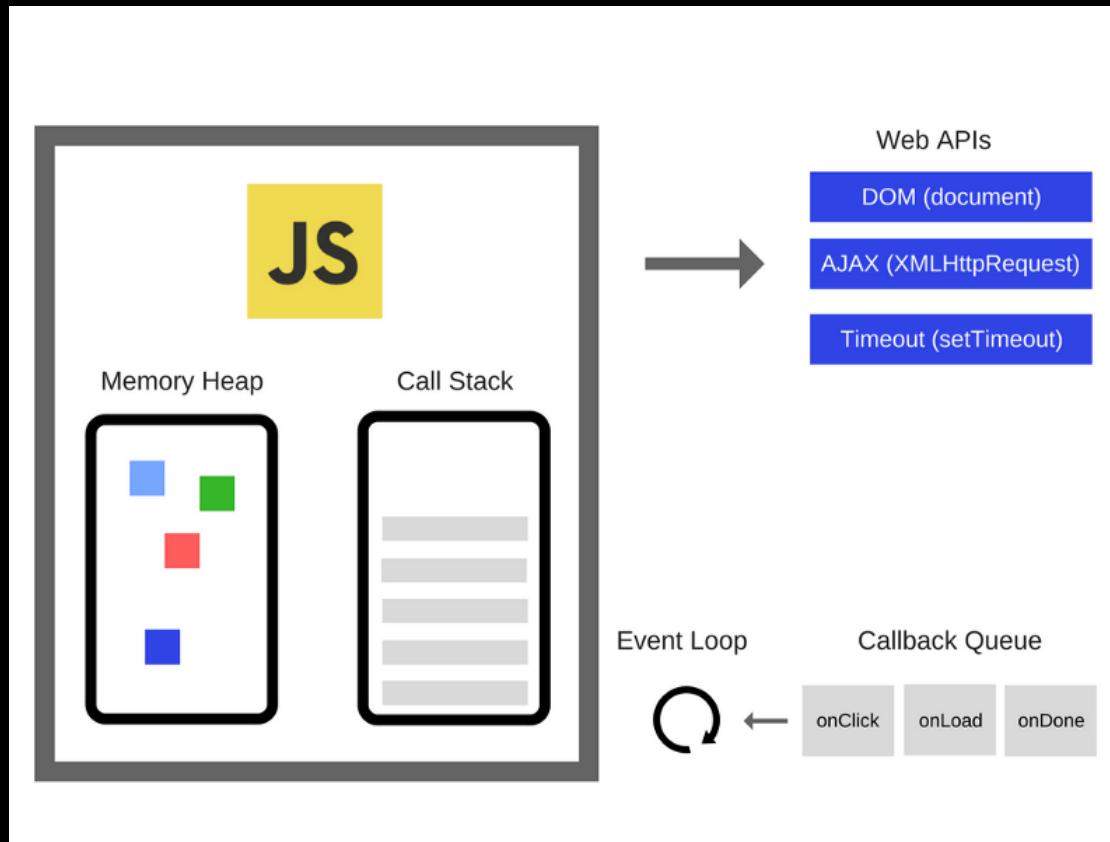
HTTP Methods

| | | |
|--------|--------------------------|------------------------|
| GET | /pet/{petId} | Find pet by ID |
| PUT | /pet | Update an existing pet |
| DELETE | /pet/{petId} | Deletes a pet |
| POST | /pet/{petId}/uploadImage | uploads an image |





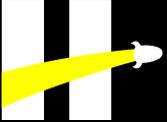
Eventos





Eventos

```
1 $('.answer').click(function(){
2   let meal = {
3     location: 'here',
4     condiments: 'ketchup',
5     idNumber: 191,
6   };
7
8   //data contains the data from the server
9   $.get('/comboMeal', meal, function(data){
10     //eat is a made-up function but you get the point
11     eat(data);
12   });
13 });
```



Demo

```
1 $('.answer').click(function(){
2     //data contains the data from the server
3     $.get('https://jsonplaceholder.typicode.com/', function(data){
4         //eat is a made-up function but you get the point
5         console.log(data);
6     });
7 });
```

Henry



Hacé click acá para dejar tu feedback sobre esta clase.



Hacé click acá completar el quiz teórico de esta lecture.

AJAX

Que es AJAX?

AJAX son las siglas de Asynchronous JavaScript and XML. XML es raramente relevante, pero cuando desarrollamos aplicaciones web, usamos Ajax para hacer cosas asincrónicas como actualizar una página, hacer acciones, etc.

En resumen, AJAX se trata de actualizar partes de una página web sin tener que recargar toda la web. Eso es muy útil si tu sitio web es grande , no querrá que tus usuarios tengan que cargar la misma información varias veces.

En que se basa Ajax ?

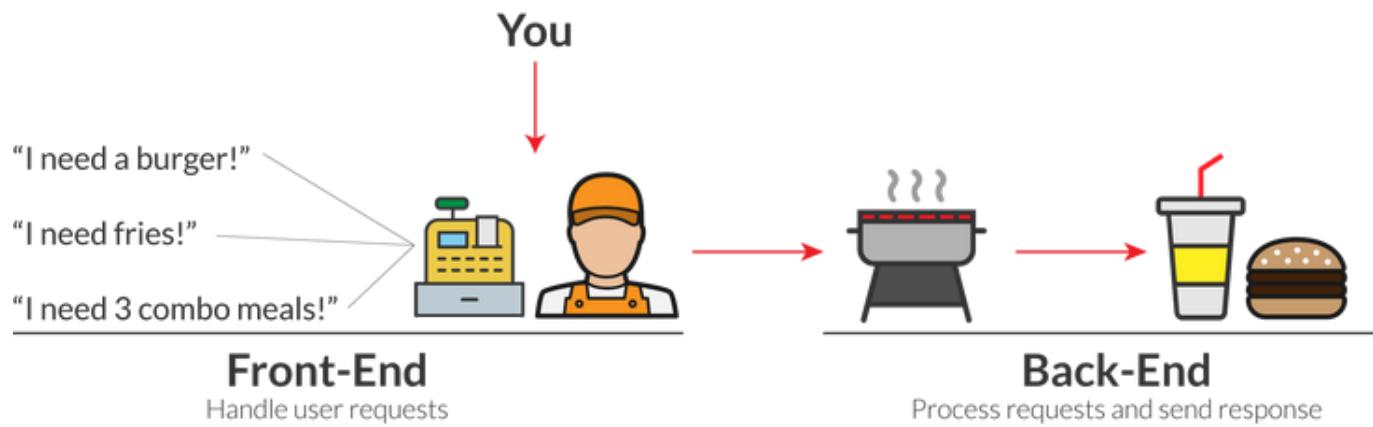
AJAX se basa en un montón de tecnologías. No tienes que ser un experto en todas ellas pero un poco de conocimiento de fondo será útil. Esto debería darte los antecedentes suficientes para que te hagas una idea de AJAX.

Las tecnologías que forman AJAX son:

- XHTML y CSS, para crear una presentación basada en estándares.
- DOM, para la interacción y manipulación dinámica de la presentación.
- XML, XSLT y JSON, para el intercambio y la manipulación de información.
- XMLHttpRequest, para el intercambio asíncrono de información.
- JavaScript, para unir todas las demás tecnologías.

Para que necesitamos Ajax?

Piensa en toda tu aplicación web como un restaurante de comida rápida. Tú eres el cajero, la persona en las primeras líneas. Manejas las **solicitudes** de los cliente

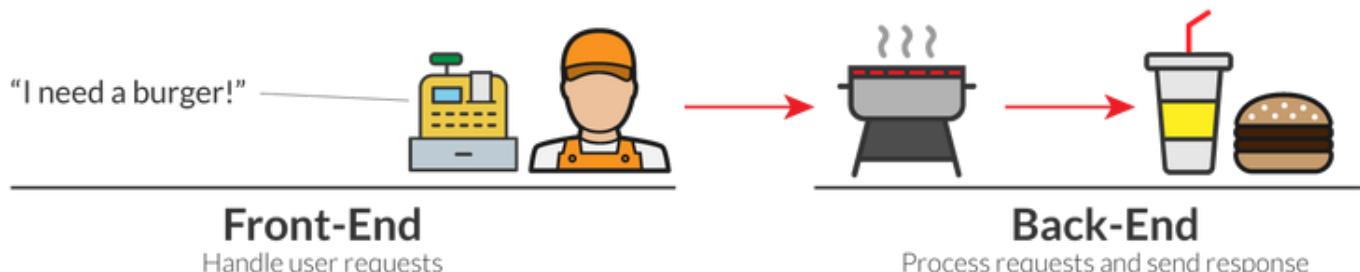


Si miras este diagrama, puedo ver tres trabajos separados que deben hacerse.

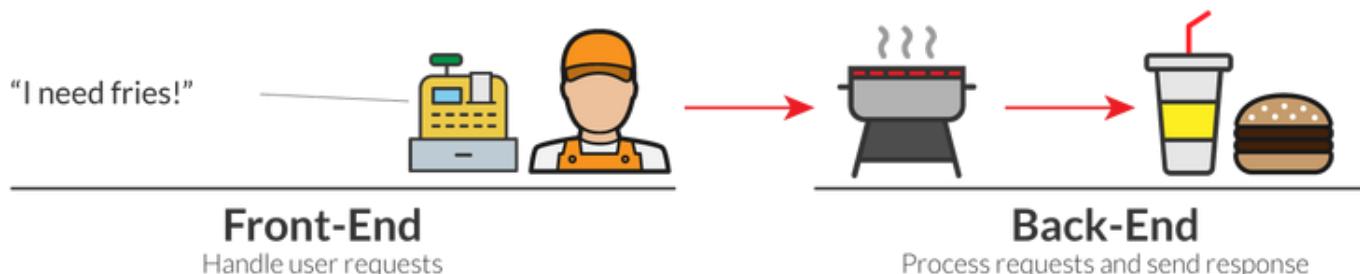
1. El cajero debe manejar las solicitudes de los usuarios a un ritmo rápido.
2. Necesita que los cocineros tiren las hamburguesas a la parrilla y cocinen toda la comida.
3. Necesitas un equipo de preparación de comida para empaquetar la comida y ponerla en una bolsa o en una bandeja.

Sin embargo, si no tuvieras AJAX, sólo se te permitiría procesar un pedido a la vez de principio a fin! Tendrías que tomar el pedido... luego cobrar al cliente... luego sentarte ahí sin hacer nada mientras la gente en la cocina cocina la comida... y luego seguir esperando mientras el equipo de preparación de la comida la empaqueta. Sólo podrías tomar el siguiente pedido después de todo eso.

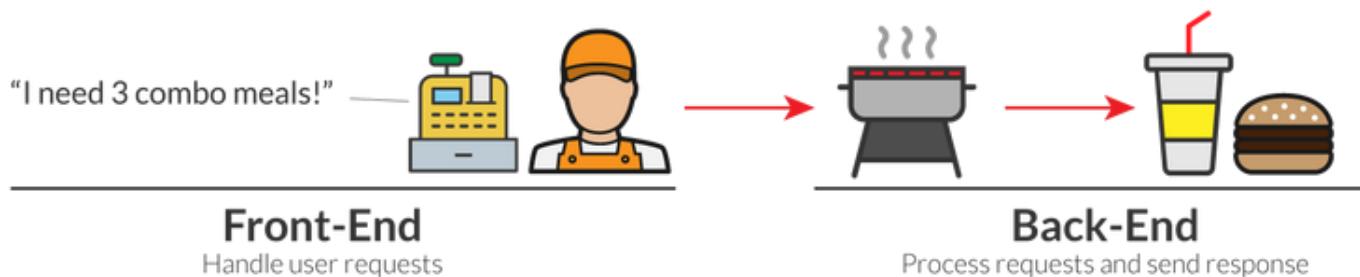
Start!



5 Minutes Later



Another 5 Minutes Later

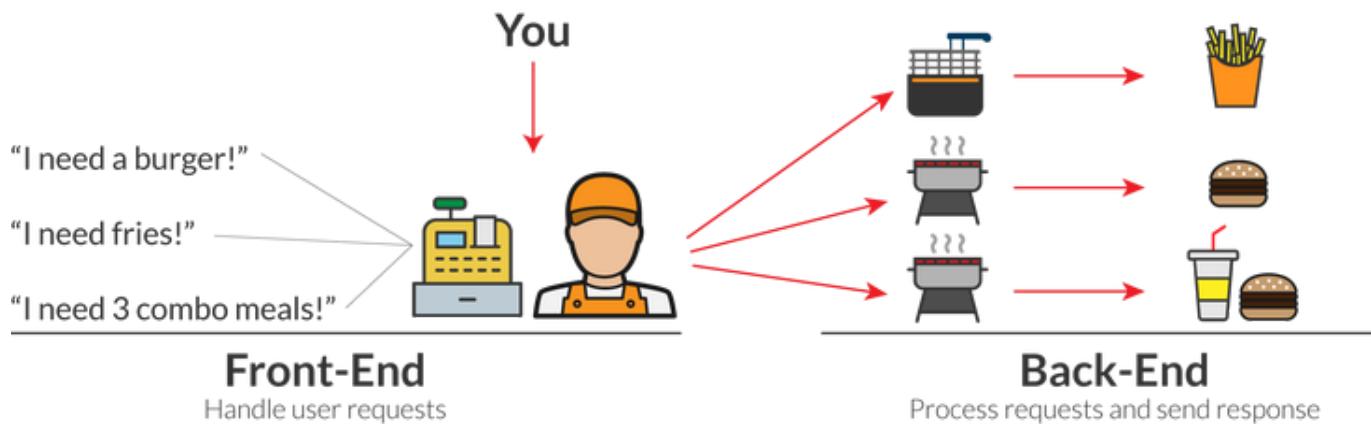


Eso es una mala experiencia para el usuario! Ya no podrías llamarlo "comida rápida". En su lugar, tendrías que llamarlo "comida mediocre"... o algo así.

AJAX permite un **modelo de procesamiento asincrónico**. Eso significa que puedes pedir datos o enviarlos sin cargar la página entera. Esto es como la forma en que funciona un restaurante de comida rápida normal. Como cajero, tomas el pedido del cliente, lo envías al equipo de cocina, y te preparas para tomar el siguiente pedido del cliente.

Los clientes pueden seguir haciendo pedidos, y no es necesario sentarse allí mientras los empleados de la cocina trabajan y hacen esperar a todo el mundo.

Esto ciertamente introduce cierta complejidad. Ahora tienes múltiples especializaciones dentro del restaurante. Además, los pedidos se están manejando a ritmos diferentes. Pero, crea una experiencia de usuario mucho mejor.



Probablemente has visto esto en acción en un restaurante. Una persona está trabajando en la máquina de papas fritas. Una persona está manejando la parrilla. Cuando llega un pedido, el cajero puede comunicarse instantáneamente con ambos y volver a tomar los pedidos.

Como crear una solicitud POST

Pongamos estos conceptos a trabajar. Como cajera, debes enviar las solicitudes de los clientes a la cocina para que el resto de tu equipo pueda preparar la comida. Puedes hacer eso con la solicitud POST.

En tu código real, una solicitud POST envía datos a tu servidor. Eso significa que estás enviando los datos del pedido al back-end, en este caso.

Tiene tres partes principales:

1. **Una URL:** esta es la ruta que seguirá la solicitud. Más en un minuto.
2. **Datos:** cualquier parámetro extra que necesites enviar al servidor.
3. **Callback:** Lo que pasa después de que hayas enviado la solicitud

¿Cuáles son algunas de las cosas comunes que la gente pide en un restaurante de comida rápida? Veamos dos ejemplos:

1. Papas Fritas
2. Un combo de una hamburguesa, papas fritas y una bebida

Estos dos requieren procesos diferentes. Una solicitud de papas fritas podría necesitar sólo una persona para meter algunas papas fritas en una manga. Pero un pedido de comida combinada requerirá el trabajo de varios miembros del equipo. Por lo tanto, estos dos necesitan diferentes URLs.

```
$.post('/comboMeal').post('/fries')
```

La URL nos permite usar la misma lógica en el back-end para ciertos tipos de solicitudes. Esa parte está fuera del alcance de este tutorial, por lo que puedes profundizar un poco más en ella cuando mires el back-end.

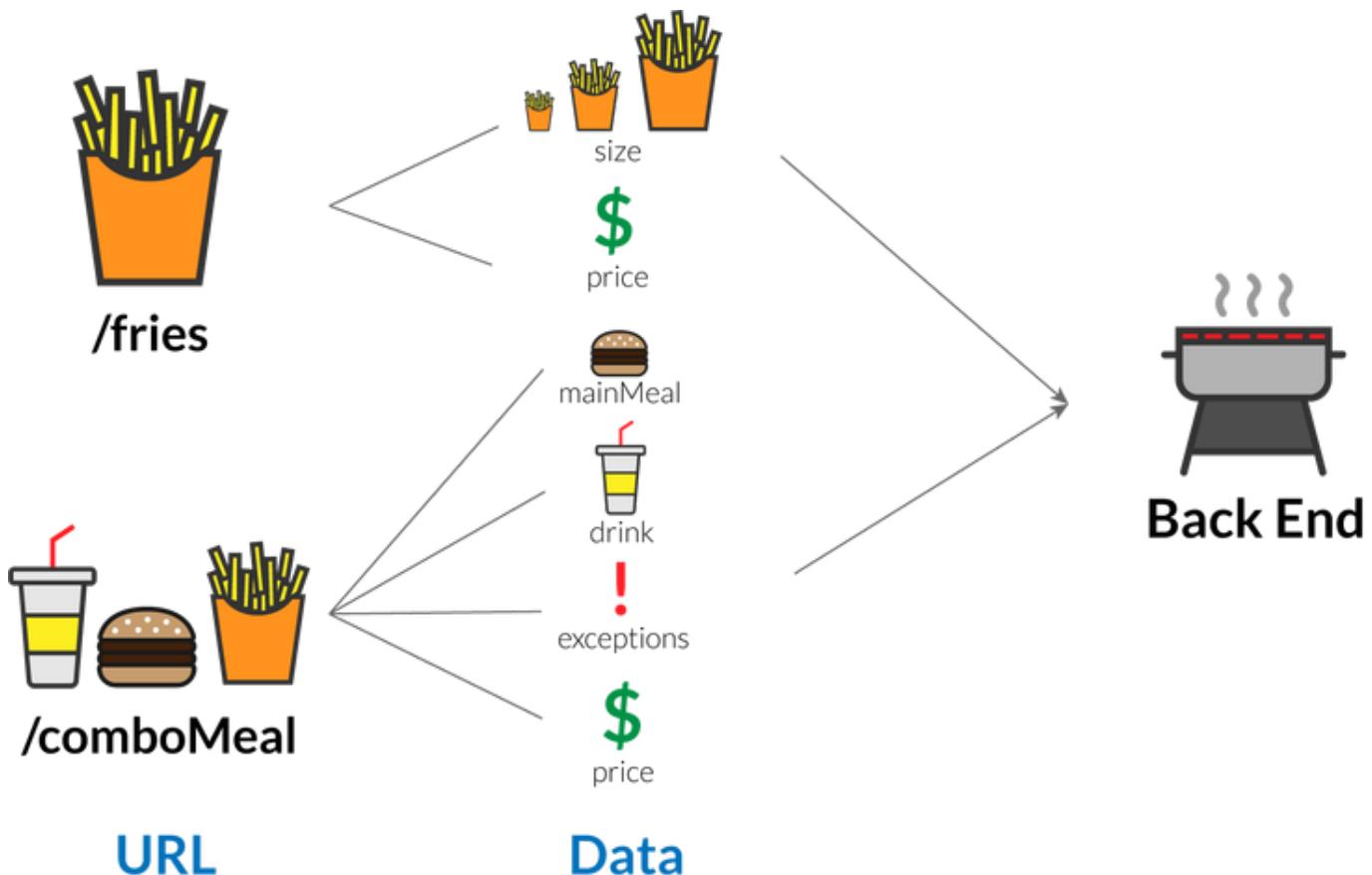
Lo siguiente es la **data**. Este es un **objeto** que nos dice un poco más sobre la petición. Para la URL de la comida combinada, probablemente necesitamos saber:

1. El tipo de comida principal
2. El tipo de bebida

3. El precio
4. Cualquier petición especial

Por las papas fritas, puede que sólo necesitemos saberlo:

1. El tamaño de las patatas
2. El precio



Veamos un ejemplo de un combo de: una hamburguesa con queso con una Pepsi que cuesta 6 dólares. Esto es lo que parece en JavaScript.

```
let order = {
  mainMeal: 'cheeseburger',
  drink: 'Pepsi',
  price: 6,
  exceptions: ''
};$.post('/comboMeal', order);
```

La variable *orden* contiene el contenido del orden. Y luego lo incluimos en el pedido POST para que nuestro personal de cocina sepa qué diablos poner en el combo de comida!

¡Pero no podemos hacer que todo este código se ejecute al azar! Necesitamos un evento de activación que active la solicitud. En este caso, un pedido de un cliente en un restaurante de comida rápida es como una persona que hace clic en un botón de "pedido" en su sitio web. Podemos usar el evento [click\(\)](#) de jQuery para ejecutar el POST cuando el usuario hace clic en un botón.

```
($('button').click(function(){
  let order = {
    mainMeal: 'cheeseburger',
    drink: 'Pepsi',
    price: 6,
    exceptions: ''
  };

  $.post('/comboMeal', order);
});
```

La última parte. Tenemos que decirle algo al cliente después de que su pedido haya sido enviado. Los cajeros suelen decir "¡El próximo cliente por favor!" ya que este es un restaurante de comida rápida, así que podemos usar eso dentro de la llamada para mostrar que el pedido ha sido enviado.

```
($('button').click(function(){
  let order = {
    mainMeal: 'cheeseburger',
    drink: 'Pepsi',
    price: 6,
    exceptions: ''
  };$.post('/comboMeal', order, function(){
    alert('Next customer please!');
  });
})
```

Como crear una solicitud GET

Hasta ahora, tenemos la capacidad de presentar una orden. Ahora, necesitamos una forma de entregar ese pedido a nuestro cliente.

Aquí es donde entran las solicitudes GET. GET nos permite solicitar datos del servidor (o de la cocina, esta analogía). Tenga en cuenta: en este momento, nuestra base de datos está llena de pedidos, no la comida en sí. Esta es una distinción importante porque las solicitudes de GET no cambian nuestra base de datos. Sólo entregan esa información al front-end. Las solicitudes POST cambian la información de la base de datos.

Estas son algunas de las preguntas típicas que te pueden hacer antes de recibir tu comida.

1. ¿Te gustaría comer aquí o recibir la comida para llevar?
2. ¿Necesitas algún condimento (como ketchup o mostaza)?
3. ¿Cuál es tu número en el recibo (para verificar que es tu comida)?

Digamos que ordenó tres comidas compuestas para su familia. Quieres comer la comida en el restaurante. Necesitas ketchup. Y el número en su recibo es 191.

Podemos crear una solicitud GET con una URL de '/comboMeal', que corresponde a la solicitud POST junto con la misma URL. Sin embargo, esta vez necesitamos datos diferentes. Es un tipo de solicitud totalmente diferente. El mismo nombre de URL sólo nos permite organizar mejor nuestro código.

```
let meal = {
  location: 'here',
  condiments: 'ketchup',
  receiptID: 191
};$.get('/comboMeal', meal);
```

2. URL



/comboMeal

3. Data



location



condiments



receiptID

También necesitamos un disparador para este. Esta solicitud se activa cuando los clientes responden a tus preguntas como cajero antes de que les entregues la comida. No hay una forma conveniente de representar las preguntas y respuestas con JavaScript. Así que voy a crear otro evento de clic para el botón con la clase "respuesta".

```
$('.answer').click(function(){
  let meal = {
    location: 'here',
    condiments: 'ketchup',
    idNumber: 191,
  };$.get('/comboMeal', meal);
});
```

1. Event

"My order is 191,
I'd like it for here
and with ketchup"



2. URL



3. Data



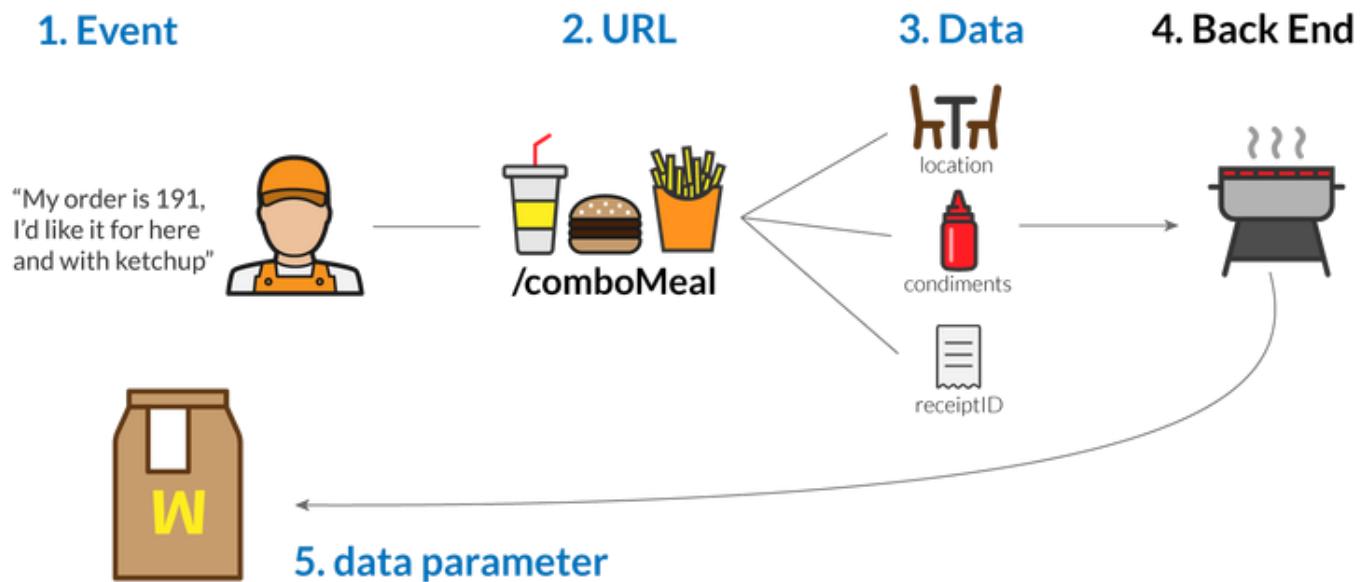
Este también necesita una función de devolución de llamada, porque vamos a recibir lo que estaba contenido en las tres comidas compuestas en el orden 191. Podemos recibir esos datos a través de un parámetro de *datos* en nuestra llamada de retorno.

Esto nos devolverá lo que sea que la retrollamada estipule para la orden 191. Voy a usar una función llamada *comer* para significar que eventualmente se puede comer la comida, pero ten en cuenta que no hay una función de comer en JavaScript!

```
$('.answer').click(function(){
  let meal = {
    location: 'here',
    condiments: 'ketchup',
    idNumber: 191,
  };

  //data contains the data from the server
  $.get('/comboMeal', meal, function(data){
    //eat is a made-up function but you get the point
    eat(data);
  });
});
```

El producto final, *datos*, contendría el contenido de las tres comidas combinadas, teóricamente. ¡Depende de cómo esté escrito en el backend!



Eventos en javascript

Como se mencionó anteriormente, los **eventos** son acciones u ocurrencias que suceden en el sistema que está programando — el sistema disparará una señal de algún tipo cuando un evento ocurra y también proporcionará un mecanismo por el cual se puede tomar algún tipo de acción automáticamente (p.e., ejecutando algún código) cuando se produce el evento. Por ejemplo, en un aeropuerto cuando la pista está despejada para que despegue un avión, se comunica una señal al piloto y, como resultado, comienzan a pilotar el avión.

En el caso de la Web, los eventos se desencadenan dentro de la ventana del navegador y tienden a estar unidos a un elemento específico que reside en ella — podría ser un solo elemento, un conjunto de elementos, el documento HTML cargado en la pestaña actual o toda la ventana del navegador. Hay muchos tipos diferentes de eventos que pueden ocurrir, por ejemplo:

- El usuario hace clic con el mouse sobre un elemento determinado o coloca el cursor sobre un elemento determinado.
- El usuario presiona una tecla en el teclado.
- El usuario cambia el tamaño o cierra la ventana del navegador.
- Una página web termina de cargar.
- Un formulario se envía
- Un video se reproduce, pausa o finaliza la reproducción.
- Un error ocurre.

Se deducirá de esto (y echar un vistazo a MDN [Referencia de eventos](#)) que hay **muchos** eventos a los que se puede responder.

Cada evento disponible tiene un **controlador de eventos**, que es un bloque de código (generalmente una función JavaScript definida por el usuario) que se ejecutará cuando se active el evento. Cuando dicho bloque de código se define para ejecutarse en respuesta a un disparo de evento, decimos que estamos **registrando un controlador de eventos**. Tenga en cuenta que los controladores de eventos a veces se llaman **oyentes de**

eventos — son bastante intercambiables para nuestros propósitos, aunque estrictamente hablando, trabajan juntos. El oyente escucha si ocurre el evento y el controlador es el código que se ejecuta en respuesta a que ocurra.

Veamos un ejemplo simple para explicar lo que queremos decir aquí. Ya has visto eventos y controladores de eventos en muchos de los ejemplos de este curso, pero vamos a recapitular solo para consolidar nuestro conocimiento. En el siguiente ejemplo, tenemos un solo `` , que cuando se presiona, hará que el fondo cambie a un color aleatorio:

```
<button>Cambiar color</button>
```

El JavaScript se ve así:

```
var btn = document.querySelector('button');

function random(number) {
    return Math.floor(Math.random()*(number+1));
}

btn.onclick = function() {
    var rndCol = 'rgb(' + random(255) + ',' + random(255) + ',' + random(255) + ')';
    document.body.style.backgroundColor = rndCol;
}
```

En este código, almacenamos una referencia al botón dentro de una variable llamada `btn`, usando la función `Document.querySelector ()`. También definimos una función que devuelve un número aleatorio. La tercera parte del código es el controlador de eventos. La variable `btn` apunta a un elemento , y este tipo de objeto tiene una serie de eventos que pueden activarse y, por lo tanto, los controladores de eventos están disponibles. Estamos escuchando el disparo del evento "click", estableciendo la propiedad del controlador de eventos `onclick` para que sea igual a una función anónima que contiene código que generó un color RGB aleatorio y establece el color de fondo igual a este.

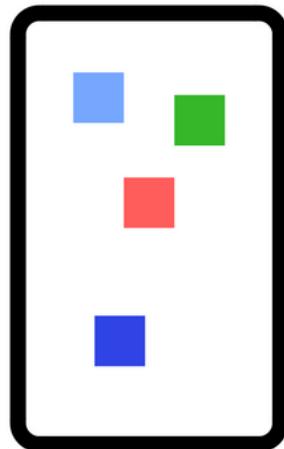
Este código ahora se ejecutará cada vez que se active el evento "click" en el elemento `` , es decir, cada vez que un usuario haga clic en él.

Event Loop

Antes que nada miremos un dibujo que representa el runtime de v8 (el runtime que usa chrome y node)



Memory Heap



Call Stack



Como se puede ver en la imagen, el engine consiste de dos elementos principales

- Memory Heap: es donde se realiza la alocación de memoria
- Call Stack: es donde el runtime mantiene un track de las llamadas a las funciones

Solo hablaremos de la call stack, que es la que se relaciona con el Event Loop.

Call Stack

Para los que no sepan un stack (también llamado pila) es una estructura simple, similar a un arreglo en el que solo se puede agregar items al final (push) , y remover el último (pop).

El proceso que realiza el call stack es simple, cuando se está a punto de ejecutar una función, esta es añadida al stack. Si la función llama a su vez, a otra función, es agregada sobre la anterior. Si en algún momento de la ejecución hay un error, este se imprimirá en la consola con un mensaje y el estado del call stack al momento en que ocurrió.

Javascript es un lenguaje single threaded. Esto quiere decir que durante la ejecución de un script existe un solo thread que ejecuta el código. Por lo tanto solo se cuenta con un call stack

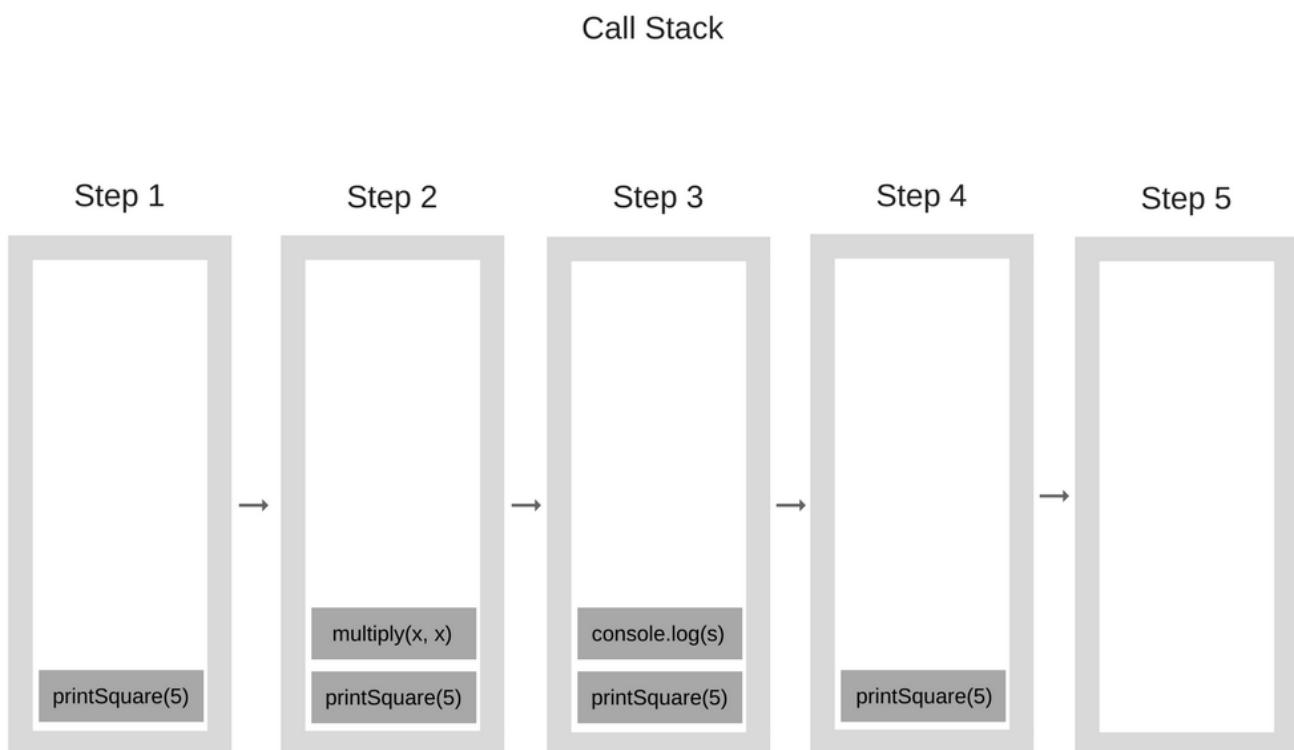
Veamos un ejemplo:

```
function multiply (x, y) {  
    return x * y;
```

```
}
```

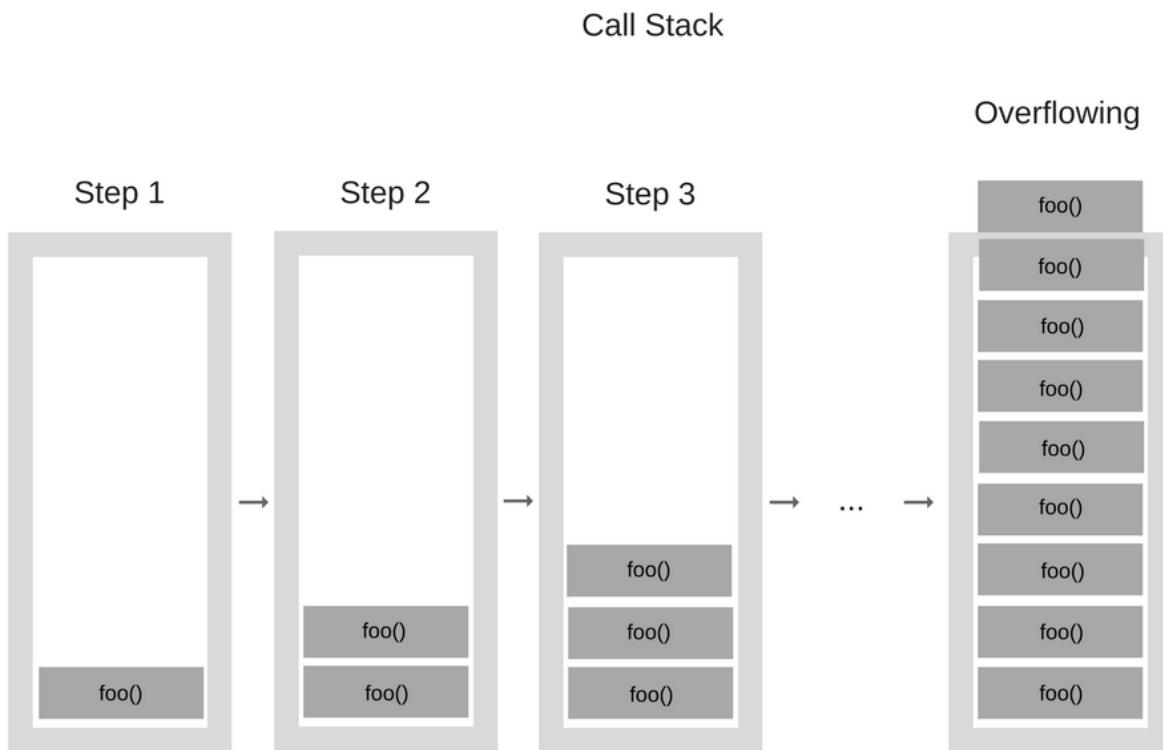
```
function printSquare (x) {  
    var s = multiply(x, x);  
    console.log(s);  
}  
  
printSquare(5);
```

Los estados del call stack serían:



Y que pasa si tenemos una función de esta manera:

```
function foo() {  
    foo();  
}  
  
foo();
```



Lo que sucedería es que en algún momento la cantidad de funciones llamadas excede el tamaño del stack , por lo que el navegador mostrará este error:

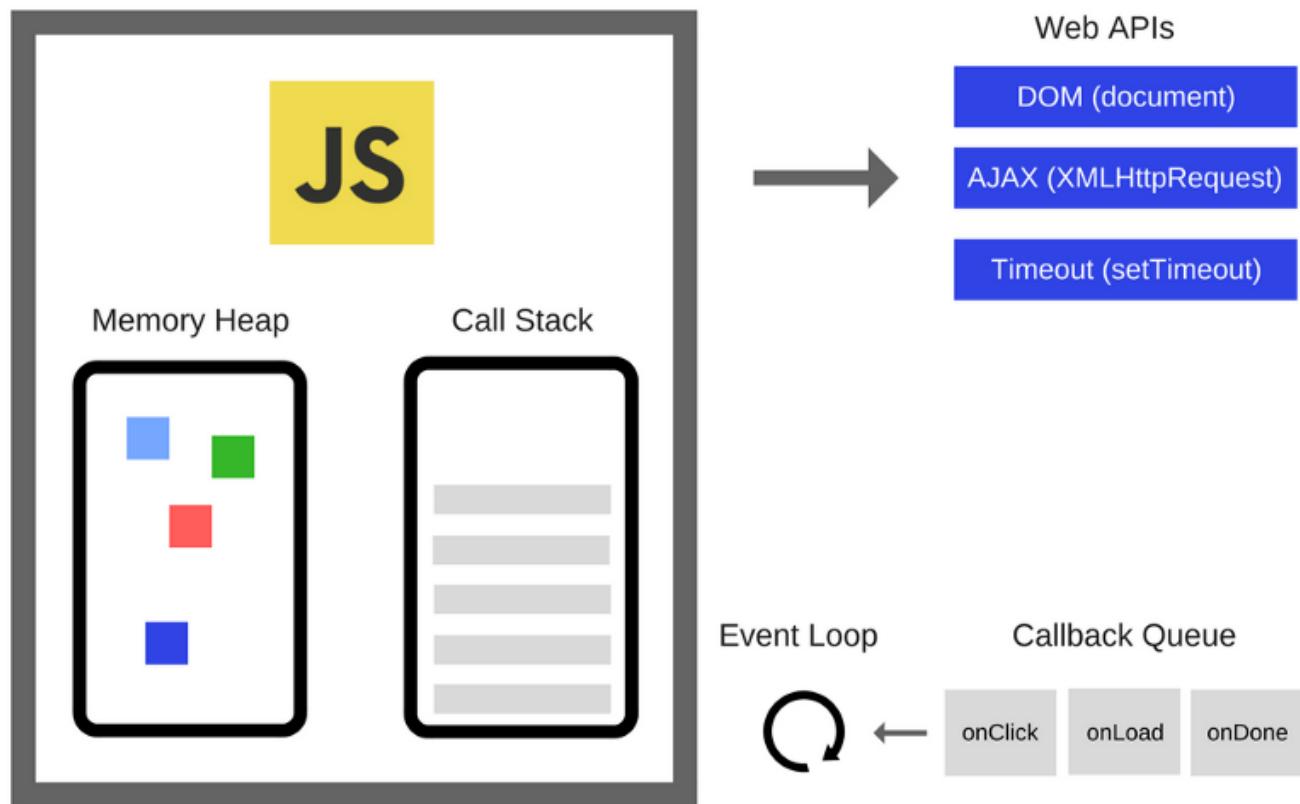
```
✖ ► Uncaught RangeError: Maximum call stack size exceeded
```

Pero qué pasa si llamamos a un timeout o hacemos un request con AJAX a un servidor. Al ser un solo thread, hay un solo call stack y por lo tanto solo se puede ejecutar una cosa a la vez. Es decir el navegador debería congelarse, no podría hacer más nada, no podría renderizar, hasta que la llamada termine de ejecutarse. Sin embargo esto no es así, javascript es asincrónico y no bloqueante. Esto es gracias al Event Loop.

Event Loop

Algo interesante acerca de javascript, o mejor dicho de los runtimes de javascript, es que no cuentan nativamente con cosas como setTimeout, DOM, o HTTP request. Estas son llamadas web apis, que el mismo navegador provee, pero no están dentro del runtime JS.

Por lo tanto este es el gráfico que muestra una visión más abarcativa de javascript. En este se puede ver el runtime, más las Web APIs y el callback queue del cual hablaremos más adelante.



Al haber un solo thread es importante no escribir código bloqueante para la UI no quede bloqueada.

Pero ¿Cómo hacemos para escribir código no bloqueante?

La solución son callbacks asincronicas. Para esto combinamos el uso de callbacks (funciones que pasamos como parámetros a otras funciones) con las WEB API's.

Por ejemplo:

```
console.log("hola");

setTimeout(function timeoutCallback() {
    console.log("mundo");
}, 500);

console.log("Ubykuo, everytime, everywhere");

/*
 * Resultados:
 * => hola
 * => Ubykuo, everytime, everywhere
 * => mundo
 */
```

Como pueden ver la ejecución no se queda bloqueada en `setTimeout()` ya que imprime la instrucción que le sigue primero) ¿Pero entonces cómo es que posible que esto sea así si solo existe un solo thread? ¿ Cómo es que la ejecución continua y al mismo tiempo el `setTimeout` hace la cuenta regresiva para ejecutar la función pasada como callback?

Esto es porque, como mencione anteriormente, el `setTimeout` NO es parte del runtime. Sino que es provista por el navegador como WEB APIs (o en el caso de Node por c++ apis). Los cuales SI se ejecutan en un thread distinto.

¿Como se maneja esto con una única call stack?

Existe otra estructura donde se guardan las funciones que deben ser ejecutadas luego de cierto evento (timeout, click, mouse move), en el caso del código de ejemplo de arriba se guarda que, cuando el timeout termine se debe ejecutar la función `timeoutCallback()`. Tener en cuenta que cuando sucede el evento, esta estructura no es la que la ejecuta y tampoco las agrega al call stack ya que sino podría pasar que la función se ejecutará en medio de otro código. Lo que hace es enviarla a la Callback Queue.

Lo que hace el event loop es fijarse el call stack, y si está vacío (es decir no hay nada ejecutándose) envía la primera función que esté en la callback queue al call stack y comienza a ejecutarse.

Luego de terminar la cuenta regresiva del `setTimeout()` (que no es ejecutada en el runtime de javascript), `timeoutCallback()` será enviada a la callback queue. El event loop chequeara el Call Stack, si este está vacío enviará `timeoutCallback()` al call stack para su ejecución.

El flujo en imágenes de todo este trabalenguas seria:



De esta manera se logra que el código sea no bloqueante, en vez de un setTimeout podría ser una llamada a un servidor, en donde habría que esperar que se procese nuestra solicitud y nos envíe una respuesta , el cual sería tiempo ocioso si no contáramos con callbacks asincronicas, de modo que el runtime pueda seguir con otro código. Una vez que la respuesta haya llegado del servidor y Call Stack esté vacío, se podrá procesar la respuesta (mediante la función pasada como callback) y hacer algo con ella , por ejemplo mostrarla al usuario.

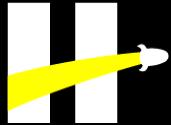
Este video explica muy bien el Event Loop <https://www.youtube.com/watch?v=8aGhZQkoFbQ>

¿Por que si bloqueamos el call stack la ui ya no responde más?

Esto se debe a que el navegador intenta realizar un proceso de renderizado cada cierto tiempo. Pero este no puede realizarse si hay código en el stack. El proceso de renderizado es similar a una callback asincrónica , ya que debe esperar a que el stack está vacío, es como una función más en la Callback Queue (aunque con cierta prioridad). Por lo que sí hay código bloqueante , el proceso de renderizado tardará más en realizarse y el usuario no podrá hacer nada, no podrá seleccionar texto, no podrá ingresar texto, no podrá apretar un botón.

¿Que pasaria si a un usuario que interactuando con nuestra página le sucediera esto?

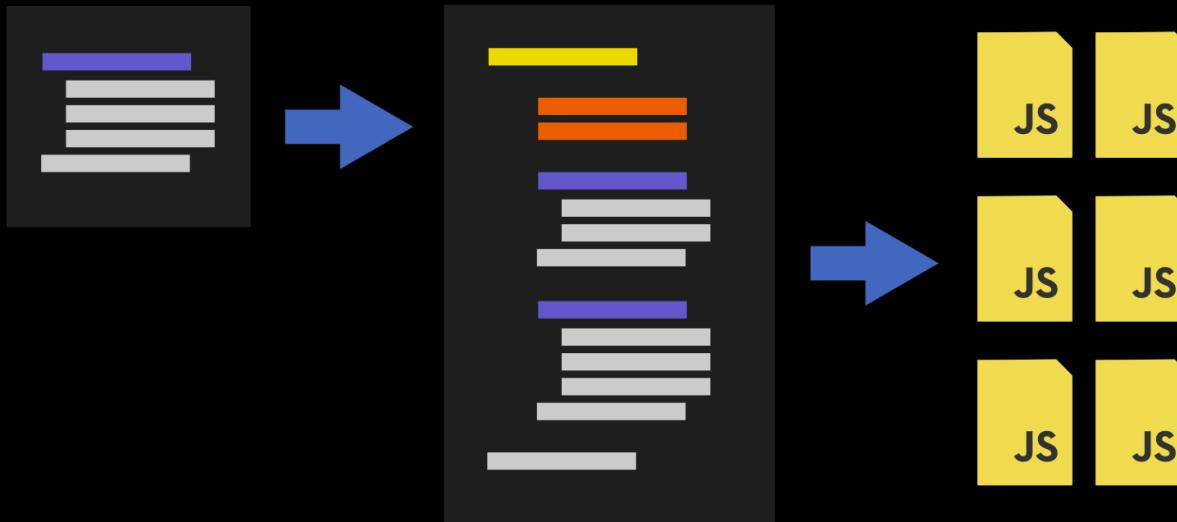
Lo más probable es que cierre el navegador y nunca más vuelva a entrar a nuestra página. No es algo que queremos que suceda.



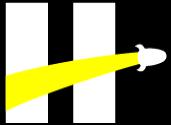
MODULOS



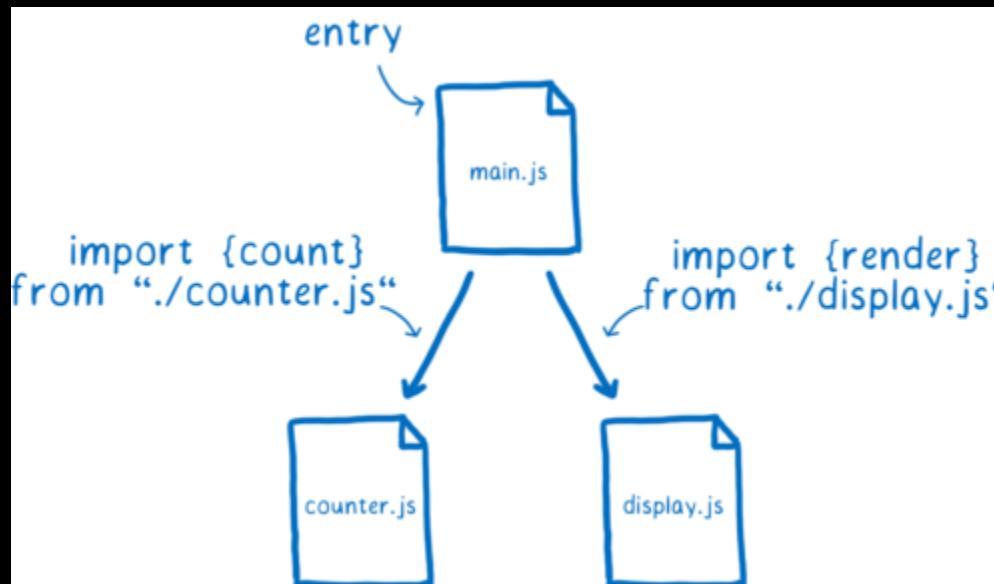
MODULOS



Un **Módulo** es un pedazo de código que cumple una tarea específica y que indica sobre qué piezas de código depende (*dependencias*).



MODULOS



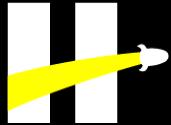
Dependencias



MODULOS

Patrón de módulos IIFE

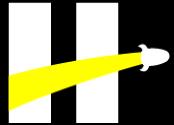
```
1 const weekDay = function() {
2     const names = ["Domingo", "Lunes", "Martes", "Miercoles",
3                     "Jueves", "Viernes", "Sabado"];
4     return {
5         name: function name(number) { return names[number]; },
6         number: function number(name) { return names.indexOf(name); }
7     };
8 }()
9
10 console.log(weekDay.name(weekDay.number("Domingo")));
```



CommonJS

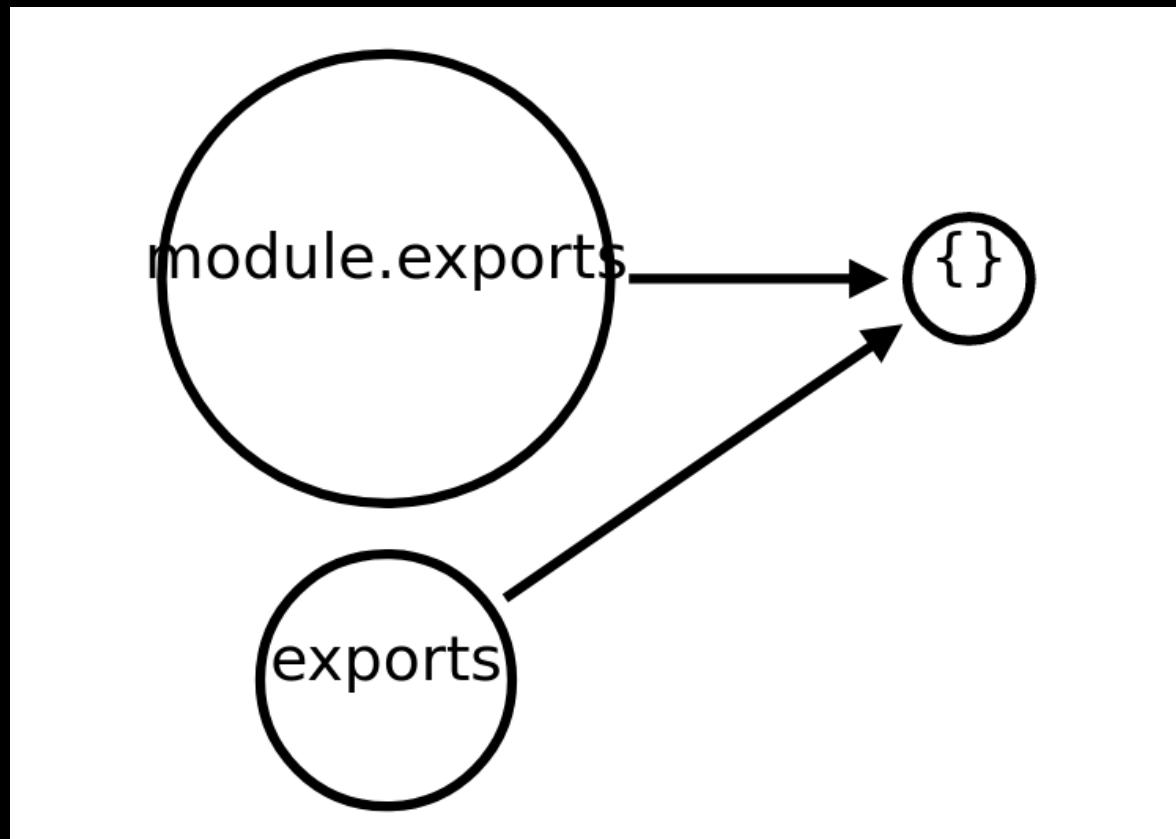
```
1 // WeekDays.js
2
3 var names = [ "Domingo", "Lunes", "Martes", "Miercoles",
4               "Jueves", "Viernes", "Sabado"];
5
6 exports.name = function name (number) { return names[number]; };
7 exports.number = function number(name) { return names.indexOf(name); };
```

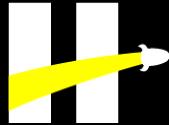
```
1 // index.js
2
3 var weekDays = require('./WeekDays.js');
4
5 console.log(weekDays.name(weekDays.number("Domingo")));
```



module.exports vs exports

Inicialmente



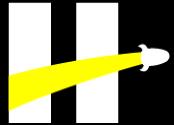


module.exports vs exports

module.exports

```
1 // module1.js
2
3 module.exports.a = function() {
4     console.log("a");
5 }
6
7 module.exports.b = function() {
8     console.log("b");
9 }
```

```
1 // module2.js
2
3 var example = require('./module1.js');
4
5 example.a(); // "a"
6
7 example.b(); // "b"
```

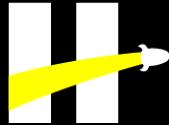


module.exports vs exports

module.exports

```
1 // module1.js
2
3 module.exports.a = function() {
4     console.log("a");
5 }
6
7 module.exports.b = function() {
8     console.log("b");
9 }
10
11 module.exports = function extra() {
12     console.log("Extra function");
13 }
```

```
1 // module2.js
2
3 var example = require('./module1.js');
4
5 example.a(); // TypeError: example.a is not a function
6
7 example.b(); // TypeError: example.a is not a function
8
9 example.extra(); // TypeError: prueba.extra is not a function
10
11 example(); // "Extra function"
```

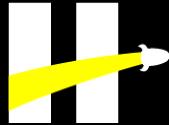


module.exports vs exports

exports

```
1 // module1.js
2
3 exports.a = function() {
4     console.log("a");
5 }
6 exports.b = function() {
7     console.log("b");
8 }
```

```
1 // module2.js
2
3 var example = require('./module1.js');
4
5 example.a(); // "a"
6
7 example.b(); // "b"
```

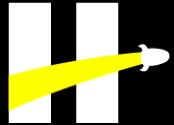


module.exports vs exports

exports

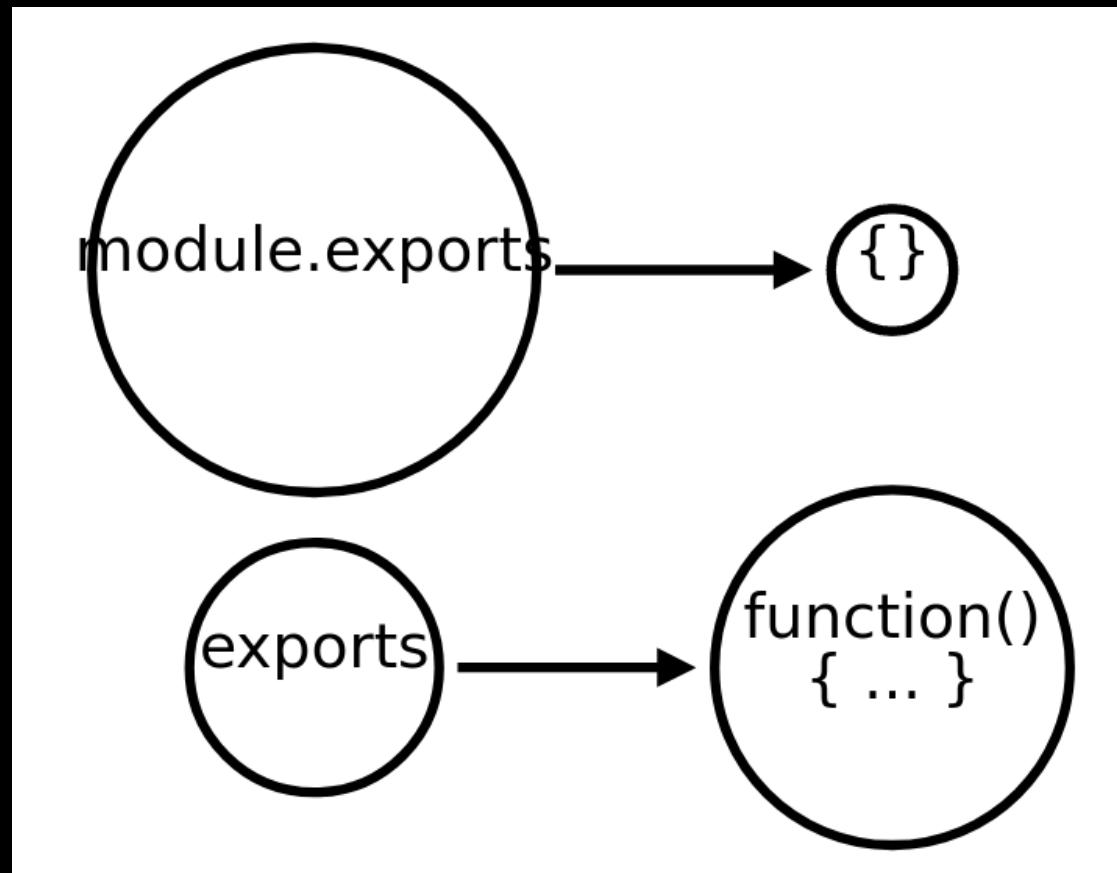
```
1 // module1.js
2
3 exports.a = function() {
4     console.log("a");
5 }
6 exports.b = function() {
7     console.log("b");
8 }
9
10 exports = function extra() {
11     console.log('Extra function');
12 }
```

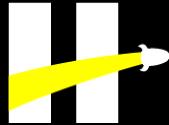
```
1 // module2.js
2
3 var example = require('./module1.js');
4
5 example.a(); // "a"
6
7 example.b(); // "b"
8
9 example.extra() // TypeError: example.extra is not a function
10
11 example(); // TypeError: example is not a function
```



module.exports vs exports

Si "pisamos" la referencia del exports...

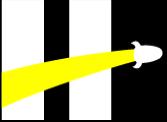




ES6

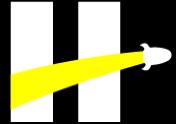
```
1 // WeekDays.js
2
3 var names = ["Domingo", "Lunes", "Martes", "Miercoles",
4               "Jueves", "Viernes", "Sabado"];
5
6 export function name (number) { return names[number]; };
7 export function number(name) { return names.indexOf(name); };
8
9 export default function myDefault () {
10   // otras cosas...
11 }
```

```
1 // index.js
2
3 import { number, name } from './WeekDays.js';
4
5 console.log(name(number("Domingo")));
```

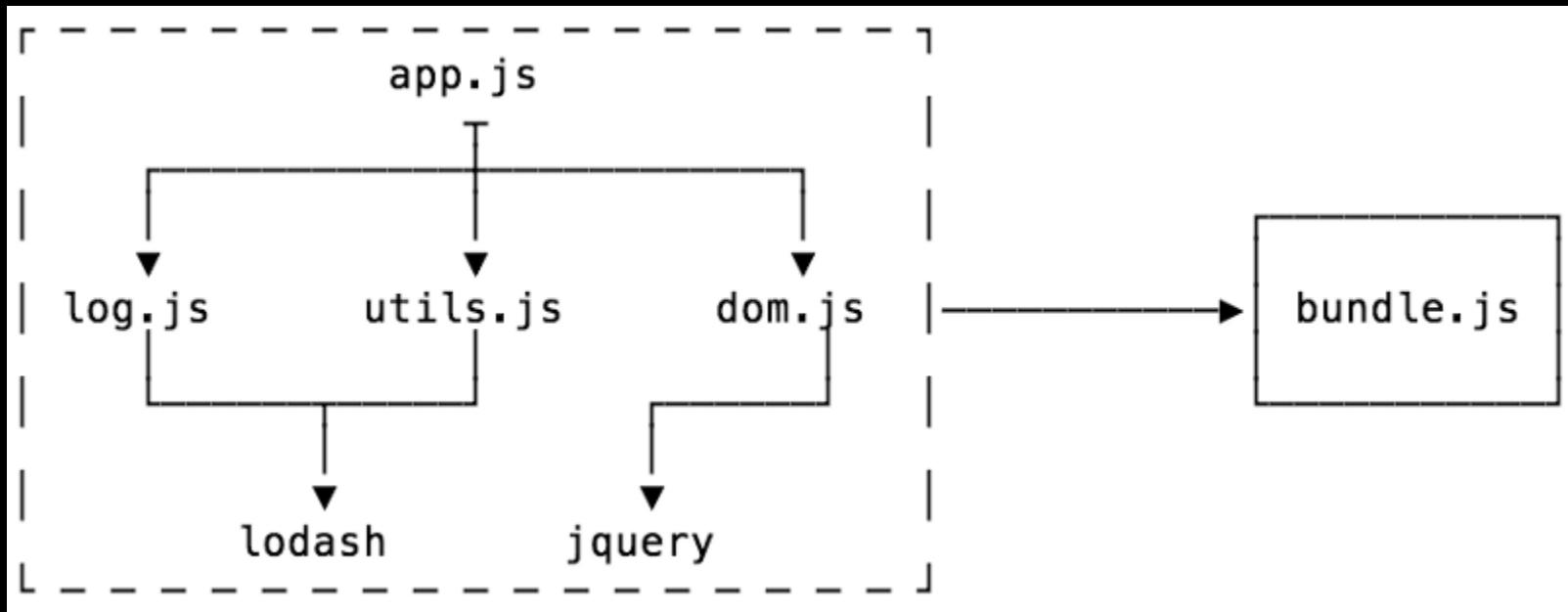


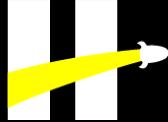
Bundlers

```
<html>
  <head>My Webpage</head>
  <body>
    <script src="script1.js"></script>
    <script src="script2.js"></script>
    <script src="script3.js"></script>
    <script src="script4.js"></script>
  </body>
</html>
```

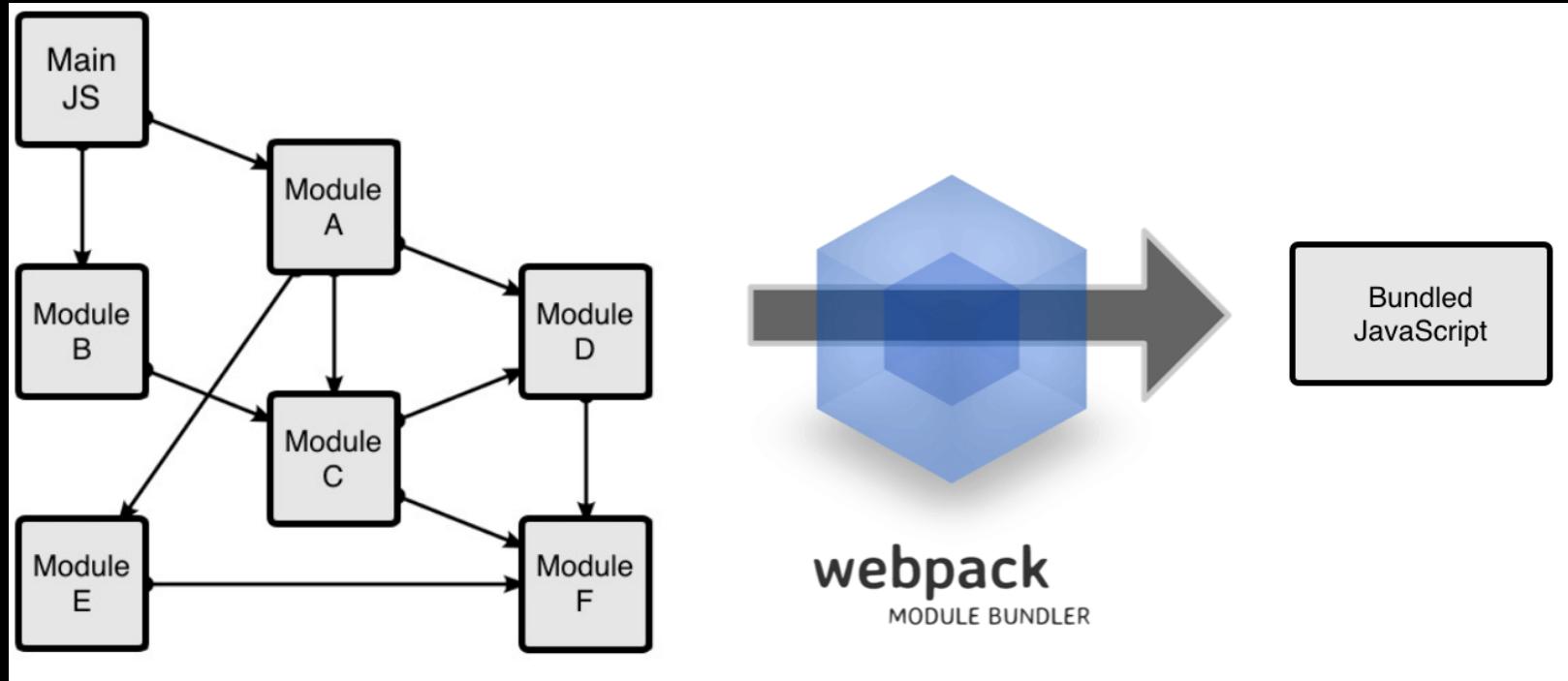


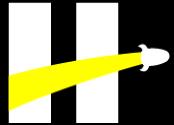
Bundlers





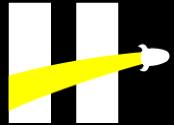
Webpack





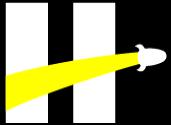
Webpack

```
1 npm init
2
3 npm install --save-dev webpack
4
5 // package.json
6
7 "scripts": {
8   "start": "node server.js",
9   "build": "webpack"
10 }
11
12 // crear archivo webpack.config.js`
```



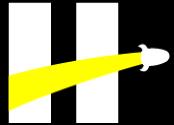
Webpack

```
1 // webpack.config.js`  
2  
3 module.exports = {  
4   entry: './browser/app.js', // el punto de arranque de nuestro programa  
5   output: {  
6     path: __dirname + '/browser', // el path absoluto para  
7           // el directorio donde queremos que el output sea colocado  
8     filename: 'bundle.js' // el nombre del archivo que va a contener  
9           //nuestro output - podemos nombrarlo como queramos pero bundle.js es lo típico  
10    }  
11 }
```



Webpack

```
1 // npm run build
2
3 Hash: 43fddf2175fdf6f7923f
4 Version: webpack 2.2.1
5 Time: 72ms
6     Asset      Size  Chunks             Chunk Names
7 bundle.js   3.1 kB      0  [emitted]  main
8 [0] ./browser/app.js 586 bytes {0} [built]
```



Webpack

```
1 <!doctype html>
2 <html>
3     <head>
4     </head>
5     <body>
6
7         <script src="./dist/bundle.js"></script>
8     </body>
9 </html>
10
```

Incluyendo el script en el HTML

Henry



Hacé click acá para dejar tu feedback sobre esta clase.

Hacé click acá completar el quiz teórico de esta lecture.

Lesson 05 - Módulos y Bundlers

Cuando desarrollamos, queremos que la estructura de nuestro programa/código sea lo mas transparente posible, facil de explicar y que cada parte cumpla una tarea definida.

Un programa típico crece de manera orgánica con el tiempo, nuevas piezas de funcionalidad van siendo agregadas a medida que surgen nuevas necesidades. Esto hace que dar una Estructura -y mantenerla mientras crece- sea fundamental. El problema que mantener esa estructura es un trabajo *extra*, el cual solo veremos los frutos en el futuro, cuando alguien nuevo trabaje en el proyecto. Por lo tanto, lo que puede terminar sucediendo, es que no se haga el trabajo extra y se deja que las partes del programa queden muy entreveradas entre sí.

Finalmente, aparecen dos problemas: el primero, es que entender un programa o sistema sin estructura clara es difícil. Si está tan entreverado que tocar una cosa puede impactar en el todo, al introducir cambios seguramente vas a crear muchos bugs que tendras que corregir. O sea que no podés trabajar de manera aislada una sola parte del código. Finalmente, te ves obligado a construir un entendimiento del código como un todo. Segundo, si quisieras **reutilizar** una parte de código en otro proyecto, es muy probable que *reescribir* esa funcionalidad sea más fácil que lograr extraerla de tu programa complejo.

Modules

Los **módulos** son un intento de evitar estos problemas. Un **Módulo** es un pedazo de código que cumple una tarea específica y que indica sobre qué piezas de código depende (*dependencias*).

Interfaz es lo que conocemos en inglés como interface ("superficie de contacto"). En informática, se utiliza para nombrar a la conexión funcional entre dos sistemas, programas, dispositivos o componentes de cualquier tipo, que proporciona una comunicación de distintos niveles permitiendo el intercambio de información. Su plural es interfaces.

Estos módulos proveen una interfaz de contacto hacia afuera, es decir que todo el funcionamiento del mismo está encapsulado del mundo externo, y sólo se permite interactuar con el módulo a través de puntos de contacto bien definidos y documentos (en el mejor de los casos).

Es muy similar a cuando interactuamos con un objeto, como un **Array** y usamos sus métodos:

```
var arreglo = [];
arreglo.push(1);
```

En el ejemplo, nosotros usamos la función `push` pero no tenemos idea como está implementada adentro, gracias a la documentación sabemos cómo funciona y cómo usarla, pero su funcionamiento está encapsulado dentro de la función.

Dependencias

Las relaciones entre módulos se llaman **dependencias**. Cuando un módulo necesita una parte de código de otro módulo, vamos a decir que ese módulo *depende* del segundo. En general los módulos van a especificar qué otros módulos son sus dependencias, de tal forma que para usarlo podemos cargar todas esas dependencias.

Encapsulando Código

En `js`, para lograr esta encapsulación vamos a necesitar que cada módulo tenga su propio **scope**.

Poner nuestro código en diferentes archivos no es suficiente, ya que si cargamos varios archivos, todos comparten el mismo contexto global. Por lo tanto podría haber colisiones entre módulos e interferir entre ellos, rompiendo el encapsulamiento.

Paquetes

Bien, ahora imaginemos que intenamos encapsular el código para poder usarlo como una pieza en nuestro proyecto (más abajo veremos cómo se hace). Cuando llegue el momento de usar ese código en otro proyecto, lo que hagamos, probablemente, es copiar ese código y *reutilizarlo*. Imaginemos ahora, que en el nuevo proyecto detecto un bug en el código y decido corregirlo. Ahora tambien debería ir al proyecto viejo y corregirlo tambien. Como se pueden imaginar, esto no se puede escalar. Cuando el número de proyectos en el que usamos ese código crezca, va a ser inmanejable la tarea de updatear cada pedazo de código en cada proyecto manualmente.

La solución a este problema son los **paquetes**. Un **paquete** es un pedazo de código que puede ser distribuido (copiado e instalado). Cada paquete puede contener uno o más módulos y a su vez tiene información sobre las dependencias que tiene con otros paquetes. Generalmente, estos paquetes viene acompañado de documentación que indican al usuario cómo usarlos y qué hacen.

Cuando un error es encontrado en algún paquete, o se le agrega funcionalidad nueva. Es corregido y updateado. Ahora los proyectos que dependen de ese paquete pueden actualizar esos paquetes a la nueva versión.

Para lograr distribuir estos paquetes y mantenerlos correctamente actualizados, vamos a necesitas la ayuda de un **gestor de paquetes**. El gestor de paquetes es un pedazo de software que se encarga de manejar esto de manera automática. En el mundo de JavaScript, el gestor de paquetes más usado es NPM (<https://npmjs.org>).

NPM es un servicio online en donde estan hosteados los paquetes que los usuarios comparten, y a su vez un programa que se puede instalar en cualquier SO, que te ayuda a descargarlos, instalarlos y mantenerlos actualizados.

Veremos NPM más en detalle en el módulo de Back-End.

Creando Módulos

Hasta 2015, JavaScript no tenía una forma *nativa* de construir módulos. Pero de todos modos, las personas lo usaron para construir grandes proyectos a lo largo de diez años. Por lo tanto, los desarrolladores crearon su propia forma de crear módulos en JavaScript. Lo lograron usando funciones para crear scopes isolados, y usaron objetos para crear las interfaces de los módulos.

Vamos a crear un módulo que nos ayude a trabajar con fechas, va a tener dos métodos que nos permiten pasar un Integer y recibir el nombre del día, y al revés.

```
const weekDay = function() {
  const names = ["Domingo", "Lunes", "Martes", "Miercoles",
    "Jueves", "Viernes", "Sabado"];
  return {
    name: function name(number) { return names[number]; },
    number: function number(name) { return names.indexOf(name); }
  };
}();

console.log(weekDay.name(weekDay.number("Domingo")));
// → Sunday
```

La interfaz está creada en el objeto que retornamos. Que tiene los dos métodos antes mencionados. Lo interesante es notar, que se logró encapsular el código a través de un **IIFE** (Immediately invoked function expression), y creando un closure con el arreglo `names`. De esta manera, en el scope global, si hubiera una variable `names` no interferiría con la que declaramos en nuestro módulo.

Pensá que ocurriría si no hubiésemos creado el close con al IIFE. ¿Qué pasaría si alguien usa nuestro código y en su proyecto ya tenía declarada una variable `names`?

Este tipo de solución solo ofrece isolación, pero no habla de dependencias, solamente pone su interfaz en el contexto global (el objeto `weekDay`). Por mucho tiempo, esta fue la forma de programar módulos en la web.

Mejorando los módulos

Una siguiente mejora lógica para nuestros módulos, sería poder tenerlos en archivos separados, por ejemplo, tener nuestro módulo de los días en el archivo: `weekDay.js`, y tener alguna manera de *importarlo*.

Para hacer eso deberíamos tener la capacidad de leer el contenido de un archivo ('strings') y poder pasarlal al interprete para que la ejecute. Hay varias formas de lograr esto en JS.

La primera es usando una función especial de JS llamada `eval`. Básicamente esta función recibe un `string` como parámetro y va a ejecutar el código en el scope actual (como si lo estuvieras copiando y pegando ahí).

```
const x = 1;
function evalAndReturnX(code) {
  eval(code);
```

```

    return x;
}

console.log(evalAndReturnX("var x = 2"));
// → 2
console.log(x);
// → 1

```

Este método no es muy efectivo, como vemos esta función puede romper el sistema de scopes tradicional, por lo tanto no es muy predecible.

Otra forma, más predecible, es usar el constructor de `Function` (es el constructor que JS utiliza internamente para crear funciones). Este recibe dos argumentos: una string que contiene una lista separada por comas de argumentos, y una string que contiene el cuerpo de la función.

```

// estas dos formas producen la misma función
function plusOne(n) {
    return n + 1;
}

let plusOne = Function("n", "return n + 1;");
console.log(plusOne(4));
// → 5

```

Utilizando esto, vamos a poder encapsular un módulo dentro de una función y usar el scope de esa función como el scope del módulo.

CommonJS

El sistema que finalmente eligió NodeJS (y por lo tanto casi haciendo que sea standart) es conocido como **CommonJS**. Es el sistema de módulos utilizado por la mayoría de paquetes de `npm`.

El concepto más importante de CommonJS es una función llamada `require`, que recibe una string que indica el nombre de una dependencia. Cuando es invocada, esta función busca el módulo, lo carga y retorna la interfaz de ese módulo. Esta función, además, envuelve todo el módulo en una función, por lo tanto cada módulo automáticamente tiene su propio scope.

Para pasar el módulo que hicimos antes a CommonJS, básicamente vamos a necesitar utilizar un objeto llamado `exports`. En CommonJS este objeto es donde debemos poner todo lo que queremos que esté en la interfaz de nuestro módulo, es decir, todo lo que querremos que el mundo exterior pueda ver.

Siguiendo con el ejemplo del módulo de las fechas, veamos como podríamos hacer un verdadero módulo usando CommonJS: Primero creamos un archivo con nombre `weekDays.js` con el siguiente contenido:

```

var names = ["Domingo", "Lunes", "Martes", "Miercoles",
             "Jueves", "Viernes", "Sabado"];

```

```
exports.name = function name (number) { return names[number]; };
exports.number = function number(name) { return names.indexOf(name); };
```

Para usarlo, tenemos que usar `require` en el archivo que quisieramos utilizar nuestro módulo:

```
var weekDays = require('./WeekDays.js');

console.log(weekDay.name(weekDay.number("Domingo")));
```

Viendo esto, podríamos imaginar cómo funciona `require` por adentro:

```
require.cache = Object.create(null);

function require(name) {
  if (!(name in require.cache)) {
    let code = readFile(name); // función que lee un archivo de texto
    let module = {exports: {}};
    require.cache[name] = module;
    let wrapper = Function("require, exports, module", code);
    wrapper(require, module.exports, module); // pasa la función require por si es
    necesario usarla adentro (otras dependencias)
  }
  return require.cache[name].exports;
}
```

En este código `readFile` es una función inventada que lee un archivo de texto y retorna su contenido como una string. JS no tiene una función tal, pero el ambiente donde se ejecuta el motor (Node o el browser), puede proveer la forma de leer archivos a JS.

Para evitar tener que cargar el mismo módulo muchas veces, `require` tiene un *cache* de módulos que ya fueron cargados. Si el módulo ya fue invocado, estará en el objeto `cache`, si no, leerá el código del módulo, lo envolverá en una función e lo invocará.

Que son los Bundlers?

Bien, ahora que sabemos algo sobre módulos, veamos cómo estos revolucionaron la forma de escribir código para el front-end con la introducción de los bundlers.

Como sabemos, la forma de importar librerías (que a su vez son módulos) en HTML es la siguiente:

```

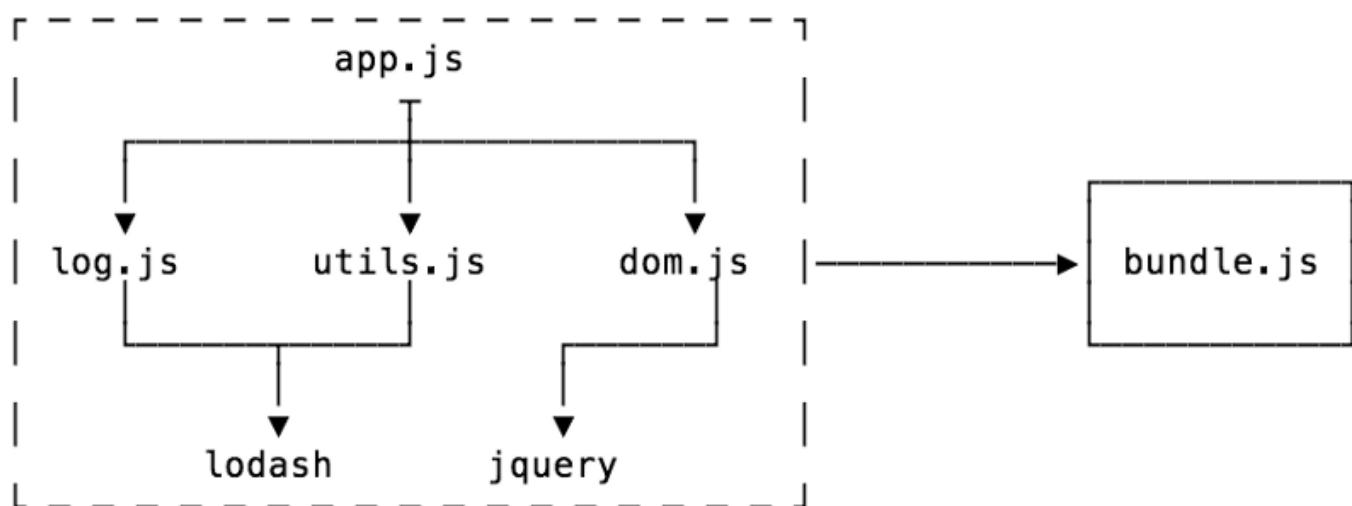
<html>
  <head>My Webpage</head>
  <body>
    <script src="script1.js"></script>
    <script src="script2.js"></script>
    <script src="script3.js"></script>
    <script src="script4.js"></script>
  </body>
</html>

```

Si pensamos en detalle que sucede cuando importamos cada uno de esos scripts, veremos que básicamente todos terminan cayendo al mismo contexto, el global. Para salvar esto, las librerías básicamente elegían arbitrariamente un nombre de variable donde exponer su funcionalidad. Por ejemplo, [jQuery](#) utilizaba el signo `$`. Ahora bien, si otra librería decidía utilizar el mismo nombre de variable para su interfaz, tendríamos un conflicto, y ambas librerías no podrían ser usadas en el mismo HTML.

Al principio, sólo se importaba una cantidad pequeñas de librerías para el front, por lo tanto esto no era un problema tan grande. pero a medida que la complejidad del front fue aumentando, y la cantidad de librerías tambien, se tuvo que pensar una nueva forma para resolver esto.

Acá aparecieron los **Module Bundlers**. Como por ejemplo: [Browserify](#), [Webpack](#), [Rollup](#), etc... Básicamente lo que hacen es ejecutar un proceso que lee todas las dependencias de nuestro proyecto, y luego genera un archivo JS que contiene todos los módulos necesarios que podemos incluir en nuestro HTML.



Hay dos cosas etapas en tarea de un Bundler:

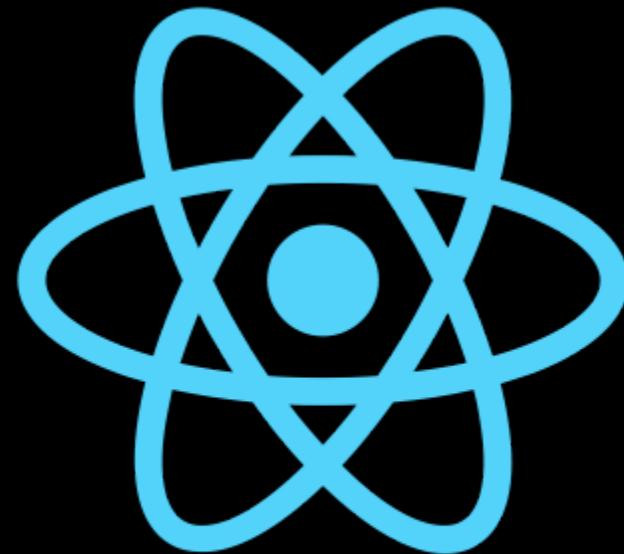
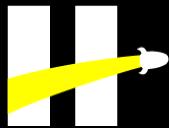
- Resolución de dependencias
- Empaqueamiento

Entrando desde un entry point (nuestro archivo `.js` principal), el objetivo de la resolución de dependencias es buscar todas las dependencias del código y construir un grafo (llamado grafo de dependencias).

Una vez hecho esto, podés empaquetar o convertir todo tu grafo de dependencias en un sólo archivo que tu aplicación va a usar. Finalmente, obtenemos un archivo único (el **bundle**) que vamos a importar en nuestro HTML. De esta forma, resolvemos los problemas de encapsulamiento que mencionamos anteriormente.

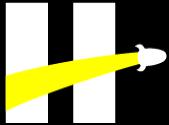
Cuando veamos *React*, vamos a aprender a usar el bundeler *webpack*.

HENRY



React

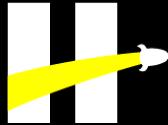
Parte I



REACT



React es una librería de **JavaScript** que es **declarativa**, **eficiente** y **flexible** y sirve para construir interfaces de usuarios. Esta librería fue creada por el equipo de *facebook* e *instagram*, que fue liberada y ahora es un proyecto *open source*.

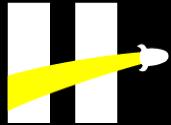


REACT

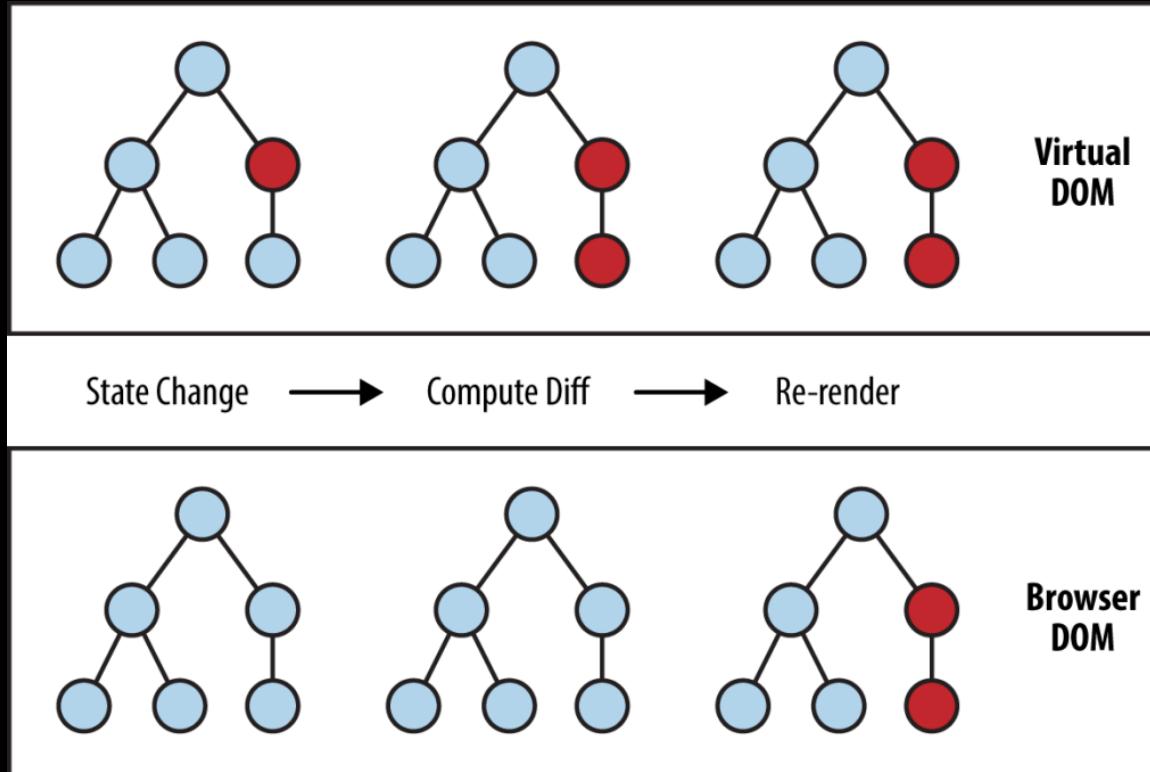
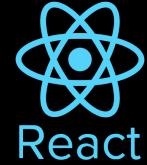


```
1
2 const numbers = [4,2,3,6];
3
4 //imperativo (le decimos COMO queremos que se hagan las cosas)
5
6 let total = 0;
7 for (let i = 0; i < numbers.length; i++){
8   total += numbers[i]
9 }
10
11
12
13 //declarativo (decime QUE queremos que se haga)
14
15 numbers.reduce(function(p, c){
16   return p + c;
17 })
```

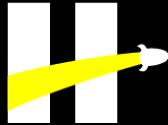
Programación declarativa vs imperativa



REACT



Virtual DOM



REACT



TfL Tube Tracker

< Network />

Bakerloo Line
<Line />
Stations Go

Central Line
<Line />
Stations Go

Circle Line
<Line />
Stations Go

District Line
<Line />
Stations Go

Hammersmith & City Line
<Line />
Stations Go

Jubilee Line
<Line />
Stations Go

Metropolitan Line
<Line />
Stations Go

Station Name Line
<Predictions />

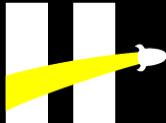
Platform 1 <DepartureBoard />

| Destination | Due | Current location |
|-----------------|------------|-------------------|
| White City | 0:00 | At Platform |
| Ealing Broadway | <Trains /> | Holland Park |
| West Ruislip | 4:30 | Notting Hill Gate |
| Ealing Broadway | 6:00 | Queensway |

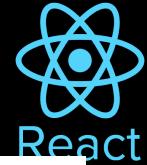
Platform 2 <DepartureBoard />

| Destination | Due | Current location |
|-----------------|------------|-------------------|
| White City | 0:00 | At Platform |
| Ealing Broadway | <Trains /> | Holland Park |
| West Ruislip | 4:30 | Notting Hill Gate |
| Ealing Broadway | 6:00 | Queensway |

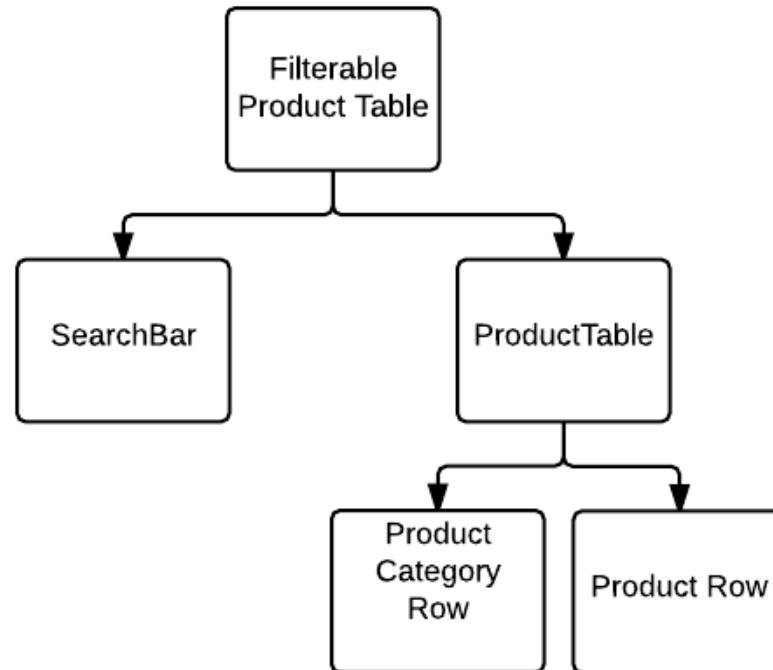
Component Driven Development



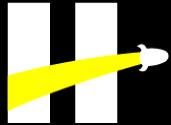
REACT



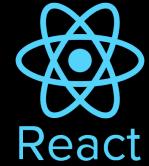
| |
|--|
| <input type="text" value="Search..."/> |
| <input type="checkbox"/> Only show products in stock |
| Name Price |
| Sporting Goods |
| Football \$49.99 |
| Baseball \$9.99 |
| Basketball \$29.99 |
| Electronics |
| iPod Touch \$99.99 |
| iPhone 5 \$399.99 |
| Nexus 7 \$199.99 |



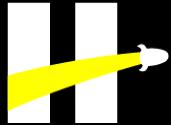
single responsibility principle, o principio de responsabilidad única



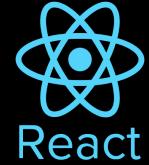
Componente



state is used for internal
communication inside a Component



JSX



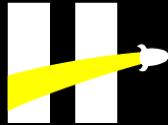
```
1  
2 const element = <h1>Hello, world!</h1>;
```



Babel compila JSX a llamadas de
React.createElement().



```
1 "use strict";  
2  
3 var element = /*#__PURE__*/React.createElement("h1", null, "Hello, world!");
```



JSX

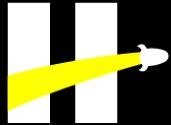


```
1 function formatName(user) {
2   return user.firstName + ' ' + user.lastName;
3 }
4 const user = {
5   firstName: 'Toni',
6   lastName: 'Tralice'
7 };
8 const element = (
9   <h1>
10   Hello, {formatName(user)}!
11   </h1>
12 );
```

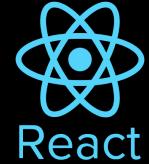
BABEL



```
1 "use strict";
2
3 function formatName(user) {
4   return user.firstName + ' ' + user.lastName;
5 }
6
7 var user = {
8   firstName: 'Toni',
9   lastName: 'Tralice'
10 };
11 var element = React.createElement("h1", null, "Hello, ", formatName(user), "!");
```



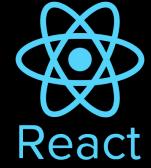
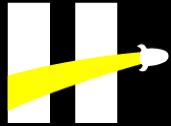
Componentes Funcionales y De Clase



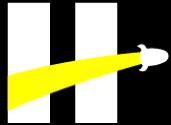
```
1 // Funcional
2
3 function Welcome(props) {
4   return <h1>Hello, {props.name}</h1>;
5 }
```



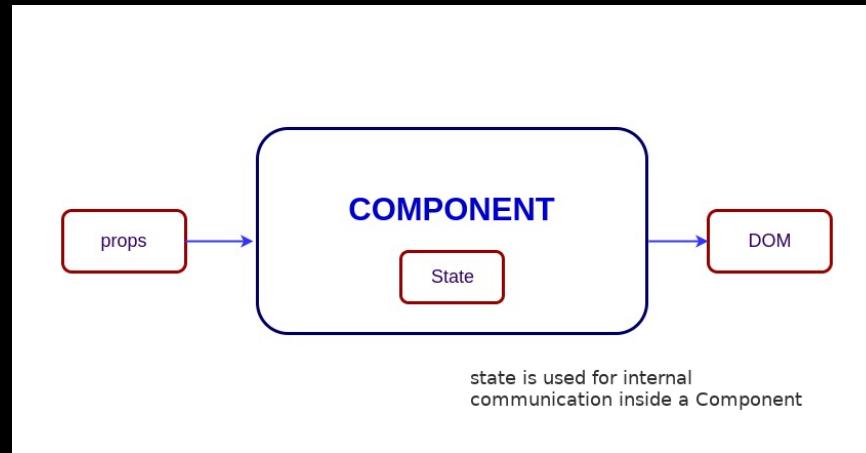
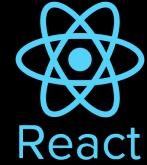
```
1 // Componente de Clase
2
3 class Welcome extends React.Component {
4   render() {
5     return <h1>Hello, {this.props.name}</h1>;
6   }
7 }
```



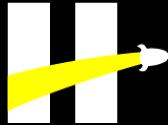
<Demo />



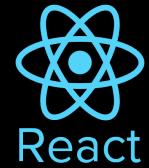
Props



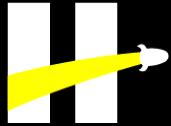
Conceptualmente, los componentes son como funciones de JS. Aceptan inputs arbitrarios (props) y retornan elementos REACT que representan lo que debería aparecer en la pantalla



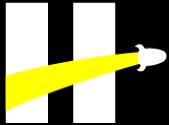
Props



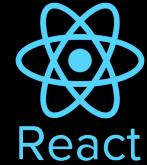
```
1
2
3 class Welcome extends React.Component {
4   render() {
5     return <h1>Hello, {this.props.name}</h1>;
6   }
7 }
8
9 function Welcome(props) {
10   return <h1>Hello, {props.name}</h1>;
11 }
```



<Demo />



JSX a JS



jsx

```
<h1 className="greeting">  
  Hello, world!  
</h1>
```

js

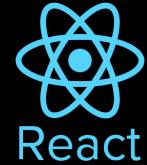
```
React.createElement(  
  'h1',  
  {className: 'greeting'},  
  'Hello, world!'  
) ;
```

@babel/preset-react



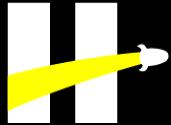


Webpack

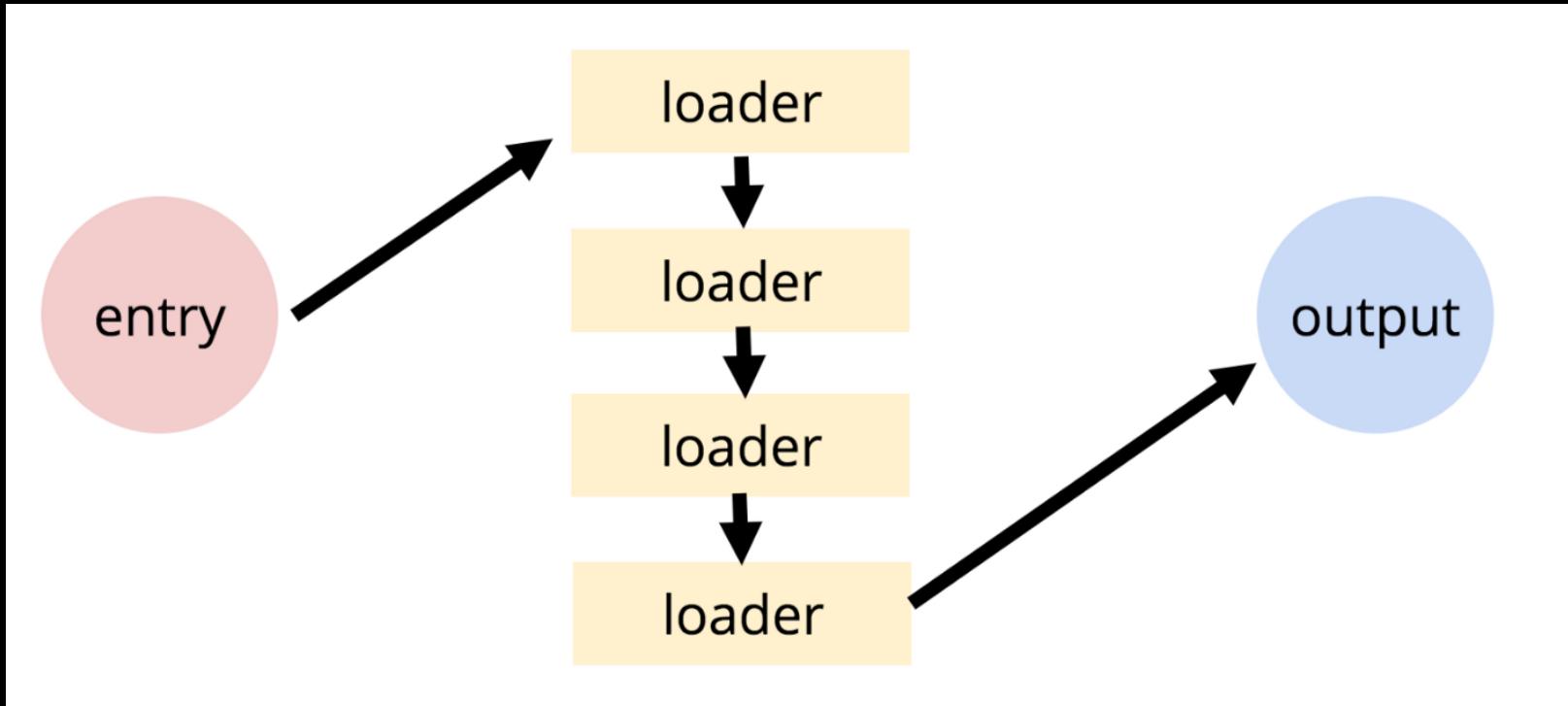
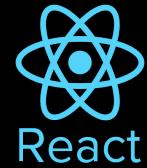


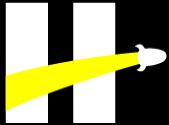
```
1 // instalamos webpack
2 npm i -D webpack webpack-cli
3
4 // instalamos React
5
6 npm i react react-dom
7
8 // instalamos babel y sus dependencias
9
10 npm i -D @babel/core @babel/preset-env @babel/preset-react babel-loader
```

- **webpack:** La base de webpack, el bundler y el superhéroe de la historia.
- **webpack-cli:** Para habilitar comandos de webpack en la consola.
- **react:** La librería de react.
- **react-dom:** La otra librería de react para trabajar con el DOM.
- **@babel/core:** La base de babel, el compilador de javascript moderno.
- **@babel/preset-env:** Para que babel pueda compilar ECMAScript 6.
- **@babel/preset-react:** Para que babel pueda compilar jsx.
- **babel-loader:** Para que webpack utilice babel.



Webpack

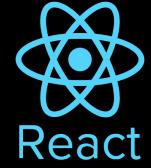
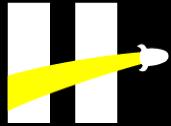




Webpack



```
1 // webpack.config.js
2
3 module.exports = {
4   entry: 'index.js',
5   output: {
6     path: __dirname + '/dist',
7     filename: '[name].js'
8   },
9   module: {
10     rules: [
11       {
12         test: /\.js|jsx$/,
13         exclude: /node_modules/,
14         use: {
15           loader: 'babel-loader',
16           options: {
17             presets: ['@babel/preset-react', '@babel/preset-env']
18           }
19         }
20       }
21     ]
22   }
23 }
```



<Demo />

Henry



Hacé click acá para dejar tu feedback sobre esta clase.

Hacé click acá completar el quiz teórico de esta lecture.

REACT

Qué es React?

React es una librería de JavaScript que es declarativa, eficiente y flexible y sirve para construir interfaces de usuarios. Esta librería fue creada por el equipo de *facebook* e *instagram*, que fue liberada y ahora es un proyecto **open source**.

La diferencia entre programación declarativa o imperativa, es que en la primera le *decimos* a la computadora **qué** queremos hacer (ella se encarga de saber cómo), mientras que en programación *imperativa* le decimos exactamente **cómo** queremos que se hagan las cosas. Puede que parezcan dos cosas iguales, pero veamos la diferencia con un ejemplo:

```
const numbers = [4,2,3,6];
//imperativo (le decimos COMO queremos que se hagan las cosas)
let total = 0;
for (let i = 0; i < numbers.length; i++){
  total += numbers[i]
}
//declarativo (decime QUE queremos que se haga)
numbers.reduce(function(p, c){
  return p + c;
})
```

Al usar *React* vamos a tener que pensar en términos de **componentes**. Cuando armemos una página con *react* vamos a tener que pensar nuestro sitio o página como una serie de **pequeños componentes**. En realidad, podemos decir que todo es un *componente*, de hecho vamos a tener *componentes* que estén formados por otros *componentes*. Esto último va a suceder cuando tengamos un problema que sea grande, y la única forma (o la mejor) para resolver en *react* es dividirlo en pequeños problemas. Esto además hace que los componentes sean altamente **reusables**.

Esta forma de desarrollar se conoce como **component driven development**.

React también es muy bueno en términos de performance, cuenta con un feature llamado **Virtual DOM**, con lo cual logra renderizar muy rápido las páginas manteniendo el código entendible y fácil de manejar.

Virtual DOM

Cuando queremos rastrear cambios en algún modelo y luego trasladarlos al DOM (rendering), tenemos que tener en cuenta dos cosas importantes:

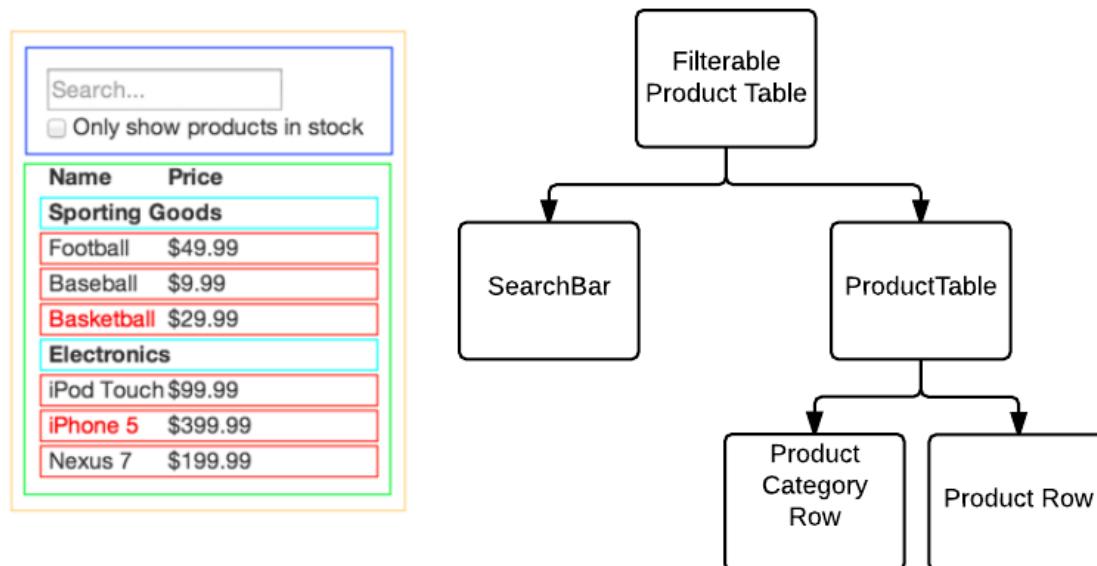
1. Cúando se cambiaron los datos.
2. Qué elementos del DOM necesitan ser actualizados.

Para el primer punto react utiliza un [modelo de observador](#) (esto quiere decir que no tiene que estar revisando continuamente por cambios). Por lo tanto cuando algo cambia este le *avisa* a react inmediatamente.

Para el segundo punto, React construye una representación del DOM en memoria y calcula que elementos del DOM van a cambiar. Hacer cambios en el DOM consume muchos recursos, es por eso que se concentraron tanto en minimizar al máximo los cambios en el DOM real. Muy básicamente, cuando algo cambia en el estado del modelo, la idea es no tocar el DOM e ir haciendo cambios en el Virtual DOM, luego se computan las diferencias entre el DOM real y el virtual DOM (para esto se utilizan [algoritmos de diferencia bastante copados](#)), y finalmente se realizan la menor cantidad de cambios posibles al DOM real.

Component Driven Development

Miremos la imagen de abajo, cada cajita con un color particular representa un componente. Esta es una de las muchas formas de poder dividir un solo elemento o feature de nuestro sitio. Según esta división tendríamos la jerarquía de componentes que se muestran a la derecha de la imagen:



Qué debería contener un Componente?

Para diseñar componentes es importante tener en cuenta el principio de diseño llamado [single responsibility principle](#), o *principio de responsabilidad única*, basicamente deberíamos diseñar cada componente para que sea

responsable de sólo una cosa. Pensar de este modo no es fácil, [acá](#) hay un tutorial de *facebook* para empezar a pensar en componentes.

De todos modos, la mejor forma de aprender es la práctica. Al usar React, te vas a ir dando cuenta cuando te conviene subdividir un componente en otros o no. De hecho, no te deberías preocupar tanto por asumir esta mentalidad antes de empezar, aceptá el hecho que mientas desarrolles con react vas a ir cambiando solo la mentalidad, y no al revés.

Usando React

Empezemos por *instalar* React y usarlo en una página estática, para que veamos cómo se siente. Primero vamos a comenzar con la forma *sencilla* de empezar con React, consiste en hacerlo en un documento HTML.

Por ahora no se preocupen en entender el código en su totalidad. Luego veremos en qué consiste cada cosa que usamos aquí, por ahora lo importante es aprender a instalar lo necesario para poder empezar a desarrollar.

Luego veremos la mejor manera de usar react, que consiste generar un pipeline con algunas herramientas para poder tener el *ambiente* preparado para React, estas herramientas pueden ser: [glup](#), [grunt](#), o [webpack](#), nosotros usaremos el último.

Vamos a ver los siguientes componentes:

- React: La librería en si.
- React Router: Nos permite mapear componentes a URLs, de esta forma podremos armar páginas tipo SPA.
- Webpack: Es una herramienta que agrupa modulos de JavaScript (entre otras cosas que hace), o mejor dicho, agrupa código, lo vamos a usar para crear el pipeline para desarrollar con React.
- Babel: Es una librería/herramienta que nos permite transformar nuestro código. Con React vamos a usar JSX, que es un lenguaje construido sobre JS, Babel lo va a transformar a JS normal.

La Manera Sencilla

La forma más fácil y rápida para poder empezar a ver *como es* React es a través de un documento HTML. Ojo, esta no es la *mejor* forma ni la que usaremos en adelante. Aca veremos dos formas de representar un componente en React. Componente basado en clases y funciones. Hasta hace poco con las clases era la única forma que teníamos de utilizar características como el estado (state), hasta la introducción de Hooks (lo veremos en profundidad más adelante), que nos permiten usar state dentro de un componente de función.

```
<!DOCTYPE html>
<html>
  <head>
    <title>React</title>
    <script crossorigin src="https://unpkg.com/react@16/umd/react.development.js">
    </script>
    <script crossorigin src="https://unpkg.com/react-dom@16/umd/react-
dom.development.js"></script>
    <script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
  </head>
```

```
<body></body>
</html>
```

Como puedes comprobar solo estamos haciendo referencia a dos ficheros ficheros JavaScript:

- **react.js**: La librería principal de ReactJS.
- **react-dom.js**: Desde React 0.14 el manejador de DOM de tus componentes React se realiza con esta libreria JavaScript.
- **browser-min.js**: Una version minima de BabelJS para el navegador.

Ahora agreguemos un poco de código de React en el body:

```
<body>
  <div id="app"></div>
  <script type="text/babel">
    class HelloWorld extends React.Component {
      render(){
        return (
          <div>
            Hola, Soy Henry!!
          </div>
        )
      }
    };

    function HelloWorldFunction() {
      return(
        <div>
          Hola, Soy Henry!!
        </div>
      )
    };
    ReactDOM.render(<HelloWorld />, document.getElementById('app'));
  </script>
</body>
```

Como ven el tag script tiene el atributo type con el valor **text/babel** y no el tan conocido text/javascript, esto es debido a que a partir de ahora vamos a escribir código EcmaScript6 y *traducirlo* usando Babel. Lo empezamos a hacer definiendo nuestra primera clase *Hola* extendiendo de la clase primitiva React.Component. Como habrás podido adivinar BabelJS ejecuta toda la transpilación entre ES6 y JavaScript desde tu navegador y es por eso que en la siguiente sección comprenderás porque es mejor utilizar un gestor de procesos que genere ficheros JavaScript precompilados.

La mejor manera: Webpack

Cuando empiezas a trabajar en un proyecto la manera anterior de incluir React no es la mejor, de hecho no puede escalar. Cuando tenemos muchas líneas de código en un sólo archivo, o muchos archivos chicos de Js, el hecho de tener que juntar todo se vuelve complicado. Por suerte, desde hace varios años existen

herramientas que nos van a automatizar este proceso, haciendo que todo el workflow sea óptimo y sobre todo mantenible.

Algunas tareas de las que se encargan estos gestores de proceso pueden ser:

- Juntar código de varios archivos en uno sólo.
- Transpilar código. Por ejemplo, de CoffeeScript, o TypeScript a Javascript.
- Minificar código.
- Concatenar archivos.
- Correr los tests automáticamente.
- etc...

Existen varios gestores de procesos buenos y populares, los más usados son: [Grunt](#), [Gulp!](#) y [Webpack](#).

Nosotros vamos a usar *Webpack*, pero podrían hacer lo mismo con los otros.

Ahora también se utiliza el concepto de **módulos** en el frontend, para lograrlo se utilizan librerías como [Browserify](#) o [CommonJS](#). Básicamente en vez de incluir librerías usando HTML, lo hacemos en el mismo JS, después estas librerías que mencionamos se encarga de incluir realmente el código necesario para que funcione.

Un ejemplo en el frontend se vería algo así:

```
//algunModulo.js
module.exports.doSomething = function() {
  return 'foo';
};
//algunOtroModulo.js
const someModule = require('someModule');
module.exports.doSomething = function() {
  return someModule.doSomething() + 'bar';
};
```

Vamos a empezar instalando y configurando Webpack. Voy a aclarar al principio que Webpack es una herramienta muy poderosa, por ende compleja, y lamentablemente su documentación no es la mejor. Por lo tanto, nos va a parecer complejo al principio, pero rápidamente nos vamos a encariñar con todas las cosas que podemos hacer con Webpack.

```
$ npm i -D webpack webpack-cli
```

Como dijimos, Webpack es una herramienta que va a aplicar ciertas *transformaciones* a nuestro código, por ende para funcionar webpack necesita saber:

1. Conocer el starting point de nuestra app, o el archivo javascript raíz.
2. Debe saber qué transformaciones tiene que hacer al código.
3. Tiene que saber dónde guardar el nuevo código transformado.

Todo esta información va a estar contenida en un archivo de configuración llamado `webpack.config.js`, que deberíamos crear en la raíz del directorio de nuestro proyecto. Este archivo va a ser en realidad un módulo, que va a exportar un objeto con las configuraciones de webpack, así que podríamos empezar escribiendo lo siguiente en ese archivo:

```
// dentro de webpack.config.js
module.exports = {}
```

Ahora empezemos a agregar la información que mencionamos antes, empezemos por el punto 1 : el entry point.

```
module.exports = {
  entry: [
    './app/index.js'
  ]
}
```

Como ven, los entry points se definen dentro del objeto que exportamos bajo el nombre `entry`, y cuyo valor es un arreglo. Dentro de este explicito los paths de todos los archivos que sirvan como entry points de nuestra app. Por ahora vamos a escribir sólo uno.

Bien, ahora para el segundo punto, tenemos que definir qué tipo de transformaciones vamos a hacer, para esto entran en juego los `loaders`, estos son los módulos encargados de realizar transformaciones, existen varios tipos de `loaders`, ya que la comunidad va creando nuevos a medida que surgen nuevas necesidades.

Para usar un loader, es necesario tenerlo instalado antes. Para eso vamos a usar `npm`. Por ejemplo, si quiero usar el `loader` de babel debería hacer: `npm i -D @babel/core @babel/preset-env @babel/preset-react babel-loader`.

```
module.exports = {
  entry: [
    './app/index.js'
  ],
  module: {
    loaders: [
      {test: /\.coffee$/, exclude: /node_modules/, loader: "coffee-loader"}
    ]
  },
}
```

En el ejemplo, usamos un `loader` de *coffeeScript*. Como se ve, también agreamos una propiedad llamada `module` que será un objeto dentro del cual aparecerá la propiedad `loaders` que será un arreglo de objetos. Cada objeto dentro de este arreglo representa una transformacion. Vemos que ese objeto tiene tres propiedades: `test`, `exclude`, y `loader`. La primera hace referencia a qué archivos deberán pasar por la transformación, y recibe como valor una **expresión regular**, en nuestro ejemplo estamos diciendo que

pasarán por la transformación todos los archivos terminados en `.coffee`. La segunda, `exclude` le indica a webpack qué directorios excluir, en nuestro ejemplo (y siempre lo haremos) excluimos `node_modules`, donde sabemos que no habrá código para transformar. Finalmente, en la propiedad `loader` vamos a poner el nombre del loader que queremos usar, en este caso el nombre es "coffee-loader".

Siempre busquen los loaders que necesiten dentro del ecosistema npm.

Por último, vamos a agregar donde queremos que webpack deposite los archivos luego de la transformación:

```
module.exports = {
  entry: [
    './app/index.js'
  ],
  module: {
    loaders: [
      {test: /\.coffee$/, exclude: /node_modules/, loader: "coffee-loader"}
    ]
  },
  output: {
    path: __dirname + '/dist',
    filename: 'bundle.js'
  },
}
```

Ya podrán imaginarse que quieren decir las cosas que hemos agregado ahora. Por empezar una nueva propiedad `output` que contiene un objeto, en este último vamos a especificar el nombre del archivo de salida (`filename`) y la carpeta donde queremos que se guarde (`path`).

Bien, entonces intentemos reproducir el mismo ejemplo que hicimos en el HTML, pero ahora usando este proceso. Por lo tanto, lo primero que hacemos es sacar el código escrito en React que estaba embebido en el HTML y lo pasamos a un archivo `js`. El primer cambio que tenemos que hacer es importar los módulos `react` y `react-dom` que antes requeríamos a través del tag `script`:

```
import React from 'react';
import ReactDOM from 'react-dom';

class HelloWorld extends React.Component {
  render(){
    return (
      <div>
        Hola, Soy Henry!!
      </div>
    )
  }
};

function HelloWorldFunction() {
  return(
    <div>
```

```

    Hola, Soy Henry!
  </div>
)
};

ReactDOM.render(<HelloWorld />, document.getElementById('app'));

```

Genial, ahora tenemos que construir el archivos de configuración de webpack, para que funcione con [babel](#). Básicamente tenemos que transformar el código que usa EcmaScript6 y JSX a JS plano.

```
// webpack.config.js

module.exports = {
  entry: [
    './app/index.js'
  ],
  output: {
    path: __dirname + '/dist',
    filename: 'bundle.js'
  },
  module: {
    rules: [
      {
        test: /\.(js|jsx)$/,
        exclude: /node_modules/,
        use: {
          loader: 'babel-loader',
          options: {
            presets: ['@babel/preset-react', '@babel/preset-env']
          }
        }
      }
    ]
  }
}
```

Por último vamos a tener que usar [npm](#) para instalar las dependencias:

```
$ npm install -D @babel/core @babel/preset-env @babel/preset-react babel-loader
```

```
$ npm install react react-dom --save
```

Para poder ejecutar webpack, debemos agregar dentro de [scripts](#) en nuestro [package.json](#) lo siguiente:

```
"scripts": {
  "build": "webpack -w",
```

```

    }
  "devDependencies": {
    "@babel/core": "^7.9.0",
    "@babel/preset-env": "^7.9.0",
    "@babel/preset-react": "^7.9.4",
    "babel-loader": "^8.1.0",
    "webpack": "^4.42.1",
    "webpack-cli": "^3.3.11"
  },
  "dependencies": {
    "react": "^16.13.1",
    "react-dom": "^16.13.1"
  }
}

```

Para probar si todo funciona bien, iremos a la carpeta donde tenemos definidos todos estos archivos, y vamos a escribir `npm run build`.

```
[atralice@arch-laptop webpack]$ npm run build

> webpackDEM0@1.0.0 build /home/atralice/Projects/henry/caronte/M2/06-React/1-Intro/demo/webpack
> webpack -w

webpack is watching the files...

Hash: 05ea77c667adec404f91
Version: webpack 4.42.1
Time: 1673ms
Built at: 04/05/2020 6:46:57 PM
    Asset      Size  Chunks      Chunk Names
bundle.js  131 KiB     0  [emitted]  main
Entrypoint main = bundle.js
[7] ./app.js + 3 modules 8.53 KiB [0] [built]
| ./app.js 940 bytes [built]
| ./src/containers/Main.jsx 3.54 KiB [built]
|   + 2 hidden modules
+ 7 hidden modules
```

Si todo funcionó bien, veremos un mensaje como el de la imagen! Y además encontraremos un archivo nuevo en la carpeta `dist`.

Bien, ahora por último tenemos que agregar ese archivo generado a un HTML para poder correrlo:

```

<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Descubre React</title>
  </head>
  <body>
    <div id="app"></div>
    <script src="./dist/index_bundle.js"></script>
  </body>
</html>
```

Si abren este archivo van a ver que vemos exactamente lo mismo que en el primer archivo HTML que creamos. La ventaja de esta forma, es que tenemos separado el código JS de todo lo demás, podemos tener múltiples

archivos y Webpack se encargará de unirlos y depositarlos en el archivo de salida.

Bien, ahora que lo hicimos funcionar y más o menos vimos cómo se escribe, aprendamos un poco más sobre React.

Componentes

Como dijimos los bloques básicos con lo que vamos a construir todo en React se llaman *Components*.

Podemos pensar a un *Componente* como una colección de HTML, CSS y JS, y un estado o datos específicos para ese componente. Estos componentes están definidos o en JavaScript puro, o usando lo que se conoce como **JSX**.

JSX

Básicamente, **JSX** es *syntactic sugar* para una función en particular de React:

`React.createElement(component, props, ...children)`. Lo que sucede es que esta función recibe un objeto que describe un elemento tipo XML (parecido a HTML, pero con tags que podemos inventar). Es contraintuitivo escribir un elemento XML, con sus propiedades y demás, codificado en un objeto, por eso mismo es que crearon este *lenguaje* que nos permite escribir directamente código **XML** embebido en **JS** y que luego será transformado a **JS** por algún loader, como *babel*.

Veamos un ejemplo:

Sin **JSX**:

```
React.createElement(  
  MyButton,  
  {color: 'blue', shadowSize: 2},  
  'Click Me'  
)
```

Con **JSX**:

```
<MyButton color="blue" shadowSize={2}>  
  Click Me  
</MyButton>
```

Esta es la razón de ser de **JSX**, podemos leer más en detalle las demás features que nos ofrece [acá](#).

Podemos ir a la página de *babel* y ver cómo se transforme el código **JSX** en Javascript plano:

```
// Código en JSX, componente basado en clases  
class HelloWorld extends React.Component{  
  render(){  
    return (  
      <div>  
        Hola, Soy Henry!!
```

```
</div>
)
}
};

// Código en JSX, componente de función
function HelloWorldFunction() {
  return(
    <div>
      Hola, Soy Henry!
    </div>
  )
};

// Código en Javascript con un componente de clases
"use strict";

var HelloWorld = function (_React$Component) {

  _createClass(HelloWorld, [{

    key: "render",
    value: function render() {
      return React.createElement(
        "div",
        null,
        "Hola, Soy Henry!!"
      );
    }
  }]);
}

return HelloWorld;
}(React.Component);

// Código en Javascript con un componente de funciones
"use strict";

function HelloWorld() {
  return React.createElement(
    'div',
    null,
    "Hola, Soy Henry!!"
  );
}
```

Como podemos ver, la transformación toma los datos y lo transforma a código Javascript. Se llamó a `React.createElement` en la función `render`, esta función crea un elemento HTML según los parámetros que les pasamos. Como podemos ver, escribir código JavaScript para crear elementos HTML puede ser engorroso. Si bien, podríamos codear todas nuestras apps de react escribiendo JS nativo, lo mejor y más productivo va a ser usar **JSX**.

Pueden ver más transformaciones que realiza *babel* [acá](#);

Creando nuestro primer componente

Usemos el ejemplo de arriba, pero pensemos paso a paso cómo crear ese componente.

Como podemos imaginar, un *Componente* en react está representado por una clase o un objeto llamado *Component*. Este tiene incorporado una serie de propiedades y métodos, los cuales logran el comportamiento y le dan el poder de React.

Cuando nosotros creamos un componente nuevo, básicamente *heredamos* todas esas propiedades y métodos del objeto *Component* y luego customizamos el nuevo componente según nuestras necesidades. Veamos un ejemplo con un componente de clases:

```
import React from 'react';
import ReactDOM from 'react-dom';

class HelloWorld extends React.Component {
  render(){
    return (
      <div>
        Hola, Soy Henry!!
      </div>
    )
  }
};

ReactDOM.render(<HelloWorld />, document.getElementById('app'));
```

Del mismo modo podemos ver el ejemplo con un componente de función:

```
import React from 'react';
import ReactDOM from 'react-dom';

function HelloWorldFunction() {
  return(
    <div>
      Hola, Soy Henry!
    </div>
  )
};

ReactDOM.render(<HelloWorldFunction />, document.getElementById('app'));
```

Primero importamos las librerías donde contienen la definición de los objetos de React. Luego creamos una clase, en este caso llamada 'HelloWorld', y extendemos a *React.Component* para formar nuestro nuevo **Componente** (este es el paso donde heredamos toda la funcionalidad de React!). Dentro de este nuevo objeto *HelloWorld* vamos a definir el método *render*, que es un método **requerido** por todos los componentes de React. Este método define el **template** de nuestro componente, por lo tanto es obligatorio!. Como verán, en este punto hemos incluido **JSX**, ya que escribimos código XML mezclado con JS, y justamente lo hemos usado para crear un nuevo tag *div* que contenga el String '*Hello World*'.

Bien, ahora ya tenemos nuestro propio componente creado. Para indicarle a React que lo incluya en la página, vamos a llamar al método `render` de `ReactDOM`. Este método recibe como parámetro el componente que queremos incluir al DOM, y en qué lugar del mismo. Por lo tanto le pasamos nuestro componente (`<HelloWorld />`), y un selector `document.getElementById('app')` para indicarle donde agregarlo.

Props

Una de las ventajas de separar todo en Componentes, es que estos pueden ser reutilizables. Para que lo sean, vamos a poder cambiar un poco su comportamiento pasandole algunos datos. En React estos datos se conocen como **props** (propiedades) de un Componente. Veamos la forma de pasar *props* a un Componente.

Las *props* funcionan como los *atributos* HTML, es decir, que cuando usamos un Componente podemos agregarle *props* escribiendo su nombre dentro del tag del mismo. Por ejemplo, para agregar la *prop* `name` al Componente que habiamos creado antes, cuando lo usamos escribimos `<HelloWorld name='Henry' />`. De esta forma, podemos pasar una o varias *props* al mismo Componente. Ahora, para utilizarlas, dentro del Componente vamos a tener un objeto que se encuentra en `this.props` que va a contener todas las *props* que le hayamos dado a ese Componente. En el ejemplo, el `name` va a estar en `this.props.name` y va a tomar el valor de `Toni`. Por último, para poder acceder al contenido de `this.props.name` vamos a tener que escribir una *expresión JavaScript* dentro de *JSX*, para hacerlo tenemos que separarla con `{}`, en el ejemplo también usamos los `{}` para pasar una variable como *prop*:

```
import React from 'react';
import ReactDOM from 'react-dom';

class HelloWorld extends React.Component {
  render(){
    return (
      <div>
        Hola, {this.props.name}!!
      </div>
    )
  }
};

const nombreVariable = 'Toni';

ReactDOM.render(<HelloWorld name={nombreVariable} />,
  document.getElementById('app'));
```

En un componente de función:

```
import React from 'react';
import ReactDOM from 'react-dom';

function HelloWorldFunction(props) {
  return(
    <div>
```

```
Hola, {props.name}!!  
</div>  
)  
}  
  
const nombreVariable = 'Toni';  
  
ReactDOM.render(<HelloWorldFunction name={nombreVariable} />,  
document.getElementById('app'));
```

Ahora tenemos un Componente que nos sirve para saludar! Sólo tenemos que pasarle una *prop* con el nombre de a quien va dirigido el saludo. 😊 Es un ejemplo simple, pero mostramos la forma de React de pasar *props* a sus Componentes. Aca vemos una de las diferencias entre un componente de clases y un componente de funciones. El uso de la palabra reservada 'this', esto nos hace mas facilita a la hora de hacer un debugging, no pensar a que hace referencia 'this' siempre es un plus. Ademas, no usar 'this' significa que no es necesario el uso de bindear los eventos para hacer referencia a eventos dentro de una clase.

Mirá como quedaría este código traducido a JavaScript plano [acá](#).

Una cosa muy importante de las *props* es que son **inmutables**, es decir, que cuando las seteamos no las vamos a poder cambiar en el futuro (por lo menos están pensadas para eso). Podemos pensar en las *props* como una *Inicialización* o una *Configuración* de un Componente antes de agregarlo. Para datos que *cambian* veremos más adelante los **estados** de un Componente.

Eventos de Usuarios y Callbacks

Veamos como podemos capturar algunos eventos disparados por el usuario en React. Vamos a usar un ejemplo, en donde el usuario pueda escribir su nombre en un *form* y luego vamos a mostrar un *alert* con lo que escribió. Primero comenzamos agregando el *form*:

```
import React from 'react';  
import ReactDOM from 'react-dom';  
  
class HelloWorld extends React.Component {  
  render(){  
    return (  
      <div>  
        <form>  
          <input type='text' ref='name'>  
          <button>Poner Nombre</button>  
        </form>  
        Hola, {this.props.name}!!  
      </div>  
    )  
  }  
};  
ReactDOM.render(<HelloWorld name='Soy Henry!' />, document.getElementById('app'));
```

Hemos creado el `form` y dentro un `input` donde el usuario va a escribir su nombre, como pueden ver hemos agregado el atributo `ref`, este es especial de React, y lo usa para poder hacer referencia luego a ese elemento. Por ahora este `form` no hace nada, le agreguemos una acción cuando es Submiteado. Para eso usamos un Evento nativo de React llamado `onSubmit`, que tambien pasamos como un atributo, y dentro suyo la función que queremos que se ejecute como callback. Generalmente estas funciones son partes del Componente, por lo tanto la vamos a definir dentro del mismo:

```
import React from 'react';
import ReactDOM from 'react-dom';

class HelloWorld extends React.Component {
  constructor(props) {
    super(props);
    // Necesitamos el binding para hacer funcionar el this en el evento
    this.onButtonClick = this.onButtonClick.bind(this);
  }
  onButtonClick(e){
    e.preventDefault();
    const name = this.refs.name.value;
    alert(name);
  }
  render(){
    return (
      <div>
        <form onSubmit={this.onButtonClick}>
          <input type='text' ref='name' />
          <button>Poner Nombre</button>
        </form>
        Hola, {this.props.name}!!
      </div>
    )
  }
}
ReactDOM.render(<HelloWorld name='Soy Henry!' />, document.getElementById('app'));
```

Como vemos, pudimos acceder al elemento `input` usando `this.refs` gracias a que agregamos el bind del `this` en nuestro constructor. En el constructos vamos a poder setear props y estados por default para nuestro componente. Cabe notar que lo que se guarda dentro de `refs` es una referencia al **elemento HTML**, por lo tanto si queremos lo que escribió el usuario, usamos `this.refs.name.value`.

Viendo el ejemplo con una función:

```
import React, { useRef } from 'react';
import ReactDOM from 'react-dom';

function HelloWorld(props) {
  let TextInput = useRef(null);
```

```

const onButtonClick = (e) => {
  e.preventDefault();
  const name = textInput.name.value;
  alert(name);
}

return (
  <div>
    <form onSubmit={onButtonClick}>
      <input type='text' ref={textInput} />
      <button>Poner Nombre</button>
    </form>
    Hola, {props.name}!!
  </div>
)
}

ReactDOM.render(<HelloWorld name='Soy Henry!' />, document.getElementById('app'));

```

En el caso de un componente de función, no podemos usar el atributo `ref` ya que esta no puede ser instanciada, y no tenemos el uso de la palabra `this` para hacer referencia a 'refs' dentro de la clase. Ahora estamos usando una nueva característica de React que son los Hooks. En este caso estamos importando el Hook `useRef`, que lo iniciamos con un argumento, en este caso null, cuya propiedad `.current` se inicializa sobre el argumento pasado. En este caso pasamos un objeto de referencia a React con `ref`, React configurará su propiedad `.current` al nodo del DOM correspondiente cuando sea que el nodo cambie.

Como vemos, logramos tener el `alert` con el nombre que escribió el usuario. Ahora, ya sabemos que los `alerts` no sirven. Intentemos hacer que cuando el usuario haga click en el botón se cambie el nombre en el saludo. Para esto vamos a introducir el concepto de **Estado** de un componente.

Anidando Componentes

Como dijimos antes, en React *todo* es un componente, por lo tanto es lógico pensar que vamos a tener *componentes dentro de componentes* todo el tiempo. Veamos con un ejemplo como funciona esto, y como se le pueden pasar datos (en React le decimos *props*) a los componentes.

Ahora armemos un ejemplo un poco más complejo, en este vamos a tener dos componentes. Uno va a llamar al otro y le va a pasar algunas propiedades, veamos como hacerlo:

```

class ContenedorAmigos extends React.Component {
  render(){
    const name = 'Soy Henry';
    const amigos = ['Santi', 'Guille', 'Facu', 'Solano'];
    return (
      <div>
        <h3> Nombre: {name} </h3>
        <MostrarLista names={amigos} />
      </div>
    )
}

```

```

    }
};
```

En este componente hemos definido un método `render` un poco más complejo, en el tenemos dos variables (`name` y `amigos`) y retornamos un XML que utiliza estas dos variables. Como ven, podemos acceder a un `prop` del mismo Componente usando los `{}`, de esta forma `{name}` va a ser reemplazado por `Soy Henry`. Luego llamamos a un componente que todavía no hemos definido con el nombre de `mostrarLista` y le pasamos como propiedad el arreglo `amigos`. Por lo tanto dentro de `mostrarLista` vamos a disponer de ese arreglo como una `prop`.

Definamos el elemento hijo o `child`:

```

class MostrarLista extends React.Component {
  render(){
    const lista = this.props.names.map(amigo => <li> {amigo} </li>);
    return (
      <div>
        <h3> Amigos </h3>
        <ul>
          {lista}
        </ul>
      </div>
    )
  }
};
```

Lo primero que notamos es que usamos `JS` para crear elementos HTML más complejos. En este caso usamos la función `map`, para crear un elemento `` por cada `amigo` en la lista o arreglo. Viendo el ejemplo anterior usando funciones:

```

function ContenedorAmigos() {
  const name = 'Soy Henry';
  const amigos = ['Santi', 'Guille', 'Facu', 'Solano'];
  return (
    <div>
      <h3> Nombre: {name} </h3>
      <MostrarLista names={amigos} />
    </div>
  )
};

function MostrarLista({ names }) {
  const lista = names.map(amigo => <li> {amigo} </li>);
  return (
    <div>
      <h3> Amigos </h3>
      <ul>
        {lista}
```

```

        </ul>
    </div>
)
};

```

Aca podemos usar **destructuring** para pasar las props directamente con el nombre de la variable **names**

Por si no se acuerdan como funciona map:

```

const amigos = ['Santi', 'Guille', 'Facu', 'Solano'];
const lista = amigos.map(amigo => "<li> " + amigo + "</li>");
console.log(lista); //['<li> Santi</li>', '<li> Guille</li>', '<li> Facu</li>',
'<li> Solano</li>'];

```

Justamente ese nuevo conjunto de **lis** que hemos creado, lo vamos a usar envuelto en tags **** para formar la lista de amigos.

Etiquetas HTML y Componentes

Los componentes que creamos en React despues los usamos escribiendolos como un tag HTML, en realidad es un tag **XML**. Por ejemplo: el tag **MostrarLista** es un Componente que creamos antes y lo usamos así:

```

<div>
  <h3> Nombre: {name} </h3>
  <MostrarLista names={amigos} />
</div>

```

Luego ese tag se renderizará a lo que sea que hayamos escrito en el método **render** de ese componente, transformandose así en HTML finalmente. Existe una convención en React para distinguir entre Componentes React y elemento HTML nativos. Para el primero usamos BumpyCase y lowercase para el último. Por ejemplo:

```

<MostrarLista /> BumpyCase
<div>           lowercase

```

Separando Componentes

Seguro estarán pensando que si tenemos Componentes, lo mejor sería poder tenerlos también en archivos y carpetas distintas. Lo bueno de React es que es TODO JavaScript, asi que vamos a poder usar **CommonJS** para exportar Componentes como módulos y luego requerirlos:

```

import React from 'react';

export class Lista extends React.Component {
  render(){

```

```

const lista = this.props.names.map(amigo => <li> {amigo} </li>);
return (
  <div>
    <h3> Amigos </h3>
    <ul>
      {lista}
    </ul>
  </div>
)
};

};

};

```

Luego en el archivo donde lo necesitemos simplemente lo requerimos y lo empezamos a usar:

```
import { Lista } from './MostrarLista.js';
```

En este caso usamos las llaves en '{ Lista }' porque le dimos nombre a nuestro export:

```
export class Lista extends React.Component
```

De haber sido un export default podemos hacer un import simple porque le indica que es lo único que se importará.

```

// Presentational
export default class Lista extends React.Component

// Container
import Lista from './MostrarLista.js';

```

De esta forma vamos a poder organizar muy bien nuestros componentes en distintos archivos y carpetas.

React y Funciones Puras

Como vimos recién vamos a poder usar todo lo que sabemos de JS para codear con React. Pensemoslo así, en vez de tener *funciones* que tomen argumentos y retornen valores y objetos, en React vamos a tener *funciones* que tomen argumentos y retornen **UI (user interfaces)**. Podemos resumir este concepto en $f(d) = V$, es decir, una función toma d argumentos y retorna una **View**. Esta es una buena forma de desarrollar interfaces, porque ahora toda tu interfaz está compuesta de invocaciones a funciones, que es la forma en que estamos acostumbrados a programar nuestras aplicaciones. Veamos este concepto en código:

```

const getFoto = function(username) {
  return 'https://photo.fb.com/' + username
}
const getLink = function(username) {

```

```
    return 'https://www.fb.com/' + username
}
const getData = function(username) {
  return {
    foto: getFoto(username),
    link: getLink(username)
  }
}
getData('atralice')
```

Si vemos el código de arriba, notamos que tenemos tres funciones y una invocación a una función. De esta forma logramos que el código sea modular y entendible. Cada función tiene un propósito específico y luego las componemos en otra función en donde generaremos un comportamiento que utiliza cada una de estas para lograr el comportamiento que deseamos.

Bien, ahora modifiquemos ese código para que devuelvan un *UI* en vez de sólo datos:

```
import React from 'React';

class Foto extends React.Component {
  render() {
    return (
      <img src={'https://photo.fb.com/' + this.props.username} />
    )
  }
};

class Link extends React.Component {
  render() {
    return (
      <a href={'https://www.fb.com/' + this.props.username}>
        {this.props.username}
      </a>
    )
  }
};

class Avatar extends React.Component {
  render(username) {
    return (
      <div>
        <Foto username={this.props.username}/>
        <Link username={this.props.username}/>
      </div>
    )
  }
};

<Avatar username='atralice' />
```

Ahora, en vez de crear componer funciones que retornen datos, estamos componiendo funciones que retornan *UIs*. Esta idea es tan importante que React en la versión **0.14** introdujo lo que se conoce como **Stateless Functional Components**, que nos permite escribir el código de arriba como simples funciones.

Lee [acá](#) que tienen de bueno las **Stateless Functional Components**.

Reescribamos nuestro código usando **Stateless Functional Components**:

```
import React from 'React';

const Foto = function(props) {
  return <img src={'https://photo.fb.com/' + props.username} />
};

const Link = function(props) {
  return (
    <a href={'https://www.fb.com/' + props.username}
      {props.username}
    </a>
  )
}

const Avatar = function(props) {
  return (
    <div>
      <Foto username={props.username}/>
      <Link username={props.username}/>
    </div>
  )
};

<Avatar username='atralice' />
```

Ahora cada componente es lo que llamamos una **pure function**. Este concepto viene de **Functional Programming**, básicamente, las funciones puras son consistente y predecible, porque tienen las siguientes características:

- Una función pura siempre retorna el mismo resultado para los mismos argumentos.
- La ejecución de una función pura **NO** depende del *estado* de la aplicación.
- Las funciones puras **NO** modifican el estado ni ninguna variable afuera de su scope.

En React el método **render** necesita ser una función pura, y por ende, todos los beneficios de programar funciones puras se transladan ahora a tu **UI**. De esta forma logramos tener lo que en React se conoce como: **Stateless Functional Components**. Si vemos le ejemplo, todos los Componentes que armamos no tiene estado, y que no hacen nada más que recibir datos a través de **props** y renderizar una **UI**, esto es, básicamente, Componentes que sólo tienen el método **render**. De esto nace un paradigma en el que se diferencian dos tipos de Componentes, los que acabamos de mencionar son los llamados **Presentational Components** y los segundos son **Containers Components**.

Como se pueden imaginar, los **Presentational Components** se preocupan en como **se ven las cosas** y los **Container Componentes** en **como funcionan las cosas**. Organizar nuestro código de esta forma trae varias ventajas:

- Mejor separación de temas. Vas a entender mejor tu aplicación y tu UI escribiendo Componentes de este modo.
- Mejor reusabilidad. Podes usar los mismos Presentational Componentes en distintos Containers.
- Podes tener a los diseñadores trabajando en los Presentational Componentes sin tener que meterse a la lógica de la aplicación.

Esto es sólo un paradigma, seguramente hay otros que tengan otras características. Si querés poder leer más de este paradigma [acá](#).

Organizando las carpetas de un Proyecto

Antes mencionamos el patrón de separar Componentes según mantengan *Estados* (**Containers**) o sólo sirvan para renderizar algo (**Presentational**). Vamos a organizar nuestra estructura de carpetas de proyecto alrededor de este.

Básicamente, vamos a guardar cada Componente en un archivo **.js** separado y *exportarlo*. Los *Presentational* van a ir en una carpeta llamada **components**, y los *Containers* en otra carpeta llamada **containers**. Como sabemos, los *Containers* van a incluir o *requerir* a los *Presentational* en su código, y desde código denuestra app vamos a *requerir* a los *Containers*.

Por convención vamos a llamar a los archivos que contengan un componente con la primera letra en Mayúsculas. Por ejemplo: **Header.jsx** o **Profile.js**.

Cuando comenzamos un proyecto nuevo de React, en vez de empezar de cero, podemos guardar un esqueleto que ya tenga todas las tareas que deberíamos repetir en cada proyecto, en inglés esto se conoce como **boilerplates project**. De hecho, podemos hacer nuestro propio **boilerplate** o buscar online alguna que se ajuste a nuestras necesidades. En general que están publicados online traen muchas cosas que tal vez no vayamos a usar, aquí algunos ejemplos:

- [react-webpack-boilerplate](#)
- [react-starter-kit](#)
- [react-native-starter](#)
- [react-webpack-boilerplate](#)

Hace poco salió [Create React APP](#) una mini app del equipo de Facebook que te ayuda a comenzar un proyecto nuevo de React en segundos (yo todavía no la probé pero parece interesante!).

Tambien pusimos nuestro propio ejemplo de un boilerPlate simplificado [acá](#). Básicamente trae un archivo de *webpack* preconfigurado y un mini servidor *express* para levantar el archivo desde la carpeta del output de *webpack*, super simple!

Como siempre, todo viene en muchos *sabores* y hay que probar y elegir el que mas le gusta a cada uno, ninguno es el mejor, todos van a tener pros y cons.

Propiedades y Estados

Ya vimos que en React las propiedades se pasan de componentes padres a hijos a través de la variable `props`. Veamos algunas propiedades más avanzadas del comportamiento de `props`.

this.props.children

Digamos que tenemos un Componente cualquiera (de React o bien HTML simple), y queremos acceder a la data que está entre el opening tag y el closing tag. Por ejemplo, quiero acceder al nombre que está dentro de `<Nombre>`:

```
<Nombre>
  SoyHenry
</Nombre>
```

React nos da una forma simple de acceder a ellos, y es con la propiedad `children` de `this.props`. En este ejemplo en particular, la propiedad `this.props.children` del Componente `Nombre` va a tomar el valor 'Soy Henry'.

Qué Pasa si lo que está adentro es un poco más complejo? Por Ejemplo:

```
<Nombre>
  <Foto />
  <Link />
</Nombre>
```

Ahora dentro de `Nombre` tenemos dos Componentes (`Foto` y `Link`). Bien, como se podrían imaginar, `this.props.children` va a evaluar a un **arreglo** de Componentes.

PropTypes

Algunas veces va a ser necesario controlar el tipo de datos de las `props` que estamos enviando a un Componente. Por suerte, React nos provee de una funcionalidad para hacer de forma *nativa*. Esta forma son las `PropTypes`. Básicamente consiste en definir un objeto de configuración en donde vamos a declarar el tipo de datos de cada propiedad, si no coinciden cuando los pasemos, React nos generará un error. Por ejemplo:

```
import React from 'react';

const PropTypes = React.PropTypes;

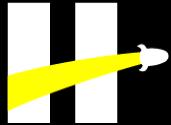
class Icono extends React.createClass {
  propTypes: {
    nombre: PropTypes.string.isRequired,
    tamano: PropTypes.number.isRequired,
    color: PropTypes.string.isRequired,
    style: PropTypes.object
  }
  render: ...
};
```

En este ejemplo estamos declarando que las *props* que le lleguen al Componente **Icono**, deberán ser:

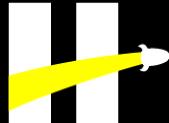
- *nombre*: Tiene que ser de tipo *String* y es obligatorio.
- *tamanio*: Tiene que ser de tipo *Number* y es obligatorio.
- *color* : Tambien un *String* y obligatorio.
- *style* : Debe ser un *Objeto*, no es obligatorio.

Como vemos, toda esta funcionalidad está contenida en el objeto **React.PropTypes** que viene nativamente en React.

Para más información sobre **React.PropTypes** y las cosas que podemos controlar con ella vamos a la documentación oficial [aquí](#).



Estilos en React



Estilos en React (Legacy)



```
1 import React from 'react';
2 import './App.css';
3
4 class App extends React.Component {
5   render() {
6     return (
7       <div className="App">
8         <h1>Título</h1>
9       </div>
10    );
11  }
12}
13
14 export default App;
```

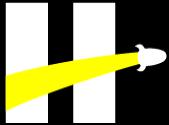
Vemos que estamos usando `import` con un archivo .css! Esto sucede gracias a webpack.



Estilos en React

Necesitamos un loader nuevo para poder importar archivos css:

```
1
2 // // $ npm install --save-dev css-loader style-loader
3
4 module.exports = {
5     ....
6     ....
7     module:{
8         rules:[
9             {
10                 test:/\.css$/,
11                 use:['style-loader','css-loader']
12             }
13         ]
14     },
15     ....
16     ....
17 }
```



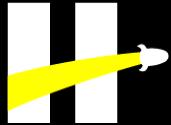
Estilos en React

Pros:

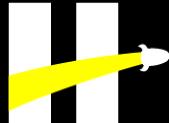
- Compatibilidad: Se da estilos igual que antes, se pueden reusar los css que ya teníamos!
- No hay que aprender nada nuevo, es el mismo paradigma que antes.

Contras:

- Los estilos son globales. Va en contra de la filosofía de los componentes.
- Tenemos los mismos problemas de organización de CSS que antes.



< Demo />



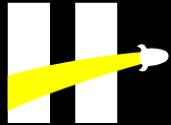
INLINE STYLING (CSS-in-JS)

```
1
2 const divStyle = {
3   color: 'blue',
4   backgroundImage: 'url(' + imgUrl + ')',
5 };
6
7 function HelloWorldComponent() {
8   return <div style={divStyle}>Hello World!</div>;
9 }
```

Podemos escribir CSS en JS!

Hacemos un objeto que tenga las reglas css, y se lo pasamos al atributo style de un tag.

Esta es la forma de dar estilos que muestra React en su documentación



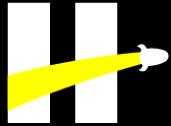
INLINE STYLING (CSS-in-JS)

Pros:

- Menos configuración: no necesitamos ningún loader.
- Estilos locales: no puede haber colisiones.

Contras:

- Perdemos los pseudoSelectores (hover, etc..)
- La sintaxis es un poco rara!



< Demo />



CSS MODULES



```
1 import React from 'react';
2 import s from './Product.css';
3
4 function Product(props) {
5   return (
6     <div className={`${s.producto} ${s.hola}`}>
7       <h3 className={s.hola}>{props.title}</h3>
8       <p>{props.price}</p>
9     </div>
10   );
11 }
12
13 export default Product;
14
15
```

La idea atrás de CSS modules es tener lo mejor de los estilos anteriores: Escribir en css propiamente dicho, y mantener scopes locales.



```
1 .producto h3 {
2   background-color: SpringGreen;
3 }
4
5 .producto {
6   color: salmon;
7 }
8
9 .hola {
10   font-size: 30px;
11 }
```



CSS MODULES



```
1 module.exports = {
2   ...
3   {
4     test: /\.css$/,
5     use: ['style-loader', {
6       loader: 'css-loader',
7       options: {
8         modules: true,
9         localIdentName: '[path][name]__[local]--[hash:base64:5]',
10        camelCase: true,
11        ignore: '/node_modules/',
12      },
13    }],
14  },
15  ...
16};
```

Para implementar CSS-Modules tenemos que agregar este loader, configurado de esta manera.



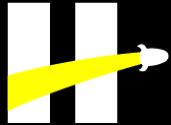
CSS MODULES

Pros:

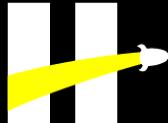
- **Componentizado:** Los estilos son locales, no puede haber colisiones.
- **Estilos locales:** Los estilos son locales para cada componente.

Contras:

- Perdemos los estilos globales, de todos modos
- podríamos combinarlo con la primera forma quevimos.



< Demo />



HOW STANDARDS PROLIFERATE:
(SEE: A/C CHARGERS, CHARACTER ENCODINGS, INSTANT MESSAGING, ETC)

SITUATION:
THERE ARE
14 COMPETING
STANDARDS.

14?! RIDICULOUS!
WE NEED TO DEVELOP
ONE UNIVERSAL STANDARD
THAT COVERS EVERYONE'S
USE CASES.



SOON:

SITUATION:
THERE ARE
15 COMPETING
STANDARDS.

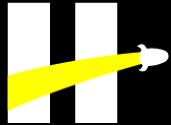


Styled Components



```
1 import styled from 'styled-components';
2
3 const DivWrapper = styled.div`  
4   width: 50%;  
5   border: 2px solid black;  
6   ${props => (props.color === 'blue') ? `background-color: blue`: null}  
7   ${props => (props.color === 'red') ? `background-color: red`: null}  
8 `;
9
10 export default function Component() {
11   return (
12     <DivWrapper>
13       <p>Hello Styled component</p>
14     </DivWrapper>
15   )
16 }
```

“The basic idea of styled-components is to enforce best practices by removing the mapping between styles and components.”



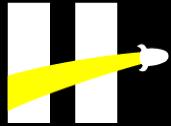
Styled Components

Pros:

- Componentizado: Creamos Componentes con estilos.
- Reutilizacion: Podemos Reutilizar componentes en vez de estilos.

Contras:

- Nuevo paradigma, hay que acostumbrarse a usarlo.



< Demo />

Henry



Hacé click acá para dejar tu feedback sobre esta clase.

Hacé click acá completar el quiz teórico de esta lecture.

Dando Estilos en React

Este es un tema muy particular, ya que se está cambiando la filosofía que teníamos de usar CSS como lo conocemos para dar estilos, es decir en forma *global* y *en cascada*. Ahora hay muchos desarrolladores que creen que esta forma no es *Escalable* y que hay que adoptar nuevas formas de dar estilos a cada Componente, en particular hablan de tener estilos *locales* y no globales.

Esta discusión de paradigmas todavía no tiene un ganador claro, hay defensores y detractores de ambas formas por todos lados. Tomen con pinzas todo lo que lean online sobre este tema.

Estilos Inline

Una de las formas de dar Estilos a los Componentes es usando el atributo `style` del mismo. En react esta propiedad recibe un *objeto JavaScript* y no una *Css String* como en HTML nativo. Por lo tanto, vamos a tener que cambiar un poco la sintaxis de las reglas CSS. Veamos el ejemplo de la [documentación](#) de react:

```
const divStyle = {  
  color: 'blue',  
  backgroundImage: 'url(' + imgUrl + ')',  
};  
  
function HelloWorldComponent() {  
  return <div style={divStyle}>Hello World!</div>;  
}
```

Como vemos, no podemos usar los mismos nombres de las propiedades CSS porque tiene conflicto con el operador `-`, por lo tanto decidieron que es mejor utilizar los nombres de las reglas usando *camelCase*.

Existen traductores de CSS a CSS JavaScript como [este](#).

Clases

En los ejemplos anteriores usamos una clase para setear el estilo de un Link activo. La clase la habíamo definido en el `index.html` en donde se cargaba todo nuestro código, por lo tanto esa definición era accesible por los Links. Del mismo modo, si tenemos clases definidas de esa manera (podríamos usar un

framework como Boostrap, por ejemplo), vamos a poder dar el nombre de las clases que tendrán nuestros Componentes y sus childrens, usando el keyword `className`. Como se imaginan el keyword `class` está reservado en JavaScript y no se puede utilizar para eso. Por ejemplo

```
function HelloWorldComponent() {
  return <div className='activado'>Hello World!</div>;
}
```

Frameworks

Veamos como podemos usar Webpack, para *requerir* un Framework y utilizarlo dentro de nuestra app. Para esto, vamos a necesitar instalar algunos `loaders` extras, para que Webpack sepa como manejar un `require('./estilo.css')` ya que no se tratan de archivos js!

Para eso vamos a instalar varios `loaders`:

- **css-loader**: Sirve para requerir archivos `.css` y tenerlos como objetos JS.
- **script-loader**: Algunos frameworks utilizan jQuery u otras librerías como dependencias, con este loader vamos a poder incluirlas también en nuestra app.
- **style-loader**: Por último nos va a faltar inyectar el CSS en nuestro HTML, lo vamos a hacer usando este loader.
- **url-loader**: Bootstrap viene con sus propias fuentes y utiliza archivos como `.woff` o `.ttf`, vamos a usar este loader para poder inyectar estos archivos en nuestra página.

Para instalar los loaders hacemos: `npm install css-loader script-loader style-loader url-loader --save`. Tambien tenemos que instalar, en este caso Bootstrap y jQuery: `npm install jquery bootstrap --save`.

Ahora tenemos que configurar de nuevo nuestro `webpack.config` para hacer uso de los loaders nuevos, para eso vamos a agregar entradas en el arreglo `loaders`:

```
module.exports = {
  entry: [
    './index.js'
  ],
  module: {
    loaders: [
      {
        test: /\.js|\.jsx$/,
        loader: 'babel-loader',
        exclude: /node_modules/,
        query: { presets: ['es2015', 'react'] }
      },
      {
        test: /\.css$/,
        loader: 'style-loader!css-loader'
      },
      {
        test: /\.woff(2)?(\.eot)?$/,
        loader: 'url-loader?limit=10000&minetype=application/font-woff'
      }
    ]
  }
}
```

```
        test: /\.(png|woff|woff2|eot|ttf|svg)$/,
        loader: 'url-loader?limit=100000'
    }
]
},
output: {
    filename: "index_bundle.js",
    path: __dirname + '/dist'
}
}
```

Hemos agregado **loaders** para tres casos distintos, cuando el archivo es **.css** vamos a usar primero el **style-loader** y luego **css-loader**. Esto requiere el archivo **.css** y luego lo inyecta a la página como un **stylesheet**.

Como dijimos, vamos a usar el **url-loader** para varios archivos que tienen que ver con fuentes como: **.woff**, **.ttf**, **.eot**, además de algunas imágenes que puede incluir el framework (**.png** o **.svg**).

Todavía no hemos usado el **script-loader**, porque por ahora sólo estamos incluyendo **Css**.

Por último nos falta agregar Bootstrap en nuestro proyecto, por lo tanto en nuestro **index.js** vamos a agregar la siguiente línea:

```
require('bootstrap/dist/css/bootstrap.css');
```

Básicamente, estamos *importando* el archivo **.css** de bootstrap que habíamos instalado con **npm**, y este es pasado por los loaders que hemos definido y así llega a nuestra página.

Genial! Ahora ya tenemos por cargado Bootstrap en nuestra App, podemos verlo en los estilos de los Headers y Links!

Ahora agregemos la clase **btn btn-default** a cada Link de nuestra Nav y vemos cómo queda:

```
<IndexLink className="btn btn-default" to="/" activeClassName="active">
  Home</IndexLink>
<Link className="btn btn-default" to="/about" activeClassName="active">
  Componente2</Link>
<Link className="btn btn-default" to="/ejemplos" activeClassName="active">
  Componente3</Link>
```

Ahora, vamos a probar agregar una NavBar de Bootstrap, podemos copiarla [acá](#).

Recuerden cambiar los **class** por **className**, y los comentarios de HTML tampoco funcionan en JSX.

Ahora, si abrimos la página vamos a ver que se ve cómo debería, pero por ejemplo, el dropdown no funciona. Esto se debe a que el NavBar de Bootstrap utiliza una librería propia de JS y además jQuery, y nosotros todavía no lo hemos incluido ninguna.

Veamos como hacerlo.

Por empezar debemos incluir el archivo `.js` en nuestro `index.js`:

```
require('script-loader!jquery/dist/jquery.min.js');
require('bootstrap/dist/css/bootstrap.css');
require('script-loader!bootstrap/dist/js/bootstrap.min.js');
```

Ahora sí estamos usando el `script-loader`, cuando cargamos jQuery. En este caso lo usamos, porque necesitamos que jQuery este accesible de manera global, así el `js` de bootstrap pueda acceder a él.

También existen otras formas de hacer lo mismo con webpack, pero indicando qué cosas necesitamos en el `webpack.config.js`. Para probarlo vamos a comentar las líneas de los requires:

```
//require('script!jquery/dist/jquery.min.js');
//require('bootstrap/dist/css/bootstrap.css');
//require('bootstrap/dist/js/bootstrap.min.js');
```

Y vamos a agregar los siguientes a nuestro `webpack.config.js`:

```
module.exports = {
  entry: [
    'script-loader!jquery/dist/jquery.min.js',
    'bootstrap/dist/js/bootstrap.min.js',
    'bootstrap/dist/css/bootstrap.css',
    './index.js'
  ],
  externals: {
    jquery: 'jQuery'
  },
  module: {
    loaders: [
      {
        test: /(\.js|\.jsx)$/,
        loader: 'babel-loader',
        exclude: /node_modules/,
        query: { presets: ['es2015', 'react'] }
      },
      {
        test: /\.css$/,
        loader: 'style-loader!css-loader'
      },
      {
        test: /\.(png|woff|woff2|eot|ttf|svg)$/,
        loader: 'url-loader?limit=100000'
      }
    ]
  },
};
```

```
output: {
  filename: "index_bundle.js",
  path: __dirname + '/dist'
}
}
```

Lo que hicimos fue decirle a Webpack que nos incluya en el **bundle** a los archivos necesarios, lo hicimos agregando cada uno de ellos en el arreglo **entry**. Noten que a jquery **.js** tuvimos que avisarle que utilize el loader **script**, ya que necesitamos que sea *global*.

Webpack es genial, pero también es muy complejo. A veces se hace difícil pensar que está haciendo por atrás. Además hay miles de formas de hacer lo mismo, lo que no lo hace más sencillo.

Nuestros propios estilos

Con *Webpack* encontramos la forma de *importar* estilos. Veamos que sucede cuando queremos importar nuestro propio **.css**. Primero comenzemos definiendo un archivo **estilos.css** simple, en una carpeta separa, por ejemplo **styles**.

```
.prueba {
  background-color: red;
}
```

Bien, ahora vamos a ir a nuestro Componente **Home** y vamos a importarlo y usar la clase prueba en un **div**:

```
var React = require('react');
var Link = require('react-router').Link;

require('../styles/estilos.css');

module.exports = React.createClass({
  render: function(){
    return (
      <div className=' prueba '>
        Hola, Henry!!
      </div>
    )
  }
});
```

Antes de mostrar el resultado, vamos a ir a otro Componente, por ejemplo **About** y usar la misma clase 'prueba', pero sin *importar* la hoja de estilos:

```
var React = require('react');
var Link = require('react-router').Link;
```

```
module.exports = React.createClass({
  render: function(){
    return (
      <div>
        <h1>About</h1>
      </div>
    )
  }
});
```

Como vemos, el estilo fue importado de manera *global* y lo podemos usar en cualquier Componente. Esto para algunos puede ser bueno, pero para otros no. Veamos porqué sucedió esto y como utilizar otro patrón.

Esto ocurre, porque en nuestro `webpack.config.js` hemos definido que cada archivo `.css` que sea importado, pase por los loaders: `style-loader` y `css-loader`. El primero justamente lo que hace, es importar el `css` como si fuera una hoja de estilo convencional, por lo tanto le da el comportamiento que todos conocemos.

Ahora, si no quisieramos que esto ocurra, tendremos que usar otro approach. Vamos a usar uno conocido como `CSS-Modules`, básicamente lo que vamos a hacer es dejar que webpack haga un poco de *magia* usando los loaders que ya tenemos, y que cuando importemos el archivo CSS nos devuelva un objeto JS, con estilos listos para usar con react y con un *namespace* local:

```
{
  test: /\.ncss$/,
  loader: 'style-loader!css-loader?modules&importLoaders=1&localIdentName=[name]__[local]__[hash:base64:5]'
},
```

Primero vamos a agregar esta entrada en los loaders, vamos a tener que usar otra extensión (`.css` ya pasa por otro loaders distintos) para que los archivos que usemos como CSS Modules puedan ser identificados por Webpack, en este caso yo elegí `.ncss`. Si se fijan, la magia sucede en el string que le pasamos al `loader`.

Ahora, vamos a crear un archivo `estilos.ncss` con el mismo contenido que `estilos.css`. Y lo vamos a requerir en nuestro Componente `Home`:

```
var prueba = require('../styles/estilos.ncss').prueba;

module.exports = React.createClass({
  render: function(){
    console.log(prueba);
    return (
      <div className= {prueba}>
        Hola, Henry!!
      </div>
    )
  }
});
```

Como ven, requerimos el archivo, y el nombre de la clase como propiedad del mismo. Y luego utilizamos la variable donde lo guardamos como `className`.

Bien! Hemos logrado importar un archivo css de manera local. El Componente `About` sigue teniendo la clase `prueba`, pero no se activa con el CSS que hemos importado. Esto se debe a que Webpack le puso un hash al nombre de la clase para que sea único, en nuestro ejemplo la clase `.prueba` terminó llamándose `.estilos_prueba_2wKns`, emulando así un *namespace* local en CSS.

Múltiples clases

Si usamos este método para importar un archivo `.css` que contenga múltiples clases, vamos a poder acceder a ellas como *propiedades* del objeto en donde importes el `css`.

Por ejemplo, si este fuera el `css` que importamos:

```
// estilos.ncss
.prueba {
    background-color: red;
}

.title {
    color: blue;
}

.size {
    font-size: 25px;
}
```

Entonces podríamos usar las clases de la siguiente manera:

```
var s = require('../styles/estilos.ncss');

module.exports = React.createClass({
  render: function(){
    return (
      <div className= {s.prueba}>
        <h1 className={s.title}>Hola, Henry!!</h1>
        <p className{[s.title, s.size].join[' ']>}Prueba</p>
      </div>
    )
  }
});
```

Como vemos, si quisieramos que un mismo elemento tenga múltiples clases, podemos usar el siguiente *truco*:

```
[s.title, s.size].join[' ']
```

Esto funciona porque lo que hace el `style-loader` es darle un nombre único a cada clase del archivo `css`, y lo guarda en el objeto donde importamos bajo en una propiedad con el nombre original de la clase, y como valor el valor nuevo de la clase. Por ejemplo, si importamos `estilos.css` en el objeto `s` este sería algo así:

```
var s = require('../styles/estilos.css');

// s = {
//   prueba: '.estilos_prueba_2wKns',
//   title: '.estilos_title_3dsns';
//   size: '.estilos_size_7d8f8';
// }
```

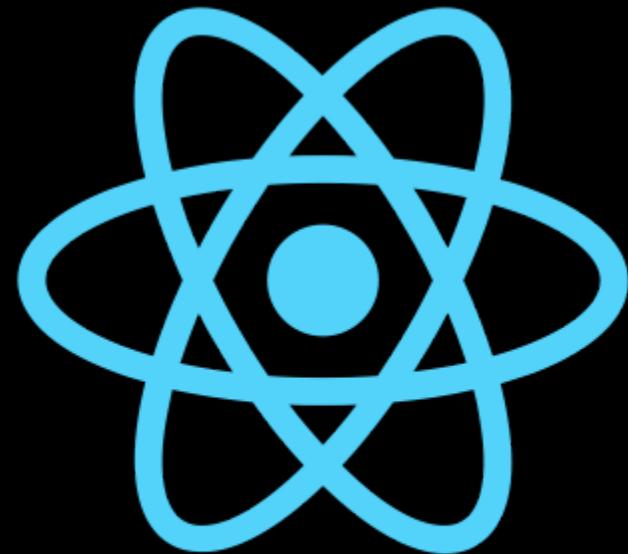
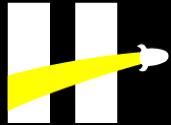
Esto es un ejemplo ilustrativo, probablemente el objeto `s` no debe ser exactamente así.

Sabiendo esto, si hacemos un `join` de un arreglo formado por cada clase que usemos, vamos a obtener un string con los nombres únicos de las clases concatenados.

Este cambio de filosofía es relativamente nuevo y todavía están surgiendo ideas nuevas y nuevas formas de hacer las cosas, así que por ahora está sucediendo lo mismo que en este comic:



Están apareciendo muchas formas distintas de incluir CSS en React, todavía no se puede decir cual es la mejor, todas tienen sus pros y sus contras. Así que hay que tener paciencia, probar varias y quedarse con la que más nos gusta. Lo importante es entender que está sucediendo por detrás.

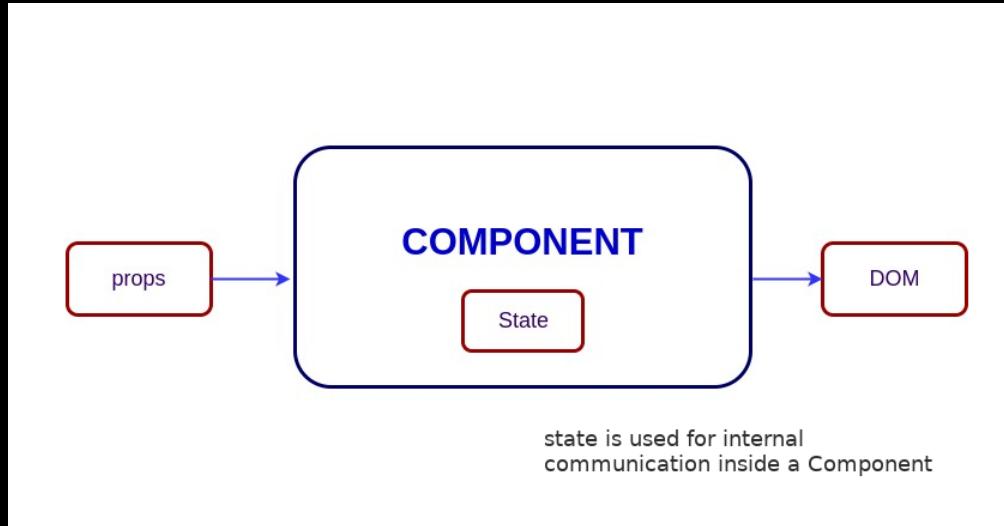


React

Estados y LifeCycles

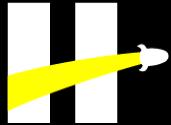


Estados



Las props son la **configuración inicial del componente**, son los datos con los que van a ser renderizados.

Pero la vida de un componente no termina ahí. De hecho, cada componente puede tener un **estado**. Podremos acceder al estado de cada componente a través del objeto en `this.state`.



Estados vs Variables



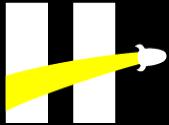
Los componentes solo van a volver a renderizarse cuando se produzcan cambios en el estado del componente o en las propiedades recibidas



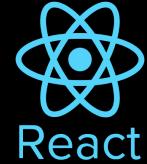
- En componentes de clase es posible invocar al método forceUpdate pero NO se recomienda su uso

<Demo indexState/>

<Demo indexStatevsVar />



Estados



setState no siempre actualiza inmediatamente, por lo que habría que evitar leer this.state luego de haber utilizado this.setState

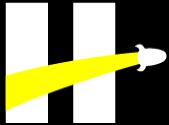
Posibles soluciones:

- Utilizar el lifecycle componentDidUpdate
- Agregar una función de callback al setState



```
1  ...
2
3  handleChange(event) {
4      this.setState({ username: event.target.username }, function() {
5          this.validateUsername();
6      });
7  }
```

<Demo setStateAsync/>



Estados



```
1 class Children extends React.Component {  
2   constructor(props) {  
3     super(props);  
4     this.state = {  
5       title: this.props.title  
6     };  
7   }  
8  
9   render(){  
10    return (  
11      <div>  
12        <h2>{this.state.title}</h2>  
13        </div>  
14      )  
15    }  
16  };
```



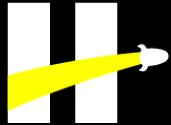
<Demo copyState/>

Bind con Arrow Function

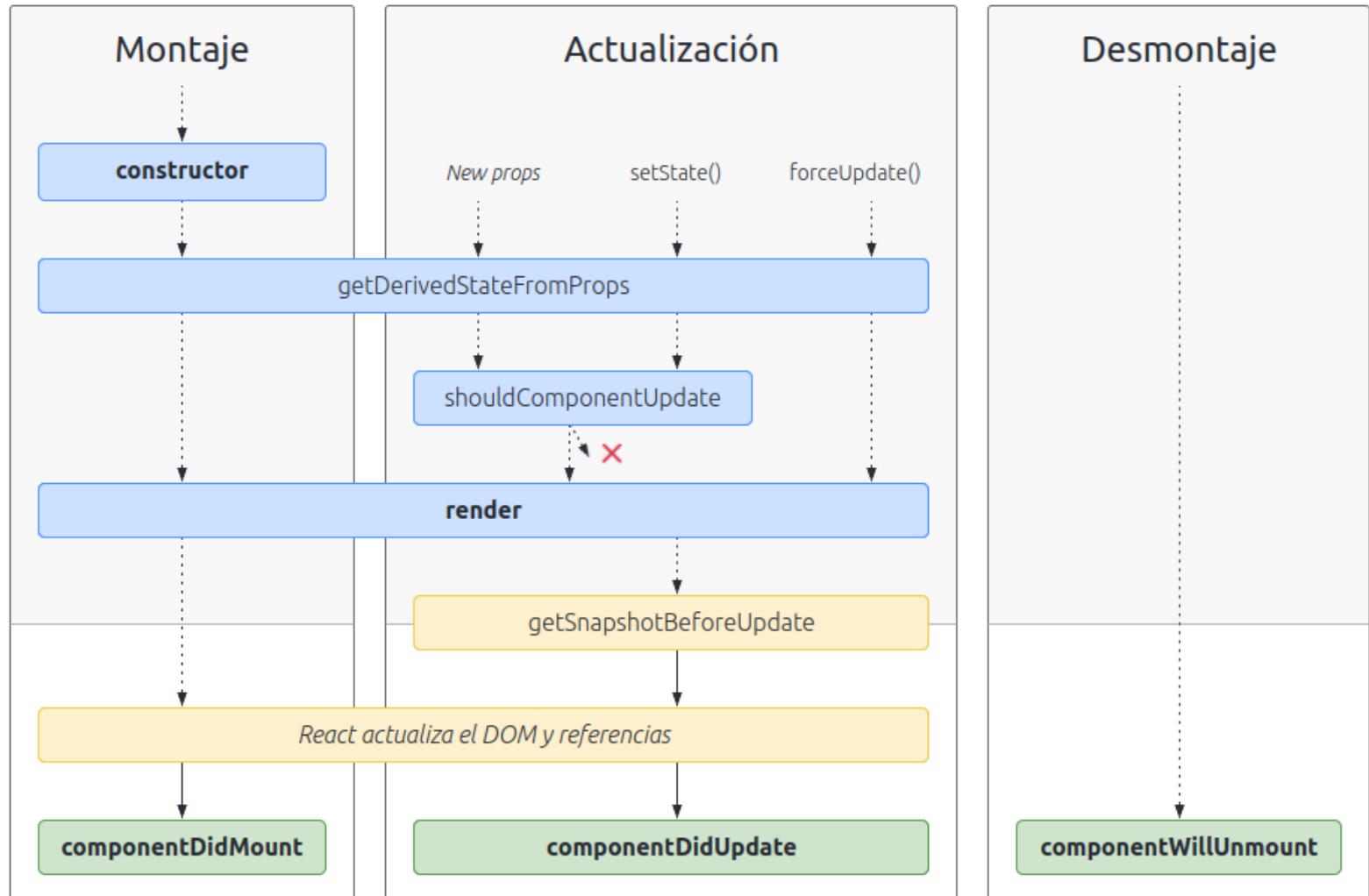


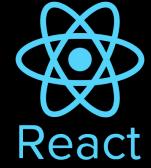
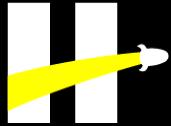
```
1 class BindArrowFunction extends React.Component {
2     ...
3     this.state = {
4         mensaje: this.props.mensaje
5     }
6 }
7 handleChange = (e) => {
8     this.setState({
9         mensaje: e.target.value
10    })
11 }
12 render() {
13     return (
14         <div>
15             <input
16                 type="text"
17                 name="message"
18                 placeholder="Ingresa mensaje"
19                 onChange={this.handleChange}
20             />
21             ...
22         </div>
23     );
24 }
25 }
```

<Demo demoArrowFunctionInClass/>

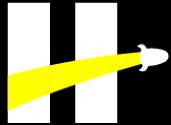


Ciclo de Vida



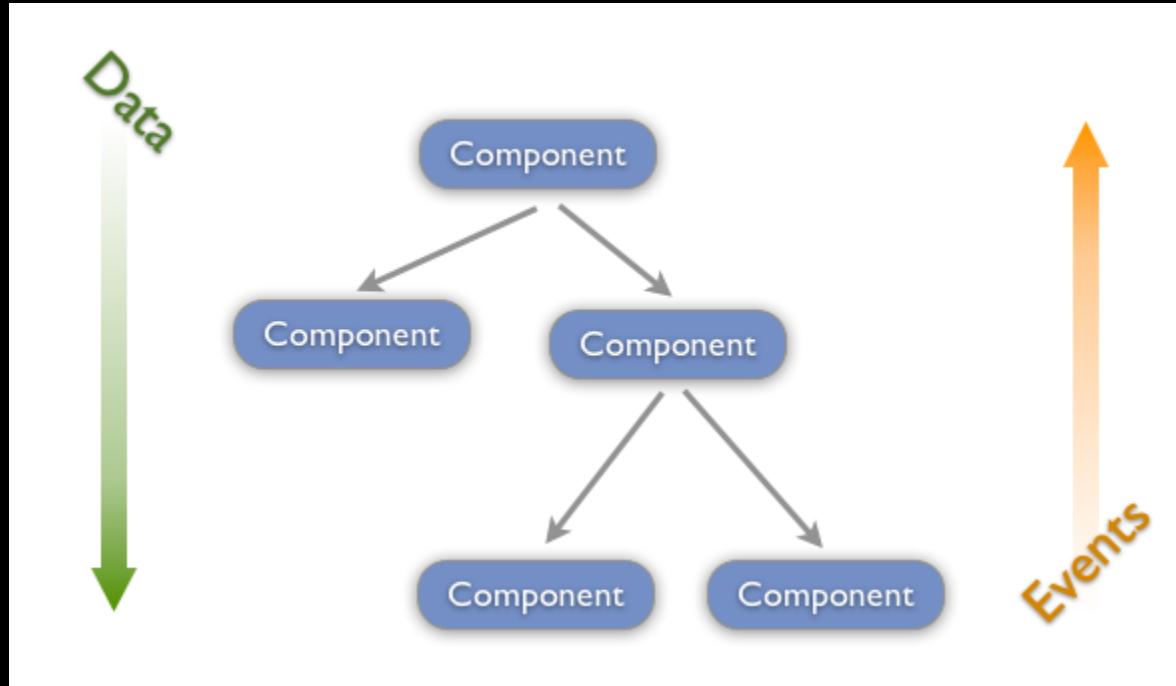
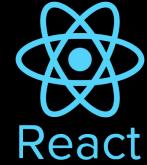


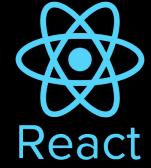
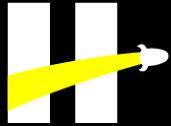
<Demo />



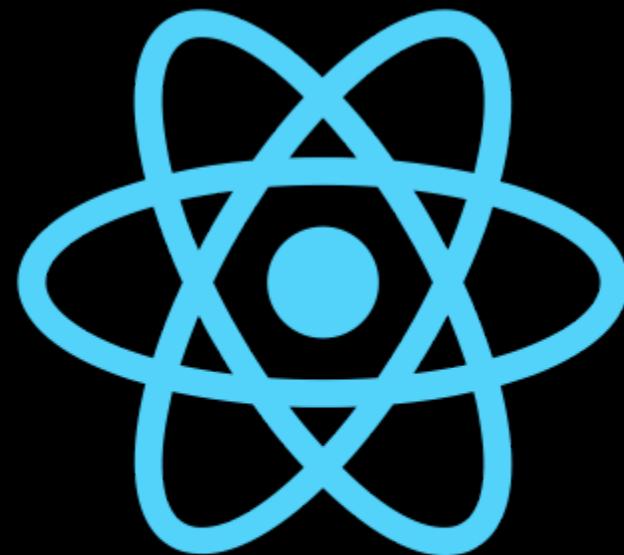
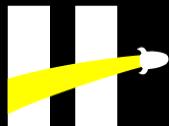
One Way Data Flow

Components y Containers



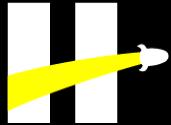


<Demo />

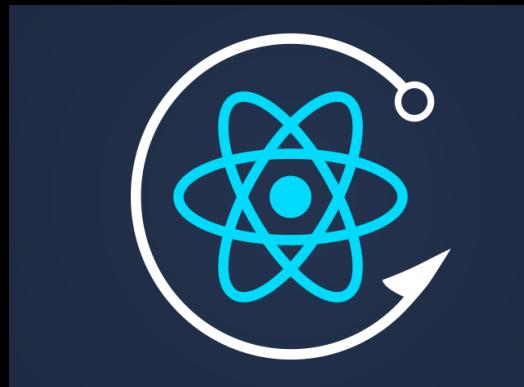


React

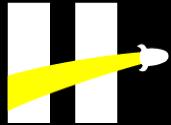
Hooks



Hooks



Los Hooks son una nueva API de la librería de React que nos permite tener estado, y otras características de React, en los componentes creados con una function. Esto, antes, no era posible y nos obligaba a crear un componente con class.

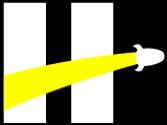


Hooks

useState

Devuelve un valor con estado y una función para actualizarlo.

```
1  
2  
3 const [state, setState] = useState(initialState);
```



Hooks

```
1 const { useState } = React;
2
3 function MuestraCuenta(props) {
4   return (
5     <p>Hola, van {props.contador}!!</p>
6   );
7 }
8
9 function Contador(props) {
10   const [contador, setContador] = useState(props.contador);
11
12   const onButtonClick = () => {
13     setContador(contador +1)
14   }
15
16   return (
17     <div>
18       <button onClick={onButtonClick}>Suma uno!</button>
19       <MuestraCuenta contador={contador} />
20     </div>
21   );
22 }
```

<https://es.reactjs.org/docs/hooks-reference.html>

Henry



Hacé click acá para dejar tu feedback sobre esta clase.



Hacé click acá completar el quiz teórico de esta lecture.

REACT

La mejor manera: Webpack

Cuando empiezas a trabajar en un proyecto la manera anterior de incluir React no es la mejor, de hecho no puede escalar. Cuando tenemos muchas líneas de código en un sólo archivo, o muchos archivos chicos de JS, el hecho de tener que juntar todo se empieza a complicar. Por suerte, desde hace varios años existen herramientas que nos van a automatizar este proceso, haciendo que todo el workflow sea óptimo y sobre todo mantenable.

Algunas tareas de las que se encargan estos gestores de proceso pueden ser:

- Juntar código de varios archivos en uno sólo.
- Transpilar código. Por ejemplo, de CoffeeScript, o TypeScript a Javascript.
- Minificar código.
- Concatenar archivos.
- Correr los tests automáticamente.
- etc...

Existen varios gestores de procesos buenos y populares, los más usados son: [Grunt](#), [Gulp!](#) y [Webpack](#). Nosotros vamos a usar [Webpack](#), pero podrían hacer lo mismo con los otros.

Ahora también se utiliza el concepto de **módulos** en el frontend, para lograrlo se utilizan librerías como [Browserify](#) o [CommonJS](#). Básicamente en vez de incluir librerías usando HTML, lo hacemos en el mismo JS, después estas librerías que mencionamos se encarga de incluir realmente el código necesario para que funcione.

Un ejemplo en el frontend se vería algo así:

```
//algunModulo.js
module.exports.doSomething = function() {
  return 'foo';
};
//algunOtroModulo.js
const someModule = require('someModule');
```

```
module.exports.doSomething = function() {
  return someModule.doSomething() + 'bar';
};
```

Vamos a empezar instalando y configurando Webpack. Voy a aclarar al principio que Webpack es una herramienta muy poderosa, por ende compleja, y lamentablemente su documentación no es la mejor. Por lo tanto, nos va a parecer complejo al principio, pero rápidamente nos vamos a encariñar con todas las cosas que podemos hacer con Webpack.

```
$ npm i -D webpack webpack-cli
```

Como dijimos, Webpack es una herramienta que va a aplicar ciertas *transformaciones* a nuestro código, por ende para funcionar webpack necesita saber:

1. Conocer el starting point de nuestra app, o el archivo javascript raíz.
2. Debe saber qué transformaciones tiene que hacer al código.
3. Tiene que saber dónde guardar el nuevo código transformado.

Todo esta información va a estar contenida en un archivo de configuración llamado `webpack.config.js`, que deberíamos crear en la raíz del directorio de nuestro proyecto. Este archivo va a ser en realidad un módulo, que va a exportar un objeto con las configuraciones de webpack, así que podríamos empezar escribiendo lo siguiente en ese archivo:

```
// dentro de webpack.config.js
module.exports = {}
```

Ahora empezemos a agregar la información que mencionamos antes, empezemos por el punto 1 : el entry point.

```
module.exports = {
  entry: [
    './app/index.js'
  ]
}
```

Como ven, los entry points se definen dentro del objeto que exportamos bajo el nombre `entry`, y cuyo valor es un arreglo. Dentro de este explicito los paths de todos los archivos que sirvan como entry points de nuestra app. Por ahora vamos a escribir sólo uno.

Bien, ahora para el segundo punto, tenemos que definir qué tipo de transformaciones vamos a hacer, para esto entran en juego los `loaders`, estos son los módulos encargados de realizar transformaciones, existen varios tipos de `loaders`, ya que la comunidad va creando nuevos a medida que surgen nuevas necesidades.

Para usar un loader, es necesario tenerlo instalado antes. Para eso vamos a usar `npm`. Por ejemplo, si quiero usar el `loader` de babel debería hacer: `npm i -D @babel/core @babel/preset-env @babel/preset-react babel-loader`.

```
module.exports = {
  entry: [
    './app/index.js'
  ],
  module: {
    loaders: [
      {test: /\.coffee$/, exclude: /node_modules/, loader: "coffee-loader"}
    ]
  },
}
```

En el ejemplo, usamos un `loader` de *coffeeScript*. Como se ve, también agreamos una propiedad llamada `module` que será un objeto dentro del cual aparecerá la propiedad `loaders` que será un arreglo de objetos. Cada objeto dentro de este arreglo representa una transformacion. Vemos que ese objeto tiene tres propiedades: `test`, `exclude`, y `loader`. La primera hace referencia a qué archivos deberán pasar por la transformación, y recibe como valor una **expresión regular**, en nuestro ejemplo estamos diciendo que pasarán por la transformación todos los archivos terminados en `.coffee`. La segunda, `exclude` le indica a webpack qué directorios excluir, en nuestro ejemplo (y siempre lo haremos) excluimos `node_modules`, donde sabemos que no habrá código para transformar. Finalmente, en la propiedad `loader` vamos a poner el nombre del loader que queremos usar, en este caso el nombre es "coffee-loader".

Siempre busquen los loaders que necesiten dentro del ecosistema npm.

Por último, vamos a agregar donde queremos que webpack deposite los archivos luego de la transformación:

```
module.exports = {
  entry: [
    './app/index.js'
  ],
  module: {
    loaders: [
      {test: /\.coffee$/, exclude: /node_modules/, loader: "coffee-loader"}
    ]
  },
  output: {
    path: __dirname + '/dist',
    filename: 'bundle.js'
  },
}
```

Ya podrán imaginarse que quieren decir las cosas que hemos agregado ahora. Por empezar una nueva propiedad `output` que contiene un objeto, en este último vamos a especificar el nombre del archivo de salida (`filename`) y la carpeta donde queremos que se guarde (`path`).

Bien, entonces intentemos reproducir el mismo ejemplo que hicimos en el HTML, pero ahora usando este proceso. Por lo tanto, lo primero que hacemos es sacar el código escrito en React que estaba embebido en el HTML y lo pasamos a un archivo `js`. El primer cambio que tenemos que hacer es importar los módulos `react` y `react-dom` que antes requeríamos a través del tag `script`:

```
import React from 'react';
import ReactDOM from 'react-dom';

class HelloWorld extends React.Component {
  render(){
    return (
      <div>
        Hola, Soy Henry!!
      </div>
    )
  }
};

function HelloWorldFunction() {
  return(
    <div>
      Hola, Soy Henry!
    </div>
  )
};
ReactDOM.render(<HelloWorld />, document.getElementById('app'));
```

Genial, ahora tenemos que construir el archivos de configuración de webpack, para que funcione con `babel`. Básicamente tenemos que transformar el código que usa EcmaScript6 y JSX a JS plano.

```
// webpack.config.js

module.exports = {
  entry: [
    './app/index.js'
  ],
  output: {
    path: __dirname + '/dist',
    filename: 'bundle.js'
  },
  module: {
    rules: [
      {
        test: /\.(js|jsx)$/,
        exclude: /node_modules/,
        use: {
          loader: 'babel-loader',
          options: {
            presets: ['@babel/preset-react', '@babel/preset-env']
          }
        }
      }
    ]
  }
};
```

```
        }
    }
}
}
```

Por último vamos a tener que usar `npm` para instalar las dependencias:

```
$ npm install -D @babel/core @babel/preset-env @babel/preset-react babel-loader
```

```
$ npm install react react-dom --save
```

Para poder ejecutar webpack, debemos agregar dentro de `scripts` en nuestro `package.json` lo siguiente:

```
"scripts": {
  "build": "webpack -w",
}
"devDependencies": {
  "@babel/core": "^7.9.0",
  "@babel/preset-env": "^7.9.0",
  "@babel/preset-react": "^7.9.4",
  "babel-loader": "^8.1.0",
  "webpack": "^4.42.1",
  "webpack-cli": "^3.3.11"
},
"dependencies": {
  "react": "^16.13.1",
  "react-dom": "^16.13.1"
}
```

Para probar si todo funciona bien, iremos a la carpeta donde tenemos definidos todos estos archivos, y vamos a escribir `npm run build`.

```
[atralice@arch-laptop webpack]$ npm run build
> webpackDEMO@1.0.0 build /home/atralice/Projects/henry/caronte/M2/06-React/1-Intro/demo/webpack
> webpack -w

webpack is watching the files...

Hash: 05ea77c667adee404f91
Version: webpack 4.42.1
Time: 1673ms
Built at: 04/05/2020 6:46:57 PM
  Asset      Size  Chunks      Chunk Names
bundle.js  131 KiB     0  [emitted]  main
Entrypoint main = bundle.js
[7] ./app.js + 3 modules 8.53 KiB {0} [built]
  | ./app.js 940 bytes [built]
  | ./src/containers/Main.jsx 3.54 KiB [built]
  |   + 2 hidden modules
  + 7 hidden modules
```

Si todo funcionó bien, veremos un mensaje como el de la imagen! Y además encontraremos un archivo nuevo en la carpeta **dist**.

Bien, ahora por último tenemos que agregar ese archivo generado a un HTML para poder correrlo:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Descubre React</title>
  </head>
  <body>
    <div id="app"></div>
    <script src="./dist/index_bundle.js"></script>
  </body>
</html>
```

Si abren este archivo van a ver que vemos exactamente lo mismo que en el primer archivo HTML que creamos. La ventaja de esta forma, es que tenemos separado el código JS de todo lo demás, podemos tener múltiples archivos y Webpack se encargará de unirlos y depositarlos en el archivo de salida.

Bien, ahora que lo hicimos funcionar y más o menos vimos cómo se escribe, aprendamos un poco más sobre React.

Introducción a Hooks

Los Hooks en React fueron introducidos en su versión 16.8, nos permiten el uso de estados y otras características sin el uso de clases. Estas además de dificultar la reutilización y organización del código, pueden ser una gran barrera para el aprendizaje de React. Se tiene que entender como funciona el 'this' en JavaScript, que es muy diferente a cómo funciona en la mayoría de los lenguajes. Agregar bind a tus manejadores de eventos hace que, por lo general, el código sea muy verboso. Para resolver estos problemas, Hooks te permiten usar más de las funciones de React sin clases. Conceptualmente, los componentes de React siempre han estado más cerca de las funciones. Los Hooks abarcan funciones, pero sin sacrificar el espíritu práctico de React.

Estado de un Componente

Dijimos que las *props* era la *configuración inicial* del Componente y que no se pueden cambiar, que son *inmutables*. Van a existir muchos casos donde un Componente mantenga adentro suyo algún dato que pueda cambiar con el tiempo, como por ejemplo lo que queremos hacer ahora de cambiar el nombre del saludo. Para hacer esto, cada Componente es capaz de mantener datos guardados en lo que React llama el *Estado* de un Componente. Un Componente por lo tanto va a poder actualizar su propio *Estado* sin restricciones. Para empezar, comenzemos dandole un *Estado* inicial al Componente, lo hacemos definiendo dentro del constructor de la clase el **this.state** que va a ser un objeto con todas las propiedades que queramos almacenar como estado interno. Cada propiedad puede almacenar cualquier tipo de dato que quedamos. Para nuestro ejemplo, vamos a darle como *Estado* inicial un objeto que contenga la propiedad **name** y que sea igual a **this.props.name**, o sea que vamos a hacer que una *prop* sea un *estado*, esto es así porque sabemos que va a cambiar en el futuro.

Genial, ya tenemos *Estado*, ahora lo único que nos falta es *cambiar* el *Estado* cuando el usuario haga click. Para hacerlo vamos a tener que usar una función llamada `setState`. No podemos simplemente asignarle un valor nuevo a `this.state.name`, esto es así por matener la arquitectura del Virtual DOM de react. Veamos cómo quedaría el ejemplo:

```
import React from 'react';
import ReactDOM from 'react-dom';

class HelloWorld extends React.Component {
  constructor(props) {
    super(props);
    this.state = {name: this.props.name}
    this.onButtonClick = this.onButtonClick.bind(this);
  }
  onButtonClick(e){
    e.preventDefault();
    const name = this.refs.name.value;
    this.setState({
      name: name
    });
  }
  render(){
    return (
      <div>
        <form onSubmit={this.onButtonClick}>
          <input type='text' ref='name' />
          <button>Poner Nombre</button>
        </form>
        Hola, {this.state.name}!!
      </div>
    )
  }
}
ReactDOM.render(<HelloWorld name='Soy Henry' />, document.getElementById('app'));
```

Como vemos, ahora tenemos un Componente que maneja un *Estado* interno, que se inicializa usando una *prop* y que está pensado en cambiar en el futuro. Siguiendo el ejemplo usando un componente de función:

```
import React, { useState, useRef } from 'react';
import ReactDOM from 'react-dom';

function HelloWorld(props) {
  const nameState = useState(props.name)
  const textInput = useRef(null);

  const onButtonClick = (e) => {
    e.preventDefault();
    const name = textInput.current.value
    nameState[1](name)
  }
}
```

```

return (
  <div>
    <form onSubmit={onButtonClick}>
      <input type='text' ref={textInput} />
      <button>Poner Nombre</button>
    </form>
    Hola, {nameState[0]}!!
  </div>
)
};

ReactDOM.render(<HelloWorld name='Soy Henry' />, document.getElementById('app'));

```

Aca vemos el uso de el Hook `useState`. Lo llamamos dentro de un componente funcional para agregarle un estado local. React mantendrá este estado entre re-renderizados. `useState` devuelve un array con 2 elementos: el valor de estado actual y una función que le permite actualizarlo. Puedes llamar a esta función desde un controlador de eventos o desde otro lugar. Es similar a `this.setState` en una clase, excepto que no combina el estado antiguo y el nuevo. El único argumento para `useState` es el estado inicial. En el ejemplo anterior, es `'props.name'`. Ten en cuenta que a diferencia de `this.state`, el estado aquí no tiene que ser un objeto — aunque puede serlo si quisieras. El argumento de estado inicial solo se usa durante el primer renderizado. Ahora veremos el mismo ejemplo escribiendo `useState` de una forma mas entendible, usando `array destructuring`. Para darle el primer valor del array nuestro state inicial y al segundo valor sera nuestra función para actualizar el estado.

```

import React, { useState, useRef } from 'react';
import ReactDOM from 'react-dom';

function HelloWorld(props) {
  const [name, setName] = useState(props.name)
  const textInput = useRef(null);

  const onButtonClick = (e) => {
    e.preventDefault();
    const name = textInput.current.value
    setName(name)
  }

  return (
    <div>
      <form onSubmit={onButtonClick}>
        <input type='text' ref={textInput} />
        <button>Poner Nombre</button>
      </form>
      Hola, {name}!!
    </div>
  )
};

ReactDOM.render(<HelloWorld name='Soy Henry' />, document.getElementById('app'));

```

Anidando Componentes

Como dijimos antes, en React *todo* es un componente, por lo tanto es lógico pensar que vamos a tener *componentes dentro de componentes* todo el tiempo. Veamos con un ejemplo como funciona esto, y como se le pueden pasar datos (en React le decimos *props*) a los componentes.

Ahora armemos un ejemplo un poco más complejo, en este vamos a tener dos componentes. Uno va a llamar al otro y le va a pasar algunas propiedades, veamos como hacerlo:

```
class ContenedorAmigos extends React.Component {
  render(){
    const name = 'Soy Henry';
    const amigos = ['Toni', 'Franco', 'Emi', 'Solano'];
    return (
      <div>
        <h3> Nombre: {name} </h3>
        <MostrarLista names={amigos} />
      </div>
    );
  }
};
```

En este componente hemos definido un método `render` un poco más complejo, en el tenemos dos variables (`name` y `amigos`) y retornamos un XML que utiliza estas dos variables. Como ven, podemos acceder a un *prop* del mismo Componente usando los `{}`, de esta forma `{name}` va a ser reemplazado por `Soy Henry`. Luego llamamos a un componente que todavía no hemos definido con el nombre de `mostrarLista` y le pasamos como propiedad el arreglo `amigos`. Por lo tanto dentro de `mostrarLista` vamos a disponer de ese arreglo como una *prop*.

Definamos el elemento hijo o *child*:

```
class MostrarLista extends React.Component {
  render(){
    const lista = this.props.names.map(amigo => <li> {amigo} </li>);
    return (
      <div>
        <h3> Amigos </h3>
        <ul>
          {lista}
        </ul>
      </div>
    )
  }
};
```

Lo primero que notamos es que usamos *JS* para crear elementos HTML más complejos. En este caso usamos la función `map`, para crear un elemento `` por cada *amigo* en la lista o arreglo. Viendo el ejemplo anterior usando funciones:

```

function ContenedorAmigos() {
  const name = 'Soy Henry';
  const amigos = ['Santi', 'Guille', 'Facu', 'Solano'];
  return (
    <div>
      <h3> Nombre: {name} </h3>
      <MostrarLista names={amigos} />
    </div>
  )
};

function MostrarLista({ names }) {
  const lista = names.map(amigo => <li> {amigo} </li>);
  return (
    <div>
      <h3> Amigos </h3>
      <ul>
        {lista}
      </ul>
    </div>
  )
};

```

Aca podemos usar **destructuring** para pasar las props directamente con el nombre de la variable **names**

Por si no se acuerdan como funciona map:

```

const amigos = ['Santi', 'Guille', 'Facu', 'Solano'];
const lista = amigos.map(amigo => "<li> " + amigo + "</li>");
console.log(lista); //['<li> Santi</li>', '<li> Guille</li>', '<li> Facu</li>', '<li> Solano</li>'];

```

Justamente ese nuevo conjunto de **lis** que hemos creado, lo vamos a usar envuelto en tags **** para formar la lista de amigos.

Etiquetas HTML y Componentes

Los componentes que creamos en React despues los usamos escribiendolos como un tag HTML, en realidad es un tag **XML**. Por ejemplo: el tag **MostrarLista** es un Componente que creamos antes y lo usamos así:

```

<div>
  <h3> Nombre: {name} </h3>
  <MostrarLista names={amigos} />
</div>

```

Luego ese tag se renderizará a lo que sea que hayamos escrito en el método **render** de ese componente, transformandose así en HTML finalmente. Existe una convención en React para distinguir entre Componentes

React y elemento HTML nativos. Para el primero usamos BumpyCase y lowercase para el último. Por ejemplo:

```
<MostrarLista /> BumpyCase  
<div>           lowercase
```

Separando Componentes

Seguro estarán pensando que si tenemos Componentes, lo mejor sería poder tenerlos también en archivos y carpetas distintas. Lo bueno de React es que es TODO JavaScript, así que vamos a poder usar [CommonJS](#) para exportar Componentes como módulos y luego requerirlos:

```
import React from 'react';

export class Lista extends React.Component {
  render(){
    const lista = this.props.names.map(amigo => <li> {amigo} </li>);
    return (
      <div>
        <h3> Amigos </h3>
        <ul>
          {lista}
        </ul>
      </div>
    )
  }
};
```

Luego en el archivo donde lo necesitemos simplemente lo requerimos y lo empezamos a usar:

```
import { Lista } from './MostrarLista.js';
```

En este caso usamos las llaves en '{ Lista }' porque le dimos nombre a nuestro export:

```
export class Lista extends React.Component
```

De haber sido un export default podemos hacer un import simple porque le indica que es lo único que se importará.

```
// Presentational
export default class Lista extends React.Component

// Container
import Lista from './MostrarLista.js';
```

De esta forma vamos a poder organizar muy bien nuestros componentes en distintos archivos y carpetas.

React y Funciones Puras

Como vimos recién vamos a poder usar todo lo que sabemos de JS para codear con React. Pensemoslo así, en vez de tener *funciones* que tomen argumentos y retornen valores y objetos, en React vamos a tener *funciones* que tomen argumentos y retornen **UI (user interfaces)**. Podemos resumir este concepto en $f(d) = V$, es decir, una función toma d argumentos y retorna una **View**. Esta es una buena forma de desarrollar interfaces, porque ahora toda tu interfaz está compuesta de invocaciones a funciones, que es la forma en que estamos acostumbrados a programar nuestras aplicaciones. Veamos este concepto en código:

```
const getFoto = function(username) {
  return 'https://photo.fb.com/' + username
}
const getLink = function(username) {
  return 'https://www.fb.com/' + username
}
const getData = function(username) {
  return {
    foto: getFoto(username),
    link: getLink(username)
  }
}
getData('atralice')
```

Si vemos el código de arriba, notamos que tenemos tres funciones y una invocación a una función. De esta forma logramos que el código sea modular y entendible. Cada función tiene un propósito específico y luego las componemos en otra función en donde generaremos un comportamiento que utiliza cada una de estas para lograr el comportamiento que deseamos.

Bien, ahora modifiquemos ese código para que devuelvan un *UI* en vez de sólo datos:

```
import React from 'React';

class Foto extends React.Component {
  render() {
    return (
      <img src={'https://photo.fb.com/' + this.props.username} />
    )
  }
};

class Link extends React.Component {
  render() {
    return (
      <a href={'https://www.fb.com/' + this.props.username}>
        {this.props.username}
      </a>
    )
  }
};
```

```
        )
    }
};

class Avatar extends React.Component {
  render(username) {
    return (
      <div>
        <Foto username={this.props.username}/>
        <Link username={this.props.username}/>
      </div>
    )
  }
};

<Avatar username='atralice' />
```

Ahora, en vez de crear componer funciones que retornen datos, estamos componiendo funciones que retornan *UIs*. Esta idea es tan importante que React en la versión **0.14** introdujo lo que se conoce como **Stateless Functional Components**, que nos permite escribir el código de arriba como simples funciones.

Lee [acá](#) que tienen de bueno las **Stateless Functional Components**.

Reescribamos nuestro código usando **Stateless Functional Components**:

```
import React from 'React';

const Foto = function(props) {
  return <img src={'https://photo.fb.com/' + props.username} />
};

const Link = function(props) {
  return (
    <a href={'https://www.fb.com/' + props.username}>
      {props.username}
    </a>
  )
}

const Avatar = function(props) {
  return (
    <div>
      <Foto username={props.username}/>
      <Link username={props.username}/>
    </div>
  )
};

<Avatar username='atralice' />
```

Ahora cada componente es lo que llamamos una **pure function**. Este concepto viene de **Functional Programming**, básicamente, las funciones puras son consistente y predecible, porque tienen las siguientes características:

- Una función pura siempre retorna el mismo resultado para los mismos argumentos.
- La ejecución de una función pura **NO** depende del *estado* de la aplicación.
- Las funciones puras **NO** modifican el estado ni ninguna variable afuera de su scope.

En React el método `render` necesita ser una función pura, y por ende, todos los beneficios de programar funciones puras se transladan ahora a tu **UI**. De esta forma logramos tener lo que en React se conoce como: **Stateless Functional Components**. Si vemos le ejemplo, todos los Componentes que armamos no tiene estado, y que no hacen nada más que recibir datos a través de `props` y renderizar una **UI**, esto es, básicamente, Componentes que sólo tienen el método `render`. De esto nace un paradigma en el que se diferencian dos tipos de Componentes, los que acabamos de mencionar son los llamados **Presentational Components** y los segundos son **Containers Components**.

Como se pueden imaginar, los **Presentational Components** se preocupan en como **se ven las cosas** y los **Container Components** en **como funcionan las cosas**. Organizar nuestro código de esta forma trae varias ventajas:

- Mejor separación de temas. Vas a entender mejor tu aplicación y tu UI escribiendo Componentes de este modo.
- Mejor reusabilidad. Podes usar los mismos Presentational Componentes en distintos Containers.
- Podes tener a los diseñadores trabajando en los Presentational Componentes sin tener que meterse a la lógica de la aplicación.

Esto es sólo un paradigma, seguramente hay otros que tengan otras características. Si querés poder leer más de este paradigma [acá](#).

Organizando las carpetas de un Proyecto

Antes mencionamos el patrón de separar Componentes según mantengan *Estados* (**Containers**) o sólo sirvan para renderizar algo (**Presentational**). Vamos a organizar nuestra estructura de carpetas de proyecto alrededor de este.

Básicamente, vamos a guardar cada Componente en un archivo `.js` separado y *exportarlo*. Los *Presentational* van a ir en una carpeta llamada `components`, y los *Containers* en otra carpeta llamada `containers`. Como sabemos, los *Containers* van a incluir o *requerir* a los *Presentational* en su código, y desde código denuestra app vamos a *requerir* a los *Containers*.

Por convención vamos a llamar a los archivos que contengan un componente con la primera letra en Mayúsculas. Por ejemplo: `Header.jsx` o `Profile.jsx`.

Cuando comenzamos un proyecto nuevo de React, en vez de empezar de cero, podemos guardar un esqueleto que ya tenga todas las tareas que deberíamos repetir en cada proyecto, en inglés esto se conoce como **boilerplates project**. De hecho, podemos hacer nuestro propio **boilerplate** o buscar online alguna que se ajuste a nuestras necesidades. En general que están publicados online traen muchas cosas que tal vez no vayamos a usar, aquí algunos ejemplos:

- [react-webpack-boilerplate](#)

- [react-starter-kit](#)
- [react-native-starter](#)
- [react-webpack-boilerplate](#)

Hace poco salió [Create React APP](#) una mini app del equipo de Facebook que te ayuda a comenzar un proyecto nuevo de React en segundos (yo todavía no la probé pero parece interesante!).

Tambien pusimos nuestro propio ejemplo de un boilerPlate simplificado [acá](#). Básicamente trae un archivo de *webpack* preconfigurado y un mini servidor *express* para levantar el archivo desde la carpeta del output de *webpack*, super simple!

Como siempre, todo viene en muchos *sabores* y hay que probar y elegir el que mas le gusta a cada uno, ninguno es el mejor, todos van a tener pros y cons.

Propiedades y Estados

Ya vimos que en React las propiedades se pasan de componentes padres a hijos a través de la variable [props](#). Veamos algunas propiedades más avanzadas del comportamiento de [props](#).

Estados

La otra forma de que los Componentes de React tengan información es a través del [State](#). Este [Estado](#) se encuentra disponible en el objeto [State](#) de cada Componente, es decir que podemos acceder a el a través de [this.State](#). Los estados **no** son inmutables, es decir que estan pensados para *cambiar* eventualmente. Justamente por esto, es que los Componentes que tienen Estados son menos *performantes* que los que no.

Cuando creamos un Componente cualquiera, su estado o [this.State](#) es igual a [null](#), o sea que no tiene Estado por defecto!. Para agregar un estado vamos a usar el método [getInitialState](#), el cual retorna un objeto que contiene el estado inicial de ese Componente. Por ejemplo:

```
class Componente extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      nombre : "Toni",
      trabajo : "Profesor"
    };
  }
  render() {
    return (
      <div>
        Mi nombre es {this.state.nombre}
        y soy el {this.state.trabajo}.
      </div>
    )
  }
}
ReactDOM.render(<Componente />, document.getElementById('appEstado'));
```

En un componente de función la nueva forma de tener un **State** es a través del Hook **useState** que vimos anteriormente. Siguiendo con el ejemplo anterior:

```
function Componente(props) {
  const [nombre, setNombre] = useState("Toni");
  const [trabajo, setTrabajo] = useState("Profesor");

  return (
    <div>
      Mi nombre es {nombre}
      y soy el {trabajo}.
    </div>
  )
}
ReactDOM.render(<Componente />, document.getElementById('appEstado'));
```

Actualizando el Estado en una clase

Para actualizar o cambiar el estado de un Componente llamamos a la función **this.setState**, pasandole por parámetros un nuevo estado, es decir un objeto con las nuevas propiedades:

```
// Agregamos este método al código anterior
...

handleClick() {
  this.setState({
    nombre : "Guille"
  });
}
render() {
  return(
    <div onClick={this.handleClick}>
      Mi nombre es {this.state.nombre}
      , y soy el {this.state.trabajo}.
    </div>;
  )
}
```

Actualizando el Estado en una función

Para actualizar el **state** en un componente de función, como vimos, llamamos a la función que nos devuelve el Hook **useState**. Y le pasamos por parámetro el nuevo **estado**. Por ejemplo:

```
// Agregamos este método al código anterior
...

const handleClick = () => {
```

```
    setNombre("Guille");
}

return(
  <div onClick={handleClick}>
    Mi nombre es {nombre}
    , y soy el {trabajo}.
  </div>
)
}
```

Cuando se cambia el estado de un Componente usando `setState` o el Hook 'useState', React re-renderiza todo el Componente. Hay que intentar hacerlo lo menos posible!

Diferencia entre Estados y props

Seguramente se preguntarán cuál es la diferencia entre props y estados, en realidad las dos mantiene información o datos que va a usar el Componente. La diferencia es el uso de cada una. Las pros de un Componente van a ser pasadas por el padre del mismo y *deberían ser inmutables*, es decir, que si se cambian las props (o sea que el padre las cambia), el Componente se debería renderizar de nuevo con las nuevas props. Podría decir que las props son un tipo de *configuración* del Componente.

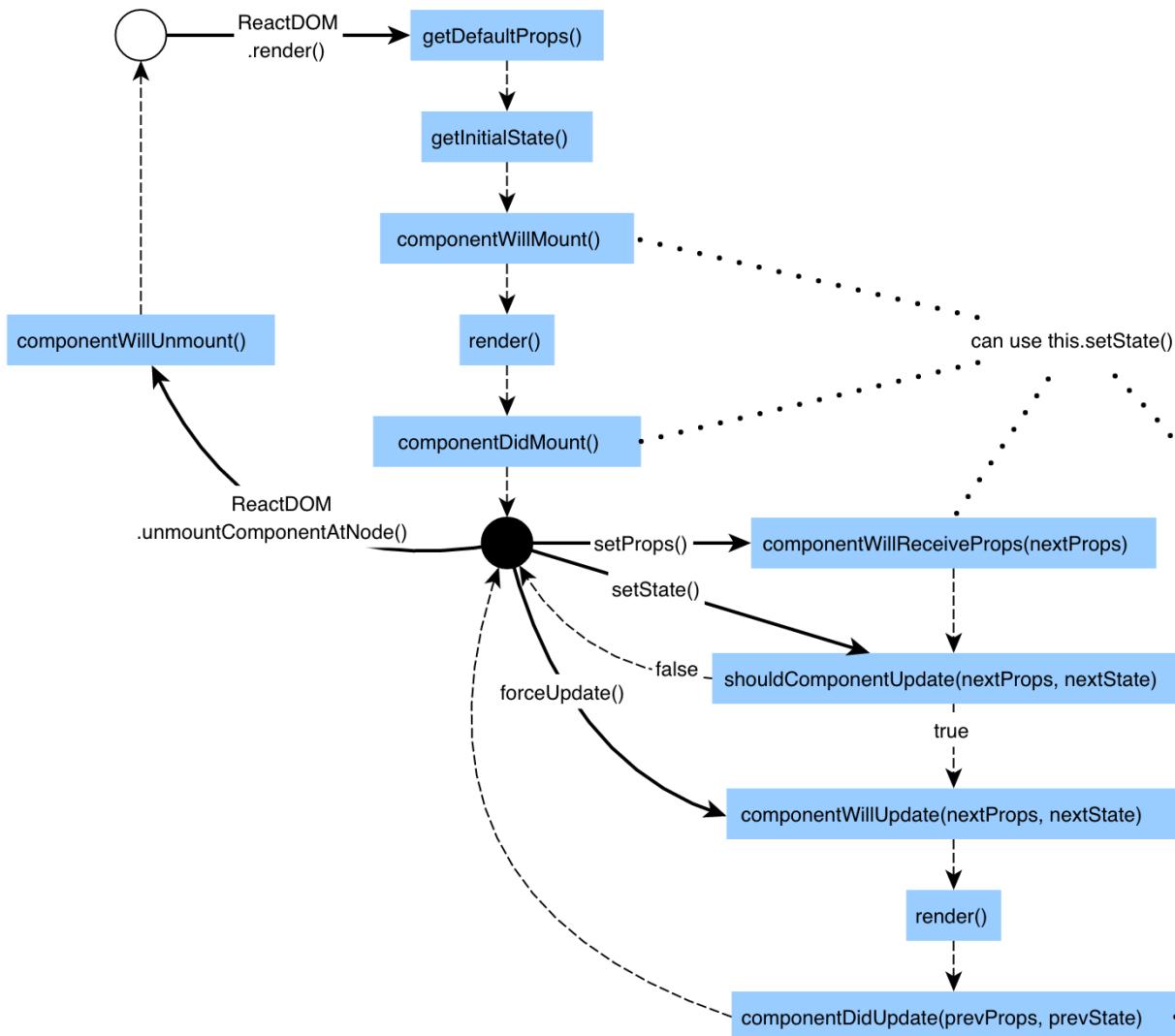
Los estados, encambio, son manejados exclusivamente e internamente por un Componente, nada tiene que ver con el estado de otros Componentes, es decir el estado es *privado*, además los estados **no** son *inmutables*!

React Life Cycle Events

El Life Cycle Completo

Como sabemos, es critico para una app tener estados, es decir poder hacer Ajax requests, etc... Dijimos que el método render de los Componentes tiene que ser *stateless*, es decir que en él no vamos a poder agregar este comportamiento. Para poder hacerlo vamos a incorporar el concepto de ciclo de vida de React y sus métodos. Con ellos, vamos a poder dar comportamiento a nuestros Componentes según sucedan ciertos eventos en nuestra app y en nuestros Componentes (por ejemplo, cuando se renderiza el Componente, o cuando le llegan datos nuevos, etc).

En la imagen de abajo, vemos el ciclo de vida Completo de cualquier Componente de React. En ella vemos también los estados en los que puede estar un Componente y qué cosas o funciones activarán el paso de estados y por ende la invocación de los métodos que nos provee React:



Pueden ver este [Gist](#) y probarlo localmente para tener un mayor entendimiento de *cuando* se invoca cada método de React.

Veamos algunos de estos eventos, primero los vamos a separar en dos categorías que cubren la mayoría de ellos:

- Cuando un Componente es Montado o Desmontado en el DOM.
- Cuando un Componente recibe nuevos datos.

Montando / Desmontado

Cuando un Componente es agregado al DOM decimos que fue Montado (mounted) y cuando es removido del DOM decimos que fue Desmontado (unmounted). Por definición estos eventos son llamados por React *solo* una vez en el ciclo de vida del Componente (cuando 'nace' y cuando 'muere'). Por lo tanto nos van a servir para setear ciertas condiciones iniciales de un Componente o bien, cerrar o eliminar ciertos listeners que sólo le servían al Componente en cuestión, en general la mayoría de las veces haremos lo siguiente en estos Eventos:

- Establecer algunas *props* por defecto.
- Establecer algunos Estados iniciales del Componente.
- Hacer alguna petición AJAX para traer datos necesarios para el Componente.
- Crear listeners si son necesarios.
- Remover listeners que ya no sirven más.

Para poder hacer uso de estos Eventos, React nos da una serie de método que son invocados según el momento del ciclo de vida del Componente. Dentro de estos métodos nosotros vamos a agregar la funcionalidad que necesitamos para nuestro Componente.

Establecer props por defecto

Varias veces vamos a tener Componentes que nos van a servir para varias cosas, por lo tanto, cuando los instanciamos vamos a querer darles distintas *props* por defecto (se toman estas props si no le pasamos una en particular). Para hacerlo, React nos provee el método `defaultProps`, en que asignamos un objeto con las *props* que finalmente tendrá nuestro Componente cuando se renderize, por ejemplo:

```
class Cargando extends React.Component {
  constructor(props) {
    super(props);
    ...
  }
  render() {
    ...
  }
};

Cargando.defaultProps = {
  text: 'cargando...',
}
```

Al igual que en una clase, tenemos el ejemplo en una `function`:

```
function Cargando(props) {
  return (
    ...
  )
}

Cargando.defaultProps = {
  text: 'cargando...',
}
```

En este ejemplo, si no le pasamos la *prop* `text` al Componente `Cargando`, esta tomará el valor por defecto: `cargando....`

Establecer un estado Inicial

Cuando queremos que nuestro Componente maneje algún tipo de Estado le tenemos que setear el *Estado Inicial* del mismo. Esto lo podemos lograr desde el constructor mediante `this.state` que se le asigna un objeto con el estado inicial. Esta será llamada cuando el Componente es montado al DOM por primera vez.

```
class Login extends React.Component{
  constructor(props) {
    super(props)
    this.state = {
      email: '',
      password: ''
    }
  }
  render() {
    ...
  }
};
```

Al usar un componente de función, seteamos el *Estado Inicial* con el Hook `useState`, en donde el primer elemento del array que nos retorna es nuestro estado, y el segundo es una función para cambiar el estado:

```
function Login() {
  const [email, setEmail] = useState('')
  const [password, setPassword] = useState('')

  return (
    ...
  )
};
```

Hacer alguna petición AJAX para traer datos necesarios para el Componente.

Este es un caso muy común, el Componente necesita datos que son traídos a través de un request tipo AJAX. En un componente de clase, React nos da el método `componentDidMount`, este método es llamado justo después que el componente se montó al DOM:

```
class Lista extends React.Component {
  componentDidMount() {
    return Axios.get(this.props.url).then(this.props.callback) // AJAX request con Axios
  }
  render() {
    ...
  }
};
```

Al usar una función, introducimos el Hook `useEffect`. Este recibe un callback que se ejecuta después de cada renderizado en el componente, y nos permite hacer peticiones de datos, establecimiento de suscripciones y actualizaciones manuales del DOM en componentes de React. Por lo general llamamos a estas operaciones “efectos secundarios” (o simplemente “efectos”). Este Hook equivale a los ciclos de vida de clase:

componentDidMount, componentDidUpdate y componentWillUnmount combinados. El segundo argumento que recibe es un `array`, al pasar un array vacío `[]`, esto le indica a React que `useEffect` no depende de ningún valor proveniente de las props o el estado, de modo que no necesita volver a ejecutarse.

```
function Lista(props) {
  useEffect(() => {
    Axios.get(props.url).then(props.callback) // AJAX request con Axios
  },[])
}

return (
  ...
)
};

##### Crear listeners si son necesarios
```

Como se pueden imaginar, también vamos a hacerlo usando el método `componentDidMount` y el Hook `useEffect`:

````javascript`

```
class Lista extends React.Component {
 componentDidMount() {
 ee.on('evento', () => this.setState({ ... }) // creamos un event listener para
 un 'evento'
 }
 render() {
 ...
 }
};
```

```
function Lista() {
 const [state, setState] = useState({})
 useEffect(() => {
 ee.on('evento', () => setState({ ... })) // creamos un event listener para un
 'evento'
 },[])
}

return (
 ...
)
```

**Remover listeners que ya no sirven más.**

Análogamente, existe el método `componentWillUnmount` que es invocado justo antes de remover el Componente del DOM:

```
class FriendsList extends React.Component {
 componentWillMount() {
 ee.off() //sacamos el listener que habiamos puesto.
 }
 render() {
 ...
 }
};
```

Para el caso de una function, usamos el mismo Hook, `useEffect`, Quizás puedes estar pensando que necesitaríamos un efecto aparte para llevar a cabo el remove del event. Pero el código para añadir y eliminarlo está tan estrechamente relacionado que `useEffect` está diseñado para mantenerlo unido. Si tu efecto devuelve una función, React la ejecutará en el momento correcto:

```
function FriendList() {
 const [state, setState] = useState({})
 useEffect(() => {
 ee.on('evento', () => setState({ ... }))

 return () => {
 ee.off() //sacamos el listener que habiamos puesto.
 };
 },[])
}

return (
 ...
)
```

En este caso, cuando retornamos una función en `useEffect`, esta es ejecutada antes de que el componente sea removido de la UI.

## Eventos cuando el componente recibe nuevos Datos

El primero método que veremos es `componentWillReceiveProps`, este evento se activa cuando el Componente recibe nuevas *props*.

El segundo, y más avanzado es `shouldComponentUpdate`. Ya sabíamos que React le pone mucho importancia a re-renderizar nuevos Componentes (ya que esto implica mucho trabajo para el cliente), por lo tanto nos da este método para que nosotros podamos controlar este comportamiento. Justamente, `shouldComponentUpdate` devuelve un **Booleano**, si es `true`, entonces se re-redenrizará el Componente (y por ende todos sus hijos), en caso de ser `false` no se hará tal cosa.

En el caso en donde tenemos un componente de función. Utilizaremos el Hook `useEffect` para el primer caso. Este Hook puede recibir como segundo parámetro un array `[]`. Por ejemplo:

```
function Ejemplo() {
 const [name, setName] = useState("Toni");
 useEffect(() => {
 document.title = name;
 }, [name]);

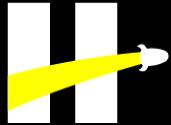
 return (
 <input value={name} onChange={event => setName(event.target.value)} />
);
};
```

En el ejemplo anterior, no necesitamos actualizar el título del documento (nuestro efecto) después de cada representación, sino solo cuando el nombre de la variable del state cambie su valor. Es por eso que pasamos array con el valor de `name` como segundo parámetro: Aca si solo queremos que nuestro Hook se invoque solo después del primer render, tenemos que pasar una matriz vacía [] (que nunca cambia) como segundo parámetro. En el segundo caso podemos utilizar `React.memo` que es un HOC (High Order Component) que ayuda en la performance de renderizado de un componente, evitar un re-renderizado. Si un componente devuelve el mismo resultado, es decir, no cambian sus props. Envolver el componente en `React.memo` puede ayudar mucho a la performance. Esta función puede recibir como segundo argumento una función de comparación personalizada, que reciba las props viejas y las nuevas. Si retorna true, se obvia la actualización. Por ejemplo:

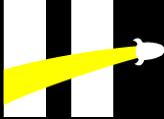
```
function Ejemplo(props) {
 return (
 <div>{props.name}</div>
);
};

const fnComparacion = function(prevProps, nextProps) {
 return prevProps.name === nextProps.name;
};

export default React.memo(Ejemplo, fnComparacion);
```



# ROUTING



# SPA

Single Page Application

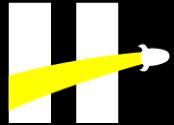


No page refresh on request

Traditional Web Application



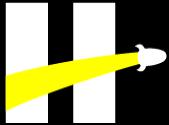
Whole page refresh on request



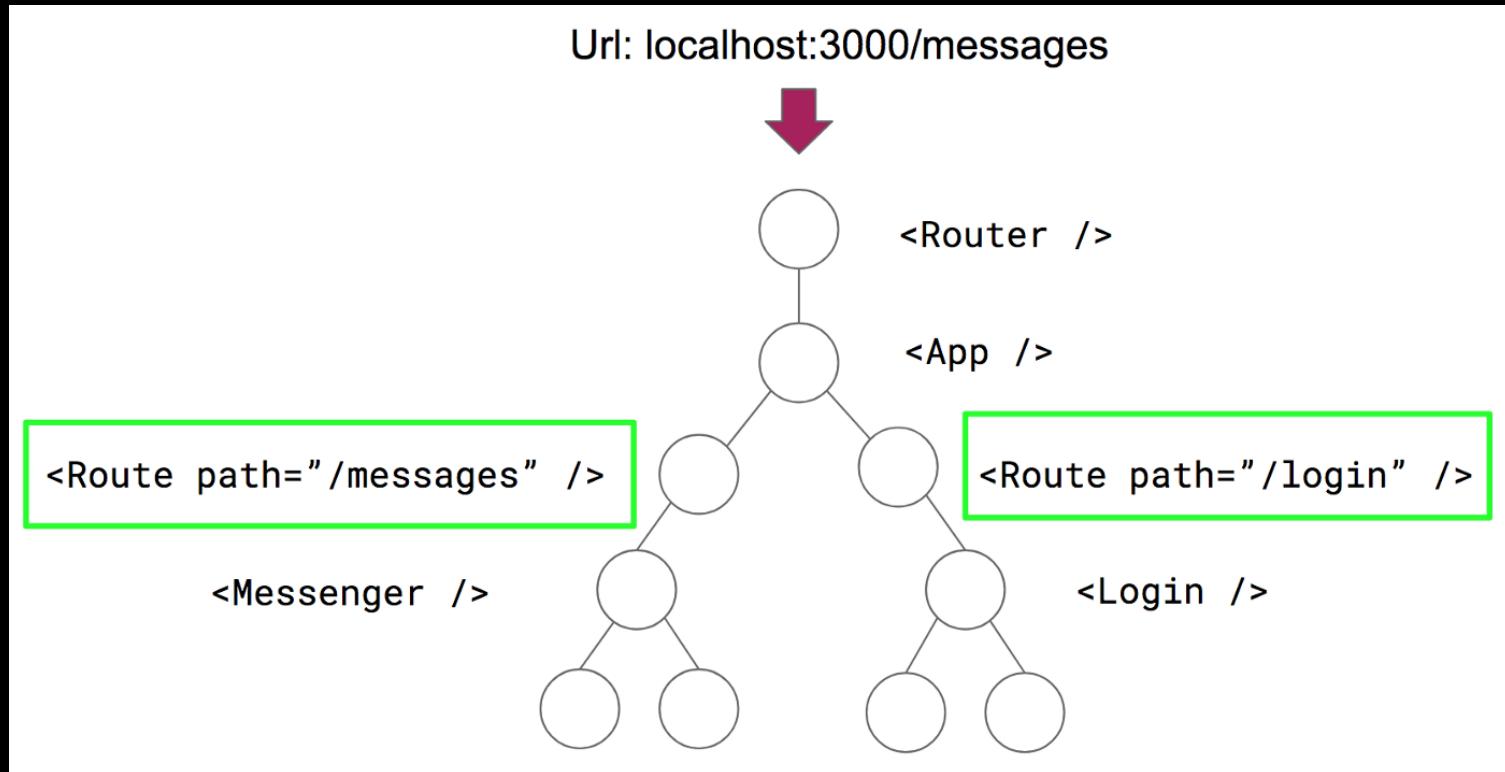
# SPA

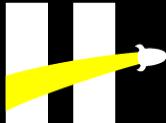


Librería para definir de forma declarativa la  
vistas que queremos renderizar  
dependiendo la URL

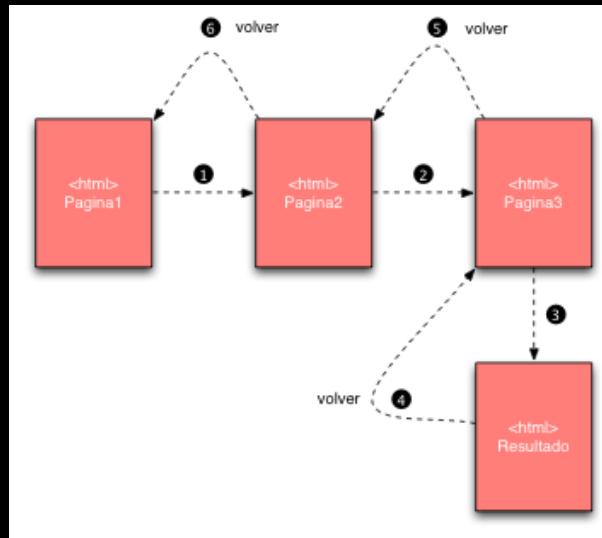


# SPA - Routing

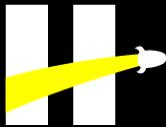




# SPA - History API



El objeto DOM `window` proporciona acceso al historial del navegador a través del objeto `history`. Este da acceso a métodos y propiedades útiles que permiten avanzar y retroceder a través del historial del usuario, así como manipular el contenido del historial.



# Routing Config

Para configurar nuestra aplicación primero tenemos que decidir que tipo de router vamos a usar:



<HashRouter />

http://example.com/#/about



Configuración del servidor más simple ya que siempre el request es a la misma URL



Agrega un # a la URL

<BrowserRouter />

http://example.com/about



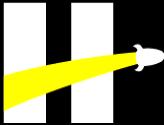
Configuración extra en ambiente de producción (DEV out of the box)



URL más prolífa



```
1 // Instalar React-Router-DOM
2 npm i react-router-dom
3
4 // Utilización
5 import { HashRouter, Route } from 'react-router-dom';
6
7 ReactDOM.render(
8 <HashRouter>
9 <App />
10 </HashRouter>
11), document.getElementById('app'))
```



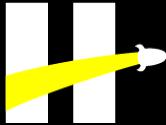
# <Route />



```
1 <Route
2 exact
3 strict
4 sensitive
5 path="..."
6 component={Component}
7 render={() => <Component />}
8 children={() => <Component />}
9 >
10 ...
11 </Route>
```

## Propiedades del componente <Route />

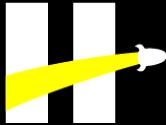
- **path**: URL o array de URLs que se van a comparar contra el path actual para ver si renderizar o no los componentes definidos en esa ruta
- **exact**: si se encuentra dentro de route, el path debe coincidir por completo y no únicamente su inicio
- **strict**: si se encuentra dentro de route, el path debe coincidir incluso en las barras (' / ') del path
- **sensitive**: si se encuentra dentro de route, el path sera case sensitive



# Basic Routing



```
1 import React from 'react';
2 import { render } from 'react-dom';
3 import { Route, Switch, HashRouter as Router, useParams } from 'react-router-dom';
4
5 const Root = (
6 <Router>
7 <NavBar />
8 <Switch>
9 <Route exact path="/">
10 <Home />
11 </Route>
12 <Route path="/about">
13 <About />
14 </Route>
15 <Route path="/ejemplo">
16 <Ejemplo nombre="Toni" apellido="Tralice"/>
17 </Route>
18 <Route path="/">
19 <h2>Default if no match</h2>
20 </Route>
21 </Switch>
22 </Router>
23);
24
25 render(Root, document.querySelector('#app'));
```



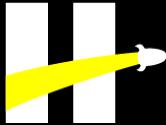
# <Link />



```
1 import React from 'react';
2 import { Link } from 'react-router-dom';
3
4 export default function NavBar() {
5 return (
6 <div className="nav-bar">
7 <h2>Barra de Navegación</h2>
8 <Link to="/">Home</Link>
9 <Link to="/about">About</Link>
10 <Link to="/ejemplo">Ejemplo</Link>
11 </div>
12);
13}
```

El componente <Link /> se va a traducir a un tag <a> que nos va a permitir redireccionar al usuario hacia una nueva URL

- **to:** nos indica el path hacia el cual debemos redirigir una vez clickeado el link



# <NavLink />

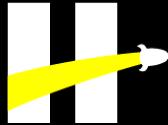


```
1 export default function NavBar() {
2 return (
3 <div className="nav-bar">
4 <h2>Barra de Navegación</h2>
5 {/* <NavLink to="/">Home</NavLink> */}
6 <NavLink exact to="/">Home</NavLink>
7 <NavLink to="/about" activeClassName="selected">About</NavLink>
8 <NavLink to="/ejemplo" activeStyle={{
9 color: 'green'
10 }}>Ejemplo</NavLink>
11 </div>
12);
13 };
```



El componente <NavLink /> mantiene la misma funcionalidad que <Link> pero permite dar estilos dependiendo la ruta actual

- **activeClassName**: nos indica la clase que se va a asignar a ese link cuando la ruta coincide con la especificada
- **exact**: si lo agregamos solo va a aplicar los estilos cuando la ruta coincide en su totalidad y no solo en su comienzo
- **activeStyle**: permite definir un style de forma inline para cada link



# <Route /> - Legacy Methods

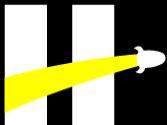


```
1 <Route
2 exact
3 strict
4 sensitive
5 path="..."
6 component={Component}
7 render={() => <Component />}
8 children={() => <Component />}
9 >
10 ...
11 </Route>
```

Métodos legacy del componente <Route />

- **component**: toma un componente y lo crea usando `React.createElement`
- **render**: toma una función que retorna UI. Útil para hacer inline render y pasar props al componente a renderizar.
- **children**: similar a render, pero siempre renderiza.

Si el path y la ubicación actual coinciden se crea un objeto conteniendo tres propiedades (match, location y history) el cual se pasa como prop del componente renderizado o como parámetro de la inline function.



# <Route /> - Route Props

```
● ● ●
1 <Route
2 path="/home"
3 component={Home}
4 />
5
6
7 <Route
8 path="/:id"
9 render={({match, location, history}) => (
10 <Home match={match} Ejemplo />
11)}
12 />
13
14 <Route
15 path="/:id"
16 children={(props) => (
17 <Home {...props} Ejemplo />
18)}
19 />
```

```
Home

props
▶ history: {action: "PUSH", block: f block() {}, createHref: f...}
▶ location: {hash: "", pathname: "/component", search: "", stat...}
▶ match: {isExact: true, params: {...}, path: "/component", ur...}
```



# <Route /> - Route Props

## match

Propiedades del objeto match

- **isExact**: boolean que será true si la ruta matchea exactamente, false de lo contrario
- **params**: objeto con los valores de los parametros de la ruta
- **path**: string que contiene el patrón utilizado para comparar contra la URL actual
- **url**: string que contiene la porción matcheada del url.

```
▼ match: {isExact: true, params: {...}, path: "/component/:fir..."}
 isExact: true
 ▶ params: {firstParam: "one", secondParam: "two"}
 path: "/component/:firstParam/:secondParam"
 url: "/component/one/two"
```



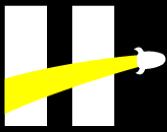
# <Route /> - Route Props

## location

Propiedades del objeto location

- **pathname**: string indicando el path actual
- **search**: string que contiene los parámetros pasados por query a la ruta actual. Si no se pasa ninguno, será un string vacío
- **state**: propiedad que por default es undefined pero en la cual podemos pasar datos extras a la nueva ruta

```
 - location: {hash: "", pathname: "/component/one/two", search: ...}
 hash: ""
 pathname: "/component/one/two"
 search: ""
 state: undefined
```



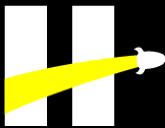
# <Route /> - Route Props

## history

Propiedades del objeto history

- **action**: string que indica la acción que se realizó para llegar a la ruta actual
- **block**: función que va a bloquear la posibilidad de navegación hacia otras rutas
- **go**: función que nos permite movernos n lugares en nuestro stack de navegación. Pueden ser números negativos para ir hacia atrás
- **goBack**: función que permite navegar una posición hacia atrás. Es equivalente a "go(-1)"
- **goForward**: función que permite navegar una posición hacia adelante. Es equivalente a "go(1)"
- **length**: número que representa la cantidad de entradas en el stack de navegación
- **push**: función que nos permite agregar otra entrada al stack de navegación
- **replace**: función que nos permite reemplazar la entrada actual en el stack de navegación por otra

```
history: {action: "PUSH", block: f block() {}, createHref: f...}
 action: "PUSH"
 block: f block() {}
 createHref: f createHref() {}
 go: f go() {}
 goBack: f goBack() {}
 goForward: f goForward() {}
 length: 50
 listen: f listen() {}
 location: {hash: "", pathname: "/component/one/two", search: ...}
 push: f push() {}
 replace: f replace() {}
```



# <Route /> - Route Props

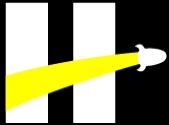
## location vs history.location



El objeto history es mutable por lo que la recomendación es que si necesitamos utilizar 'location' lo hagamos desde esa propiedad y no desde history.



```
1 class Comp extends React.Component {
2 componentDidUpdate(prevProps) {
3 // will be true
4 const locationChanged =
5 this.props.location !== prevProps.location;
6
7 // INCORRECT, will *always* be false because history is mutable.
8 const locationChanged =
9 this.props.history.location !== prevProps.history.location;
10 }
11 }
12
13 <Route component={Comp} />;
```



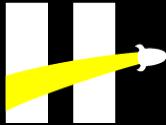
# useParams



```
1 import React from 'react';
2 import { useParams } from 'react-router-dom';
3
4 export default function Mostrar() {
5 let params = useParams();
6 return (Estoy en la ciudad {params.ciudadId})
7 }
```

```
Params: ▾ Object { ciudadId: "5" }
 | ciudadId: "5"
 | > <prototype>: Object { ... }
```

Cuando la URL especificada dentro de `<Route>` es dinámica, es decir, que puede variar podemos obtener el o los valores pasados como parámetro dentro del path mediante el hook **useParams**



# useHistory

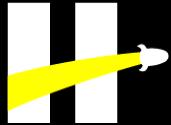


```
1 import React from 'react';
2 import { useHistory } from "react-router-dom";
3
4 export default function History() {
5 let history = useHistory();
6
7 ...
8 }
```

```
History:
▼ Object { length: 31, action: "PUSH", location: {...},
 createHref: createHref(location) ↗,
 push: push(path, state) ↗,
 replace: replace(path, state) ↗,
 go: go(n) ↗,
 goBack: goBack() ↗,
 goForward: goForward() ↗,
 block: block(prompt) ↗,
 ... }

 action: "PUSH"
 ▶ block: function block(prompt) ↗
 ▶ createHref: function createHref(location) ↗
 ▶ go: function go(n) ↗
 ▶ goBack: function goBack() ↗
 ▶ goForward: function goForward() ↗
 ▶ length: 31
 ▶ listen: function listen(listener) ↗
 ▶ location: Object { pathname: "/history", search: "", hash: "", ... }
 ▶ push: function push(path, state) ↗
 ▶ replace: function replace(path, state) ↗
```

Hook que nos permite acceder a la instancia de History sin necesidad de pasarla explícitamente como vimos con las propiedades del componente <Route /> previamente



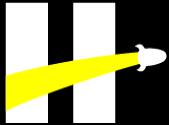
# useLocation



```
1 import React from 'react';
2 import { useLocation } from "react-router-dom";
3
4 export default function Location() {
5 let location = useLocation();
6
7 ...
8 }
```

```
Location:
 Object { pathname: "/location", state: {...}, search: "",
hash: "" }
 hash: ""
 pathname: "/location"
 search: ""
 state: Object { extraData: "Henry" }
 extraData: "Henry"
```

Hook que nos permite acceder al objeto Location sin necesidad de pasarla explícitamente como vimos con las propiedades del componente <Route /> previamente



# Nested Routing

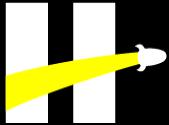


```
1 const Root = (
2 <Router>
3 <NavBar />
4 <Switch>
5 <Route path="/home/linkRelative">
6 <h2>Estoy en /home/linkRelative</h2>
7 </Route>
8 <Route path="/home">
9 <Home />
10 </Route>
11 <Route path="/">
12 <h2>Default</h2>
13 </Route>
14 </Switch>
15 </Router>
16);
```



```
1 function Home() {
2 let match = useRouteMatch();
3 return (
4 <div>
5 <h2>Home, Soy Henry!!</h2>
6 <Link to='/linkAbsolute'>Link Absolute</Link>
7

8 <Link to={`${match.url}/linkRelative`}>Link Relative</Link>
9 </div>
10);
11 };
```



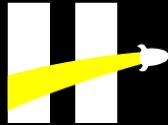
# <Prompt />

Componente que nos va a permitir mostrar un pop-up de confirmación si el usuario quiere irse del path actual.



```
1 <Prompt
2 when={condition}
3 message="Are you sure you want to leave?"
4 />
```

- **when**: booleano que va a determinar si el prompt debe mostrarse o no cuando se intente navegar hacia otro path
- **message**: podemos pasarle un string o una función. En el primer caso será simplemente el mensaje del pop-up y el segundo permite mayor customización, posibilitando el acceso a 'location' y 'action'



# <Redirect />

Componente que al renderizarse automáticamente nos redirigirá hacia un nuevo path reemplazando la ubicación actual en el stack de navegación.



```
1 <Route exact path="/">
2 {loggedIn ? <Redirect to="/dashboard" /> : <PublicHomePage />}
3 </Route>
```

- **to**: puede ser un string indicando la URL a la cual debe redirigir o un objeto con mayor información (pathname, search y state)
- **from**: string indicando una URL que si la aplicación intenta acceder automáticamente redirigirá hacia la URL indicando en el **to**.

**exact, strict, sensitive**



< demo-extra />

# Henry

---



Hacé click acá para dejar tu feedback sobre esta clase.



Hacé click acá completar el quiz teórico de esta lecture.

## Front-End Routing

---

Vamos a ver que muchísimos proyectos de *React* están construidos como **SPA** (Single Page Applications), es decir que sin recargar una página se van mostrando algunos *Containers* u otros según donde vaya navegando el usuario. Gracias a que tenemos todo encapsulado en *Componentes* pensar nuestra *app* como **SPA** no puede ser muy complejo, lo que sí puede serlo es el *ruteo* interno en el *front-end*, es decir, el trabajo de **mapear** cada *link* con algún *Componente*. Por suerte no estamos solos, existen varias librerías en *npm* listas para ayudarnos con esto. Nosotros vamos a usar una llamada *react-router*.

### React Router

Según su descripción en su [repo](#), **react-router** sirve para mantener sincronizados tu *UI* con la *url* de una forma *declarativa*.

Nosotros utilizaremos *react-router-dom*, que contiene los componentes básicos de *react-router* más componentes extras que iremos utilizando. Básicamente, lo que nos da *react-router-dom* son una serie de *Componentes*, los cuales van a recibir ciertas *props* que le van cambiar el comportamiento. La idea entonces será tener un *Componente* principal que se va a cargar en nuestra página (la única que vamos a tener), y este se encargará de llamar a nuestro **Componentes** que queremos mostrar según a donde navegue el usuario. Por ejemplo:

```
ReactDOM.render((
 <HashRouter>
 <Route path="/" component={Home} />
 </HashRouter>
) , document.getElementById('root'));
```

### Empezando con *react-router-dom*

Como siempre, vamos a empezar por instalar *react-router-dom* en nuestro proyecto con `$ npm install -save react-router-dom`.

Luego, vamos a *importar* los *Componentes* que necesitamos:

```
import { HashRouter, Route } from 'react-router-dom';

ReactDOM.render(
 <HashRouter>
 <Route path="/" component={App}/>
 </HashRouter>
), document.getElementById('app'))
```

En una SPA no vamos a poder usar la barra de URL *out of the box* para manejarnos entre nuestras URLs. Por eso existen componentes como **HashRouter** que es un Router que usa partes de los hashes de la URL para mantener en sincronía nuestra UI con la URL y **BrowserRouter** que usa la API de HTML5 para mantener la sincronía con la URL. El componente **BrowserRouter** es más utilizado en servidores que manejan peticiones dinámicas, mientras que **HashRouter** se utiliza en sitios web estáticos. En este caso utilizaremos **HashRouter**.

Para ver cómo funciona, vamos a usar los *Componentes* que habíamos creado en el ejemplo anterior: **MostrarLista**, **ContenedorAmigos**, y **HelloWorld**. Vamos a crear una SPA con dos **rutas**, una para **MostrarLista** y otra para **HelloWorld**.

Para hacerlo, primero vamos a importar todos los *Componentes* y librerías necesarias y luego usar los Componentes de *react-router-dom* para definir nuestras rutas de la siguiente manera:

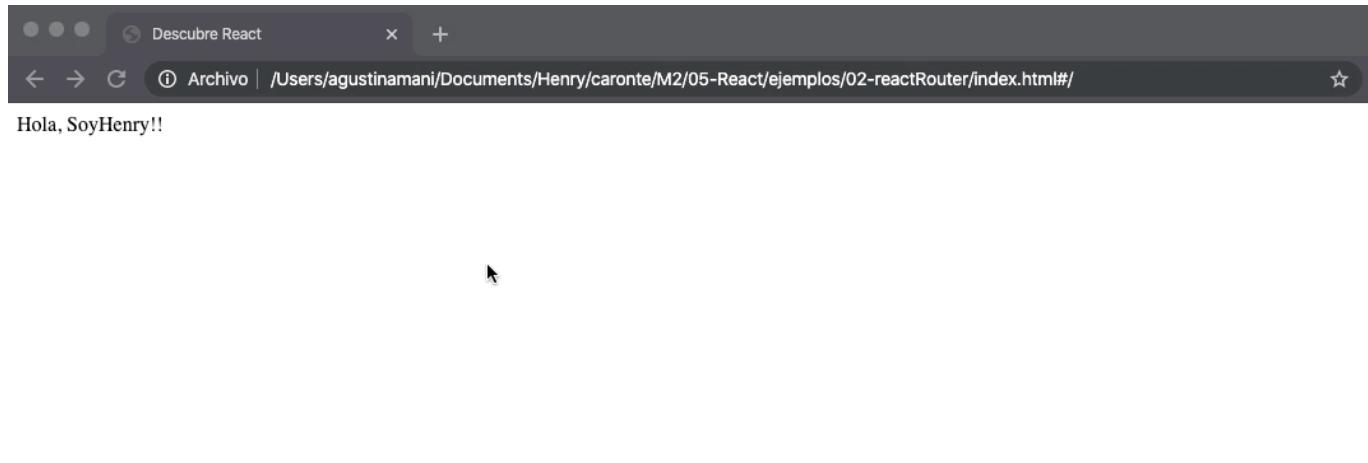
```
import MostrarLista from './MostrarLista.js';
import HelloWorld from 'HelloWorld.js';

// ReactDOM.render(<ContenedorAmigos>Hola!</ContenedorAmigos>,
document.getElementById('app'));

ReactDOM.render(
 <HashRouter>
 <Route path="/lista" component={ContenedorAmigos}/>
 <Route exact path="/" component={HelloWorld}/>
 </HashRouter>
), document.getElementById('app'));
```

Primero vemos que vamos a envolver todo en un Componente llamador **BrowserRouter**. Dentro de **BrowserRouter**, vamos a agregar nuestras rutas. En este caso agregamos dos, usando **<Route>**. A este Componente hay que pasarle dos propiedades:

- **path**: Es el path de la **url** que va a activar esta ruta.
- **component**: Es el componente que se va a cargar cuando ingresemos a la ruta definida en el **path**.
- **exact**: Agregamos esta keyword para que matchee exactamente con el path que le pasamos. Este parametro entra en juego cuando tenemos path anidados.



Ahora, si probamos este ejemplo (siguiendo los pasos mencionados anteriormente sobre como usar `webpack`), vamos a ver en el browser que se carga el Componente que declaramos en la ruta `/`, y si escribimos `/lista` en la URL, vemos que automáticamente se carga el componente declarado en la ruta `/lista`.

Si son observadores se habrán dado cuenta del `#` en la barra de direcciones después del `index.html`, esto se debe a que estamos usando el middleware `hashHistory` para mapear URLs, más adelante vamos a ver otras formas.

## Links

Bien, ahora lo que necesitamos es crear Links para navegar entre rutas. Podríamos usar el tag `<a>` y en el `href` agregar el `/`, por dos razones simples no vamos a usar este método y sí vamos a usar un nuevo Componente de `react-router-dom` llamado `Link`. La primera razón es que si por alguna razón dejamos de usar `BrowserRouter`, probablemente (no es seguro) nuestros links dejarían de funcionar, la segunda es que los `Links` de `react-router-dom` no producen un 'refresh' en nuestra pagina como lo hacen los hags

Como siempre, primero vamos a tener que importar el Componente `Link` de `react-router-dom`. Luego, lo vamos a utilizar de manera muy similar a `<a>` sólo que en vez de `href` vamos a pasarle la *propiedad* `to`, y en ella indicarle a que *path* nos debería llevar. Por ejemplo, agreguemos un Link en el Componente `Home` del ejemplo anterior (también agregamos un link `<a>` para ver qué sucede):

```
import React from 'react';
import { Link } from 'react-router-dom';

function Home(){
 return (
 <div>
 Hola, Soy Henry!!
 Link normal a Lista
 <Link to="/lista">Link de react-router a Lista</Link>
 <Link to="/">Link de react-router a Home</Link>
 </div>
)
};

});
```

En el Componente **ContenedorAmigos** también voy a agregar un Link, para poder navegar ida y vuelta entre los dos *paths*.

También agregamos un Link a la misma página como si fuera una barra de navegación.

Existe tambien una nueva version de los que son los que nos ayuda a agregar estilos a nuestros links que matcheen con nuestra URL actual.

Tambien veamos como usar el feature de mostrar distinto los **Links** de páginas activas. Por empezar definamos una clase CSS con algún color llamativo en nuestro archivo **index.html** :

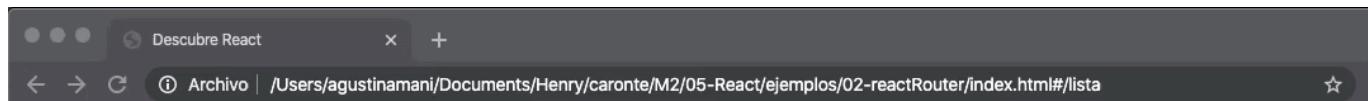
```
<style>
 .active {
 color: SpringGreen;
 font-size: 24px;
 }
</style>
```

Ahora en los **NavLink**s vamos a agregar la propiedad **activeClassName** y pasarle el nombre de la clase que queremos que tenga el **Link** cuando coincidan la url que tiene en su propiedad **to** con la url en la que estamos. Esto nos va a servir para, por ejemplo, tener un sólo Componente que sea una barra de navegación y cambiar el estilo del link cuando el usuario haya navegado a la página correspondiente.

```
Link normal a Lista
<NavLink to='/lista' activeClassName= 'active'>Link de react-router a
Lista</NavLink>
<NavLink to='/' activeClassName= 'active'>Link de react-router a
Home</NavLink>
```

Por ahora no estamos realizando las mejores prácticas en términos de reutilizar Componentes, por lo tanto vamos a escribir la propiedad en todos los Links.

Veamos el resultado:



**Nombre: Soy Henry!**

### Amigos

- Emi
- Toni
- Facu

[Link de react-router-dom a Home](#) [Link de react-router-dom a Lista](#)

Con esto vamos a poder navegar entre páginas de nuestra SPA, y además ya tenemos resuelto el tema de mantener los estilos de los Links de páginas activas!

## Reutilizando Componentes con React-Router-Dom

Recién vimos en el ejemplo que usamos la misma barra de navegación en varios Componentes. Pero en este caso no *reutilizamos* código ya que tuvimos que escribir el código de los [Links](#) en cada Componente.

Veamos como podríamos hacer para tener siempre un Componente que se cargue siempre (con cosas que queremos mostrar siempre como Navbars, Footers, etc...), y dentro de este que se carguen distintos Componentes según la página a la que naveguemos.

Lo primero que necesitamos es tener algo que queremos que se repita, en este caso una barra de navegación con algunos Links:

```
import React from 'react';
import { NavLink } from 'react-router-dom';

export default function NavBar() {
 return (
 <div>
 <h2>Barra de Navegación</h2>
 <NavLink to="/" activeClassName="active" >Home</NavLink>
 <NavLink to="/about" activeClassName="active" >About</NavLink>
 <NavLink to="/ejemplo" activeClassName="active">Ejemplo</NavLink>
 </div>
);
}

});
```

Ahora que tenemos nuestra NavBar, vamos a necesitar un Componente que se cargue siempre, y que dentro de él estén la Navbar (o todo lo que queramos que se renderize siempre) y también que *llame* a los Componentes que corresponden según la url. Para entender esto, mejor empezemos viendo cómo serán las rutas de [react-router-dom](#) para diagramar lo que queremos hacer:

```
ReactDOM.render(
 <HashRouter>
 <NavBar />
 <Switch>
 <Route path="/about" component={About}/>
 <Route path="/ejemplo" component={Ejemplo}/>
 <Route exact path="/" component={Home}/>
 </Switch>
 </HashRouter>,
 document.getElementById('app')
);
```

Ahora introducimos otro componente de 'react-router-dom', , este es único en el sentido de que representa una ruta exclusivamente. Por ej, si estamos en /about, comenzará a buscar una coincidente. coincidirá y dejará de buscar coincidencias y mostrará . Básicamente lo que va a ocurrir, es que cuando se cargue la página, se va a invocar al Componente raíz que matchee con la ruta ( en este ejemplo: / ), y si cambiamos de ruta por ejemplo /ejemplos, empezara a buscar y en el primer match renderizara la ruta, en este caso el componente . Ponemos el path "/" ya que siempre sera match entonces sera nuestra ruta por default. Afuera de nuestro **Switch** pondremos nuestro componente , que no tendra ningun path para nuestra url, y renderizara siempre. Este sera nuestro componente que nos permitira movernos a las rutas que tengamos en nuestro **Switch**.

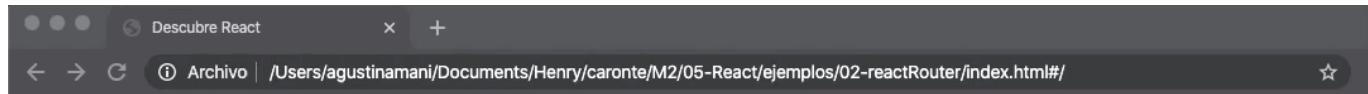
Como vemos, importamos los Componentes como siempre y en el método render retornamos un **div** ( JSX sólo nos deja retornar un único elemento que envuelta todo), que tiene adentro a nuestro **Nav** y además usamos la referencia **this.props.children** para retornar el Componente que nos haya pasado **react-router-dom** según la ruta donde haya navegado el usuario.

En la siguiente imagen, digramamos que sería cada Componente:



El componente violeta llamado *Componentes* va a ser el que se renderizé a través nuestras rutas.

Veamos nuestro código en funcionamiento:



## Barra de Navegación

[Home](#) [About](#) [Ejemplo](#)

Hola, SoyHenry!!

Bien, vemos que renderiza bien la barra de navegación en todas las páginas, pero está pasando algo raro ya que el **NavLink** al home ( '/' ) está siempre como activo. Esto se debe a que la ruta de ese link matchea siempre con la ruta de la ruta *raíz*. Para resolver este problema debemos agregar lo siguiente:

```
<div>
 <h2>Barra de Navegación</h2>
 <NavLink exact to="/" activeClassName="active" >Home</NavLink>
 <NavLink to="/about" activeClassName="active" >Componente2</NavLink>
 <NavLink to="/ejemplo" activeClassName="active">Componente3</NavLink>
</div>
```

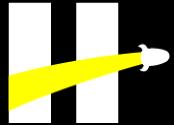
El parametro **exact** cuando lo pasamos nuestra **activeClassName** solo se aplicará si la ubicación coincide exactamente.



## Barra de Navegación

[Home](#) [About](#) [Ejemplo](#)

Hola, SoyHenry!!



# Formularios



# Formularios

## Controlled (Controlados)

Los valores de los inputs están *bindeados* al estado del Componente.

- Esta es la forma recomendada por el equipo de React.
- Sigue los patrones de React.
- Es más predecible.

## Uncontrolled (No Controlado)

Sacas el valor del DOM.

- Fácil de hacer, no hay que aprender nada nuevo.
- Lo que se muestra en el input no lo podemos controlar.
- Es menos predecible.



# Formularios

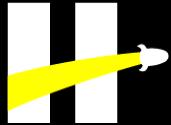
| feature                                                      | controlled | uncontrolled |
|--------------------------------------------------------------|------------|--------------|
| one-time value retrieval (e.g. on submit)                    | ✓          | ✓            |
| <a href="#"><u>validating on submit</u></a>                  | ✓          | ✓            |
| <a href="#"><u>instant field validation</u></a>              | ✓          | ✗            |
| <a href="#"><u>conditionally disabling submit button</u></a> | ✓          | ✗            |
| enforcing input format                                       | ✓          | ✗            |
| several inputs for one piece of data                         | ✓          | ✗            |
| <a href="#"><u>dynamic inputs</u></a>                        | ✓          | ✗            |

<https://goshakkk.name/controlled-vs-uncontrolled-inputs-react/>



# Formularios - Importancia de Key

```
1 function Ejemplo({lang}) {
2 if (lang === 'hun') {
3 return (
4 <form>
5 <input key="lastName" type="text" placeholder="Vezetéknév" name="lastName"/>
6 <input key="firstName" type="text" placeholder="Keresztnév" name="firstName"/>
7 <input key="middleInitial" type="text" placeholder="KB" style={{width: 30}} name="middleInitial"/>
8 </form>
9)
10 }
11 return (
12 <form>
13 <input key="firstName" type="text" placeholder="First Name" name="firstName"/>
14 <input key="middleInitial" type="text" placeholder="MI" style={{width: 30}} name="middleInitial"/>
15 <input key="lastName" type="text" placeholder="Last Name" name="lastName"/>
16 </form>
17);
18 }
```



< Demo />

# Henry

---



Hacé click acá para dejar tu feedback sobre esta clase.

Hacé click acá completar el quiz teórico de esta lecture.

## Lesson 10 - React Forms

---

Los formularios son muy útiles en cualquier aplicación WEB. En React tenemos que manejar estos formularios nosotros mismos. Por ejemplo: obtener los valores que se ingresan, cómo administramos el state del form, las validaciones de cada valor ingresado, mostrar los mensajes de validación, etc. Existen diferentes métodos y librerías que nos ayudan con esto, pero como no queremos depender de código de otro, lo haremos nosotros.

## Tipos de Componentes para un Formulario:

En React tenemos dos tipos de componentes para crear nuestro Form: **Controlled Components** y **Uncontrolled Components**.

## Componentes Controlados

---

Como sabemos, el estado en React es mutable, y lo mantenemos dentro del componente. En un componente controlado, que renderiza el Formulario, también controla lo que sucede con él. Es decir que a medida que cambien los valores del Form, el componente guarda esos valores en su state. Aquí vemos un pequeño ejemplo:

Con Hooks:

```
import React, { useState } from 'react';

function Form() {
 const [name, setName] = useState('');

 function handleChange(e) {
 setName(e.target.value);
 }

 return (
 <form>
 <input
 type="text"
 value={name}
 onChange={handleChange}
 />
 </form>
);
}
```

```
 name="name"
 value={name}
 onChange={handleChange}
 />
 </form>
)
}
```

Con Class:

```
import React from 'react';

class Form extends React.Component {
 constructor () {
 super();
 this.state = {
 name: ''
 }
 this.handleChange = this.handleChange.bind(this);
 }

 handleChange(e) {
 this.setState({
 name: e.target.value
 });
 }

 render () {
 return (
 <form>
 <input
 type="text"
 name="name"
 value={this.state.name}
 onChange={this.handleChange}
 />
 </form>
);
 }
}

export default Form;
```

Nuestro objetivo es que cada vez que cambien los valores de nuestro Form lo vayamos almacenando en nuestro state. Esto significa que nuestro Form puede responder a los cambios que se hagan en cada input. Por ejemplo agregar validaciones, deshabilitar un boton hasta que todos los campos esten llenos o validados, forzar un formato especifico en cada input.

Manejando multiples inputs:

En la mayoria de los casos tendremos mas de un solo input. Para manejarlos podemos agregar el atributo `name` a cada uno y dejar que nuestra funcion que maneje los cambios decida que hacer dependiendo de cada valor de `e.target.name`:

Con Hooks:

```
import React, { useState } from 'react';

function Form() {
 const [input, setInput] = useState({
 name: '',
 lastname: ''
 });

 function handleChange(e) {
 const value = e.target.value;
 const name = e.target.name;

 setInput({
 [name]: value // Sintaxis ES6 para actualizar la key correspondiente
 });
 }

 return (
 <form>
 <input
 name="name"
 type="text"
 value={name}
 onChange={handleChange} />
 <input
 name="lastname"
 type="text"
 value={lastname}
 onChange={handleChange} />
 </form>
)
}

export default Form;
```

Con Class:

```
import React from 'react';

class Form extends React.Component {
 constructor() {
 super();
 this.state = {
 name: '',
 lastname: ''
 };
 }

 handleChange(e) {
 const value = e.target.value;
 const name = e.target.name;

 this.setState({
 [name]: value
 });
 }

 render() {
 return (
 <form>
 <input
 name="name"
 type="text"
 value={this.state.name}
 onChange={this.handleChange} />
 <input
 name="lastname"
 type="text"
 value={this.state.lastname}
 onChange={this.handleChange} />
 </form>
);
 }
}
```

```
 lastname: ''
 };

 this.handleChange = this.handleChange.bind(this);
}

handleChange(e) {
 const value = e.target.value;
 const name = e.target.name;

 this.setState({
 [name]: value // Sintaxis ES6 para actualizar la key correspondiente
 });
}

render() {
 return (
 <form>
 <input
 name="name"
 type="name"
 value={this.state.name}
 onChange={this.handleChange} />
 <input
 name="lastname"
 type="name"
 value={this.state.lastname}
 onChange={this.handleChange} />
 </form>
);
}
}

export default Form;
```

## Validando nuestros Inputs

Otra de las cosas que queremos hacer en nuestro componente es validar los input dependiendo de que datos se tengan que ingresar. Por ejemplo: en el caso de validar un email, en el momento de cambiar nuestro state queremos 'filtrar' el valor de ese input por una funcion que valide los valores ingresados. En el caso que no sea valido mostraremos un mensaje de error.

Con Hooks:

```
import React, { useState } from 'react';

function Form() {
 const [input, setInput] = useState({
 name: '',
 lastname: '',
 user: ''
 })

 const handleChange = (e) => {
 const { name, value } = e.target;
 setInput({
 ...input,
 [name]: value
 })
 }

 const handleSubmit = (e) => {
 e.preventDefault();
 if (input.name === '') {
 alert('Name is required');
 return;
 }
 if (input.lastname === '') {
 alert('Lastname is required');
 return;
 }
 if (input.user === '') {
 alert('User is required');
 return;
 }
 alert(`Name: ${input.name}, Lastname: ${input.lastname}, User: ${input.user}`);
 }

 return (
 <form>
 <input
 type="text"
 name="name"
 value={input.name}
 onChange={handleChange} />
 <input
 type="text"
 name="lastname"
 value={input.lastname}
 onChange={handleChange} />
 <input
 type="text"
 name="user"
 value={input.user}
 onChange={handleChange} />

 <button type="button" onClick={handleSubmit}>Submit</button>
 </form>
);
}
```

```
});
const [error, setError] = useState('');

function validateEmail(value) {
 var emailPattern = /\S+@\S+\.\S+/; // Expresion Regular para validar Emails.

 if(!emailPattern.test(value)) {
 console.log('entro al if')
 setError('El usuario debe ser un email');
 } else {
 setError('')
 }
}

function handleChange(e) {
 const { value, name } = e.target;

 if(name === 'user') {
 validateEmail(input.user)
 }

 setInput({
 ...input,
 [name]: value // Sintaxis ES6 para actualizar la key correspondiente
 });
}
return (
 <form>
 <input
 name="name"
 type="text"
 value={input.name}
 onChange={handleChange}
 placeholder="Nombre" />
 <input
 name="lastname"
 type="text"
 value={input.lastname}
 onChange={handleChange}
 placeholder="Apellido" />
 <input
 name="user"
 type="text"
 value={input.user}
 onChange={handleChange}
 placeholder="Usuario" />
 {!error ? null : <div>{error}</div>}
 <input type="submit" value="Submit" />
 </form>
)
}

export default Form;
```

Con Class:

```
import React from 'react';

class Form extends React.Component {
 constructor() {
 super();
 this.state = {
 name: '',
 lastname: '',
 user: '',
 error: ''
 };
 this.handleChange = this.handleChange.bind(this);
 }

 validateEmail(value) {
 var emailPattern = /\S+@\S+\.\S+/; // Expression Regular para validar Emails.

 if(!emailPattern.test(value)) {
 this.setState({
 error: 'El usuario debe ser un email'
 })
 } else {
 this.setState({
 error: ''
 })
 }
 }

 handleChange(e) {
 const { value, name } = e.target;

 if(name === 'user') {
 this.validateEmail(this.state.user)
 }

 this.setState({
 [name]: value // Sintaxis ES6 para actualizar la key correspondiente
 });
 }

 render() {
 return (
 <form>
 <input
 name="name"
 type="text"
 value={this.state.name}
 onChange={this.handleChange}
 placeholder="Nombre" />

```

```

<input
 name="lastname"
 type="text"
 value={this.state.lastname}
 onChange={this.handleChange}
 placeholder="Apellido" />
<input
 name="user"
 type="text"
 value={this.state.user}
 onChange={this.handleChange}
 placeholder="Usuario" />
{!this.state.error ? null : <div>{this.state.error}</div>}
<input type="submit" value="Submit" />
</form>
);
}
}

export default Form;

```

Para tener un Formulario completo. Tenemos que hacer algun tipo de validacion en cada input dependiendo de que es lo que se quiera ingresar. Y para finalizar, podemos agregar una ultima validacion en nuestro boton de submit para saber si pasamos o no las validaciones anteriores. Si tenemos algun error, desabilitamos el input de submit. Ahora nuestro state **error**, pasara a ser un objeto con cada tipo de error segun nuestro input, y agregaremos **disabled** para saber si nuestro form esta habilidado o no.

```

this.state = {
 name: '',
 lastname: '',
 user: '',
 errors: {
 name: '',
 lastname: '',
 user: ''
 },
 disabled: true
};

```

Dentro de **handleChange** agregamos un switch statement para validar cada input:

```

handleChange(e) {
 const { name, value } = e.target;
 let errors = this.state.errors;

 switch (name) {
 case 'name':
 errors.name = value.length < 5 ? 'Nombre debe tener almenos 5 caracteres'
 : '';

```

```

 break;
 case 'lastname':
 errors.lastname = value.length < 5 ? 'Apellido debe tener al menos 5
caracteres' : '';
 break;
 case 'user':
 var emailPattern = /\S+@\S+\.\S+/;
 errors.user = emailPattern.test(value) ? '' : 'El usuario debe ser un
email';
 break;
 default:
 break;
}
this.setState({
 [name]: value,
 errors
});
}
}

```

Crearemos una función que valide que nuestro Formulario no tenga ningún error para habilitar el botón submit:

```

validarForm(errors) {
 let valid = true;
 Object.values(errors).forEach((val) => val.length > 0 && (valid = false));
 if(valid) {
 this.setState({
 disabled: false
 })
 } else {
 this.setState({
 disabled: true
 })
 }
}

```

Y así tendríamos un Formulario Controlado, en donde estamos haciendo validaciones por cada input, y deshabilitamos el input de submit hasta pasar todas las validaciones.

```

import React from 'react';

class Form extends React.Component {
 constructor() {
 super();
 this.state = {
 name: '',
 lastname: '',
 user: '',
 errors: {

```

```
 name: '',
 lastname: '',
 user: '',
 },
 disabled: true
 };

 this.handleChange = this.handleChange.bind(this);
}

validarForm(errors) {
 let valid = true;
 Object.values(errors).forEach((val) => val.length > 0 && (valid = false));
 if(valid) {
 this.setState({
 disabled: false
 })
 } else {
 this.setState({
 disabled: true
 })
 }
}

handleChange(e) {
 const { name, value } = e.target;
 let errors = this.state.errors;

 switch (name) {
 case 'name':
 errors.name = value.length < 5 ? 'Nombre debe tener 5 caracteres' : '';
 break;
 case 'lastname':
 errors.lastname = value.length < 5 ? 'Apellido debe tener 5 caracteres' :
 '';
 break;
 case 'user':
 var emailPattern = /\S+@\S+\.\S+/; // Expresion Regular para validar
Emails.
 errors.user = emailPattern.test(value) ? '' : 'El usuario debe ser un
email';
 break;
 default:
 break;
 }
 this.setState({
 [name]: value, // Sintaxis ES6 para actualizar la key correspondiente
 errors
 });

 this.validarForm(this.state.errors)
}
}
```

```

render() {
 return (
 <form style={{display: 'flex', flexDirection: 'column', width: '150px'}}>
 <input
 name="name"
 type="name"
 value={this.state.name}
 onChange={this.handleChange}
 placeholder="Nombre" />
 {!this.state.errors.name ? null : <div>{this.state.errors.name}</div>}
 <input
 name="lastname"
 type="name"
 value={this.state.lastname}
 onChange={this.handleChange}
 placeholder="Apellido" />
 {!this.state.errors.lastname ? null : <div>{this.state.errors.lastname}</div>}
 </div>
 <input
 name="user"
 type="name"
 value={this.state.user}
 onChange={this.handleChange}
 placeholder="Usuario" />
 {!this.state.errors.user ? null : <div>{this.state.errors.user}</div>}
 <input disabled={this.state.disabled} type="submit" value="Submit" />
 </form>
);
}
};

export default Form;

```

## Inputs Dinamicos en Forms Controlados

Otra de las cosas que podemos hacer en un Form controlado es tener Inputs Dinamicos, es decir, dependiendo de el usuario que utilice nuestro formulario, dinamicamente podemos crear inputs que se adapten a cada usuario. Por ejemplo, cuando en un Formulario agregamos miembros en nuestra familia, dinamicamente los inputs se van agregando.

Vamos paso a paso, utilizando Hooks. La idea de este ejemplo que haremos sera crear un simple input para que registre el nombre de una persona, y nos permita agregar familiares. Lo primero que haremos es crear el Form con los datos estaticos, en este caso el nombre de una persona.

```

import React from 'react';

function Form() {

 return (
 <form>
 <label htmlFor="nombre">Nombre:</label>

```

```
<input
 type="text"
 name="nombre"
/>
<input type="submit" value="Submit" />
</form>
);
};

export default Form;
```

Para crear los inputs dinamicos utilizaremos un arreglo de objetos. Cada objeto tendra los datos de cada familiar. En este caso tendremos solo la key `nombre`. Agregaremos un botón que nos permite agregar un nuevo familiar, es decir, un nuevo objeto al arreglo. Lo que generara un cambio de state y que el componente se actualice con los cambios. Tendremos que iterar sobre ese arreglo y por cada elemento renderizar un nuevo input, dependiendo la cantidad de keys que tenga en el objeto.

```
import React, { useState } from 'react';

function Form() {
 const [familiar, setFamiliar] = useState([
 { nombre: '' },
]);

 return (
 <form>
 <label htmlFor="nombre">Nombre:</label>
 <input
 type="text"
 name="nombre"
 />
 <input
 type="button"
 value="Agrega un Familiar"
 />
 {
 familiar.map((el, i) => (
 <div key={`persona-${i}`}>
 <label htmlFor={`nombre-${i}`}`>`Familiar ${i + 1}`</label>
 <input
 type="text"
 name={`nombre-${i}`}
 id={i}
 data-name="nombre"
 value={el.nombre}
 />
 </div>
))
 }
 <input type="submit" value="Submit" />
 </form>
);
}
```

```
};
export default Form;
```

## Agregando inputs

Por el momento estamos iterando sobre el state `familiar` para mostrar un input. Todavia no es dinamico. Para eso tenemos que dejar que el usuario agregue inputs haciendo click en el boton que creamos. Vamos a crear un metodo que agregue un nuevo objeto a nuestro state.

```
import React, { useState } from 'react';

function Form() {
 const modeloFamiliar = { nombre: '' }; // Creamos un modelo de Familiar para
 // poder usar este objeto para agregar al state cada vez que agregamos un familiar
 const [familiar, setFamiliar] = useState([
 { ...modeloFamiliar },
]);

 const agregaFamiliar = () => {
 setFamiliar([...familiar, { ...modeloFamiliar }]); // Hacemos una copia del
 // state que teniamos y agregamos un objeto nuevo al state.
 };

 return (
 <form>
 <label htmlFor="nombre">Nombre:</label>
 <input
 type="text"
 name="nombre"
 />
 <input
 type="button"
 value="Agrega un Familiar"
 onClick={agregaFamiliar}
 />
 {
 familiar.map((el, i) => (
 <div key={`persona-${i}`}>
 <label htmlFor={`nombre-${i}`}`>`Familiar ${i + 1}`</label>
 <input
 type="text"
 name={`nombre-${i}`}
 id={i}
 data-name="nombre"
 value={el.nombre}
 />
 </div>
))
 }
 <input type="submit" value="Submit" />
 </form>
```

```
);
};

export default Form;
```

## Controlando los inputs.

Ya tenemos nuestros inputs, ahora tenemos que, utilizando lo antes visto. Capturar los valores ingresados utilizando un estado.

```
function Form() {
 const modeloFamiliar = { nombre: '' };
 const [familiar, setFamiliar] = useState([
 { ...modeloFamiliar },
]);

 const [persona, setPersona] = useState({
 nombre: '',
 });

 const agregaFamiliar = () => {
 setFamiliar([...familiar, { ...modeloFamiliar }]);
 };

 // Este metodo lo usamos para capturar el valor ingresado en nuestro input
 estatico
 const handlePersonaChange = (e) => setPersona({
 ...persona,
 [e.target.name]: e.target.value,
 });

 // Este metodo lo usamos para controlar los inputs que se van creando
 dinamicamente
 // Primero hacemos una copia del estado `familiar`
 // Utilizamos el `id` y el atributo `data-name` que le asignamos a cada input
 para poder reconocerlos entre si.
 // Y por ultimo modificamos el state para actualizar segun los cambios que se
 realizaron en los inputs.
 const handleFamiliarChange = (e) => {
 const familiares = [...familiar];
 familiares[e.target.id][e.target.dataset.name] = e.target.value;
 setFamiliar(familiares);
 };

 return (
 <form>
 <label htmlFor="nombre">Nombre:</label>
 <input
 type="text"
 name="nombre"
 value={persona.nombre}
 onChange={handlePersonaChange}
 </input>
 </form>
);
}

export default Form;
```

```

 />
 <input
 type="button"
 value="Agrega un Familiar"
 onClick={agregaFamiliar}
 />
 {
 familiar.map((el, i) => (
 <div key={`persona-${i}`}
 <label htmlFor={`nombre-${i}`}>`Familiar ${i + 1}`</label>
 <input
 type="text"
 name={`nombre-${i}`}
 id={i}
 data-name="nombre"
 value={el.nombre}
 onChange={handleFamiliarChange} // Agregamos el metodo a cada input
 que generamos
 />
 </div>
))
 }
 <input type="submit" value="Submit" />
 </form>
);
};

export default Form;

```

Y con ese ultimo paso creamos un simple Formulario con inputs dinamicos.

## Uncontrolled Components

Siempre se recomienda trabajar con Componentes Controlados para la implementacion de Forms. La alternativa a esto son los Componentes no controlados, en donde los datos del Form son manejados por el propio DOM. En lugar de escribir eventos para las actualizaciones del state, podemos usar una referencia para obtener los valores del formulario desde el DOM, o utilizar los metodos de [document](#) como por ejemplo [document.querySelector\(\)](#) para obtener los valores del Formulario. En este tipo de formularios solo tenemos la posibilidad de validar en el momento en que hacemos el submit del form, cuando obtenemos los datos ingresados en el form.

```

// Utilizando Referencias

class UncontrolledForm extends React.Component {
 constructor() {
 super();
 this.handleSubmit = this.handleSubmit.bind(this);
 this.input = React.createRef();
 }
}

```

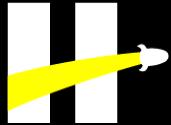
```
handleSubmit(e) {
 e.preventDefault();
 const name = this.input.current.value;
 console.log(name);
}

render() {
 return (
 <form onSubmit={this.handleSubmit}>
 <label>
 Name:
 <input type="text" ref={this.input} />
 </label>
 <input type="submit" value="Submit" />
 </form>
);
}
}

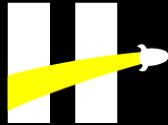
// Utilizando Selectores

function Uncontrolled() {

 function handleSubmit(e) {
 e.preventDefault();
 const username = document.querySelector('input[name=username]').value;
 const password = document.querySelector('input[name=password]').value;
 console.log(username, password);
 }
 render() {
 return (
 <form onSubmit={handleSubmit}>
 <input name="username" placeholder="username ej: toni@gmail.com" />
 <input name="password" type="password" placeholder="password" />
 <input type="submit" />
 </form>
);
 }
}
```

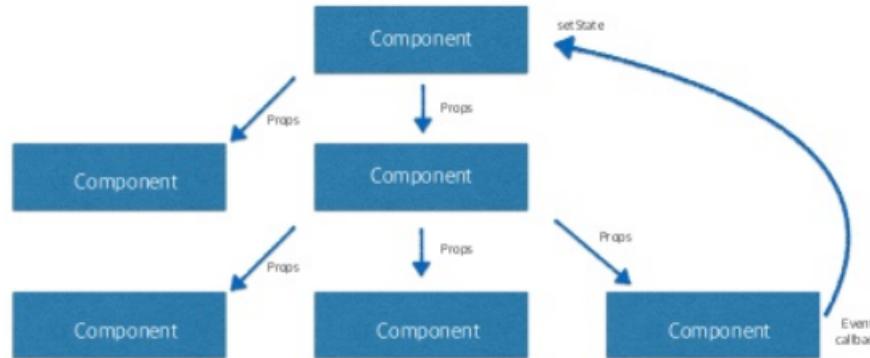


# Redux

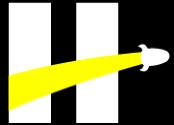


# Redux

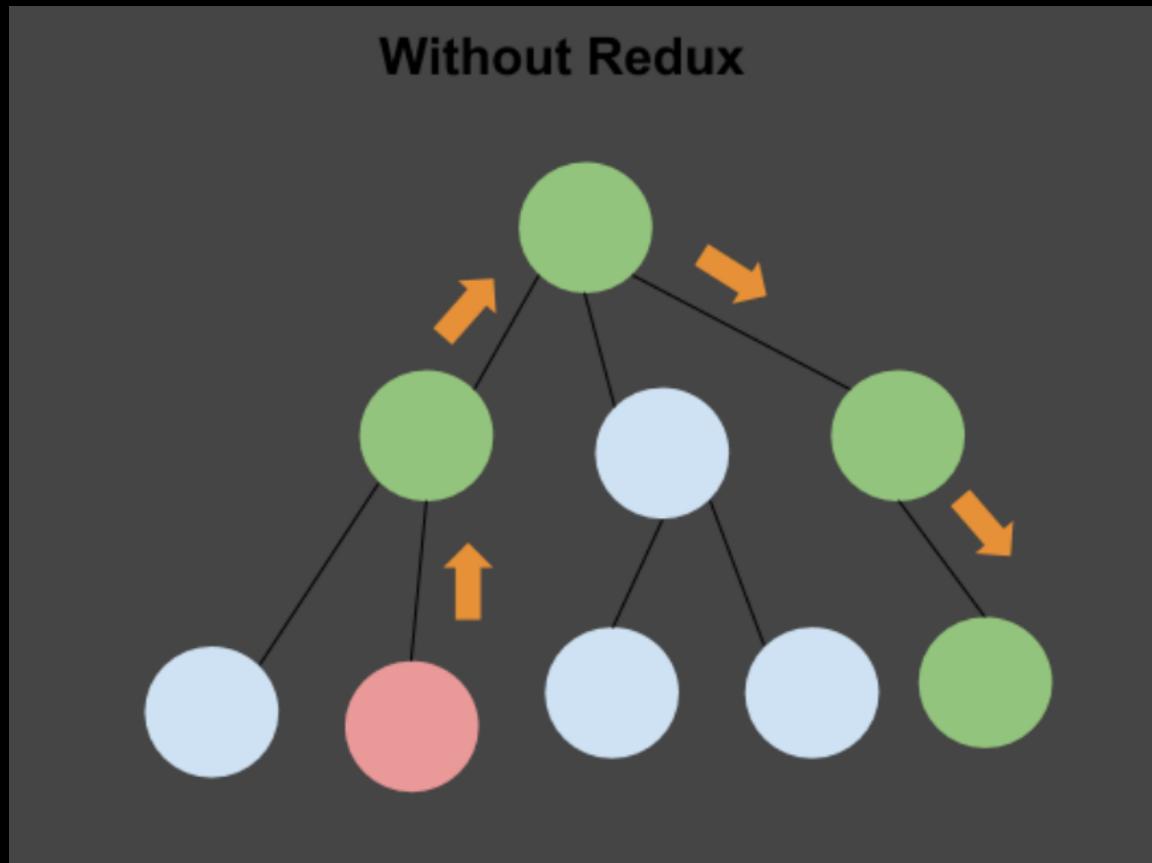
## Data flow



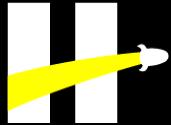
¿Se acuerdan del one-way Data Flow?



# Redux



Podría generar ciertos problemas.

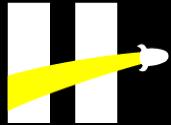


# Redux



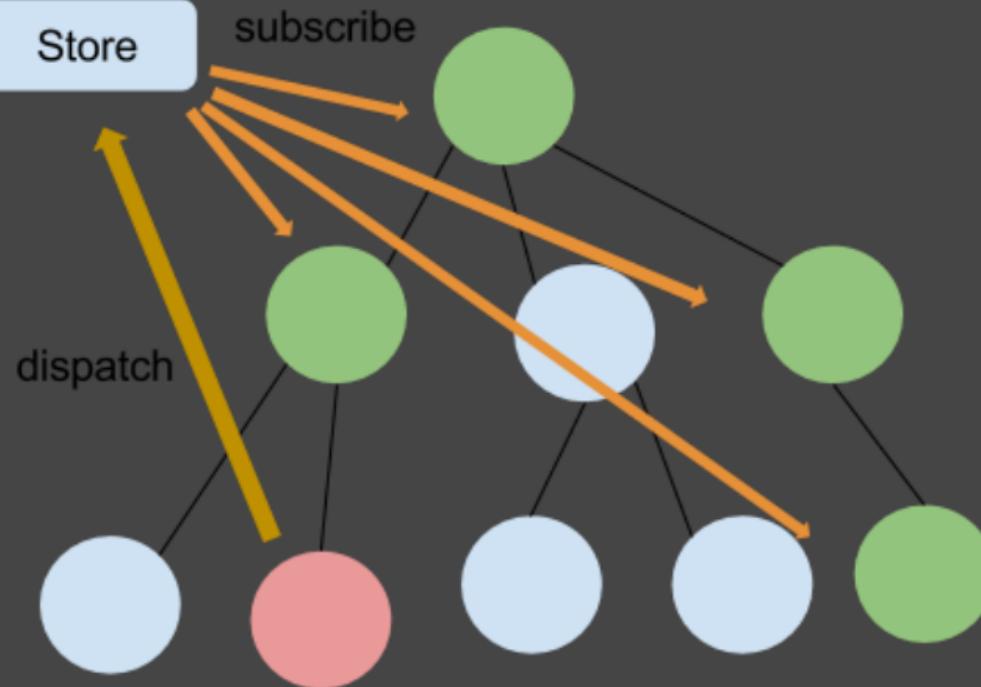
# Redux

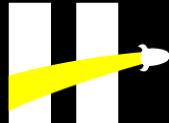
A Predictable State Container for JS Apps



# Redux

**With Redux**



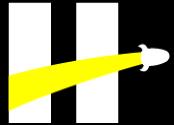


# Los tres principios de Redux

## Single source of truth

The **state** of your whole application is stored in an object tree within a single **store**.

```
1 console.log(store.getState())
2
3 /* Prints
4 {
5 visibilityFilter: 'SHOW_ALL',
6 todos: [
7 {
8 text: 'Consider using Redux',
9 completed: true,
10 },
11 {
12 text: 'Keep all state in a single tree',
13 completed: false
14 }
15]
16 }
17 */
```

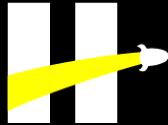


# Los tres principios de Redux

## State is read-only

The only way to change the state is to emit an **action**, an object describing what happened.

```
1
2 store.dispatch({
3 type: 'COMPLETE_TODO',
4 index: 1
5 })
6
7 store.dispatch({
8 type: 'SET_VISIBILITY_FILTER',
9 filter: 'SHOW_COMPLETED'
10 })
```

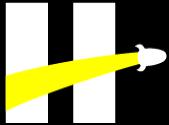


# Los tres principios de Redux

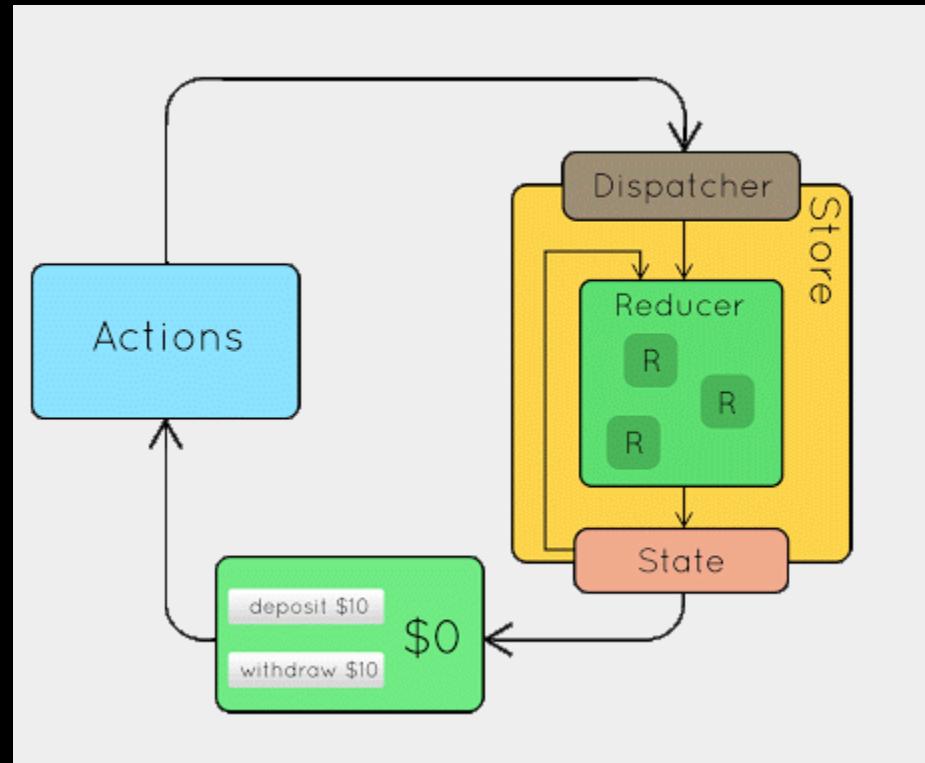
Changes are made with pure functions

To specify how the state tree is transformed by actions, you write pure reducers.

```
1 function visibilityFilter(state = 'SHOW_ALL', action) {
2 switch (action.type) {
3 case 'SET_VISIBILITY_FILTER':
4 return action.filter
5 default:
6 return state
7 }
8 }
9
10 function todos(state = [], action) {
11 switch (action.type) {
12 case 'ADD_TODO':
13 return [
14 ...state,
15 {
16 text: action.text,
17 completed: false
18 }
19]
20 case 'COMPLETE_TODO':
21 return state.map((todo, index) => {
22 if (index === action.index) {
23 return Object.assign({}, todo, {
24 completed: true
25 })
26 }
27 return todo
28 })
29 default:
30 return state
31 }
32 }
33
34 import { combineReducers, createStore } from 'redux'
35 const reducer = combineReducers({ visibilityFilter, todos })
36 const store = createStore(reducer)
```

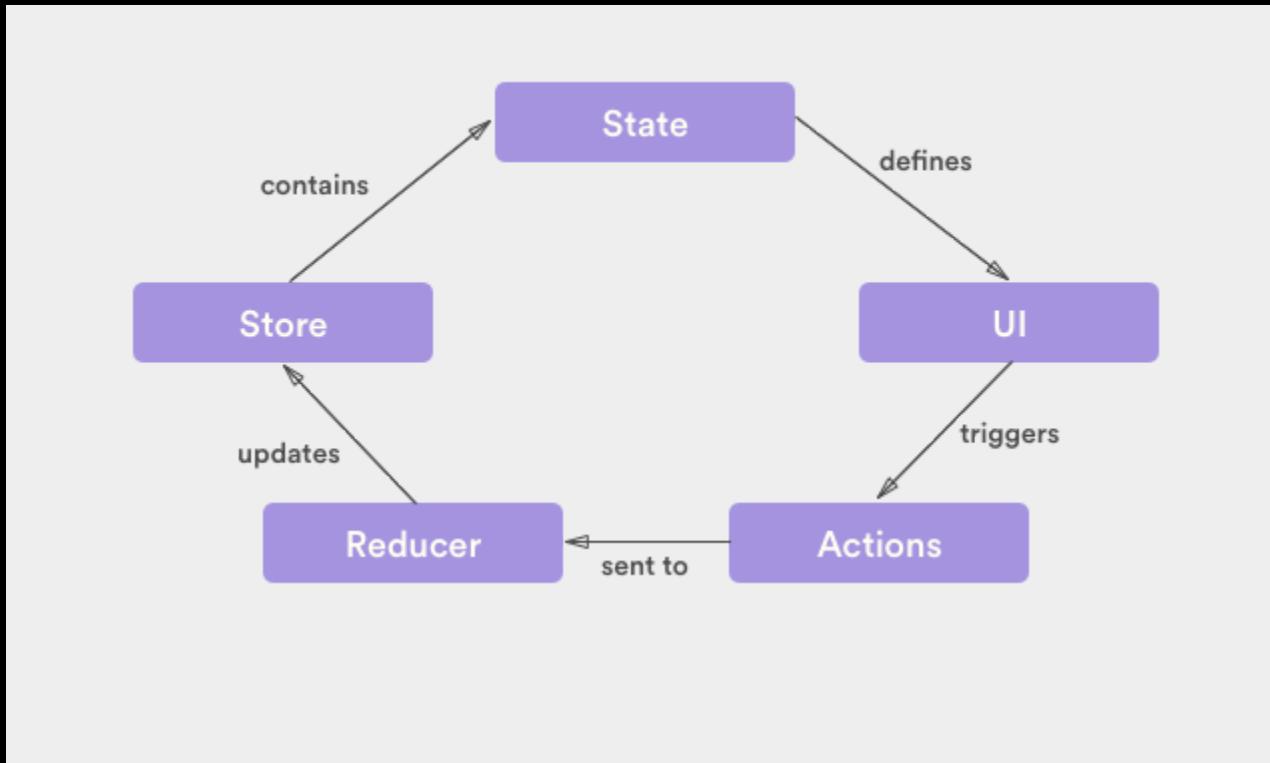


# Flow de Redux





# Flow de Redux





# Actions

```
1 const ADD_TODO = 'ADD_TODO'
2
3 {
4 type: ADD_TODO,
5 text: 'Build my first Redux app'
6 }
```

Las **acciones** son un bloque de información que envia datos desde tu aplicación a tu store. Son la *única* fuente de información para el store. Las envias al store usando `store.dispatch()`



# Actions Creators

```
1 function addTodo(text) {
2 return {
3 type: ADD_TODO,
4 text
5 }
6 }
```

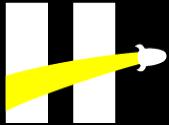
Los **creadores de acciones** son exactamente eso—funciones que crean acciones.



# Dispatch()

```
1 import * as actions from './actionsCreators';
2
3 store.dispatch(actions.increment());
4 store.dispatch(actions.addComment());
5 store.dispatch(actions.removeComment());
```

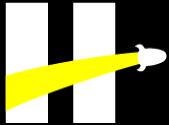
La función *dispatch* es la encargada de *enviar* las acciones al store.



# Reducers

Las **acciones** describen que *algo pasó*, pero no especifican cómo cambió el estado de la aplicación en respuesta. Esto es trabajo de los reducers.

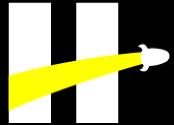
```
1 const addContact = (state, action) => {
2 switch (action.type) {
3 case 'NEW_CONTACT':
4 return {
5 ...state, contacts:
6 [...state.contacts, action.payload]
7 }
8 case 'UPDATE_CONTACT':
9 return {
10 // Handle contact update
11 }
12 case 'DELETE_CONTACT':
13 return {
14 // Handle contact delete
15 }
16 case 'EMPTY_CONTACT_LIST':
17 return {
18 // Handle contact list
19 }
20 default:
21 return state
22 }
23 }
```



# Reducers

Cuando una aplicación es muy grande, podemos dividir nuestros reducers en archivos separados y mantenerlos completamente independientes y controlando datos específicos.

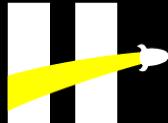
```
1 import { combineReducers } from 'redux'
2
3 const todoApp = combineReducers({
4 visibilityFilter,
5 todos
6 })
7
8 export default todoApp
```



# Store

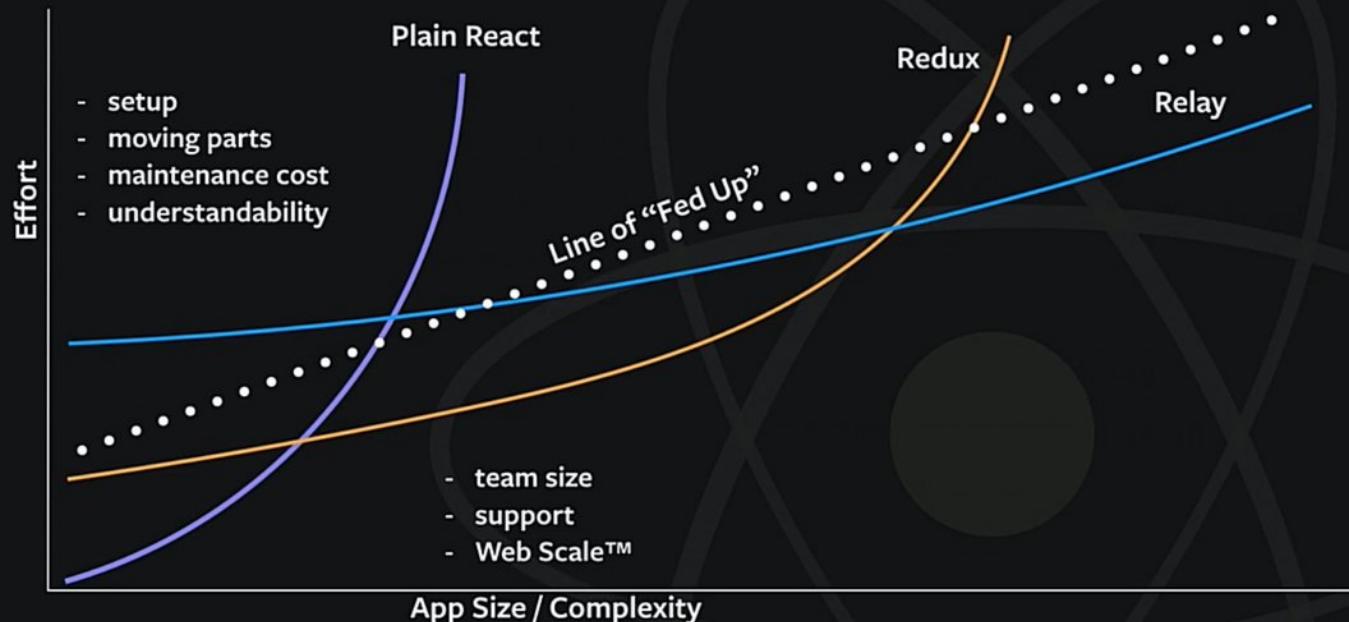
- Contiene el estado de la aplicación;
- Permite el acceso al estado via `getState()`;
- Permite que el estado sea actualizado via `dispatch(action)`;
- Registra los *listeners* via `subscribe(listener)`;
- Maneja la anulación del registro de los *listeners* via el retorno de la función de `subscribe(listener)`.

```
1 import { createStore } from 'redux'
2 import todoApp from './reducers'
3
4
5 let store = createStore(todoApp)
```



# Redux

## Recommendations



Note: not to scale. margin of error is large and not shown. axes might be logarithmic.

# Henry



Hacé click acá para dejar tu feedback sobre esta clase.

Hacé click acá completar el quiz teórico de esta lecture.

## Redux



# Redux

Redux is a predictable state container for JavaScript apps.

Redux es una librería que nos va a ayudar a mantener el estado *global* de nuestra aplicación.

Podemos usar Redux fuera de React, y React sin redux también. Pero ambas funcionan muy bien juntas, por esto la comunidad las adoptó rápidamente para usarlas juntas.

Si bien, podemos crear *containers* que mantengan el estado de sus *childrens*, lo que termina ocurriendo es que los Componentes *presentacionales* terminan demasiado acoplados a los *containers*, bajando su *resusabilidad*. Otro tema, es que en estos *containers* vamos a tener que escribir funciones que manejen el estado de varios Componentes, haciendo que este archivo se convierta en inmanejable de forma rápida. Justamente, **Redux** nos va a ayudar a resolver estos problemas con el paradigma que implementa.

## Los tres principios de Redux

Vamos a ir introduciendo cierta terminología específica de Redux, pueden leer el [glosario completo aca](#)

Única fuente de verdad (Single source of truth)

El **estado** de toda tu aplicación está guardado en un árbol en una sola **store**.

Esto hace que sea fácil crear apps universales, ya que el *estado* de tu servidor puede ser *serializado* fácilmente a todos los clientes sin esfuerzo extra. Además hace que sea más fácil *debuggear* e *inspeccionar* tu aplicación.

```
console.log(store.getState())

/* Prints
{
 visibilityFilter: 'SHOW_ALL',
 todos: [
 {
 text: 'Consider using Redux',
 completed: true,
 },
 {
 text: 'Keep all state in a single tree',
 completed: false
 }
]
}
*/
```

## Los Estados son sólo lectura (State is read-only)

La única forma de cambiar un estado es emitiendo una **acción**, que es un objeto que describe lo que ocurrió.

Esto asegura que ni la vista, ni ningún callback escriban directamente sobre el *estado*. En cambio, tienen que expresar una *intención* de transformar el estado. Como todas los cambios están *centralizados* y ocurren en un orden estricto, no vamos a tener que preocuparnos por qué cosas suceden primero (debido a la naturaleza *asincrónica*). Como las acciones son *objetos*, pueden ser logeados, serializados, guardados y pueden ser reproducidas en el futuro para *debuggear* o *testear* la aplicación.

```
store.dispatch({
 type: 'COMPLETE_TODO',
 index: 1
})

store.dispatch({
 type: 'SET_VISIBILITY_FILTER',
 filter: 'SHOW_COMPLETED'
})
```

## Los cambios se hacen con funciones puras (Changes are made with pure functions)

Para especificar cómo se transforma el *árbol de estado* se escriben funciones llamadas **reducers**.

Las funciones *reducers* son funciones puras, que toman el *estado anterior* y un *acción* y retornan el *nuevo estado*. Estas funciones deben retornar *nuevos objetos de estado* y no *mutar el estado anterior*. Como las *reducers* son sólo funciones, podemos manejar el orden en el que se ejecutan, pasar datos adicionales, o inclusive hacer *reducers* reutilizables para tareas comunes.

```

function visibilityFilter(state = 'SHOW_ALL', action) {
 switch (action.type) {
 case 'SET_VISIBILITY_FILTER':
 return action.filter
 default:
 return state
 }
}

function todos(state = [], action) {
 switch (action.type) {
 case 'ADD_TODO':
 return [
 ...state,
 {
 text: action.text,
 completed: false
 }
]
 case 'COMPLETE_TODO':
 return state.map((todo, index) => {
 if (index === action.index) {
 return Object.assign({}, todo, {
 completed: true
 })
 }
 return todo
 })
 default:
 return state
 }
}

import { combineReducers, createStore } from 'redux'
let reducer = combineReducers({ visibilityFilter, todos })
let store = createStore(reducer)

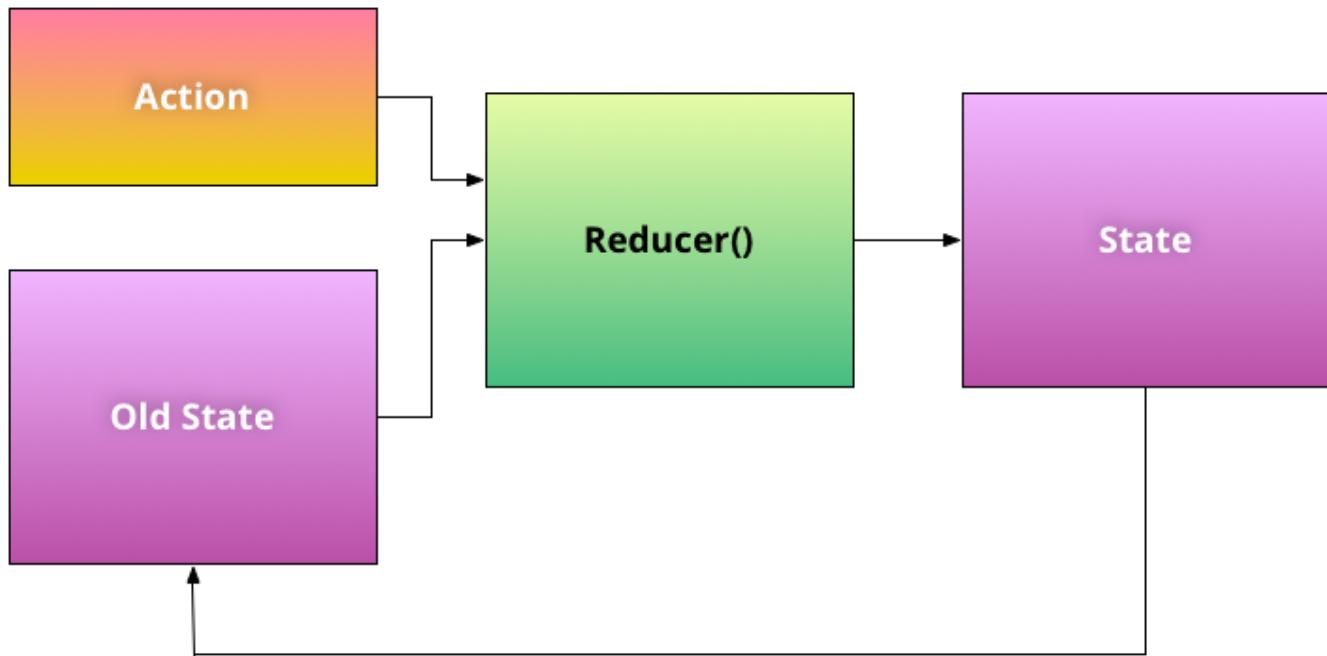
```

## Nuestra propia implementación de Redux

Para entender bien el concepto de *Redux*, vamos a crear nuestra propia mini-implementación del paradigma que utiliza. Luego vamos a usar la librería `redux`.

Para hacerlo, empecemos repasando las ideas principales que tenemos que implementar:

- Toda la data mantenida por nuestra aplicación tiene que estar contenida en una única estructura de datos llamada el `state` de nuestra app. Esta estructura de datos debe estar guardada en el `store`.
- Nuestra app lee el `state` desde nuestra `store`.
- El `store` no puede ser manipulado directamente por el usuario.
- Los usuarios disparan `acciones` que describen qué sucedió.
- Un *nuevo estado* es generado, resultado de combinar el *viejo estado* y la *acción* del usuario. Este proceso lo realiza una función llamada `reducir`.



## Reducers

Un *reducer* toma un *estado viejo (actual)* y una *acción* y devuelve un *estado nuevo*. Un *reducer* **debe ser una función pura**, es decir:

- No debe mutar directamente el estado.
- No debe usar datos que no hayan sido pasada por argumentos.

Los *reducers* siempre deben tratar el *estado actual* como **sólo lectura**. De hecho, el *reducer* **no cambia el estado**, si no que **devuelve un estado nuevo**. Para crear nuestro primer *reducer*, entonces, vamos a necesitar:

- Una **acción**, que nos define que hacer (opcionalmente con argumentos).
- El **estado**, que *guarda* toda la información de nuestro *app*.
- El **reducer per se** que recibe el *estado* y la *acción* y retorna el *nuevo estado*.

## Creando un Reducer

Empezemos por el *reducer* más simple posible, es el que devuelve el **estado actual**. Similar a la función [Identidad](#).

```

var reducer = function(state, action) {
 return state;
}

```

Para los siguientes ejemplos, consideraremos que el **estado** es un entero y que empieza en 0, podríamos llamarlo **counter**.

**En el común de los casos, los estados van a ser *sumamente* más complejos que sólo un entero!**

Para incrementar o decrementar en uno nuestro estado, vamos a necesitar definir una **acción**, para que el usuario pueda indicar que tiene la *intención* de cambiar el estado de nuestra *app*.

Una acción sólo necesita una propiedad **type**:

```
// incrementar en uno el counter
var INCREMENTAR_COUNTER = {
 type: 'INCREMENTAR_COUNTER'
}
// decrementar en uno el counter
var DECREMENTAR_COUNTER = {
 type: 'DECREMENTAR_COUNTER'
}
```

Ahora que tenemos estas acciones las usemos en nuestros *reducers*:

```
var reducer = function(state, action){
 switch(action.type){
 case "INCREMENTAR_COUNTER":
 return state + 1;
 case "DECREMENTAR_COUNTER":
 return state - 1;
 default:
 return state; // caso por defecto
 }
}
```

Listo! Ahora vamos a tener el *nuevo estado* retornado según qué acciones hayamos pasado al reducer.

Noten que dejamos el caso por **defecto** de tal modo que si no pasamos una acción *conocida*, el reducer vuelve el estado como estaba.

## Argumentos en las acciones

En el ejemplo anterior, la acción siempre incrementaba en uno. Pero a veces, para describir la acción es necesario contar una serie de parámetros, por ejemplo en el caso que queramos incrementar nuestro **counter**, pero en un valor mayor a uno. Primero vamos a definir una **acción** en la cual podamos incrementar **7 veces el counter**:

```
// incrementar en uno el counter
var INCREMENTAR_COUNTER = {
 type: 'INCREMENTAR_COUNTER'
}
// decrementar en uno el counter
var DECREMENTAR_COUNTER = {
 type: 'DECREMENTAR_COUNTER'
}
// incrementar el counter en n
var INCREMENTAR_7 = {
 type: 'INCREMENTAR_N',
```

```
 payload: 7
}
```

y la agregamos en nuestro **reducer**:

```
var reducer = function(state, action){
 switch(action.type){
 case "INCREMENTAR_COUNTER":
 return state + 1;
 case "DECREMENTAR_COUNTER":
 return state - 1;
 case "INCREMENTAR_N":
 return state + action.payload;
 default:
 return state; // caso por defecto
 }
}
```

En el ejemplo vemos que el type de acción es **INCREMENTAR\_N**, pero sólo hemos creado la acción de ese tipo, pero con el **payload** en un número fijo, para generar varias acciones de este tipo vamos a hacer un *generador de acciones*, que no es otra cosa que una **factory**:

```
function increment(n) {
 return {
 type: 'INCREMENTAR_N',
 payload: n
 }
}
```

Ahora podríamos invocar a nuestro reducer de la siguiente manera:

```
var state = 0;
reducer(state, increment(7)); // Incrementa siete // 7
reducer(state, INCREMENTAR_COUNTER) // incrementa de a uno. // 1
```

Si vemos el ejemplo, notamos que a pesar de pasar por los *reducers*, nuestro estado sigue siendo **0** siempre! Por lo tanto nos vemos en la necesidad de implementar un **Store**, que es la forma de guardar el estado en la filosofía **redux**:

```
class Store {
 constructor(estadoinicial, reducer) {
 this._state = estadoinicial;
 this.reducer = reducer;
 }
}
```

```

 getState(): {
 return this._state;
 }

 dispatch(action) {
 this._state = this.reducer(this._state, action);
 }
}

```

En Redux, generalmente, tenemos un Store y un top level reducer.

Veamos que contiene nuestra **Store**:

- Cuando la inicializamos vamos a pasarle un *Estado Inicial* y un *reducer*.
- `getState()` retorna el *Estado Actual*.
- Por último tenemos la función `dispatch` que recibe una acción e invoca al `reducer` con el estado actual y la acción, y actualiza el estado con lo que retorna el `reducer` (el nuevo estado).

Noten que `dispatch` no retorna nada, simplemente *actualiza* el estado de la aplicación. Esto es un concepto importante de `redux`: cuando *despachamos* una acción, lo hacemos y nos olvidamos. **Despachar una acción no es una manipulación directa del estado, y no devuelve el nuevo estado.**

Usando el Store

Veamos como podemos usar nuestro **Store**:

```

var store = new Store(0, reducer);
console.log(store.getState()); //0
store.dispatch(INCREMENTAR_7);
console.log(store.getState()); //7
store.dispatch(INCREMENTAR_COUNTER)
console.log(store.getState()); //8
store.dispatch(DECREMENTAR)
console.log(store.getState()); //7

```

Empezamos por crear una **Store** nueva y la guardamos, en este caso, en una variable también llamada `store`. Luego usaremos el objeto que guardamos para consultar el estado en el que se encuentra nuestra aplicación.

Pedirle al Store que nos notifique los cambios

Si vemos el ejemplo anterior, siempre que queremos ver el estado del **Store** tenemos que pedirle explicitamente. Sería interesante que nos enteremos que una acción fue despachada para que podamos responder si es necesario. Para esto vamos a implementar el [patrón Observador](#), es decir que vamos a *suscribir* un callback para que escuche por cambios.

Esto es lo que queremos hacer:

- Registrar una función *listener* usando `suscribe`.

- Cuando `dispatch` es invocada, iteraremos sobre todos los *listeners* que tenga subscriptos y los invocaremos, notificandolos que el Estado ha cambiado.

Para esto vamos a agregar la funcionalidad de `suscribe` y `unsubscribe` a nuestro `Store`:

```
class Store {
 constructor(estadoInicial, reducer) {
 this._state = estadoInicial;
 this.reducer = reducer;
 this._listeners = []; // Empezamos con un arreglo vacío.
 }

 getState() {
 return this._state;
 }

 dispatch(action) {
 this._state = this.reducer(this._state, action);
 }

 suscribe(listener) {
 _listeners.push(listener);
 return () => { // retorna una función unsubscribe
 this._listeners = this._listeners.filter(function(l){l !== listener});
 };
 }
}
```

La función `suscribe` recibe una función como `listener`, y retorna una función que al ser ejecutada, va a `filtrar` el `listener` con el que fue generada de la lista de `listeners`.

Ahora, para notificar a los `listeners` subscriptos, vamos a invocarlos cuando se ejecute el método `dispatch`:

```
dispatch(action) {
 this._state = this.reducer(this._state, action);
 this._listeners.forEach(function(listener){
 listener();
 })
}
```

Ahora que nos podemos suscribir para obtener novedades, probemos lo siguiente:

```
var store = new Store(0, reducer);
var unsubscribe = store.suscribe(function() {
 console.log(store.getState());
})

store.dispatch(INCREMENTAR_7); // --> 7
```

```

store.dispatch(INCREMENTAR_COUNTER); // --> 8
store.dispatch(DECREMENTAR); // --> 7

unsubscribe(); // dejamos de recibir notificaciones;

store.dispatch(DECREMENTAR);
console.log(store.getState()); // 6

```

Si entendiste lo anterior, entonces ya tienes una idea de las bases sobre las que siente **redux**. La librería implementa otros patrones más complejos, y resuelve problemas comunes que pueden llegar a aparecer.

## Instalando Redux

Redux viene en un paquete con el mismo nombre, así que para instalarlo podemos hacer `npm install redux react-redux`. Con esto instalamos la librería de redux en sí, y los **helpers** de `react-redux`, donde están las funciones `bindActionsCreator`, `connect`, etc..

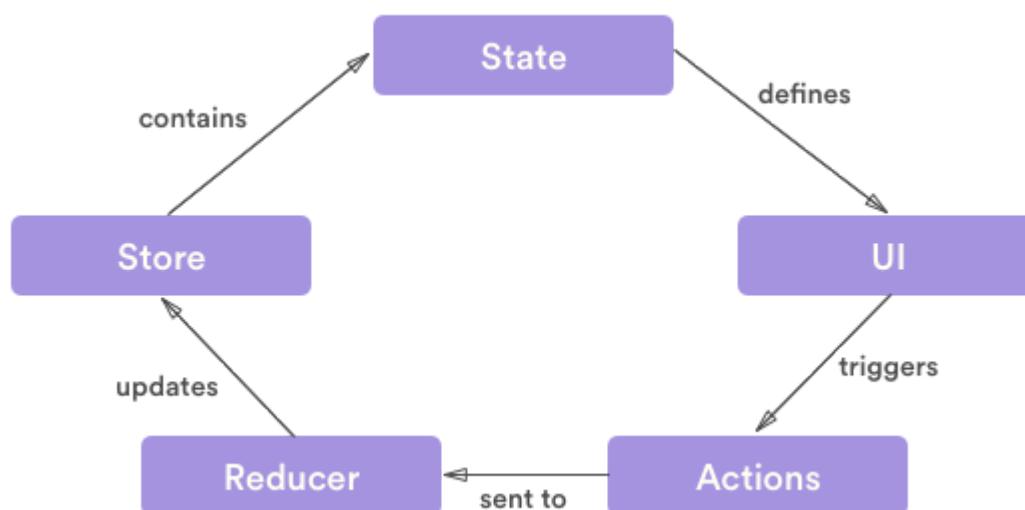
Redux se instala así, asumiendo que ya tenemos un proyecto de React funcionando, con su webpack configurado.

## Workflow de Redux

Lo primero que tenemos que incorporar para trabajar con Redux, es el workflow que nos propone.

Básicamente, nuestra aplicación debería funcionar siempre siguiente este ciclo de vida, en el medio van a aparecer elementos de la API de redux que vamos a ir explicando.

Se podría considerar a **redux** como una implementación del patrón **flux** para react, también se podría considerar cómo un patrón por si mismo. ( De hecho, [ni sus autores se ponen de acuerdo en eso](#) ) Lo cierto es que está influenciado por el patrón **flux**, usando por facebook.



1. El **árbol de Estado** define la UI y las acciones posibles a través de **props**.
2. Acciones realizadas por los usuarios son enviadas a un **action creator** que las normaliza.
3. Estas acciones normalizadas son pasadas a un **reducer**, que es donde se ejecuta el código que contiene la lógica de la acción.

4. El reducer crea un nuevo Estado y lo devuelve (`dispatches it`) al `Store`.

5. La UI es actualizada acorde al nuevo estado.

## Acciones

Las **acciones** representan una *intención* de cambiar el estado de nuestra `store`. Las *acciones* son las **únicas** fuente de información que llega a nuestras `stores`.

Las **acciones** son **objetos JavaScript planos**, deben tener una propiedad llamada `type`, que indica o describa la acción que se realiza. Por convención los `types` debe estar escritos como `string constants`, es decir, todo en mayúsculas y con SNAKE\_CASE. Además del `type`, las acciones van a contener cualquier otra propiedad que necesitemos para su funcionamiento. Se recomienda que las acciones tengan la menor cantidad de propiedades posibles.

Podemos ver algunas recomendaciones de cómo diseñar nuestras acciones en [esta guía](#).

Cuando tu app empieze a crecer, es buena idea separar las acciones en distintos módulos (archivos).

```
// incrementas likes
const INCREMENT_LIKES = {
 type: 'INCREMENT_LIKES',
 index: 4,
}

// agregar comentarios
const ADD_COMMENT = {
 type: 'ADD_COMMENT',
 postId: 'X4Yb4',
 author: 'Toni',
 comment: 'Esto es una acción en particular.',
}

// remover comentarios
const REMOVE_COMMENT = {
 type: 'REMOVE_COMMENT',
 postId: 'X4Yb5',
 index: 5,
}
```

## Action Creators

Los **actions creators** son funciones que **crean** acciones, o sea que *retornan* un objeto que representa una acción. Es fácil confundir los términos *action* y *action creator* así que hay que usarlos con cuidado.

```
// incrementas likes
export function increment(index) {
 return {
 type: 'INCREMENT_LIKES',
 index
}
```

```

}
// agregar comentarios
export function addComment(postId, author, comment) {
 return {
 type: 'ADD_COMMENT',
 postId,
 author,
 comment
 }
}
// remover comentarios
export function removeComment(postId, index) {
 return {
 type: 'REMOVE_COMMENT',
 postId,
 index
 }
}

```

## Dispatch

Las *acciones* tienen que ser enviadas al *Store* para que surgen efecto. La función `dispatch` base es un método de `store`, esta manda una acción de forma sincrónica a los reductores del store, junto con el estado previo retornado por el store, para calcular el nuevo estado. Espera que las acciones que les pasemos sean objetos planos, listas para ser consumidas por los reducers.

También se puede envolver a los dispatchers en una serie de `Middlewares`, esto permite que los dispatchers puedan manejar *acciones asincrónicas*, además de poder transformar, demorar, ignorar o interpretar acciones antes de pasarla al siguiente Middleware.

When we use redux with react, the dispatcher functions are bound to the component

## Reducers

Las *acciones* describen que algo sucedió en nuestra app, pero no especifican *cómo* esta acción impacta en el estado actual. Saber eso es el trabajo de los **reducers**.

Un *reducer* es una función que recibe el estado previo de un *Store* y una acción y retorna el nuevo estado. Justamente se llama reducer, porque toma al estado como una *acumulación* de acciones. Los reducers tienen que ser siempre funciones **puras**, estas son cosas que **nunca** deberías hacer en un reducer:

- Mutar sus argumentos.
- Realizar cosas que tengan efectos secundarios, como llamadas a APIs, o routeo.
- Llamar a funciones no puras adentro, como `Math.random()` o `Date.now()`

**Given the same arguments, a reducer should calculate the next state and return it. No surprises.  
No side effects. No API calls. Just a calculation.**

```

function posts(state = [], action) {
 switch(action.type){

```

```

 case 'INCREMENT_LIKES':
 console.log('increment Likes');
 const i = action.index;
 return [
 ...state.slice(0,i), // antes del que estamos actualizando
 {...state[i], likes: state[i].likes + 1},
 ...state.slice(i+1)// despues del actualizado
]
 default:
 return state;

 }
}

export default posts;

```

Cuando la app es grande, podemos poner cada reducer en archivos separados, agrupandolos según los datos que manejen, haciendolos independientes entre ellos. Para hacer esto, redux nos ofrece la funcionalidad de `combineReducers()`, lo que hace es convertir un objeto que tiene varios reducers en una sola función reducers que los contiene, y que puede ser pasada a `createStore`.

```

// Redux sólo puede tener un reducer.
// por eso vamos a tener el Root Reducer
// donde vamos a incorporar los demás

import { combineReducers } from 'redux';
import { routerReducer } from 'react-router-redux';

import posts from './posts.js';
import comments from './comments.js';

const rootReducer = combineReducers({posts, comments, routing: routerReducer})

export default rootReducer;

```

## Store

La **Store** tiene las siguientes responsabilidades:

- Mantiene el estado de la aplicación.
- Permite el acceso al estado a través de `getState()`.
- Permite actualizar el estado a través de `dispatch(action)`.
- Registra *listeners* con `subscribe(listener)`.
- Maneja la desuscripción de *listeners*.

Hay que notar que vamos a tener exactamente *una* Store por cada aplicación que hagamos usando Redux. Para crear una Store, vamos a usar la función `createStore` que recibe un reducer como argumento, y opcionalmente el *estado inicial* de la app.

```
import { createStore, compose } from 'redux';
import { syncHistoryWithStore} from 'react-router-redux';
import { browserHistory } from 'react-router';

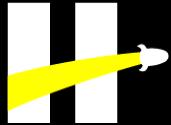
// Import root reducer
import rootReducer from './reducers/index.js';

import comments from './data/comments.js';
import posts from './data/posts.js';

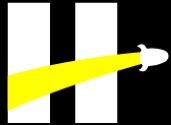
const defaultState = {
 posts,
 comments
}

const store = createStore(rootReducer, defaultState);
```

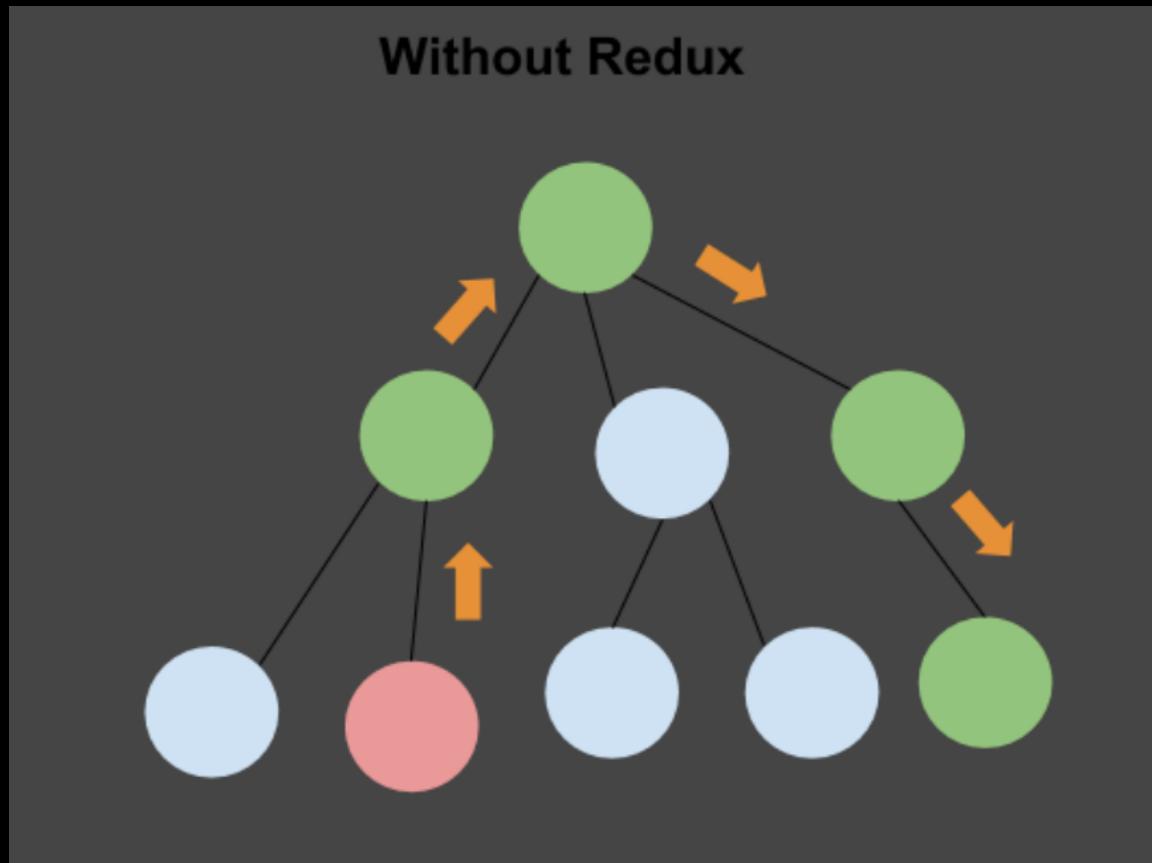
HENRY



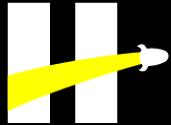
# Redux



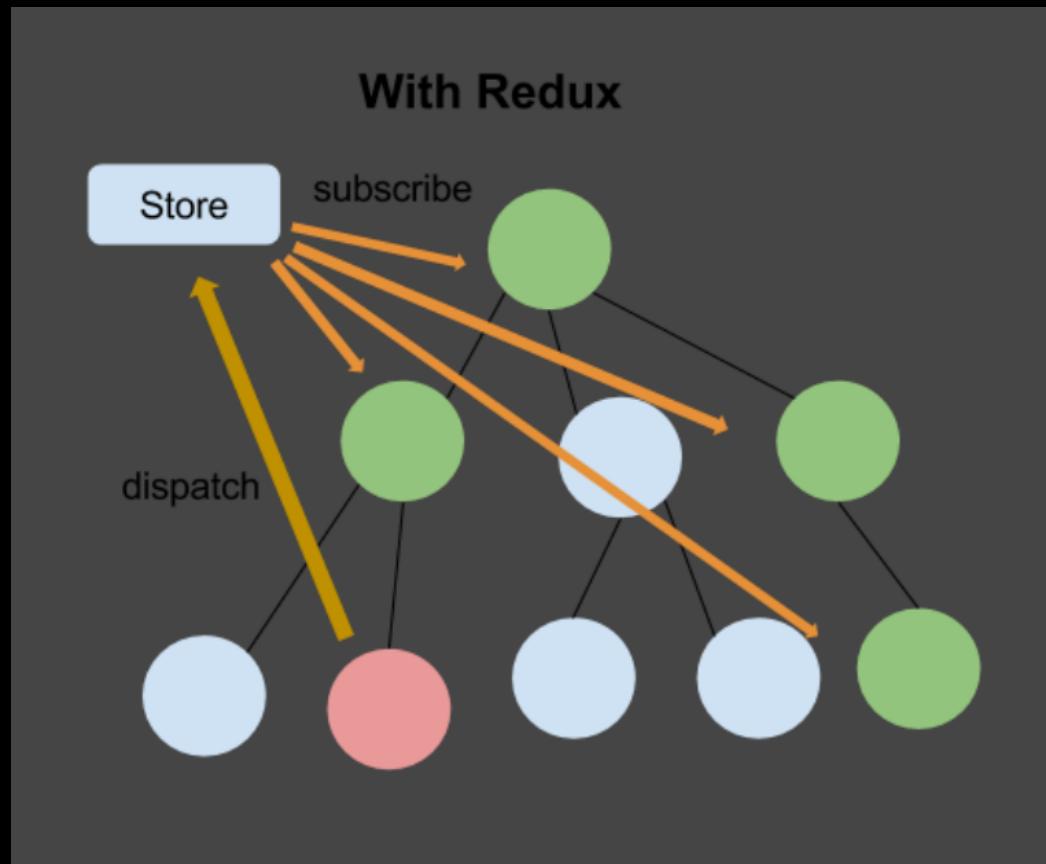
# Redux



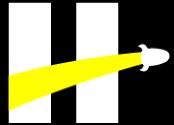
Podría generar ciertos problemas.



# Redux



¿Qué Componentes *suscribimos* al store?



# Componentes Presentacionales VS Containers

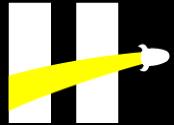
Tambien llamados smarts y dumbs components.

Smart:

- Cómo funcionan las cosas.
- Poco o nada de DOM.
- Sin estilos
- Provee datos
- Invoca las acciones

Dumb:

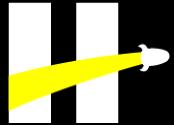
- Cómo se ven las cosas.
- Trabaja con sus props.
- Generalmente no tienen estado propio. (Algunos sí!)



# Componentes Presentacionales VS Containers

Separarlos de esta manera traer varias ventajas:

- Separa los problemas de la lógica de lo presentacional.
- Obtenemos componentes reutilizables (los presentacionales mayormente).
- Localizamos la complejidad en los containers



# Componentes Presentacionales VS Containers

|                    | Presentacionales                           | Containers                                                    |
|--------------------|--------------------------------------------|---------------------------------------------------------------|
| Próposito          | Cómo se ven las cosas<br>(markup, estilos) | Cómo funcionan las cosas<br>(traer datos, actualizar estados) |
| Sabe de Redux      | NO                                         | SI                                                            |
| Para leer datos    | Lee de props                               | Se suscribe a los estados de Redux                            |
| Para cambiar datos | Invoca callbacks de sus props              | Envía acciones a Redux                                        |



# React-Redux

Vamos a conectar a redux a los Containers.



```
1 import { bindActionCreators } from 'redux';
2 import { connect } from 'react-redux';
3 import * as actionsCreators from '../actions/actionsCreators.js';
4
5 import Main from './Main.js';
6
7 function mapStateToProps(state) {
8 return {
9 posts: state.posts,
10 comments: state.comments
11 }
12 }
13
14 function mapDispatchToProps(dispatch) {
15 return bindActionCreators(actionsCreators, dispatch);
16 }
17
18 const App = connect(mapStateToProps, mapDispatchToProps)(Main);
19
20 export default App;
```

**mapStateToProps:** Recibe el estado de la aplicación y lo mapea a props de react.

**mapDispatchToProps:** Recibe el método dispatch y retorna callbacks props que vamos a poder pasar a los Componentes presentacionales



# React-Redux

Todos los Componentes Containers deben tener acceso al Store para que puedan suscribirse a ella.



```
1 ...
2 import { Provider } from 'react-redux'; //Bindings from redux and React
3 import store, { history } from './store.js';
4
5 const router = (
6 <Provider store={store}>
7 <Router history={ history }>
8 <Route path="/" component={App}>
9 <IndexRoute component={PhotoGrid}/>
10 <Route path="view/:postId" component={Single}/>
11 </Route>
12 </Router>
13 </Provider>
14)
15 ...
```

Lo que nos recomienda redux es usar un Componente especial de react-redux llamado `<Provider>` que **mágicamente** hace que el Store esté disponible para todos los Container de nuestra app, sin pasarla explícitamente.

# Henry



Hacé click acá para dejar tu feedback sobre esta clase.

Hacé click acá completar el quiz teórico de esta lecture.

## Redux



# Redux

Redux is a predictable state container for JavaScript apps.

Redux es una librería que nos va a ayudar a mantener el estado *global* de nuestra aplicación.

Podemos usar Redux fuera de React, y React sin redux también. Pero ambas funcionan muy bien juntas, por esto la comunidad las adoptó rápidamente para usarlas juntas.

Si bien, podemos crear *containers* que mantengan el estado de sus *childrens*, lo que termina ocurriendo es que los Componentes *presentacionales* terminan demasiado acoplados a los *containers*, bajando su *resusabilidad*. Otro tema, es que en estos *containers* vamos a tener que escribir funciones que manejen el estado de varios Componentes, haciendo que este archivo se convierta en inmanejable de forma rápida. Justamente, **Redux** nos va a ayudar a resolver estos problemas con el paradigma que implementa.

## Los tres principios de Redux

Vamos a ir introduciendo cierta terminología específica de Redux, pueden leer el [glosario completo aca](#)

Única fuente de verdad (Single source of truth)

El **estado** de toda tu aplicación está guardado en un árbol en una sola **store**.

Esto hace que se fácil crear apps universale, ya que el *estado* de tu servidor puede ser *serializado* fácilmente a todos los clientes sin esfuerzo extra. Además hace que sea más fácil *debuggear* e *inspeccionar* tu aplicación.

```
console.log(store.getState())

/* Prints
{
 visibilityFilter: 'SHOW_ALL',
 todos: [
 {
 text: 'Consider using Redux',
 completed: true,
 },
 {
 text: 'Keep all state in a single tree',
 completed: false
 }
]
}
*/
```

## Los Estados son sólo lectura (State is read-only)

La única forma de cambiar un estado es emitiendo una **acción**, que es un objeto que describe lo que ocurrió.

Esto asegura que ni la vista, ni ningún callback escriban directamente sobre el *estado*. En cambio, tienen que expresar una *intención* de transformar el estado. Como todas los cambios están *centralizados* y ocurren en un orden estricto, no vamos a tener que preocuparnos por qué cosas suceden primero (debido a la naturaleza *asincrónica*). Como las acciones son *objetos*, pueden ser logeados, serializados, guardados y pueden ser reproducidas en el futuro para *debuggear* o *testear* la aplicación.

```
store.dispatch({
 type: 'COMPLETE_TODO',
 index: 1
})

store.dispatch({
 type: 'SET_VISIBILITY_FILTER',
 filter: 'SHOW_COMPLETED'
})
```

## Los cambios se hacen con funciones puras (Changes are made with pure functions)

Para especificar cómo se transforma el *árbol de estado* se escriben funciones llamadas **reducers**.

Las funciones *reducers* son funciones puras, que toman el *estado anterior* y un *acción* y retornan el *nuevo estado*. Estas funciones deben retornar *nuevos objetos de estado* y no *mutar el estado anterior*. Como las *reducers* son sólo funciones, podemos manejar el orden en el que se ejecutan, pasar datos adicionales, o inclusive hacer *reducers* reutilizables para tareas comunes.

```

function visibilityFilter(state = 'SHOW_ALL', action) {
 switch (action.type) {
 case 'SET_VISIBILITY_FILTER':
 return action.filter
 default:
 return state
 }
}

function todos(state = [], action) {
 switch (action.type) {
 case 'ADD_TODO':
 return [
 ...state,
 {
 text: action.text,
 completed: false
 }
]
 case 'COMPLETE_TODO':
 return state.map((todo, index) => {
 if (index === action.index) {
 return Object.assign({}, todo, {
 completed: true
 })
 }
 return todo
 })
 default:
 return state
 }
}

import { combineReducers, createStore } from 'redux'
let reducer = combineReducers({ visibilityFilter, todos })
let store = createStore(reducer)

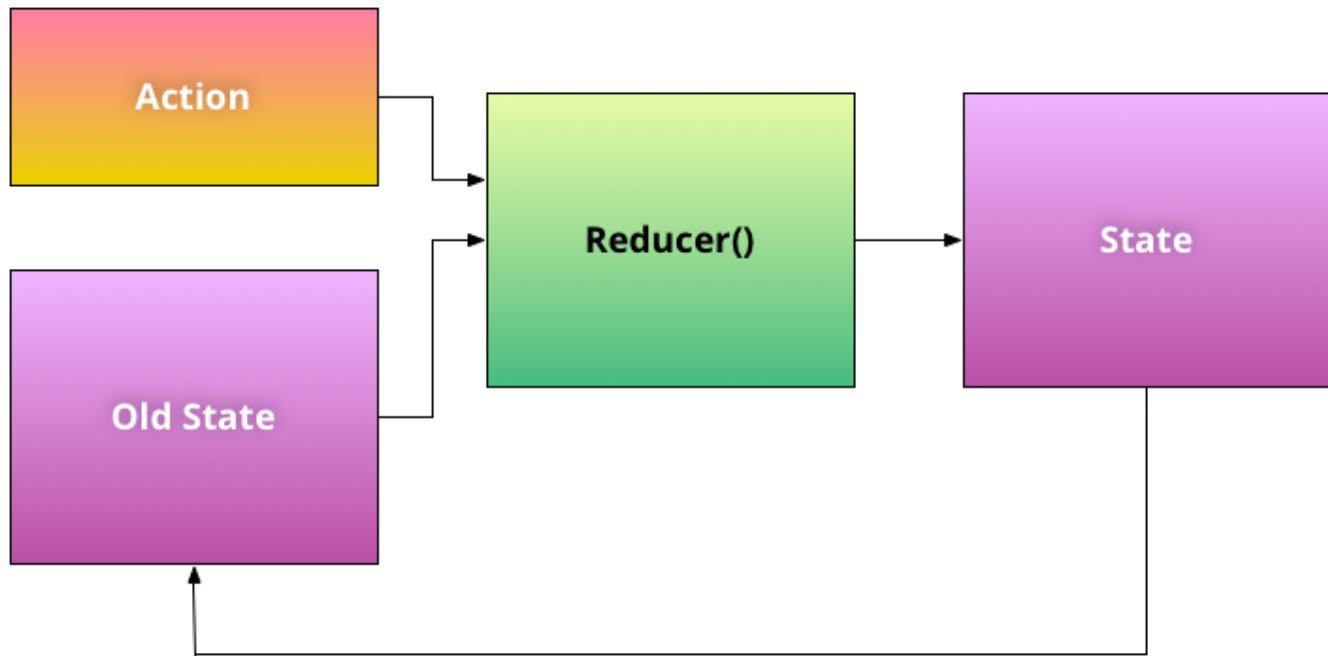
```

## Nuestra propia implementación de Redux

Para entender bien el concepto de *Redux*, vamos a crear nuestra propia mini-implementación del paradigma que utiliza. Luego vamos a usar la librería `redux`.

Para hacerlo, empecemos repasando las ideas principales que tenemos que implementar:

- Toda la data mantenida por nuestra aplicación tiene que estar contenida en una única estructura de datos llamada el `state` de nuestra app. Esta estructura de datos debe estar guardada en el `store`.
- Nuestra app lee el `state` desde nuestra `store`.
- El `store` no puede ser manipulado directamente por el usuario.
- Los usuarios disparan `acciones` que describen qué sucedió.
- Un *nuevo estado* es generado, resultado de combinar el *viejo estado* y la *acción* del usuario. Este proceso lo realiza una función llamada `reducir`.



## Reducers

Un *reducer* toma un *estado viejo (actual)* y una *acción* y devuelve un *estado nuevo*. Un *reducer* **debe ser una función pura**, es decir:

- No debe mutar directamente el estado.
- No debe usar datos que no hayan sido pasada por argumentos.

Los *reducers* siempre deben tratar el *estado actual* como **sólo lectura**. De hecho, el *reducer* **no cambia el estado**, si no que **devuelve un estado nuevo**. Para crear nuestro primer *reducer*, entonces, vamos a necesitar:

- Una **acción**, que nos define que hacer (opcionalmente con argumentos).
- El **estado**, que *guarda* toda la información de nuestro *app*.
- El **reducer per se** que recibe el *estado* y la *acción* y retorna el *nuevo estado*.

## Creando un Reducer

Empezemos por el *reducer* más simple posible, es el que devuelve el **estado actual**. Similar a la función [Identidad](#).

```

var reducer = function(state, action) {
 return state;
}

```

Para los siguientes ejemplos, consideraremos que el **estado** es un entero y que empieza en 0, podríamos llamarlo **counter**.

**En el común de los casos, los estados van a ser *sumamente* más complejos que sólo un entero!**

Para incrementar o decrementar en uno nuestro estado, vamos a necesitar definir una **acción**, para que el usuario pueda indicar que tiene la *intención* de cambiar el estado de nuestra app.

Una acción sólo necesita una propiedad **type**:

```
// incrementar en uno el counter
var INCREMENTAR_COUNTER = {
 type: 'INCREMENTAR_COUNTER'
}
// decrementar en uno el counter
var DECREMENTAR_COUNTER = {
 type: 'DECREMENTAR_COUNTER'
}
```

Ahora que tenemos estas acciones las usemos en nuestros *reducers*:

```
var reducer = function(state, action){
 switch(action.type){
 case "INCREMENTAR_COUNTER":
 return state + 1;
 case "DECREMENTAR_COUNTER":
 return state - 1;
 default:
 return state; // caso por defecto
 }
}
```

Listo! Ahora vamos a tener el *nuevo estado* retornado según qué acciones hayamos pasado al reducer.

Noten que dejamos el caso por **defecto** de tal modo que si no pasamos una acción *conocida*, el reducer vuelve el estado como estaba.

## Argumentos en las acciones

En el ejemplo anterior, la acción siempre incrementaba en uno. Pero a veces, para describir la acción es necesario contar una serie de parámetros, por ejemplo en el caso que queramos incrementar nuestro **counter**, pero en un valor mayor a uno. Primero vamos a definir una **acción** en la cual podamos incrementar **7 veces el counter**:

```
// incrementar en uno el counter
var INCREMENTAR_COUNTER = {
 type: 'INCREMENTAR_COUNTER'
}
// decrementar en uno el counter
var DECREMENTAR_COUNTER = {
 type: 'DECREMENTAR_COUNTER'
}
// incrementar el counter en n
var INCREMENTAR_7 = {
 type: 'INCREMENTAR_N',
```

```
 payload: 7
}
```

y la agregamos en nuestro **reducer**:

```
var reducer = function(state, action){
 switch(action.type){
 case "INCREMENTAR_COUNTER":
 return state + 1;
 case "DECREMENTAR_COUNTER":
 return state - 1;
 case "INCREMENTAR_N":
 return state + action.payload;
 default:
 return state; // caso por defecto
 }
}
```

En el ejemplo vemos que el type de acción es **INCREMENTAR\_N**, pero sólo hemos creado la acción de ese tipo, pero con el **payload** en un número fijo, para generar varias acciones de este tipo vamos a hacer un *generador de acciones*, que no es otra cosa que una **factory**:

```
function increment(n) {
 return {
 type: 'INCREMENTAR_N',
 payload: n
 }
}
```

Ahora podríamos invocar a nuestro reducer de la siguiente manera:

```
var state = 0;
reducer(state, increment(7)); // Incrementa siete // 7
reducer(state, INCREMENTAR_COUNTER) // incrementa de a uno. // 1
```

Si vemos el ejemplo, notamos que a pesar de pasar por los *reducers*, nuestro estado sigue siendo **0** siempre! Por lo tanto nos vemos en la necesidad de implementar un **Store**, que es la forma de guardar el estado en la filosofía **redux**:

```
class Store {
 constructor(estadoinicial, reducer) {
 this._state = estadoinicial;
 this.reducer = reducer;
 }
}
```

```

 getState(): {
 return this._state;
 }

 dispatch(action) {
 this._state = this.reducer(this._state, action);
 }
}

```

En Redux, generalmente, tenemos un Store y un top level reducer.

Veamos que contiene nuestra **Store**:

- Cuando la inicializamos vamos a pasarle un *Estado Inicial* y un *reducer*.
- `getState()` retorna el *Estado Actual*.
- Por último tenemos la función `dispatch` que recibe una acción e invoca al `reducer` con el estado actual y la acción, y actualiza el estado con lo que retorna el `reducer` (el nuevo estado).

Noten que `dispatch` no retorna nada, simplemente *actualiza* el estado de la aplicación. Esto es un concepto importante de `redux`: cuando *despachamos* una acción, lo hacemos y nos olvidamos. **Despachar una acción no es una manipulación directa del estado, y no devuelve el nuevo estado.**

Usando el Store

Veamos como podemos usar nuestro **Store**:

```

var store = new Store(0, reducer);
console.log(store.getState()); //0
store.dispatch(INCREMENTAR_7);
console.log(store.getState()); //7
store.dispatch(INCREMENTAR_COUNTER)
console.log(store.getState()); //8
store.dispatch(DECREMENTAR)
console.log(store.getState()); //7

```

Empezamos por crear una **Store** nueva y la guardamos, en este caso, en una variable también llamada `store`. Luego usaremos el objeto que guardamos para consultar el estado en el que se encuentra nuestra aplicación.

Pedirle al Store que nos notifique los cambios

Si vemos el ejemplo anterior, siempre que queremos ver el estado del **Store** tenemos que pedirle explicitamente. Sería interesante que nos enteremos que una acción fue despachada para que podamos responder si es necesario. Para esto vamos a implementar el [patrón Observador](#), es decir que vamos a *suscribir* un callback para que escuche por cambios.

Esto es lo que queremos hacer:

- Registrar una función *listener* usando `suscribe`.

- Cuando `dispatch` es invocada, iteraremos sobre todos los *listeners* que tenga subscriptos y los invocaremos, notificandolos que el Estado ha cambiado.

Para esto vamos a agregar la funcionalidad de `suscribe` y `unsubscribe` a nuestro `Store`:

```
class Store {
 constructor(estadoInicial, reducer) {
 this._state = estadoInicial;
 this.reducer = reducer;
 this._listeners = []; // Empezamos con un arreglo vacío.
 }

 getState() {
 return this._state;
 }

 dispatch(action) {
 this._state = this.reducer(this._state, action);
 }

 suscribe(listener) {
 _listeners.push(listener);
 return () => { // retorna una función unsubscribe
 this._listeners = this._listeners.filter(function(l){l !== listener});
 };
 }
}
```

La función `suscribe` recibe una función como `listener`, y retorna una función que al ser ejecutada, va a `filtrar` el `listener` con el que fue generada de la lista de `listeners`.

Ahora, para notificar a los `listeners` subscriptos, vamos a invocarlos cuando se ejecute el método `dispatch`:

```
dispatch(action) {
 this._state = this.reducer(this._state, action);
 this._listeners.forEach(function(listener){
 listener();
 })
}
```

Ahora que nos podemos suscribir para obtener novedades, probemos lo siguiente:

```
var store = new Store(0, reducer);
var unsubscribe = store.suscribe(function() {
 console.log(store.getState());
})

store.dispatch(INCREMENTAR_7); // --> 7
```

```

store.dispatch(INCREMENTAR_COUNTER); // --> 8
store.dispatch(DECREMENTAR); // --> 7

unsubscribe(); // dejamos de recibir notificaciones;

store.dispatch(DECREMENTAR);
console.log(store.getState()); // 6

```

Si entendiste lo anterior, entonces ya tienes una idea de las bases sobre las que siente **redux**. La librería implementa otros patrones más complejos, y resuelve problemas comunes que pueden llegar a aparecer.

## Instalando Redux

Redux viene en un paquete con el mismo nombre, así que para instalarlo podemos hacer `npm install redux react-redux`. Con esto instalamos la librería de redux en sí, y los **helpers** de `react-redux`, donde están las funciones `bindActionsCreator`, `connect`, etc..

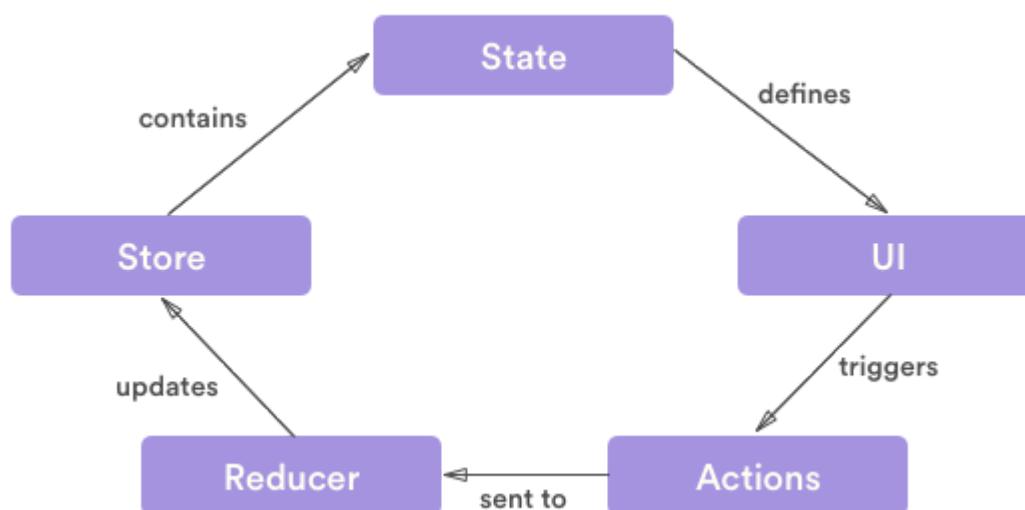
Redux se instala así, asumiendo que ya tenemos un proyecto de React funcionando, con su webpack configurado.

## Workflow de Redux

Lo primero que tenemos que incorporar para trabajar con Redux, es el workflow que nos propone.

Básicamente, nuestra aplicación debería funcionar siempre siguiente este ciclo de vida, en el medio van a aparecer elementos de la API de redux que vamos a ir explicando.

Se podría considerar a **redux** como una implementación del patrón **flux** para react, también se podría considerar cómo un patrón por si mismo. ( De hecho, [ni sus autores se ponen de acuerdo en eso](#) ) Lo cierto es que está influenciado por el patrón **flux**, usando por facebook.



1. El **árbol de Estado** define la UI y las acciones posibles a través de **props**.
2. Acciones realizadas por los usuarios son enviadas a un **action creator** que las normaliza.
3. Estas acciones normalizadas son pasadas a un **reducer**, que es donde se ejecuta el código que contiene la lógica de la acción.

4. El reducer crea un nuevo Estado y lo devuelve (`dispatches it`) al `Store`.

5. La UI es actualizada acorde al nuevo estado.

## Acciones

Las **acciones** representan una *intención* de cambiar el estado de nuestra `store`. Las *acciones* son las **únicas** fuente de información que llega a nuestras `stores`.

Las **acciones** son **objetos JavaScript planos**, deben tener una propiedad llamada `type`, que indica o describa la acción que se realiza. Por convención los `types` debe estar escritos como `string constants`, es decir, todo en mayúsculas y con SNAKE\_CASE. Además del `type`, las acciones van a contener cualquier otra propiedad que necesitemos para su funcionamiento. Se recomienda que las acciones tengan la menor cantidad de propiedades posibles.

Podemos ver algunas recomendaciones de cómo diseñar nuestras acciones en [esta guía](#).

Cuando tu app empieze a crecer, es buena idea separar las acciones en distintos módulos (archivos).

```
// incrementas likes
const INCREMENT_LIKES = {
 type: 'INCREMENT_LIKES',
 index: 4,
}

// agregar comentarios
const ADD_COMMENT = {
 type: 'ADD_COMMENT',
 postId: 'X4Yb4',
 author: 'Toni',
 comment: 'Esto es una acción en particular.',
}

// remover comentarios
const REMOVE_COMMENT = {
 type: 'REMOVE_COMMENT',
 postId: 'X4Yb5',
 index: 5,
}
```

## Action Creators

Los **actions creators** son funciones que **crean** acciones, o sea que *retornan* un objeto que representa una acción. Es fácil confundir los términos *action* y *action creator* así que hay que usarlos con cuidado.

```
// incrementas likes
export function increment(index) {
 return {
 type: 'INCREMENT_LIKES',
 index
}
```

```

}
// agregar comentarios
export function addComment(postId, author, comment) {
 return {
 type: 'ADD_COMMENT',
 postId,
 author,
 comment
 }
}
// remover comentarios
export function removeComment(postId, index) {
 return {
 type: 'REMOVE_COMMENT',
 postId,
 index
 }
}

```

## Dispatch

Las *acciones* tienen que ser enviadas al *Store* para que surgen efecto. La función `dispatch` base es un método de `store`, esta manda una acción de forma sincrónica a los reductores del store, junto con el estado previo retornado por el store, para calcular el nuevo estado. Espera que las acciones que les pasemos sean objetos planos, listas para ser consumidas por los reducers.

También se puede envolver a los dispatchers en una serie de `Middlewares`, esto permite que los dispatchers puedan manejar *acciones asincrónicas*, además de poder transformar, demorar, ignorar o interpretar acciones antes de pasarla al siguiente Middleware.

When we use redux with react, the dispatcher functions are bound to the component

## Reducers

Las *acciones* describen que algo sucedió en nuestra app, pero no especifican *cómo* esta acción impacta en el estado actual. Saber eso es el trabajo de los **reducers**.

Un *reducer* es una función que recibe el estado previo de un *Store* y una acción y retorna el nuevo estado. Justamente se llama reducer, porque toma al estado como una *acumulación* de acciones. Los reducers tienen que ser siempre funciones **puras**, estas son cosas que **nunca** deberías hacer en un reducer:

- Mutar sus argumentos.
- Realizar cosas que tengan efectos secundarios, como llamadas a APIs, o routeo.
- Llamar a funciones no puras adentro, como `Math.random()` o `Date.now()`

**Given the same arguments, a reducer should calculate the next state and return it. No surprises.  
No side effects. No API calls. Just a calculation.**

```

function posts(state = [], action) {
 switch(action.type){

```

```

 case 'INCREMENT_LIKES':
 console.log('increment Likes');
 const i = action.index;
 return [
 ...state.slice(0,i), // antes del que estamos actualizando
 {...state[i], likes: state[i].likes + 1},
 ...state.slice(i+1)// despues del actualizado
]
 default:
 return state;

 }
}

export default posts;

```

Cuando la app es grande, podemos poner cada reducer en archivos separados, agrupandolos según los datos que manejen, haciendolos independientes entre ellos. Para hacer esto, redux nos ofrece la funcionalidad de `combineReducers()`, lo que hace es convertir un objeto que tiene varios reducers en una sola función reducers que los contiene, y que puede ser pasada a `createStore`.

```

// Redux sólo puede tener un reducer.
// por eso vamos a tener el Root Reducer
// donde vamos a incorporar los demás

import { combineReducers } from 'redux';
import { routerReducer } from 'react-router-redux';

import posts from './posts.js';
import comments from './comments.js';

const rootReducer = combineReducers({posts, comments, routing: routerReducer})

export default rootReducer;

```

## Store

La **Store** tiene las siguientes responsabilidades:

- Mantiene el estado de la aplicación.
- Permite el acceso al estado a través de `getState()`.
- Permite actualizar el estado a través de `dispatch(action)`.
- Registra *listeners* con `subscribe(listener)`.
- Maneja la desuscripción de *listeners*.

Hay que notar que vamos a tener exactamente *una* Store por cada aplicación que hagamos usando Redux. Para crear una Store, vamos a usar la función `createStore` que recibe un reducer como argumento, y opcionalmente el *estado inicial* de la app.

```

import { createStore, compose } from 'redux';
import { syncHistoryWithStore} from 'react-router-redux';
import { browserHistory } from 'react-router';

// Import root reducer
import rootReducer from './reducers/index.js';

import comments from './data/comments.js';
import posts from './data/posts.js';

const defaultState = {
 posts,
 comments
}

const store = createStore(rootReducer, defaultState);

```

## Usando Redux con React

Para usar redux con react, vamos a usar un paquete llamado `react-redux` que nos ofrece los **bindings** de redux con react. Para instalarlo hacemos:

```
npm install --save react-redux
```

## Componentes

Los bindings de `react-redux` están realizados pensados en el patrón de **separar los Componentes Presentacionales de los Containers**.

|                           | <b>Presentacionales</b>                 | <b>Containers</b>                                          |
|---------------------------|-----------------------------------------|------------------------------------------------------------|
| <b>Próposito</b>          | Cómo se ven las cosas (markup, estilos) | Cómo funcionan las cosas (traer datos, actualizar estados) |
| <b>Sabe de Redux</b>      | NO                                      | SI                                                         |
| <b>Para leer datos</b>    | Lee de props                            | Se suscribe a los estados de Redux                         |
| <b>Para cambiar datos</b> | Invoca callbacks de sus props           | Envía acciones a Redux                                     |
| <b>Son escritos</b>       | A mano                                  | Generados por React Redux                                  |

Técnicamente podríamos codear los Containers por nosotros mismos usando `subscribe`, pero Redux nos desaconseja de hacer esto, ya que nos proveen de la función `connect`, que nos permite generarlos, y estos Componentes generados están optimizados en términos de performance.

Para generar un Componente que tenga todos los bindings de redux con react, primero vamos a tener que definir las siguientes funciones:

- **mapStateToProps:** Recibe el estado de la aplicación y lo mapea a props de react.
- **mapDispatchToProps:** Recibe el método `dispatch` y retorna callbacks props que vamos a poder pasar a los Componentes presentacionales.

Finalmente, usando `connect` de `react-redux` y pasandole estas dos funciones obtenemos una función lista para darle los binding a un Componente React. Finalmente elegimos que Componente queremos que tenga los bindings y luego lo exportamos. Por ejemplo, vamos a darle los binding al Componente `Main` y lo vamos a exportar como `App`:

```
import { bindActionCreators } from 'redux';
import { connect } from 'react-redux';
import * as actionsCreators from '../actions/actionsCreators.js';

import Main from './Main.js';

function mapStateToProps(state) {
 return {
 posts: state.posts,
 comments: state.comments
 }
}

function mapDispatchToProps(dispatch) {
 return bindActionCreators(actionsCreators, dispatch);
}

const App = connect(mapStateToProps, mapDispatchToProps)(Main);

export default App;
```

## Provider

Todos los Componentes Containers deben tener acceso al `Store` para que puedan suscribirse a ella. Una opción seria pasar el store como un prop a cada componente Container, pero esto se volvería tedioso muy rápidamente, y un posible punto de error. Lo que nos recomienda `redux` es usar un Componente especial de `react-redux` llamado `<Provider>` que **mágicamente** hace que el Store esté disponible para todos los Container de nuestra app, sin pasarla explícitamente.

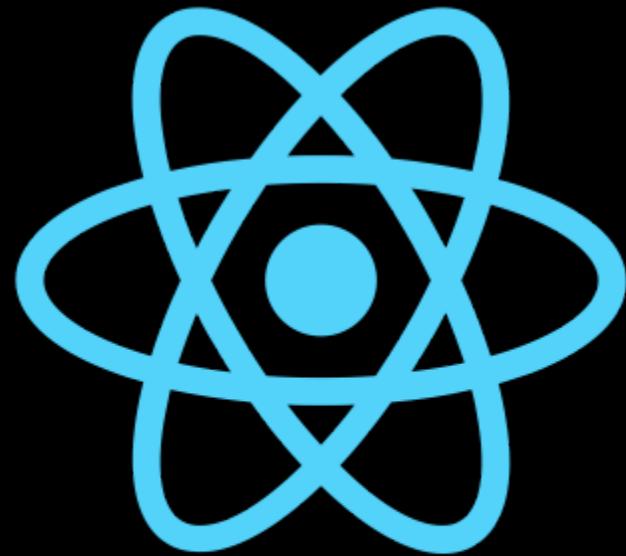
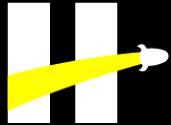
```
...
import { Provider } from 'react-redux'; //Bindings from redux and React
import store, { history } from './store.js';

const router = (
 <Provider store={store}>
 <Router history={ history }>
 <Route path="/" component={App}>
 <IndexRoute component={PhotoGrid}>
 <Route path="view/:postId" component={Single}>
 </Route>
 </Route>
 </Router>
 </Provider>
)
```

```
 </Router>
 </Provider>
)
...

```

Ahora vamos a poder acceder a nuestro `store` en cada Componente a través de sus props.



# React

## Hooks



# Hook de estado

## useState

Permite guardar estados en componentes funcionales.



```
1 function Example() {
2 const [count, setCount] = useState(0);
3 ...
4 function increment() {
5 setCount(count + 1)
6 }
7 return <div>
8 <button onClick={increment}>Suma 1</button>
9 <p>{count}</p>
10 </div>
11 ...
12 }
```

useState() es una función que devuelve un arreglo con dos valores. El primero es una variable con el valor del estado y el segundo valor es una función se usa para actualizar el estado. Acepta un nuevo valor de estado y sitúa en la cola una nueva renderización del componente.



# Hook de efecto

## useEffect

Permite realizar efectos secundarios en componentes funcionales.  
Efectos secundarios pueden ser: Traer data con AJAX, cambiar el  
DOM manualmente, suscribirse a algo, etc..



```
1 function Example() {
2 const [count, setCount] = useState(0);
3 ...
4 useEffect(() => {
5 // Update the document title using the browser API
6 document.title = `You clicked ${count} times`;
7 });
8 ...
9 }
```

De forma predeterminada, React ejecuta los efectos después del primer renderizado y después de cada actualización



# Hook de efecto

## componentDidMount con useEffect



```
1 function Example() {
2 const [count, setCount] = useState(0);
3 ...
4 useEffect(() => {
5 // Update the document title using the browser API
6 console.log("this component has been mounted");
7 }, []);
8 ...
9 }
```

Podemos realizar un efecto que actúe de manera similar al método componentDidMount de los class component



# Hook de efecto

## componentDidUpdate con useEffect



```
1 function Example() {
2 const [count, setCount] = useState(0);
3 ...
4 useEffect(() => {
5 // Update the document title using the browser API
6 console.log("First time or the count state was updated");
7 }, [count]);
8 ...
9 }
```

Podemos realizar un efecto que actúe de manera similar al método componentDidUpdate de los class component



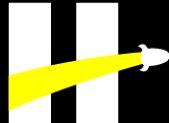
# Hook de efecto

## componentWillUnmount con useEffect



```
1 function Example() {
2 const [count, setCount] = useState(0);
3 ...
4 useEffect(() => {
5 // Update the document title using the browser API
6 return () => console.log("this component has been removed");
7 }, []);
8 ...
9 }
```

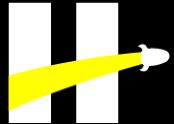
Podemos realizar un efecto que actúe de manera similar al método `componentWillUnmount` de los class component



# Hook useReducer



```
1 const initialState = {count: 0};
2
3 function reducer(state, action) {
4 switch (action.type) {
5 case 'increment':
6 return {count: state.count + 1};
7 case 'decrement':
8 return {count: state.count - 1};
9 default:
10 throw new Error();
11 }
12 }
13
14 function Counter() {
15 const [state, dispatch] = useReducer(reducer, initialState);
16 return (
17 <>
18 Count: {state.count}
19 <button onClick={() => dispatch({type: 'decrement'})}>-</button>
20 <button onClick={() => dispatch({type: 'increment'})}>+</button>
21 </>
22);
23}
```



# Hook useRef



```
1 function TextInputWithFocusButton() {
2 const inputEl = useRef(null);
3 const onButtonClick = () => {
4 // `current` apunta al elemento de entrada de texto montado
5 inputEl.current.focus();
6 };
7 return (
8 <>
9 <input ref={inputEl} type="text" />
10 <button onClick={onButtonClick}>Focus the input</button>
11 </>
12);
13}
```

IMPORTANTE: podemos manipular el DOM manualmente con este hook, pero es desaconsejado hacerlo.

# Henry

---



Hacé click acá para dejar tu feedback sobre esta clase.

Hacé click acá completar el quiz teórico de esta lecture.

## Presentando Hooks

---

Hooks son una nueva característica en React 16.8. Estos te permiten usar el estado y otras características de React sin escribir una clase.

```
import React, { useState } from 'react';

function Example() {
 // Declara una nueva variable de estado, la cual llamaremos "count"
 const [count, setCount] = useState(0);

 return (
 <div>
 <p>You clicked {count} times</p>
 <button onClick={() => setCount(count + 1)}>
 Click me
 </button>
 </div>
);
}
```

## Motivación

Los Hooks resuelven una amplia variedad de problemas aparentemente desconectados en React que hemos encontrado durante más de cinco años de escribir y mantener decenas de miles de componentes. Ya sea que estés aprendiendo React, usándolo diariamente o incluso prefieras una librería diferente con un modelo de componentes similar, es posible que reconozcas algunos de estos problemas.

### Es difícil reutilizar la lógica de estado entre componentes

React no ofrece una forma de “acoplar” comportamientos re-utilizables a un componente (Por ejemplo, al conectarse a un store). Si llevas un tiempo trabajando con React, puedes estar familiarizado con patrones como render props y componentes de orden superior que tratan resolver esto. Pero estos patrones requieren que reestructures tus componentes al usarlos, lo cual puede ser complicado y hacen que tu código sea más difícil de seguir. Si observas una aplicación típica de React usando React DevTools, Lo más probable es que

encuentras un “wrapper hell” de componentes envueltos en capas de providers, consumers, componentes de orden superior, render props, y otras abstracciones. Aunque podemos filtrarlos usando las DevTools, esto apunta a un problema aún más profundo: React necesita una mejor primitiva para compartir lógica de estado.

Con Hooks, puedes extraer lógica de estado de un componente de tal forma que este pueda ser probado y re-usado independientemente. Los Hooks te permiten reutilizar lógica de estado sin cambiar la jerarquía de tu componente. Esto facilita el compartir Hooks entre muchos componentes o incluso con la comunidad.

Discutiremos esto más a fondo en [Construyendo tus propios Hooks](#).

## Los componentes complejos se vuelven difíciles de entender

A menudo tenemos que mantener componentes que empiezan simples pero con el pasar del tiempo crecen y se convierten en un lío inmanejable de múltiples lógicas de estado y efectos secundarios. Cada método del ciclo de vida a menudo contiene una mezcla de lógica no relacionada entre sí. Por ejemplo, los componentes pueden realizar alguna consulta de datos en el `componentDidMount` y `componentDidUpdate`. Sin embargo, el mismo método `componentDidMount` también puede contener lógica no relacionada que cree escuchadores de eventos, y los limpia en el `componentWillUnmount`. El código relacionado entre sí y que cambia a la vez es separado, pero el código que no tiene nada que ver termina combinado en un solo método. Esto hace que sea demasiado fácil introducir errores e inconsistencias.

En muchos casos no es posible dividir estos componentes en otros más pequeños porque la lógica de estado está por todas partes. También es difícil probarlos. Esta es una de las razones por las que muchas personas prefieren combinar React con una librería de administración de estado separada. Sin embargo, esto a menudo introduce demasiada abstracción, requiere que saltes entre diferentes archivos, y hace que la reutilización de componentes sea más difícil.

Para resolver esto, Hooks te permite dividir un componente en funciones más pequeñas basadas en las piezas relacionadas (como la configuración de una suscripción o la consulta de datos), en lugar de forzar una división basada en los métodos del ciclo de vida. También puedes optar por administrar el estado local del componente con un reducer para hacerlo más predecible.

Discutiremos esto más a fondo en [Usando el Hook de efecto](#).

## Las clases confunden tanto a las personas como a las máquinas

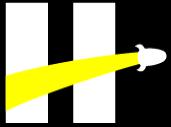
Además de dificultar la reutilización y organización del código, hemos descubierto que las clases pueden ser una gran barrera para el aprendizaje de React. Tienes que entender cómo funciona `this` en JavaScript, que es muy diferente a cómo funciona en la mayoría de los lenguajes. Tienes que recordar agregar `bind` a tus manejadores de eventos. Sin inestables propuestas de sintaxis, el código es muy verboso. Las personas pueden entender `props`, el estado, y el flujo de datos de arriba hacia abajo perfectamente, pero todavía tiene dificultades con las clases. La distinción entre componentes de función y de clase en React y cuándo usar cada uno de ellos lleva a desacuerdos incluso entre los desarrolladores experimentados de React.

Además, React ha estado en el mercado durante unos cinco años, y queremos asegurarnos de que siga siendo relevante en los próximos cinco años. Como muestran Svelte, Angular, Glimmer, y otros, la compilación anticipada de componentes tiene mucho potencial a futuro. Especialmente si no se limita a las plantillas. Recientemente, hemos estado experimentando con el encarpetado de componentes usando Prepack, y hemos visto resultados preliminares prometedores. Sin embargo, encontramos que los componentes de clase

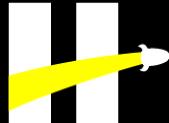
pueden fomentar patrones involuntarios que hacen que estas optimizaciones nos lleven a un camino más lento. Las clases también presentan problemas para las herramientas de hoy en día. Por ejemplo, las clases no minifican muy bien, y hacen que la recarga en caliente sea confusa y poco fiable. Queremos presentar una API que hace más probable que el código se mantenga en la ruta optimizable.

Para resolver estos problemas, Hooks te permiten usar más de las funciones de React sin clases. Conceptualmente, los componentes de React siempre han estado más cerca de las funciones. Los Hooks abarcan funciones, pero sin sacrificar el espíritu práctico de React. Los Hooks proporcionan acceso a vías de escape imprescindibles y no requieren que aprendas técnicas complejas de programación funcional o reactiva.

HENRY



# Node



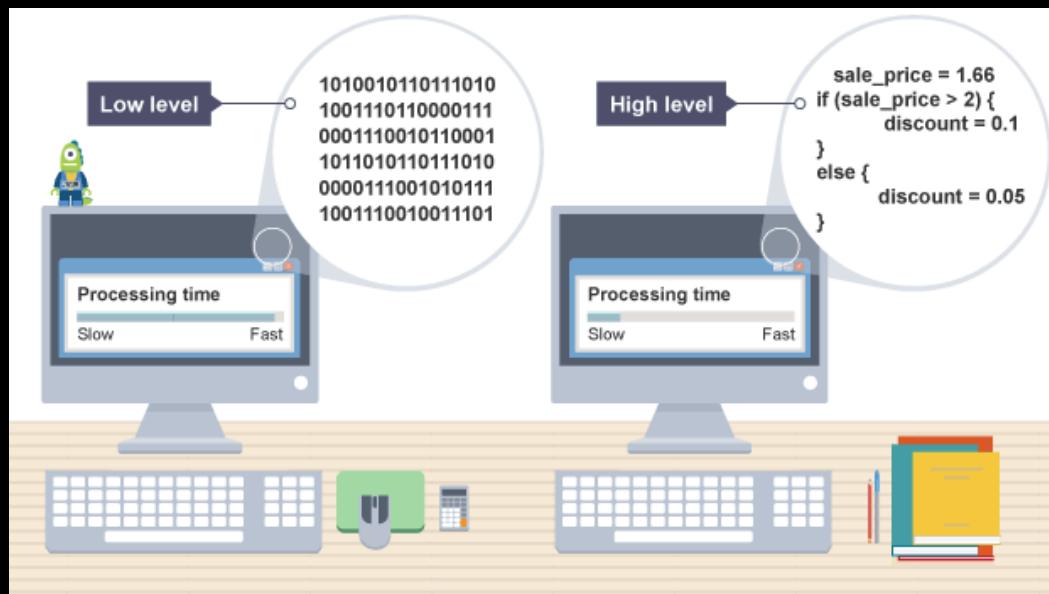
# Lenguajes de Bajo Nivel y de Alto Nivel



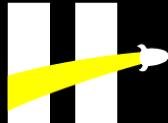
```
-u 100 1a
OCFD:0100 BA0B01 MOV DX,010B
OCFD:0103 B409 MOV AH,09
OCFD:0105 CD21 INT 21
OCFD:0107 B400 MOV AH,00
OCFD:0109 CD21 INT 21

-d 10b 13f
OCFD:0100 20 65 73 74 65 20 65 73-20 75 48 6F 6C 61 2C
OCFD:0110 72 61 6D 61 20 68 65 63-68 6F 20 70 72 6F 67
OCFD:0120 73 65 6D 62 6C 65 72 20-70 61 72 61 20 6C 61 73
OCFD:0130 57 69 6B 69 70 65 64 69-61 24
OCFD:0140
Hola,
este es un progra
ma hecho en as
sembler para la
Wikipedia$
```

Assembler



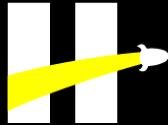
Node está  
escrito en C++!



# V8



- V8 JavaScript Engine: *ya sabíamos.*
- V8 is Google's open source JavaScript engine: *really?*
- V8 implements ECMAScript as specified in ECMA-262: *ah!, sigue estándares, bien!.*
- V8 is written in C++ and is used in Google Chrome, the open source browser from Google: *Alguien conoce ese browser Chrome, es bueno?.*
- V8 can run standalone, or can be embedded into any C++ application: *Atención con esto, se puede embeber (usar) en cualquier aplicación C++*



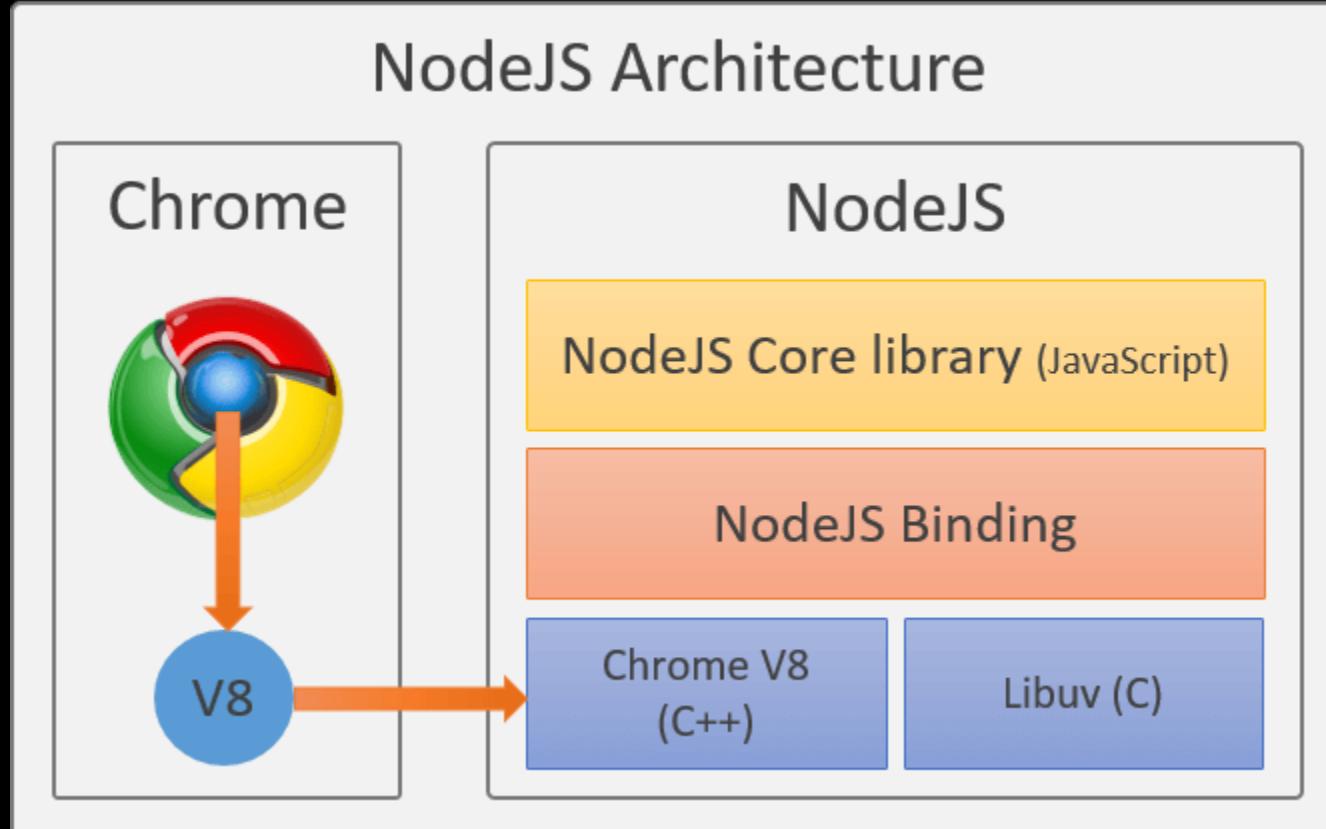
# V8 + libuv

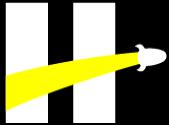


- Maneras de organizar nuestro código para que sea reusable (**módulos**)
- Poder leer y escribir archivos ( input/output)
- Leer y escribir en Bases de Datos.
- Poder enviar y recibir datos de internet.
- Poder interpretar los formatos estándares.
- Alguna forma de manejar procesos que lleven mucho tiempo.



# NodeJS





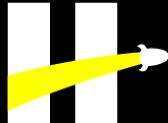
# NodeJS



```
1 // ejecutando el archivo
2
3 $ node index.js
4 $ 1
```



```
1 // index.js
2
3 const hola = 1;
4
5 console.log(hola);
```



# Módulos



Def: Un bloque de código reusable, cuya existencia no altera accidentalmente el comportamiento de otros bloques de código.

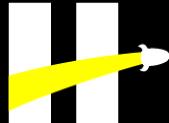
## Cómo funciona CommonJs Modules?

Básicamente, el standart dice lo siguiente:

- Cada archivo es un módulo, y cada módulo es un archivo separado.
- Todo lo que queremos exportar va a ser expuesto desde un único punto.

## Require con modulos Core o nativos

```
1 var util = require('util'); // No usamos ./ porque es un modulo core
2
3 var nombre = 'Toni';
4 var saludo = util.format('Hola, %s', nombre);
5 util.log(saludo);
```



# Gestor de Paquetes

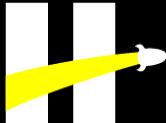


Primero definamos lo que es un paquete. Básicamente es.. `código` (un módulo podría ser un paquete)! Es cualquier pieza de código manejada y mantenida por un gestor de paquetes.

Ahora, un gestor de paquetes es un software que automatiza la instalación y actualización de paquetes.



NPM: Node Package Manager



# Package.json

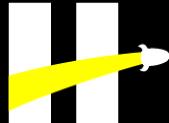
Para poder trackear las dependencias y los paquetes que tenemos instalados (entre otras cosas), npm hace uso de un archivo de **configuración** al que llama **package.json**.

```
{
 "main": "node_modules/expo/AppEntry.js",
 "scripts": {
 "start": "expo start",
 "android": "expo start --android",
 "ios": "expo start --ios",
 "web": "expo start --web",
 "eject": "expo eject",
 "test": "jest"
},
 "dependencies": {
 "expo": "^35.0.0",
 "jest": "^24.9.0",
 "react": "16.8.3",

```



```
1 // Para iniciar una carpeta
2 // con un package.json podemos hacer
3
4 npm init
```



# Semantic Versioning



1 . 3 . 1

MAJOR . MINOR . PATCH

Breaking  
Changes

Nueva  
funcionalidad,  
retro-compatible

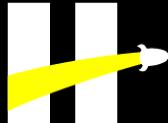
Bug fix  
retro-compatible



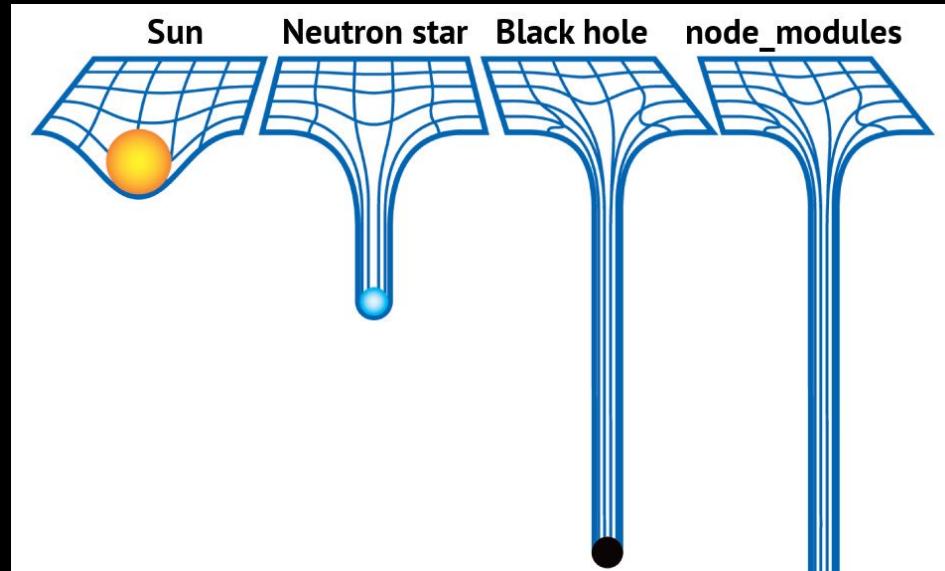
```
1 ~1.2.3 is >=1.2.3 <1.3.0
```

```
2
```

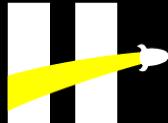
```
3 ^1.2.3 is >=1.2.3 <2.0.0
```



# NPM



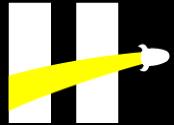
Los paquetes instalados de forma local serán guardados en una carpeta llamada `node\_modules` creada dentro de la carpeta donde ejecuté el comando



# NPM



```
1 // Algunos comandos
2
3 npm install --save {nombrePaquete}
4
5 npm install -g {nombre paquete}
6
7 npm update
8
9 npm audit
10
11 npm start
12
13 npm test
14
15 npm run {nombreScript}
```



< DEMO />

# Henry

---



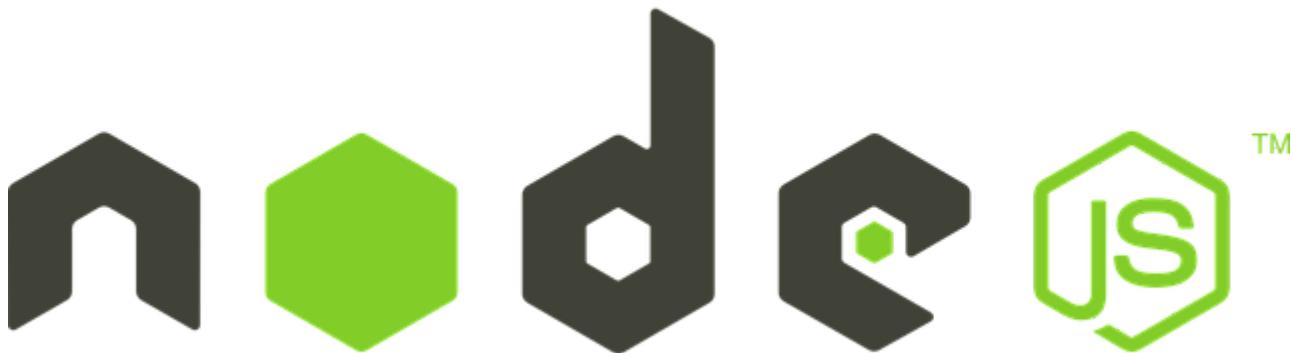
Hacé click acá para dejar tu feedback sobre esta clase.



Hacé click acá completar el quizz teórico de esta lecture.

## Introducción a Nodejs

---



## ¿Qué es Nodejs?

### Conceptos

Para poder entender el concepto de node, primero vamos a tener que conocer otros conceptos más básicos sobre estos temas:

- Procesadores
- Lenguaje de máquina
- C++

Podemos pensar a procesador (microprocesador) como una pequeña máquina que recibe impulsos eléctricos como entrada y genera salidas al que nosotros le pasamos *instrucciones*. Pero el procesador no entiende cualquier lenguaje, sólo habla *lenguaje de máquina* (es un lenguaje binario, secuencia de unos y ceros) y que está detallado en lo que se conoce como set de instrucciones. Como se imaginarán cada fabricante tiene set de instrucciones distintos y por lo tanto distintos lenguajes, mencionemos algunos:

- IA-32

- x86, x86-64
- ARM
- MIPS

Para pasarle instrucciones al procesador, no escribimos 1's y 0's, sino que usamos un lenguaje que se traduce directamente a esa secuencia, llamado **assembler**.

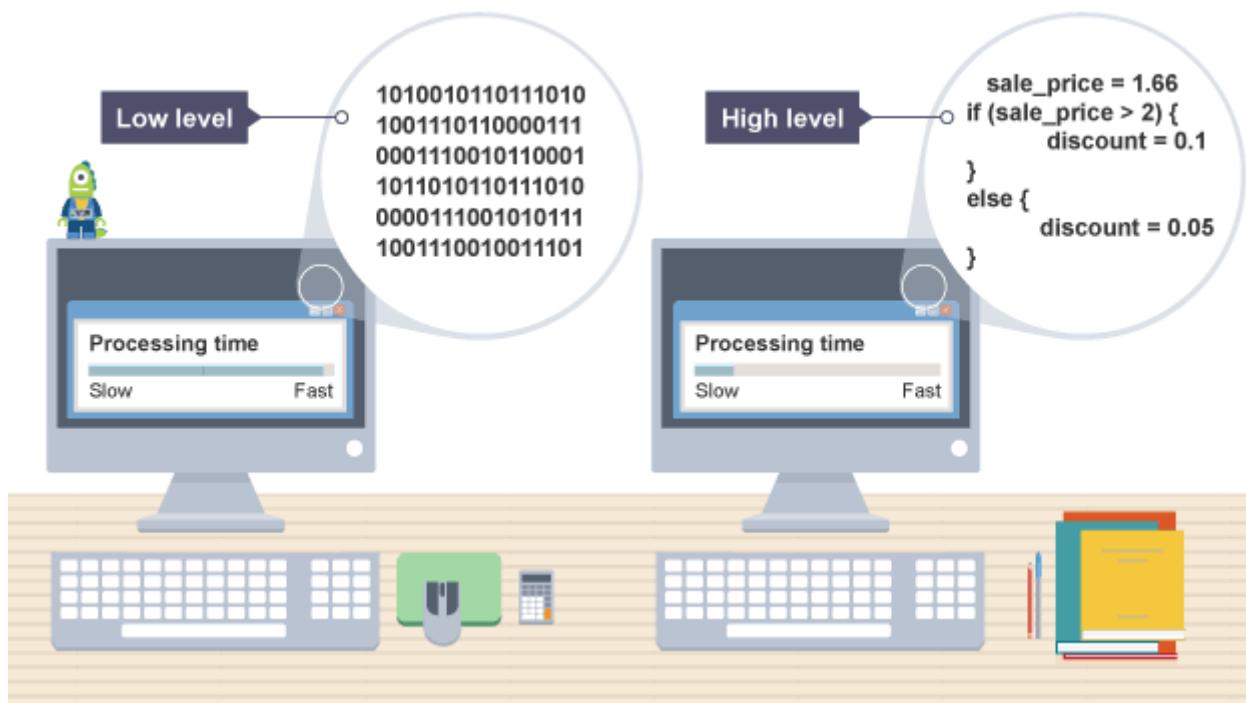
```
-u 100 1a
OCFD:0100 BA0B01
OCFD:0103 B409
OCFD:0105 CD21
OCFD:0107 B400
OCFD:0109 CD21
MOV DX,010B
MOV AH,09
INT 21
MOV AH,00
INT 21

-d 10b 13f
OCFD:0100 48 6F 6C 61 2C
OCFD:0110 20 65 73 74 65 20 65 73-20 75 6E 20 70 72 6F 67
OCFD:0120 72 61 6D 61 20 68 65 63-68 6F 20 65 6E 20 61 73
OCFD:0130 73 65 6D 62 6C 65 72 20-70 61 72 61 20 6C 61 20
OCFD:0140 57 69 6B 69 70 65 64 69-61 24

Hola,
este es un progra
rama hecho en as
sembler para la
Wikipedia$
```

Hoy en día no se programa en assembler (**lenguaje de bajo nivel**), ya que es muy complejo y hacer un simple 'Hello World' podría llevar muchas líneas de código: Simplemente no escala. Para solucionar esto, se crearon lenguajes más fáciles y rápidos de programar y que compilan a lenguaje de máquina, estos son los conocidos *lenguajes de alto nivel*, JAVA, C++, Javascript son ejemplos de estos lenguajes. Es importante notar, que no importa que lenguaje usemos, en algún momento el código será *traducido o compilado* a lenguaje de máquina, que es el único lenguaje que entiende verdaderamente la computadora.

Como pueden pensar mientras nos alejamos del lenguaje de máquina (lenguajes de más alto nivel) y nos abstraemos vamos ganando velocidad para codear, pero también vamos perdiendo performance. Piensen que si codeamos en lenguaje de máquina, podemos controlar cada slot de memoria nosotros mismos y hacerlo de la mejor forma posible. Cuando subimos de nivel, alguien hace ese trabajo por nosotros, y como tiene que ser genérico no puede lograr ser tan óptimo. Es por eso que según la performance y los recursos que se necesite o tengamos vamos a elegir lenguajes de altísimo o bajísimo nivel.



Por ejemplo, los microcontroladores embebidos en lavarropas están programados en C compilado a lenguaje de máquina, esto es porque tienen muy poca memoria y tienen que optimizarla al máximo

## C++

**C++** es un lenguaje de programación de bajo nivel, estaría justo por arriba de Assembly. Este lenguaje se hizo muy popular porque es fácil para codear, pero a su vez te deja tener bastante control sobre lo que está pasando en nivel hardware. Muchos otros lenguajes se construyeron en base a C++, o siguiendo sus sintaxis o usándolo en su cadena de abstracción.

Justamente *Nodejs* está programado en C++. La razón por la que nodejs está escrito en C++ es porque **V8** está escrito en C++, ahora veamos qué es V8.

## Motor de Javascript

Antes que nada es importante entender que el lenguaje JavaScript está basado en un standard que se conoce como **ECMASCRIPT**. Este standard setea las bases de qué cosas deberá hacer el lenguaje y cuales no, y de qué manera. Ahora bien, en el mundo real no se respetan los estándares al 100%, es por eso que hay muchas implementaciones distintas de JavaScript, que va a interpretar (convertir el código a lenguaje que pueda ser corrido por la computadora) el código de una manera particular, estas implementaciones son los llamados *motores javascript*.

[Ve ocho](#) es el motor de javascript creado por **Google** para su browser *Chrome*. Es un proyecto Open Source así que podemos investigar su [código](#), en su página Google define a v8 como:

- V8 JavaScript Engine: *ya sabíamos*.
- V8 is Google's open source JavaScript engine: *really?*.
- V8 implements ECMAScript as specified in ECMA-262: *ah!, sigue estándares, bien!*.
- V8 is written in C++ and is used in Google Chrome, the open source browser from Google: *Alguien conoce ese browser Chrome, es bueno?*.
- V8 can run standalone, or can be embedded into any C++ application: *Atención con esto, se puede embeber (usar) en cualquier aplicación C++*

Releamos el último bullet en detalle: Es Standalone, eso quiere decir que puedo bajar V8 y correrlo en mi computadora, no necesariamente dentro del browser, genial!; puede ser embedido, es decir que puedo codear una aplicación en C++ y agregarle todas las funciones de V8, copado!! Ya se imaginan donde empezó a nacer *Nodejs*??

Sí! Nodejs es justamente una aplicación escrita en C++ que embebe el motor *V8* en su código. Por lo tanto puede interpretar código javascript, 'traducirlo' a lenguaje de máquina y finalmente ejecutarlo. Pero eso no es lo mejor de Nodejs, lo mejor es que el creador agregó algunos features que no están definidos en el estándar. Javascript originalmente estaba diseñado para correr en el browser, o sea que nadie pensó en que pudiera leer archivos, o conectarse a una base de datos, etc... Justamente estas features son las que Nodejs agrega usando código C++. O sea, crea nuevas funciones Javascript que envuelven en realidad funciones de C++, como por ejemplo la función de leer un archivo del filesystem.

Esto hace que NodeJs sea muy poderoso! De hecho, gracias a esto NodeJs tiene todas las features necesarias para poder manejar un servidor.

- Maneras de organizar nuestro código para que sea reusable

- Poder leer y escribir archivos ( input/output)
- Leer y escribir en Bases de Datos
- Poder enviar y recibir datos de internet
- Poder interpretar los formatos estándares
- Alguna forma de manejar procesos que lleven mucho tiempo

## Instalar Nodejs

Para instalar node vamos a ir [acá](#) y seguir las instrucciones según el sistema operativo que estés usando.

Una vez terminada la instalación, podemos probar si funciona correctamente escribiendo el siguiente comando en la consola:

```
node -v
```

Si el resultado es algo de la forma: `v6.3.1` entonces habremos instalado Node de manera correcta!

## Core libraries

Nodejs cuenta con un set de librerías (les vamos a llamar módulos) que vienen por defecto en la instalación. Básicamente es código escrito para hacer tareas muy comunes. Podemos separar estas librerías en las que están escritas en *C++* y las que son nativas o están escritas en *javascript*.

### C++

Como dijimos antes, usando el motor *V8* los desarrolladores de Nodejs agregaron funcionalidad que ECMAScript no tenía en el standard. La mayoría de estas funciones tiene que ver con tareas que también involucran al Sistema operativo, como leer archivos, mandar y recibir datos por la red, comprimir y descomprimir archivos y otras de manejo de streams.

### Javascript

Estas librerías están escritas en Javascript, algunas de ellas implementan la funcionalidad usando javascript, pero la mayoría son sólo envoltorios a las funciones escritas en *C++* para que puedan ser fácilmente utilizadas por los desarrolladores de Nodejs. Éstas son el tipo de librerías que podrías haber escrito vos mismo, por suerte alguien ya se tomó el trabajo!

## Organizando nuestro código en Nodejs

Una de las features que hizo que Nodejs creciera tanto, es justamente el ecosistema de librerías desarrolladas por los usuarios, que sumadas a las librerías *Core* hacen que sea muy potente. Ahora vamos a ver cómo poder utilizar módulos en nuestro código, y luego veremos cómo instalar módulos externos.

### Módulos

Def: Un bloque de código reusable, cuya existencia no altera accidentalmente el comportamiento de otros bloques de código. Este concepto ya existía en otros lenguajes de programación y era muy usado para estructurar proyectos. De todos modos, los módulos no eran parte del standard en ECMAScript (lo agregaron

en la última versión), pero Nodejs lo introdujo bajo el nombre de *Commonjs Modules*, que era básicamente un acuerdo (un standard) sobre cómo deberían estar estructurados los módulos.

## Cómo funciona CommonJs Modules?

Básicamente, el standart dice lo siguiente:

- Cada archivo es un módulo, y cada módulo es un archivo separado.
- Todo lo que queremos exportar va a ser expuesto desde un único punto.

Para entender el standard tenemos que tener dos conceptos en claro:

- First-Class Functions: Las funciones en javascript son tratadas igual que cualquier otro objeto, es decir que podés guardarlas en variables, pasárlas como argumentos, guardarlas en arreglos, etc...
- Function Expressions: Como las funciones son first-class, al escribir una expresión de función estoy creando la misma, eso quiere decir que puedo crear funciones en cualquier parte del código.

Vamos a empezar construyendo nuestro propio módulo! una vez que lo tengamos, vamos a ver como usarlo. Entendiendo así todo el ciclo.

## Construyendo un módulo

Empezamos con un archivo `js` vacío. Lo llamaremos `hola.js`, este será nuestro primer módulo. Lo único que hará este módulo es saludar:

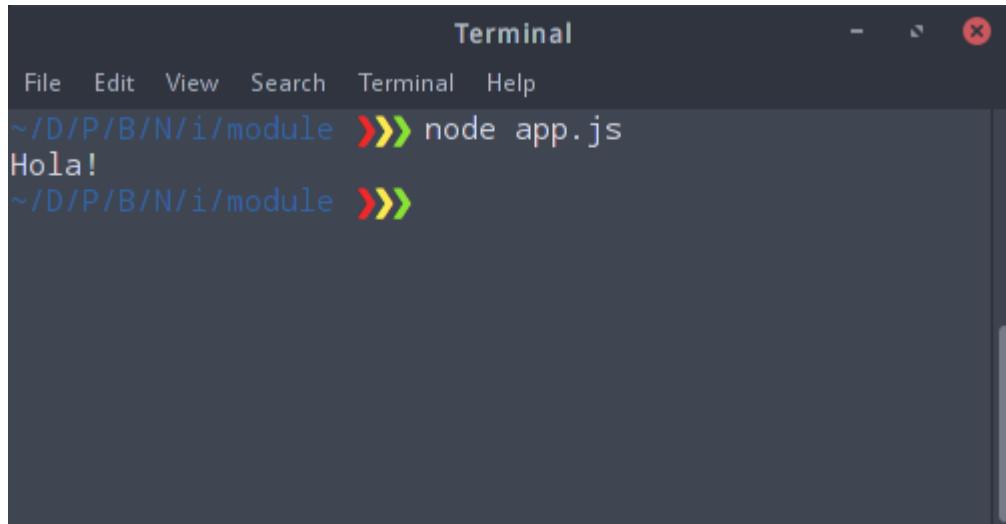
```
console.log('Hola!');
```

Tambien vamos a crear un archivo `js` que se llame `app.js`, en donde vamos a utilizar el código de `hola.js`. Dentro de `app.js` vamos a llamar a la función `require` que es parte de las core libraries de Nodejs:

```
require('./hola.js');
```

**Require** recibe como argumento, un string que es el path en donde encontrará el código o el módulo que queremos agregar, en este caso como `hola.js` está en la misma carpeta el path es: `./hola.js`.

Ahora, si corremos el archivo `app.js` usando `node app.js` en la terminal vamos a ver que se ejecutó el código que escribimos en `hola.js`:

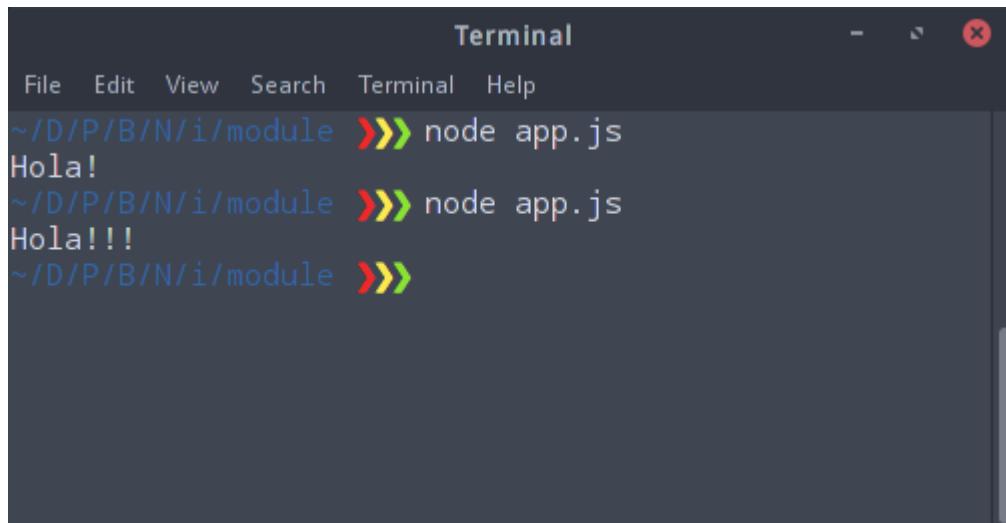


The screenshot shows a terminal window titled "Terminal". The menu bar includes "File", "Edit", "View", "Search", "Terminal", and "Help". The command entered is "node app.js". The output displayed is "Hola!".

Hagamos algo más interesante, vamos a [hola.js](#) y creemos una función y luego la usemos para saludar:

```
var saludar = function() {
 console.log('Hola!!!');
}
saludar();
```

Corramos de nuevo [app.js](#):



The screenshot shows a terminal window titled "Terminal". The menu bar includes "File", "Edit", "View", "Search", "Terminal", and "Help". The command entered is "node app.js". The output displayed is "Hola!" followed by "Hola!!!".

El resultado es el mismo! Ahora, si no invocamos `saludar()` dentro de `hola.js`, creen que la podremos invocar (usar) en `app.js`? Hagamos la prueba!

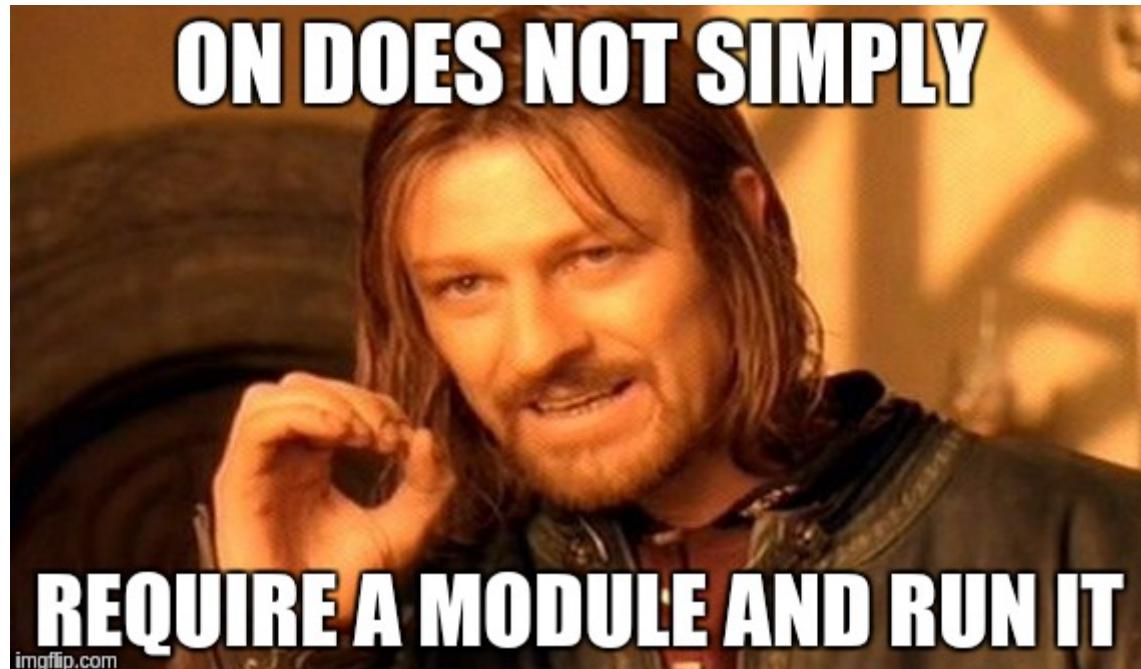
Comentamos la invocación en [hola.js](#):

```
var saludar = function() {
 console.log('Hola!!!');
}
//saludar();
```

Invocamos en [app.js](#):

```
require('./hola.js');
saludar();
```

saludar no está definido! recibimos un error. De hecho, esto está bien que suceda. Se acuerdan que dijimos que un módulo no debería afectar otro código accidentalmente? Eso quiere decir que el código está protegido y que no podemos simplemente usarlo y acceder a los objetos fuera de ese módulo.



Para usarlo afuera, vamos a tener que explicitarlo, veamos como hacer que podamos usar `saludar()` desde `app.js`.

Nodejs nos permite hacerlo usando `module.exports`, que es una variable especial cuyo objetivo es pasar su contenido a otro pedazo de código cuando llamamos a `require`:

```
var saludar = function() {
 console.log('Hola!!!');
};

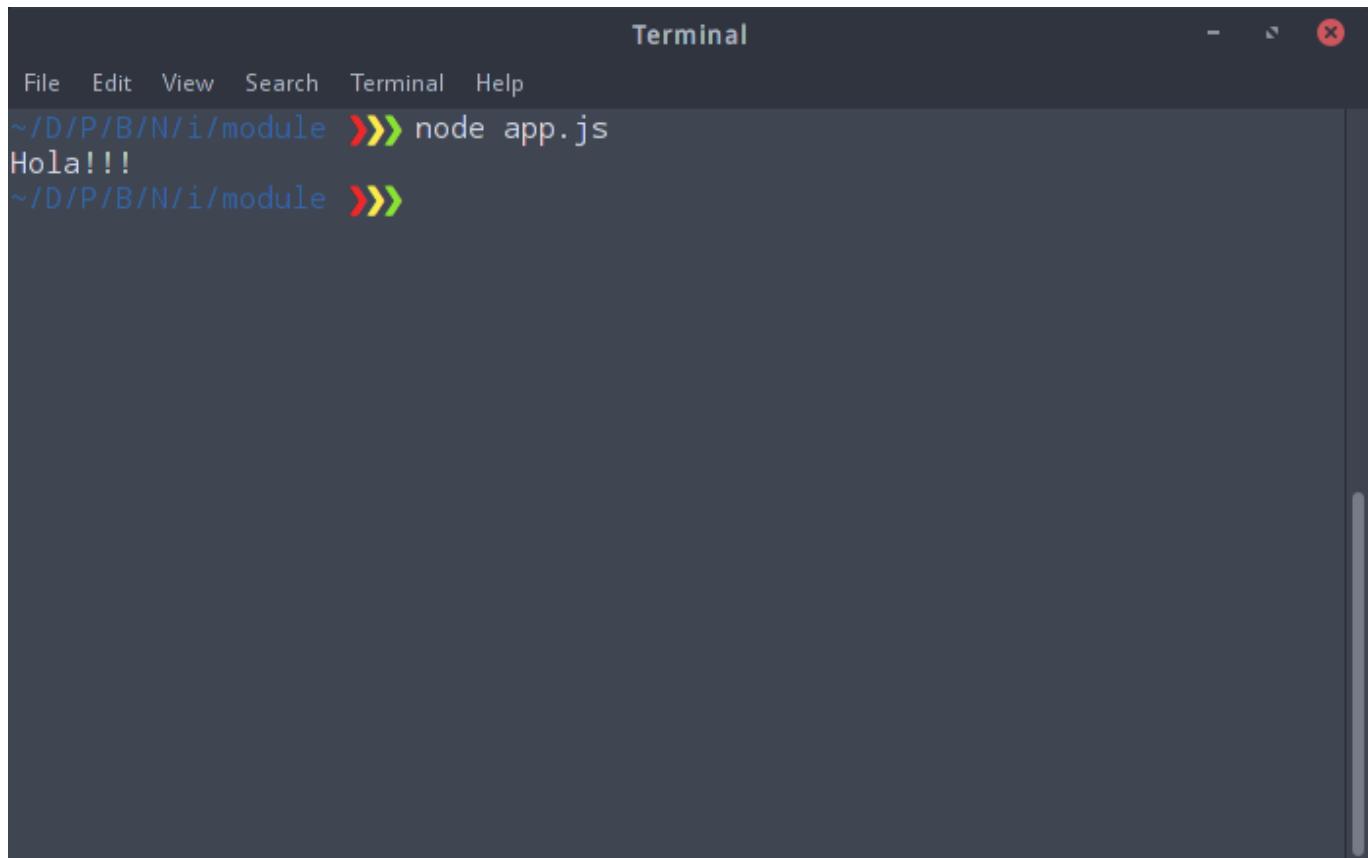
module.exports = saludar;
```

Ahora este módulo está exponiendo el objeto `saludar`. Para usarlo en nuestro módulo tenemos que guardar lo que devuelve `require` en una variable (puedo llamar a la nueva variable como quiera):

```
var hola = require('./hola.js');

hola();
```

Ahora voy a poder invocar `hola()`, porque lo hemos pasado a través de `module.exports`. Si ejecuto `app.js`:



```
Terminal
File Edit View Search Terminal Help
~/D/P/B/N/i/module ➤ node app.js
Hola!!!
~/D/P/B/N/i/module ➤
```

Ahora pudimos acceder al código de `hola.js`, porque lo expusimos a propósito.

Resumiendo:

- **Require** es una función que recibe un *path*.
- **module.exports** es lo que retorna la función *require*.

### Algunos patrones de Require.

Como siempre en Node.js, hay muchas formas de hacer lo mismo (esto puede ser bueno o malo - según cómo lo usemos) y crear módulos no es la excepción. Veamos algunos patrones comunes en la creación de módulos!

#### Múltiples imports

La función *require* busca primero un archivo con el nombre que le pasamos en la carpeta que le pasamos (de hecho no es necesario poner la extensión 'js'). Si no encuentra ese archivo, entonces busca una carpeta con ese nombre y busca un archivo `index.js` el cual importa. Esta funcionalidad la podemos usar como patrón para importar varios archivos: dentro de `index.js` importamos los demás archivos.

Imaginemos que queremos hacer una función que salude en distintos idiomas, vamos a tener un `app.js` de esta forma:

```
var saludos = require('./saludos');

saludos.english();
saludos.spanish();
```

Estamos importando solamente el path `./saludos`, como no hay ningún archivo `.js` con ese nombre, `require` busca una carpeta, por lo tanto vamos a crear una carpeta `saludos` con los siguientes archivos:

### index.js

```
var english = require('./english');
var spanish = require('./spanish');

module.exports = {
 english: english,
 spanish: spanish
};
```

En este archivo estamos importando otros dos módulos, uno por cada idioma en el que queremos saludar. Lo bueno de esto, es que va a ser muy fácil agregar y sacar idiomas de nuestra aplicación, ya que solo debemos agregar o borrar los idiomas que exportamos!

Ahora veamos como sería cada idioma:

### spanish.js

```
var saludos = require('./greetings.json');

var greet = function() {
 console.log(saludos.es);
}

module.exports = greet;
```

### english.js

```
var saludos = require('./greetings.json');

var greet = function() {
 console.log(saludos.en);
}

module.exports = greet;
```

y en `greetings.json` vamos a tener los saludos per se:

```
{
 "en": "Hello",
 "es": "Hola"
}
```

Si ejecutamos nuestra `app.js`:

```
Terminal
File Edit View Search Terminal Help
~/D/P/B/N/i/patrones >>> node app.js
Hello
Hola
~/D/P/B/N/i/patrones >>>
```

## Más Patrones

En la carpeta `./patrones/otro` hemos puesto varios files distintos llamados `saludos` que van del uno al cinco, y luego los importamos en el archivo `app.js`. En cada módulo exportamos lo que necesitamos de manera distinta, cada una de esas formas constituye un patrón.

Dentro de cada archivo en los comentarios está explicado en más detalle el patrón.

De nuevo, **no hay una mejor forma, prueben todos los patrones y vean cual es el que les gusta y cual los hace felices!**

Require con modulos Core o nativos

También podemos importar módulos CORE (los que hablamos al principio) usando `require`. Esta función está preparada para que si no encuentra un archivo con el nombre que le pasamos, busca una carpeta, y si no encuentra busca en los módulos nativos. Podemos ir a la [documentación](#) y ver los módulos que podemos usar.

Veamos como podemos importar algo de esa funcionalidad. Como ejemplo vamos a importar el módulo llamado `utilities`, en la doc vemos que el módulo se llama `util`. Por lo tanto lo vamos a importar escribiendo

```
var util = require('util'); // No usamos ./ porque es un modulo core

var nombre = 'Toni';
var saludo = util.format('Hola, %s', nombre);
util.log(saludo);
```

¿Que hace este código? Busquen en la [documentación](#) por la función `format()` y `log()` e intenten predecir que hará ese código.

## Eventos: Events emitter

Una gran porción del core de Nodejs está construida usando como base este concepto. Un *Evento* es algo que ha ocurrido en nuestra aplicación y que dispara una acción. Este concepto no es exclusivo a *Nodejs*, si no que aparece en muchos lenguajes y arquitecturas de software. En Node hay dos tipo de eventos:

- Eventos del sistemas: Estos vienen del código en C++, gracias a una librería llamada *libuv* y manejan eventos que vienen del sistema operativo como por ejemplo: Terminé de leer una archivo, o recibí datos de la red, etc...
- Eventos Customs: Estos eventos estan dentro de la parte Javascript del Core. Maneja eventos que podemos crear nosotros mismos, para eso vamos a usar el *Event Emitter*. Varias veces cuando un evento de *libuv* sucede, genera un evento usando el *event emitter*.

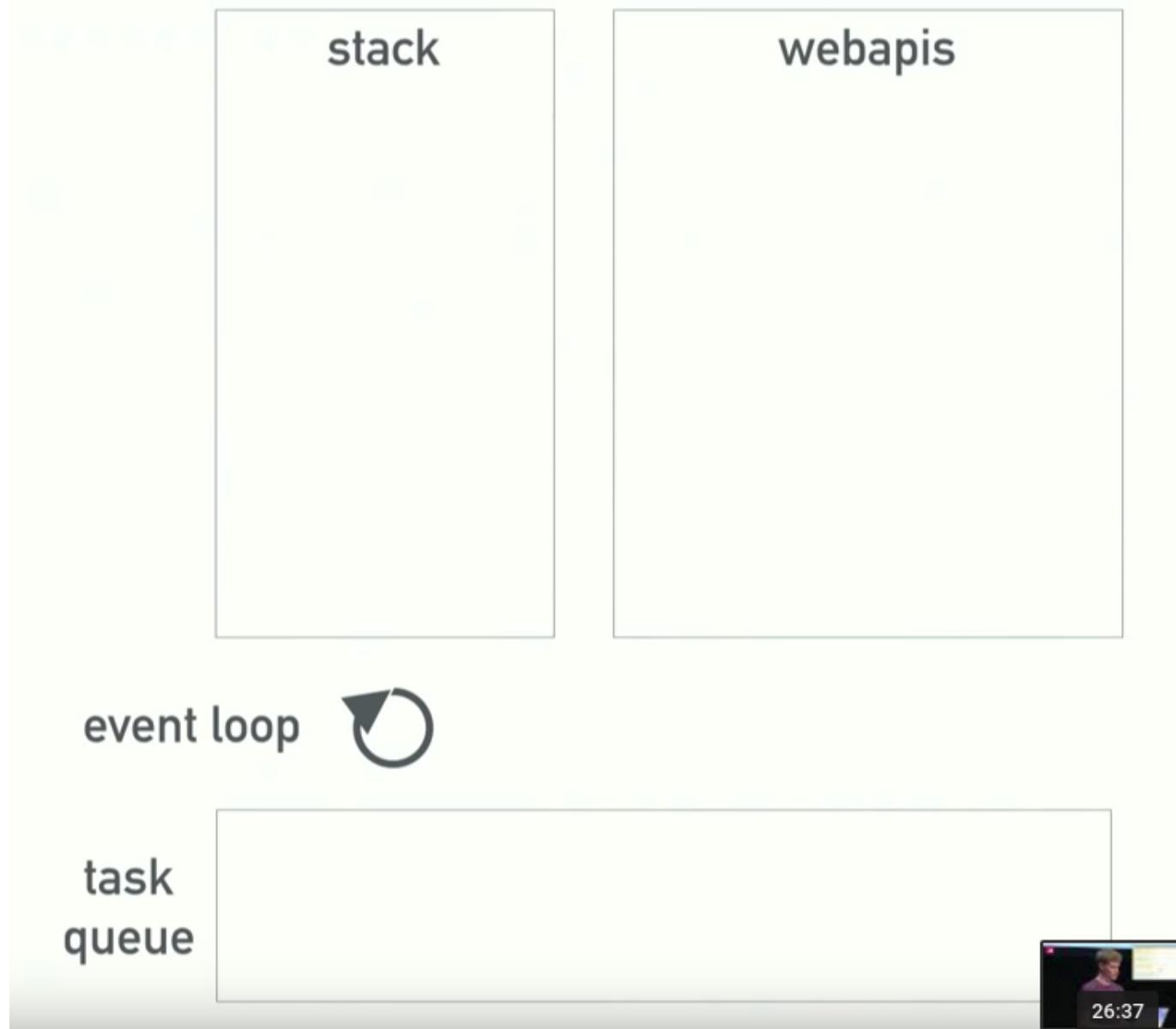
*Ejercicio* Crear un forma simple de emitir eventos y capturarlos.

## Event Listener

Dijimos que cuando ocurre un evento, queremos capturarlo y responder de alguna forma, para eso vamos a hacer uso de los *Events Listeners*, básicamente es el código que *escucha* por un evento y que hace algo (ejecuta código) cuando ese evento sucede. Podemos tener varios listeners escuchando el mismo evento.

## Event Loop

Repasemos como funcionaba V8 internamente y veamos qué es exactamente el **event loop**:



Los componentes que vemos en la figura son:

- Stack: es el runtime de V8, éste va leyendo el código del programa que esté corriendo y va poniendo cada instrucción en el stack para ejecutarla. **Javascript es Single threaded, o sea que sólo puede ejecutar una instrucción a la vez**
- SO / Webapis: Esta pila es manejada por el sistema Operativo. Básicamente V8 le delega tareas al SO, por ejemplo: Bajame esto de internet, o leeme este archivo, o comprimí esta imagen, etc... **El SO puede hacer estas tareas en un thread o varios, es transparente para nosotros**. Lo único que nos importa es cuando el SO *completó* la tarea que se le pidió. Cuando lo hizo, nos avisa y pasa el *callback* al task queue.
- Task Queue: Cuando el SO nos indica que terminó una tarea, no podemos simplemente pasar el *callback* al stack de JS (no sabemos que está haciendo y podría correr código en un momento inoportuno), por lo tanto, lo que hace es dejar el *callback* en el *Task Queue* (es una cola tipo FIFO). Ahora, cuando el Stack JS está vacío el TANTAN TATAN... **EVENT LOOP** agarra el *callback* del Queue y lo pasa al Stack JS para ser ejecutado!

Toda esta interacción con el SO, el manejo del Task Queue y el Event loop está implementado en la librería [libuv](#) que es la piedra angular de Nodejs. De hecho, su logo es un Unicornio T-rex, es demasiado copada:



## ¿Qué es un gestor de paquetes?

---

Primero definamos lo que es un paquete. Básicamente es.. [código!](#) Es cualquier pieza de **código** manejada y mantenida por un gestor de paquetes.

Ahora, un gestor de paquetes es un software que automatiza la instalación y actualización de paquetes. Maneja qué versión de los paquetes tenés o necesitas y maneja sus *dependencias*. Una **dependencia** es código del cual dependen uno o más pedazos de código para funcionar. Por ejemplo, si usás [fs](#) en tu servidor, entonces [fs](#) es una dependencia de tu server. Sin el código de [fs](#) tu servidor no podría ejecutarse. De hecho, cada paquete podría tener sus propias dependencias, es por esto que manejarlos a mano se podría volver un poco engorroso, por suerte tenemos los gestores de paquetes que nos solucionan este problema.

## NPM: Node Package Manager

Npm es el gestor de paquetes que viene con Nodejs y que gestiona los paquetes para este. Para probar si tenés npm podés hacer [npm -v](#) en tu consola, y si recibis algo como esto: [3.10.5](#) quiere decir que lo tenés instalado.

Para ver el registro de paquetes podés ir a [esta](#) página. Acá podés buscar paquetes individuales, ver su información y bajarlos. Cualquiera puede subir sus paquetes en npm, así que tenés que tener cuidado con qué paquetes bajar.

Para instalar un paquete se utiliza el comando [npm](#) con el argumento [install](#) y el nombre del paquete. Por ejemplo, para instalar el paquete 'express' vamos a la consola y tipeamos: [npm install express](#).

Para poder trackear las dependencias y los paquetes que tenemos instalados, npm hace uso de un archivo de *configuración* al que llama **package.json**. Este es básicamente un archivo de texto en formato JSON con el listado de dependencias de tu aplicación, de esta forma con sólo compartir ese archivo cualquiera sabrá qué paquetes se deben instalar, e incluso hacerlo de forma automática.

Para crear este archivo npm nos da el comando **npm init**, que es una forma interactiva de crear el 'package.json'.

- entry point: Indica cuál es el archivo javascript que Node debe correr para arrancar la aplicación.

En la imagen vemos los datos que nos piden y a continuación vemos el archivo **package.json** que creó el comando:

```
{
 "name": "test-app",
 "version": "1.0.0",
 "description": "Esta es una aplicación de prueba",
 "main": "index.js",
 "scripts": {
 "test": "echo \\\"Error: no test specified\\\" && exit 1"
 },
 "keywords": [
 "Prueba"
],
 "author": "Toni Tralice",
 "license": "ISC"
}
```

Para que veamos como funcionan las dependencias, vamos a instalar algunos paquetes y requerirlos dentro de nuestra aplicación de prueba.

*Los paquetes instalados de forma local serán guardados en una carpeta llamada **node\_modules** creada dentro de la carpeta donde ejecuté el comando*

## moment

Es una librería de javascript para manejar fechas. Para instalarlo hacemos: **npm install moment --save**.

**usamos el argumento **--save** para indicar a npm que además de instalar el paquete, lo agregue a la lista de dependencias en el archivo **package.json**.**

Luego de ejecutar el comando, vemos que en **package.json** hay una nueva propiedad llamada 'dependencies' que es un objeto que contiene los nombres y las versiones de los paquetes que hemos instalado (siempre que instalemos usando **--save**).

```
{
 "name": "test-app",
 "version": "1.0.0",
 "description": "Esta es una aplicación de prueba",
```

```
"main": "index.js",
"scripts": {
 "test": "echo \"Error: no test specified\" && exit 1"
},
"keywords": [
 "Prueba"
],
"author": "Toni Tralice",
"license": "ISC",
"dependencies": {
 "moment": "^2.14.1"
}
}
```

Ahora creamos un archivo llamado `index.js` (el entry point de nuestra app), y dentro de él vamos a incorporar el nuevo módulo instalado.

```
var moment = require('moment');
console.log(moment().format("ddd/MM/YYYY, hA"));
```

Como verán, Nodejs ya sabe en qué carpetas buscar el módulo 'moment' y no tenemos que explicitarlo diciendo en qué carpeta está, de la forma `./node_modulos/moment`.

## NPM paquete globales

Con lo anterior hemos instalado el paquete 'moment' al que podremos acceder desde la aplicación test.

Ahora vamos a ver que hay paquetes de npm que nos son útiles en todas las aplicaciones que hacemos, por lo tanto los queremos instalar de manera *global*. Para hacerlos usamos el argumento `-g` en `npm install`.

*Según el sistema operativo pueden llegar a tener algún problema con permisos, si es el caso pueden buscar soluciones en [esta página](#). La carpeta donde se instalan los módulo globales también dependen del SO.*

## Nodemon

Se acuerdan cuando hicimos el servidor web con Node? Cada vez que cambiamos algo en el código, teníamos que reiniciar el servidor para que los cambios sean reflejados. Con *Nodemon* podemos olvidarnos de eso, ya que hace eso por nosotros.

```
nodemon will watch the files in the directory in which nodemon was started, and if any files change, nodemon will automatically restart your node application.
```

Como verán este es un paquete que vamos a usar en casi todos los proyectos de node que hagamos, por lo tanto es un buen ejemplo de algo que instalaríamos globalmente. Para hacerlo hacemos:

```
npm install -g nodemon
```

o

`sudo npm install -g nodemon` en Linux o Mac.

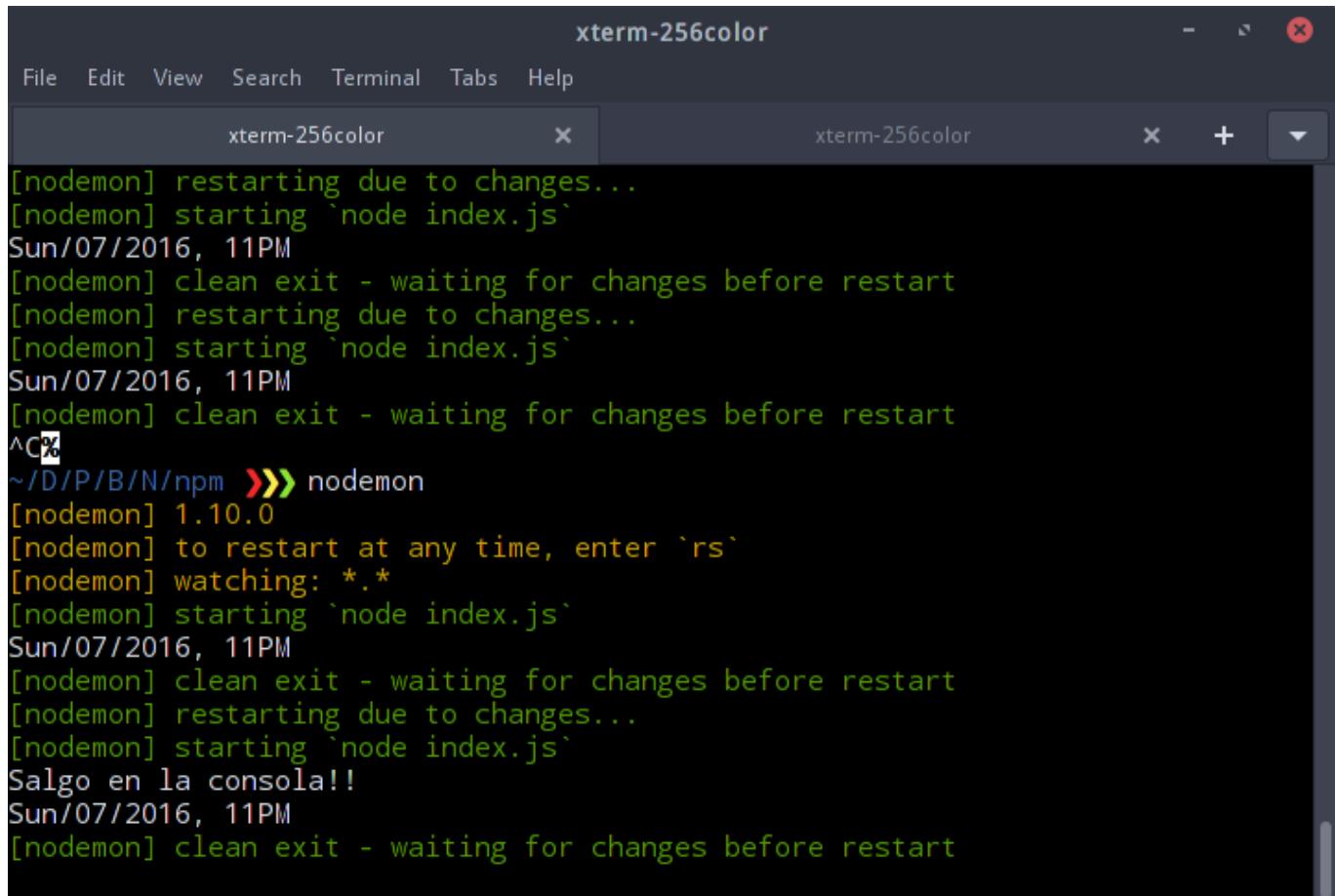
Una vez instalado globalmente, puedo simplemente utilizarlo desde la línea de comandos, ya que viene con una interfaz CLI (*command line interface*). De hecho, varios paquetes de npm vienen con una interfaz CLI para ser usados desde la consola o terminal.

En nuestro ejemplo al ejecutar `nodemon` obtenemos:

```
[nodemon] 1.10.0
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: ***!
[nodemon] starting `node index.js`
Sun/07/2016, 11PM
[nodemon] clean exit - waiting for changes before restart
```

Para probar si funciona, hagamos un cambio en el archivo `index.js` y salvemos. Para mi ejemplo voy a agregar un `console.log()`.

```
var moment = require('moment');
console.log('Salgo en la consola!!');
console.log(moment().format("ddd/MM/YYYY, hA"));
```



The screenshot shows a terminal window titled "xterm-256color". The window has a menu bar with File, Edit, View, Search, Terminal, Tabs, and Help. There are two tabs open: "xterm-256color" and another "xterm-256color" tab which is currently active. The terminal output is as follows:

```
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
Sun/07/2016, 11PM
[nodemon] clean exit - waiting for changes before restart
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
Sun/07/2016, 11PM
[nodemon] clean exit - waiting for changes before restart
^C%
~/D/P/B/N/npm ➜ nodemon
[nodemon] 1.10.0
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: ***!
[nodemon] starting `node index.js`
Sun/07/2016, 11PM
[nodemon] clean exit - waiting for changes before restart
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
Salgo en la consola!!
Sun/07/2016, 11PM
[nodemon] clean exit - waiting for changes before restart
```

Como vemos, no fue necesario volver a correr el archivo `index.js` a mano para ver los cambios, Nodemon hizo todo el trabajo por nosotros!

## Actualizar paquetes

Como dijimos, `npm` también nos sirve para mantener actualizados los paquetes. Para hacerlo sólo tenemos que escribir el siguiente comando:

```
npm update
```

Para mantener la compatibilidad con las aplicaciones, `npm` sólo actualiza automáticamente los *patches* y *minor changes* ([Semantic Versioning](#)) de un paquete, por defecto.

De hecho, el `^` antes del número de versión en 'dependencies' indica qué puede actualizar los *minors* automáticamente, si quisieramos restringir ese comportamiento para que actualice sólo los *patches* deberíamos usar el carácter `~` antes de la versión:

```
"dependencies": {
 "moment": "^2.14.1" //Actualiza sólo Minors
}

"dependencies": {
 "moment": "~2.14.1" //Actualiza sólo Patches
}
```

## Leyendo un archivo con Node

Para trabajar con archivos en Node, vamos a usar el módulo `fs` que tiene la funcionalidad para leer y escribir archivos, como se encuentra en las librerías core de node, vamos a requerirlo usando `require('fs')`.

Ahora vamos a crear un archivo llamado `greet.txt` y vamos a escribir un saludo dentro, lo vamos a dejar en la misma carpeta que nuestro archivo `.js`.

Ahora vamos a leer el contenido de ese archivo y mostrarlo por consola:

```
var fs = require('fs');

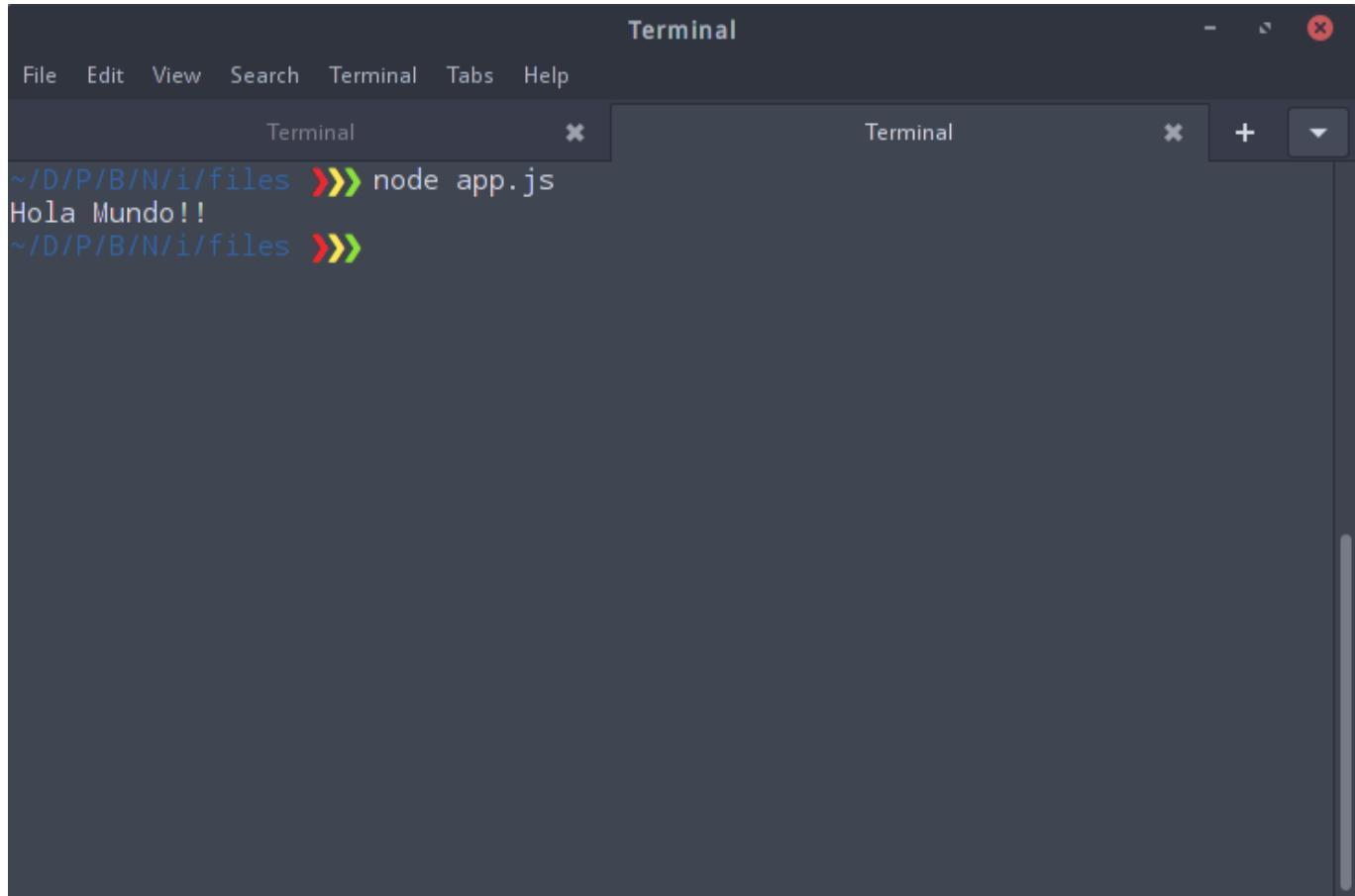
var saludo = fs.readFileSync(__dirname + '/greet.txt', 'utf8');
console.log(saludo);
```

Como podemos ver en la [documentación](#) de `fs`, la función `readFileSync`, recibe un path como parametro y el encoding del file.

**`__dirname: esta variable tiene guardado el path completo del directorio donde está el archivo que está ejecutando el código`**

Por lo tanto le estamos diciendo que lea el archivo `greet` que creamos *DE MANERA SINCRONICA* (esto quiere decir que no va avanzar hasta que no lea de manera completa) y que guarde el resultado en la variable

saludo, luego que muestre el contenido de saludo por consola. Como vemos, el resultado es que en saludo se guarda el contenido del archivo greet.txt!!



The screenshot shows a terminal window with two tabs. The active tab is titled 'Terminal' and contains the command 'node app.js' followed by the output 'Hola Mundo!!'. The second tab is also titled 'Terminal' and is currently inactive.

Al hacerlo sincronico, el programa se *bloquea* hasta que no termine de leer el archivo completo, si el archivo fuera muy grande, veríamos que el programa queda *trabado* y causa una mala experiencia de uso.

Para resolver ese problema, vamos a hacer lo mismo, pero haciendo que la lectura del archivo sea en forma asincrónica. Para hacerlo, `fs` nos da la función `readFile`, que recibe el path del archivo a leer, el encoding, y además recibe una función, que será el `callback` para cuando se termine de leer el archivo:

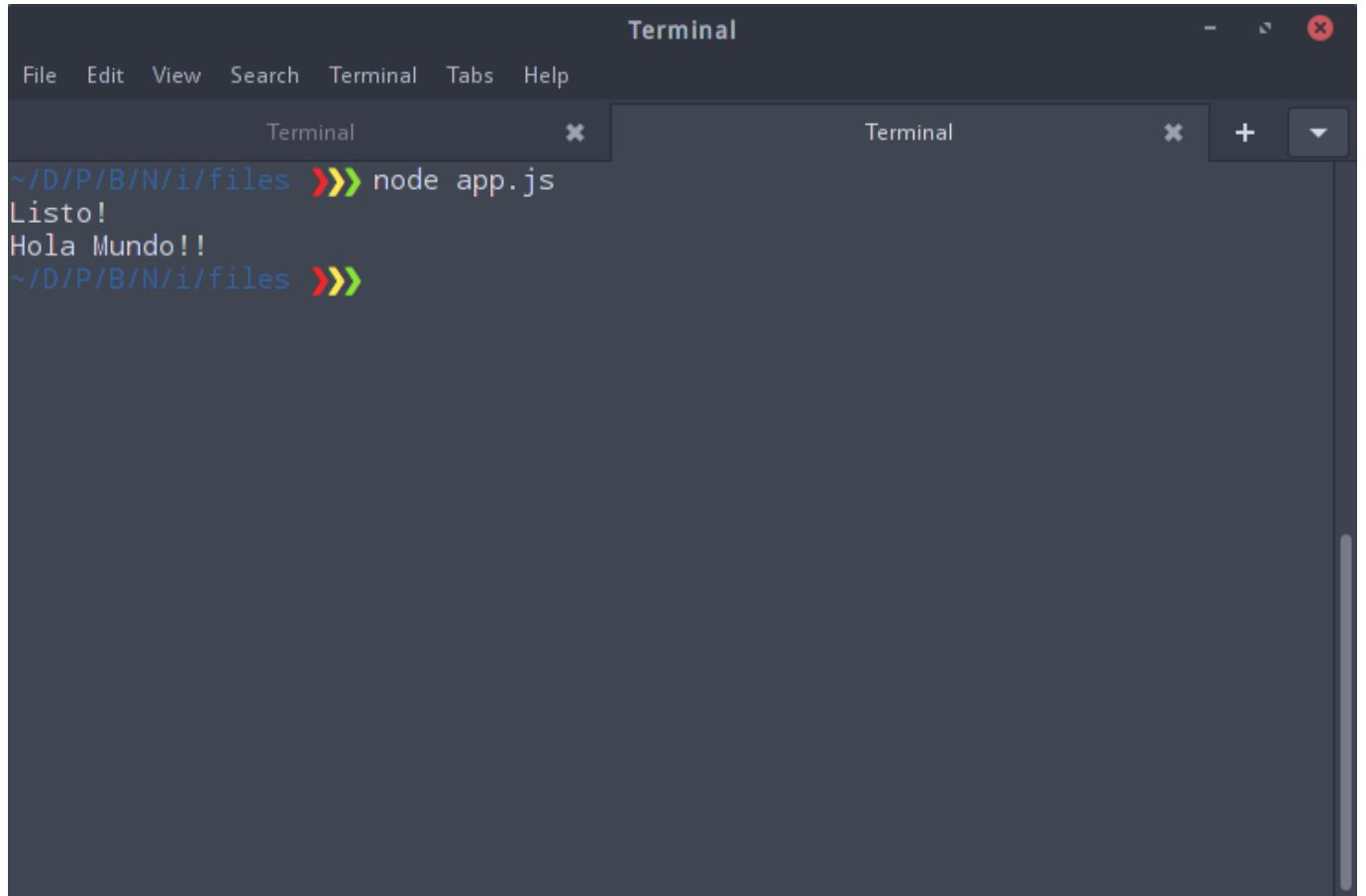
```
var fs = require('fs');
fs.readFile(__dirname + '/greet.txt', 'utf8', function(err, data) {
 console.log(data);
});
console.log('Listo!');
```

Hemos agregado al final el `console.log('Listo!')` para que vean en qué momento se ejecuta esa linea de código y en qué momento se ejecuta el callback.

**Es super importante que comprendan el porqué del orden en el que aparecen los `console.logs`, no avances antes de comprenderlo. Pista: Mirá más arriba como funciona el event loop de JS.**

Otra cosa a notar es que la función anónima que le pasamos tiene dos parámetros: `err` y `data`. Esto se debe a que existe un standard llamado **error-first callback** que dice que cada vez que escribas una función que ejecute un callback, le pases a ese `cb` como **primer** parámetro una variable de `Error`. En el caso que no haya

habido errores en la ejecución, entonces esa variable tendrá *null*, en caso contrario tendrá algo que informe el error.

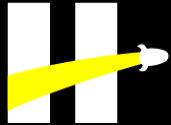


The screenshot shows a terminal window with two tabs. The active tab is titled 'Terminal' and contains the command `node app.js`. The output of the script is displayed: 'Listo!' followed by 'Hola Mundo!!'. The second tab is also titled 'Terminal' and is currently inactive.

Como vemos, primero se ejecutó el segundo console log, luego el console log del *callback*.

## Combinando Todo

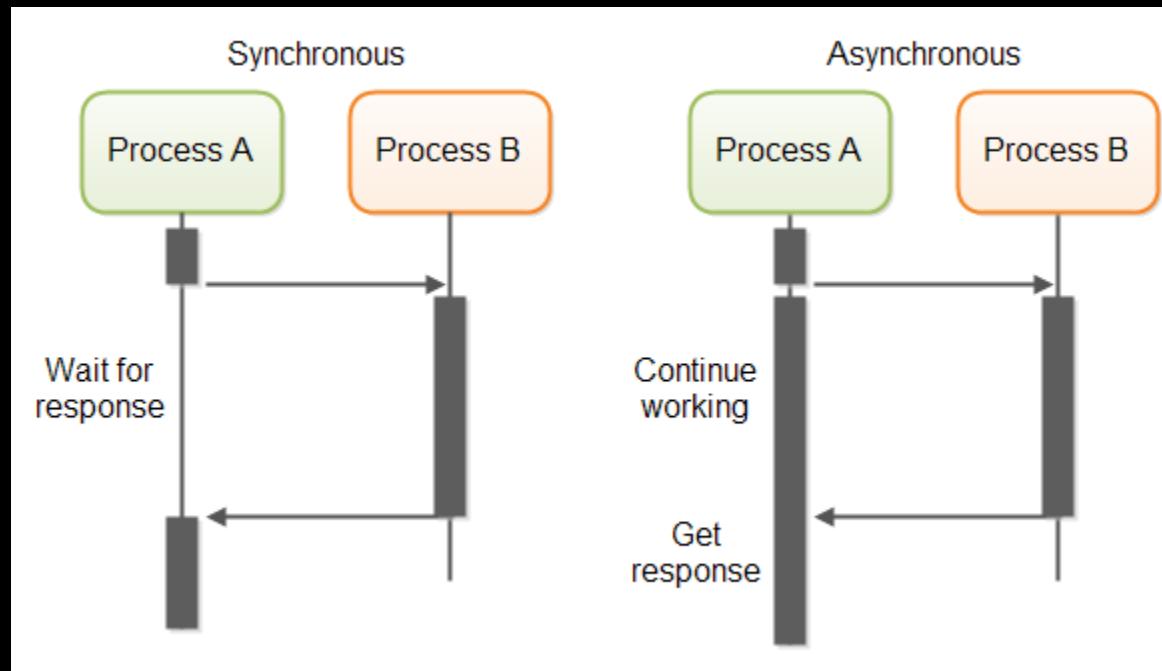
Bien! Ahora estamos listos para empezar a crear nuestro propio servidor web usando estos conceptos!

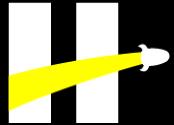


# Promises



# Promises

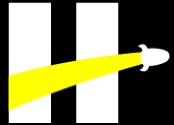




# ¿Qué es una Promise?

*“ Una promesa representa el eventual resultado de una operación asíncrona.*

- The Promise A+ Spec



# Las Promesas son Objetos

Es un objeto que **representa** y **gestiona** el lifecycle de una respuesta futura.

El objeto mantiene dentro suyo:

status( pending, fulfilled, rejected)  
information (value or a reason)

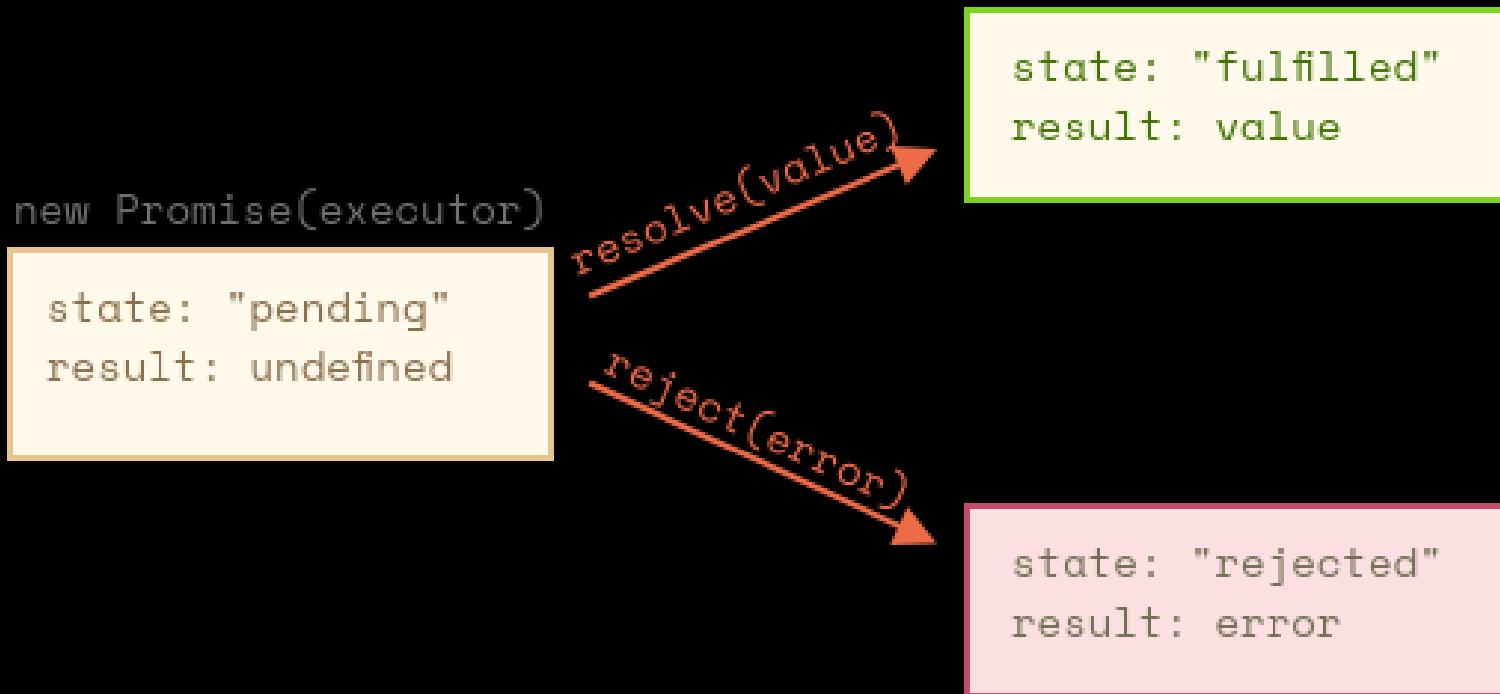
Sólo podemos acceder al método:

.then()



# Las Promesas son Objetos

El estado de las promesas sólo cambia una vez.



unaPromise.then(succesfullHandler, errorHandler)

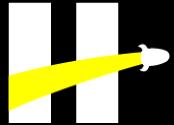


# Promises

Un vez que una promesa se fullfilea, o es rechazada.  
Ejecuta sus handlers

Si le agregamos un handler nuevo, lo ejecuta con el mismo  
valor al que se fullfileó.

```
1 aPromise.then(handler, errorHandler);
2
3 // se fullfilea la promesa, por ejempllo con un valor (sin error);
4 // se ejecuta el handler.
5
6 aPromise.then(handler2);
7
8 // si agregamos un nuevo handler, se ejecuta con el mismo valor!
```



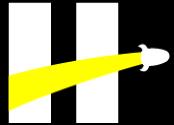
# Creando Promesas

```
1
2 var promise = new Promise(function(resolve, reject) {
3 // Hacer cosas acá dentro, probablemente asincrónicas.
4
5 if /* Todo funcionó como esperabamos */) {
6 resolve("Jooya!");
7 }
8 else {
9 reject(Error("Algo se rompió"));
10 }
11});
```



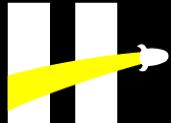
# Callback Hell

```
1 const verifyUser = function(username, password, callback){
2 DataBase.verifyUser(username, password, (error, userInfo) => {
3 if (error) {
4 callback(error)
5 }else{
6 DataBase.getRoles(username, (error, roles) => {
7 if (error){
8 callback(error)
9 }else {
10 DataBase.logAccess(username, (error) => {
11 if (error){
12 callback(error);
13 }else{
14 callback(null, userInfo, roles);
15 }
16 })
17 }
18 })
19 }
20 })
21 };
```



# Callback Hell

```
1 const verifyUser = function(username, password) {
2 database.verifyUser(username, password)
3 .then(userInfo => DataBase.getRoles(userInfo))
4 .then(rolesInfo => DataBase.logAccess(rolesInfo))
5 .then(finalResult => {
6 //do whatever the 'callback' would do
7 })
8 .catch((err) => {
9 //do whatever the error handler needs
10 });
11 };
```



# Encadenado Promesas

.then() retorna una promesa! por eso las podemos encadenar!

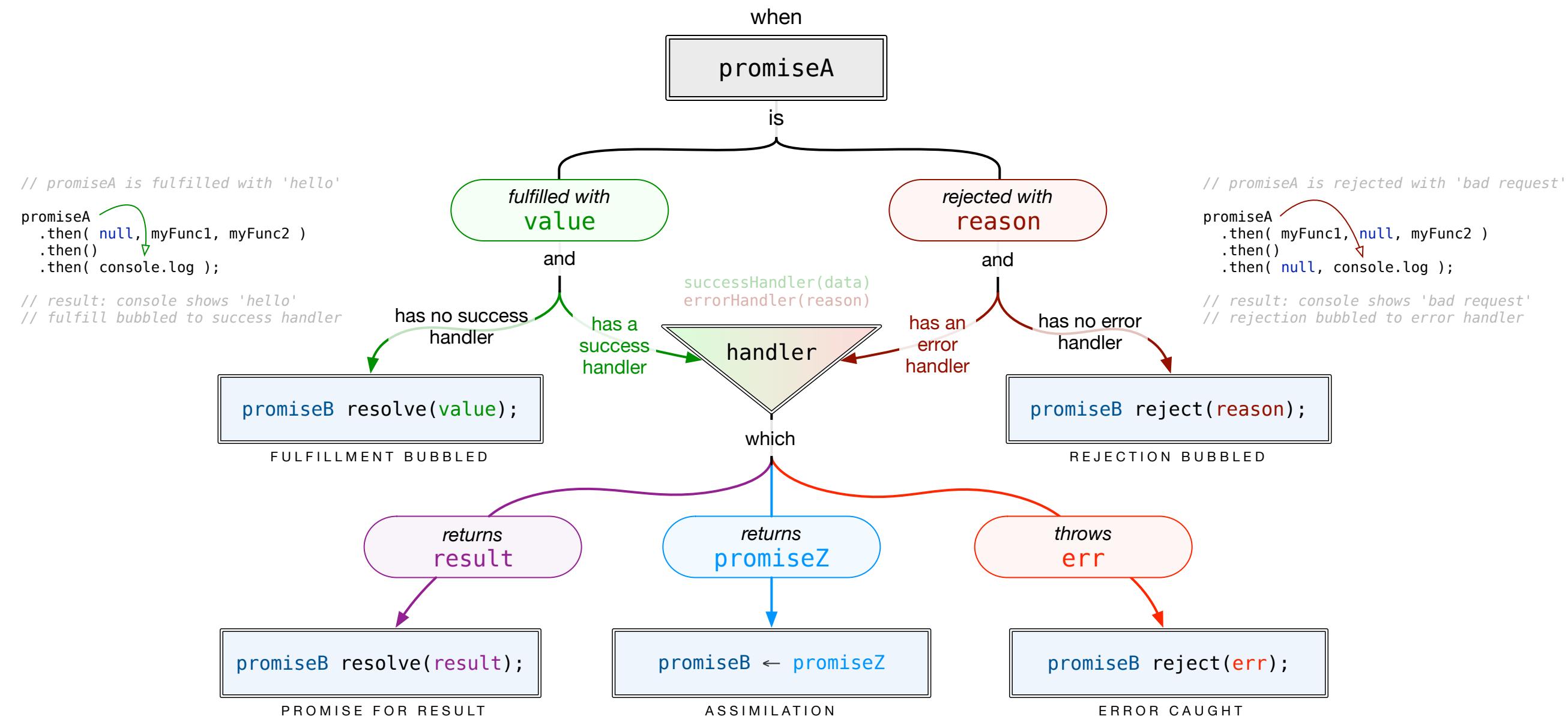
```
1 var primerMetodo = function() {
2 var promise = new Promise(function(resolve, reject){
3 setTimeout(function() {
4 console.log('Terminó el primer método');
5 resolve({num: '123'}); //pasamos unos datos para ver como los manejamos
6 }, 2000); // para simular algo asincronico hacemos un setTimeOut de 2 s
7 });
8 return promise;
9 };
10
11
12 var segundoMetodo = function(datos) {
13 var promise = new Promise(function(resolve, reject){
14 setTimeout(function() {
15 console.log('Terminó el segundo método');
16 resolve({nuevosDatos: datos.num + ' concatenamos texto y lo pasamos'});
17 }, 2000);
18 });
19 return promise;
20 };
21
22 var tercerMetodo = function(datos) {
23 var promise = new Promise(function(resolve, reject){
24 setTimeout(function() {
25 console.log('Terminó el tercer método');
26 console.log(datos.nuevosDatos); //imprimos los datos concatenados
27 resolve('hola');
28 }, 3000);
29 });
30 return promise;
31 };
32
33 primerMetodo()
34 .then(segundoMetodo)
35 .then(tercerMetodo)
36 .then(function(datos){
37 console.log(datos); //debería ser el 'hola' que pasamos en tercerMetodo
38 });

```

< DEMO />

.then() always returns a new promise. What happens to that promise?

```
promiseB = promiseA.then([successHandler], [errorHandler]);
```



// output promise is for returned val

```
promiseForVal2 = promiseForVal1
 .then(function success (val1) {
 val2 = ++val1;
 return val2;
 });

```

// same idea, shown in a direct chain:

```
promiseForVal1
 .then(function success (val1) {
 // do some code to make val2
 return val2;
 })
 .then(function success (val2) {
 console.log(val2);
 });

```

// output promise "becomes" returned promise

```
promiseForMessages = promiseForUser
 .then(function success (user) {
 // do some code to get a new promise
 return promiseForMessages;
 });

```

// same idea, shown in a direct chain:

```
promiseForUser
 .then(function success (user) {
 // do some code to get a new promise
 return promiseForMessages;
 })
 .then(function success (messages) {
 console.log(messages);
 });

```

// output promise will be rejected with error

```
promiseForVal2 = promiseForVal1
 .then(function success (val1) {
 // THROWN ERROR '404' trying to make val2
 return val2;
 });

```

// same idea, shown in a direct chain:

```
promiseForVal1
 .then(function success (val1) {
 // THROWN ERROR '404' trying to make val2
 return val2;
 })
 .then(null, function failed (err) {
 console.log('Oops!', err);
 });

```

# Henry

---



Hacé click acá para dejar tu feedback sobre esta clase.



Hacé click acá completar el quizz teórico de esta lecture.

## Promises

---

Javascript es muy potente a la hora de trabajar con tareas asincrónicas, de hecho, ya venimos programando funciones y código que hacen uso de **callbacks** para ejecutar código en el futuro cercano, cuando un evento sucede o cuando se termina la ejecución de un proceso (escribir en la bd, o escribir en el disco, hacer un request http, etc...). Esto es genial, pero a veces nos sucede que tenemos callbacks anidadas, es decir, que dentro de un callback tenemos otro callback y así sucesivamente ( inception de callbacks ), y tambien tenemos problemas donde tenemos que esperar que *dos o más* eventos terminen para continuar la ejecución de nuestro código. Si bien podemos resolverlo sin problemas con callbacks, vamos a ver que nuestro código empieza a hacerse difícil de leer, muy difícil de controlar si hay errores (no sabemos qué función es la que realmente produjo el error ), y si tenemos que buscar un bug dentro del código nos daremos cuenta que, sin querer, hemos terminado dentro del **Callback Hell**:

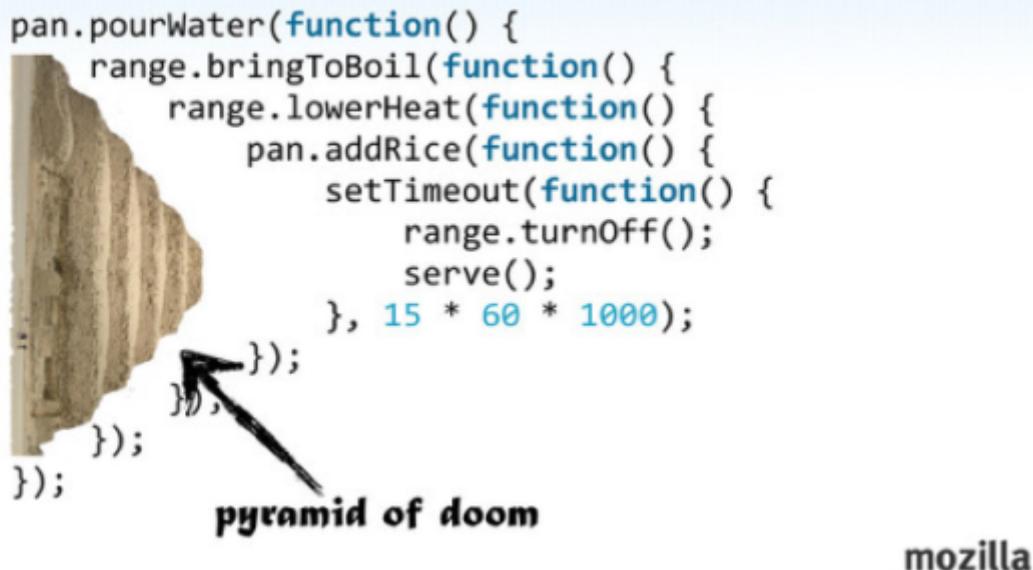
```

function register() {
 if (empty($_POST)) {
 $msg = '';
 if (!$_POST['user_name']) {
 if (!$_POST['user_password_new']) {
 if (!$_POST['user_password_repeat']) {
 if (strlen($_POST['user_name']) < 65 || strlen($_POST['user_password_new']) > 3) {
 if (strlen($_POST['user_name']) < 65 || strlen($_POST['user_name']) > 1) {
 if (preg_match('/^(a-zA-Z)([a-zA-Z\d]{1,14})$/i', $_POST['user_name'])) {
 $user = read_user($_POST['user_name']);
 if (!isset($user['user_name'])) {
 if (!$_POST['user_email']) {
 if (strlen($_POST['user_email']) < 65) {
 if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
 create_user();
 $SESSION['msg'] = 'You are now registered so please login';
 header('Location: ' . $_SERVER['PHP_SELF']);
 exit();
 } else $msg = 'You must provide a valid email address';
 } else $msg = 'Email must be less than 64 characters';
 } else $msg = 'Email cannot be empty';
 } else $msg = 'Username already exists';
 } else $msg = 'Username must be only a-z, A-Z, 0-9';
 } else $msg = 'Username must be between 2 and 64 characters';
 } else $msg = 'Password must be at least 6 characters';
 } else $msg = 'Passwords do not match';
 } else $msg = 'Empty Password';
 } else $msg = 'Empty Username';
 $SESSION['msg'] = $msg;
 }
 return register_form();
}

```

icompile.eladkarako.com

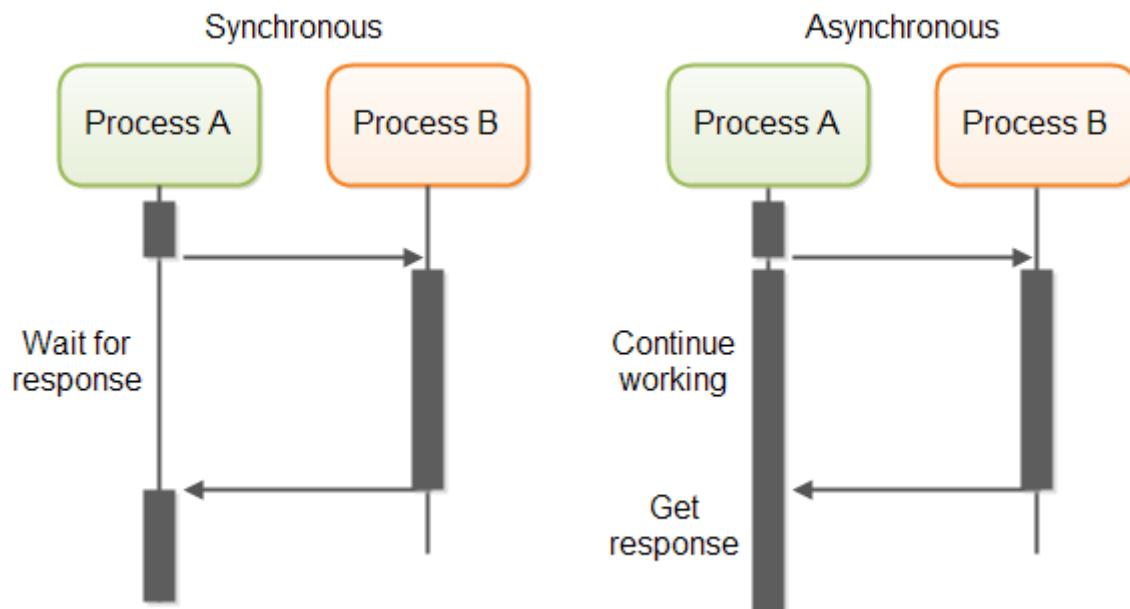
También conocido como **Pyramid of Doom**



Se pueden imaginar, por los nombres que eligieron para esto, que no es una situación deseada en nuestro código. También van a pensar, que es un mal necesario, ya que de esta forma puedo lograr que mi aplicación se comporte de acuerdo a los requerimientos asincrónicos que hay en el medio.

## Promises al rescate

Cuando programamos en lenguajes que son bloqueantes, como C++ o python, perdemos el poder del asincronismo, pero ganamos legibilidad, ya que una línea de código se ejecuta exactamente cuando termina al anterior (si la anterior tarda 3 horas, vamos a esperar a que termine), esto hace que el código sea fácil de leer ya que siguiendo la línea de ejecución vamos a ver que cosas suceden antes o después, esto mismo no lo podemos hacer con callbacks (o por lo menos sin entrar al *Callback hell*):



¿No sería genial si pudieras escribir código como si fuera sincrónico, pero que la ejecución fuera asincrónica? Esta pregunta seguramente se hicieron los inventores de las **Promises** de javascript! Justamente las **Promises**

en Js intentan solucionar el problema del *callback hell* y lograr que el código sea más legible, más fácil de debuggear y que tengamos mayor control sobre los errores. Veamos como funcionan las promises.

## ¿Qué es una Promesa?

Una *promesa* representa el resultado eventual (en un futuro incierto) de una operación asincrónica, como por ejemplo: un registro de una db, una página que pedimos por http, un objeto JSON que es respuesta de un request a un API. Es decir, que representa un *placeholder* (un lugar reservado) donde vamos a guardar la respuesta del método asincrónico ( o un posible error en caso que no sea existosa ). Como nos podemos imaginar, una promesa puede ser exitosa o no, y una misma promesa no se puede ejecutar dos veces, una vez que termina se convierte en **immutable** (no puede cambiar). Además si a una promesa le agregamos un callback (le podemos agregar en cualquier momento), este será ejecutará cuando esta termine. Esto es genial, porque ya no nos interesa en qué momento se producen las cosas, si no en reaccionar al resultado de esas cosas.

Terminología de promises:

Una promesa puede estar:

- *Pendiente* (pending): El estado inicial de un promise.
- *Completada* (fullfilled): Representa que se completó de manera exitosa.
- *Rechazada* (rejected): La operación terminó, pero de manera fallida.
- *Terminada* (settled): La operación terminó, de cualquiera de las dos maneras anteriores.

## Promises en JavaScript

Antes de **ES6** (EcmaScript 6) javascript no soportaba nativamente las *Promises*, pero como el concepto ya estaba dando vuelta en la cabeza de algunos desarrolladores aparecieron varias librerías que lo implementaban, por ejemplo:

- [Q](#)
- [When](#)
- [WinJS](#)
- [RSPV.js](#)
- [Bluebird](#)

Si bien el concepto es el mismo, y de hecho se creó un standard para las promises: el [Promises/A+](#), no todas las implementaciones son iguales ni respetan el standard. Ustedes se preguntarán porque no usamos directamente las promises que definieron los científicos de EcmaScript, pero resulta que hay un *serio* debate online por las performances de las mismas en términos de tiempo y memoria, la gente de Bluebird hizo algunos [benchmarks](#), y si los examinamos vemos que las promises *nativas* de **ES6** están todavía muy abajo en la lista. Esto es debido a la implementación de cada librería sobre las promises, y por lo visto la implementación nativa no es tan buena todavía. Así que por ahora todavía nos conviene usar alguna librería externa, como por ejemplo *Bluebird* que veremos más abajo.

El autor de *Bluebird* explica un poco porqué tanta diferencia de performance en este [link](#).

## Creando una promesa

Una *Promise* es una clase en JavaScript, así que para instanciar una nueva vamos a el keyword `new`. Una *Promise* recibe un sólo argumento: una función con dos parámetros: `reject` y `resolve`. Dentro de esta función vamos a hacer lo que necesitemos y, si todo salió bien, llamamos `resolve` y si no, (algo salió mal), llamamos `reject`.

```
var promise = new Promise(function(resolve, reject) {
 // Hacer cosas acá dentro, probablemente asincrónicas.

 if /* Todo funcionó como esperabamos*/) {
 resolve("Jooya!");
 }
 else {
 reject(Error("Algo se rompió"));
 }
});
```

En el método `reject` podemos pasar cualquier cosa, pero se recomienda devolver un objeto de tipo `Error` ya que estos tienen el `stack trace` y es más fácil de debugear.

## Haciendo algo cuando una promesa se '*cumple*'

En el statement anterior hemos *definido* una promesa y la hemos guardado en la variable `promise`. Ahora, para usar esa promesa, debemos de alguna forma poder decirle qué hacer cuando se resuelva o se rechaze la promesa. Para eso vamos a usar el método `then`:

```
promise.then(function(data) {
 // Ejecuto código sabiendo que todo salió bien
 // Siguiendo el ejemplo de arriba, data tendría adentro el string: 'Jooya!'
}, function(error) {
 // La promesa fue rechazada, aca ejecutamos código para ese caso
 // Siguiendo el ejemplo de arriba, error tendría adentro el string: 'Algo se
 rompió'
});
```

La función `then` de las *promises* recibe dos argumentos, un callback de `success` y un callback de `failure`, que van a ser llamadas según si el promise terminó en un `resolve` o un `reject` respectivamente. Los parámetros que le llegan a esta función (en el ejemplo `data` y `error`) son los mismos parámetros con los que llamamos a las funciones `resolve` y `reject`. Podemos escribir los mismo que arriba, pero en vez de pasar dos callbacks a `then`, vamos a usar otro método similar llamado `catch`. Básicamente, al separarlos de este modo, con el `then` vamos a llamar un callback cuando el Promise termina en éxito, y con `catch` vamos a llamar un callback cuando termina en error:

```
promise.then(function(data) {
 // Ejecuto código sabiendo que todo salió bien
 // Siguiendo el ejemplo de arriba, data tendría adentro el string: 'Jooya!'
}).catch(function(error) {
```

```
// La promesa fue rechazada, aca ejecutamos código para ese caso
// Siguiendo el ejemplo de arriba, error tendría adentro el string: 'Algo se
rompió'
});
```

Este último pedazo de código con **then-catch** es equivalente (hace lo mismo) que el anterior donde **then** recibe dos callbacks. Es simplemente una forma más simple de escribir código.

## Encadenando Promesas

Algo muy potente de las *Promises* es que vamos a poder encadenarlas (*chaining*), ya que podemos hacer que un *Promise* **retorne** otro *Promise*, y de esa forma ir encadenando métodos. Por ejemplo:

```
var primerMetodo = function() {
 var promise = new Promise(function(resolve, reject){
 setTimeout(function() {
 console.log('Terminó el primer método');
 resolve({num: '123'}); //pasamos unos datos para ver como los manejamos
 }, 2000); // para simular algo asincronico hacemos un setTimeOut de 2 s
 });
 return promise;
};

var segundoMetodo = function(datos) {
 var promise = new Promise(function(resolve, reject){
 setTimeout(function() {
 console.log('Terminó el segundo método');
 resolve({nuevosDatos: datos.num + ' concatenamos texto y lo pasamos'});
 }, 2000);
 });
 return promise;
};

var tercerMetodo = function(datos) {
 var promise = new Promise(function(resolve, reject){
 setTimeout(function() {
 console.log('Terminó el tercer método');
 console.log(datos.nuevosDatos); //imprimos los datos concatenados
 resolve('hola');
 }, 3000);
 });
 return promise;
};

primerMetodo()
 .then(segundoMetodo)
 .then(tercerMetodo)
 .then(function(datos){
 console.log(datos); //debería ser el 'hola' que pasamos en tercerMetodo
 });
```

En el ejemplo hemos creado tres métodos donde simulamos algo asincrónico, o sea que no sabemos cuando se va a terminar de ejecutar y como vemos, todos crean una *Promise* nueva dentro de ellos y la **retornan**. Para llamarlos, invocamos al primer métodos y le decimos con **then** que si termina exitosamente ejecute la función **segundoMetodo**, esta también devuelve una *Promise*, por lo tanto también podemos llamar a **then** sobre ella, con esto invocamos **tercerMetodo** (que tambien retorna una *Promise*) y a este última le pasamos una función anónima pidiendo que imprima por consola los *datos* que recibió como argumento en **resolve**. Si lo ejecutan en su browser verán cómo es el flujo de datos y en qué orden se imprimen los **console.logs**.

Intenten hacer el mismo ejemplo, pero sin usar Promises. Haciéndolo van a notar la diferencia y vean cuan inentendible puede ser el  *Callback Hell*

Esperando que varias Promesas se cumplan para hacer algo

A veces no sólo necesitamos esperar por un evento para hacer algo, si no que queremos que varios eventos hayan sucedido para actuar. Por ejemplo, quiero guardar un arreglo de **alumnos** en una base de datos, y cuando estén todos correctamente guardados mostrar la respuesta al usuario. Para eso, la **API** de *Promises* no da el método **all**, el cual toma un arreglo de *Promises* y crea un *Promise* que se completa cuando todas las *Promises* que les pasamos estén completas. Al devolver una *Promise* vamos a poder invocar el método **then** sobre ella, y en el callback que les pasemos vamos a recibir como argumento un arreglo de los resultados de las *Promises* que les pasamos, en el mismo orden que se las pasamos:

```
Promise.all(arregloDePromesas).then(function(arregloDeResultados) {
 //...
});
```

Vamos a usar el ejemplo anterior, pero ahora queremos que los métodos se ejecuten sólo cuando se completaron los tres, para eso vamos a tener que cambiar cada método para que no utilice ningún dato del método anterior (ahora estamos llamando los tres al mismo tiempo) y la última parte y reemplazarla por **all**, en el callback de esta vamos a hacer un **console.log** del arreglo que nos devolvió:

```
var primerMetodo = function() {
 var promise = new Promise(function(resolve, reject){
 setTimeout(function() {
 console.log('Terminó el primer método');
 resolve({num: '123'}); //pasamos unos datos para ver como los manejamos
 }, 2000); // para simular algo asincronico hacemos un setTimeOut de 2 s
 });
 return promise;
};

var segundoMetodo = function() {
 var promise = new Promise(function(resolve, reject){
 setTimeout(function() {
 console.log('Terminó el segundo método');
 resolve({texto: 'segundo metodo'});
 }, 2000);
 });
 return promise;
};

Promise.all([primerMetodo(), segundoMetodo()]).then(function(resultados) {
 console.log(resultados);
});
```

```

 }, 1000);
 });
 return promise;
};

var tercerMetodo = function(datos) {
 var promise = new Promise(function(resolve, reject){
 setTimeout(function() {
 console.log('Terminó el tercer método');
 resolve({hola:1});
 }, 3000);
 });
 return promise;
};

Promise.all([primerMetodo(), segundoMetodo(), tercerMetodo()])
.then(function(resultado){
 console.log(resultado); //un arreglo con los valores pasamos a resolve en
cada metodo
});

```

Genial, como vemos las promesas se completaron según cuanto tardaba cada una, en el ejemplo pusimos a propósito que el segundo método tarde menos que los demás. Lo importante a notar es que en el arreglo que recibimos al final los resultados vienen ordenados **en el mismo orden en el que pasamos las promises a all y no en el orden en que se resuelven.**

Seguramente se preguntarán qué pasa cuando una *Promise* es rechazada ( o sea que su ejecución termina en un *reject*). *Promise.all* fue diseñada con un comportamiento llamado *fail-fast*, esto quiere decir que ni bien existe un *reject*, *all* ya lo reporta y devuelve el error. Es decir que el método *catch* va a seguir cómo antes, recibiendo sólo un parámetro (el error), y este corresponderá a la *Promise* que falló primero.

Modifiquemos el código para que el método del medio termine en error. Vamos a rechazar dos promesas, una que termine primero y otra al final, vamos a ver que siempre se mostrará el error de la que **falla primero**:

```

var primerMetodo = function() {
 var promise = new Promise(function(resolve, reject){
 setTimeout(function() {
 console.log('Terminó el primer método');
 reject('Esto es una prueba controlada');
 }, 2000); //
 });
 return promise;
};

var segundoMetodo = function() {
 var promise = new Promise(function(resolve, reject){
 setTimeout(function() {
 console.log('Terminó el segundo método');
 resolve({texto:' segundo metodo'});
 }, 1000);
 });
}

```

```
 return promise;
};

var tercerMetodo = function(datos) {
 var promise = new Promise(function(resolve, reject){
 setTimeout(function() {
 console.log('Terminó el tercer método');
 reject('Tercer método falla');
 }, 3000);
 });
 return promise;
};

Promise.all([primerMetodo(), segundoMetodo(), tercerMetodo()])
 .then(function(resultado){
 console.log(resultado); //un arreglo con los valores pasamos a resolve en
cada metodo
 })
 .catch(function(err){
 console.warn(err); //mostramos el error por consola. Veremos que es el que
falló primero, o sea el del primer metodo
 });
}
```

Noten que la Promise retorna y ejecuta el `catch` inclusive antes que termine el tercer método, esto se debe a que el primer método falla antes que se ejecute el tercero.

## BlueBird

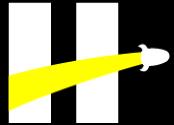
BlueBird es una librería enfocada en *Promises*, esta cumple con el standard de [Promises/A+](#). Como dijimos antes, esta librería es más performante que las *Promises* nativas de **EC6** y sus diseñadores se aprovecharon de ello ya que las dos librerías son perfectamente intercambiables, es decir, que la sintaxis de *Promises* en nativo es idéntica a la de *Bluebird* (excepto en un [estos](#) casos puntuales). Por lo tanto, para usarla vamos primero a instalarla: `npm install bluebird` y luego, vamos a reemplazar el objeto `Promise` que contiene la clase de las *Promises* con el módulo de *Bluebird*:

```
var Promise = require("bluebird");
```

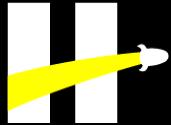
Listo! ahora podemos usar las Promesas de Bluebird mejorando la performance de nuestra aplicación!



Para ver todo lo que se puede hacer con *Promises* podemos ir a leer la [documentación](#) de Bluebird.

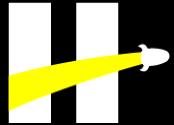


# Web Server

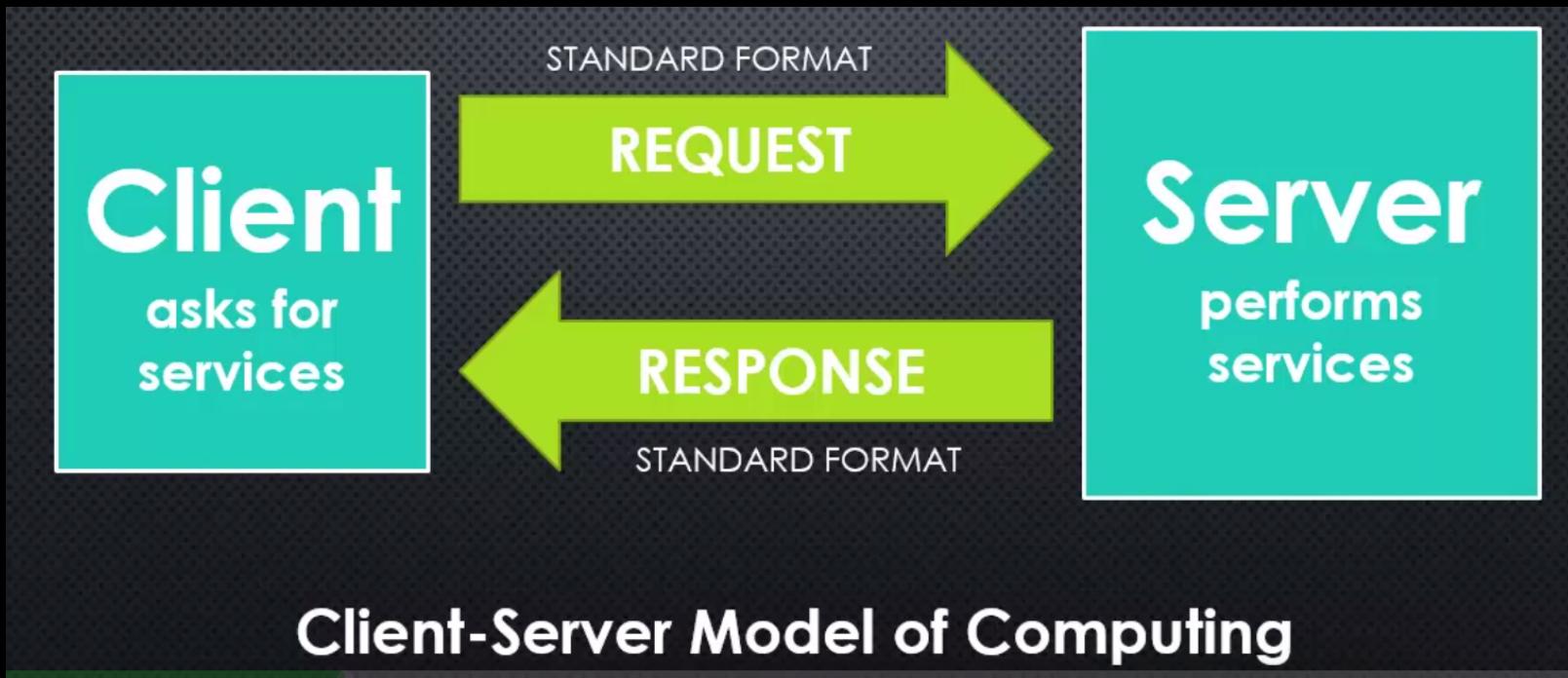


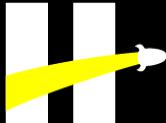
# Web Server

*“ Un servidor web es cualquier computadora o sistema que procese solicitudes (requests) y que devuelva una respuesta (response) a través de un protocolo de red.*

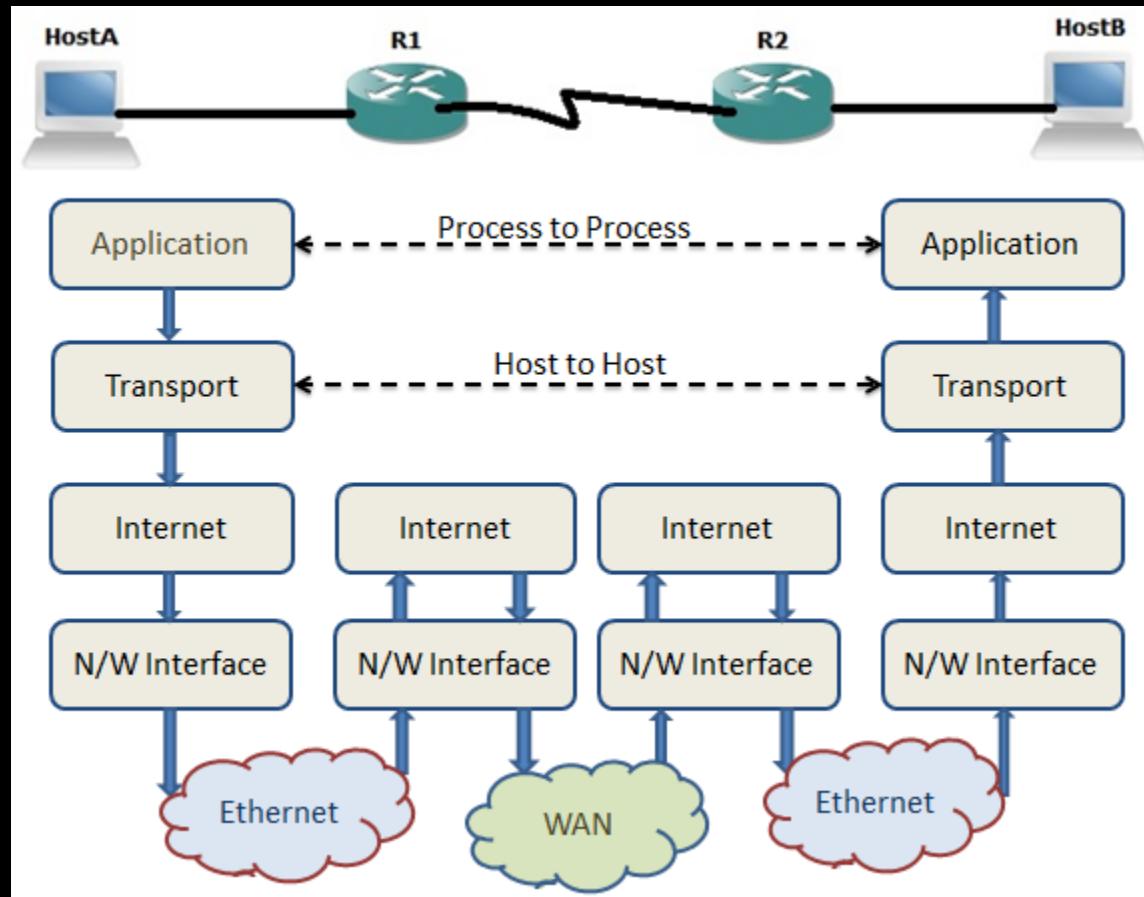


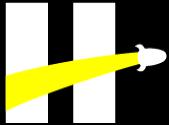
# Modelo Cliente-Servidor



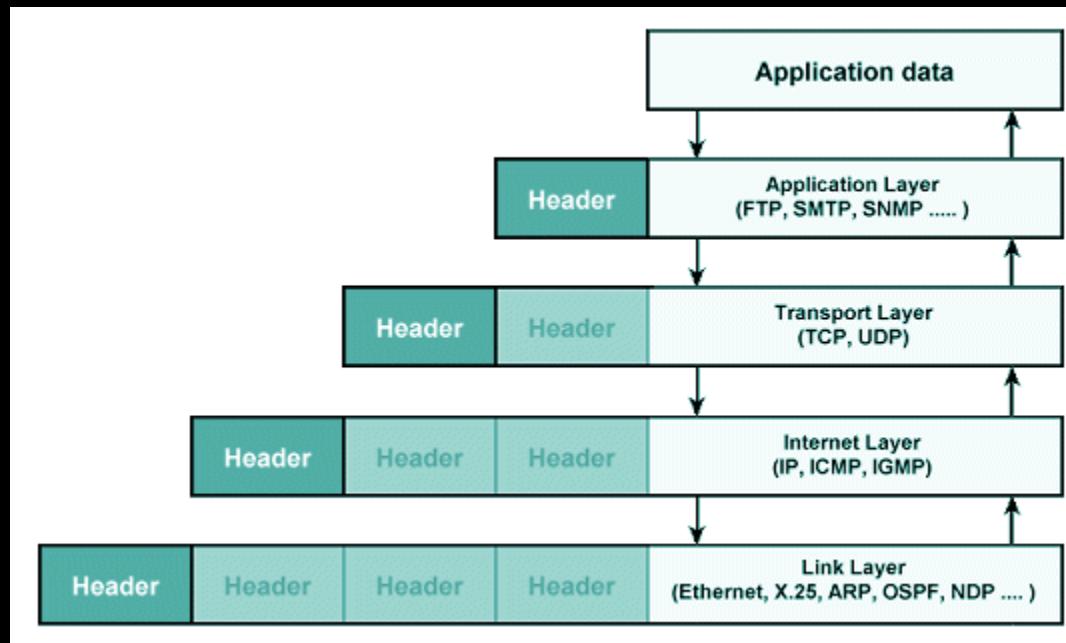


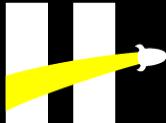
# Networks



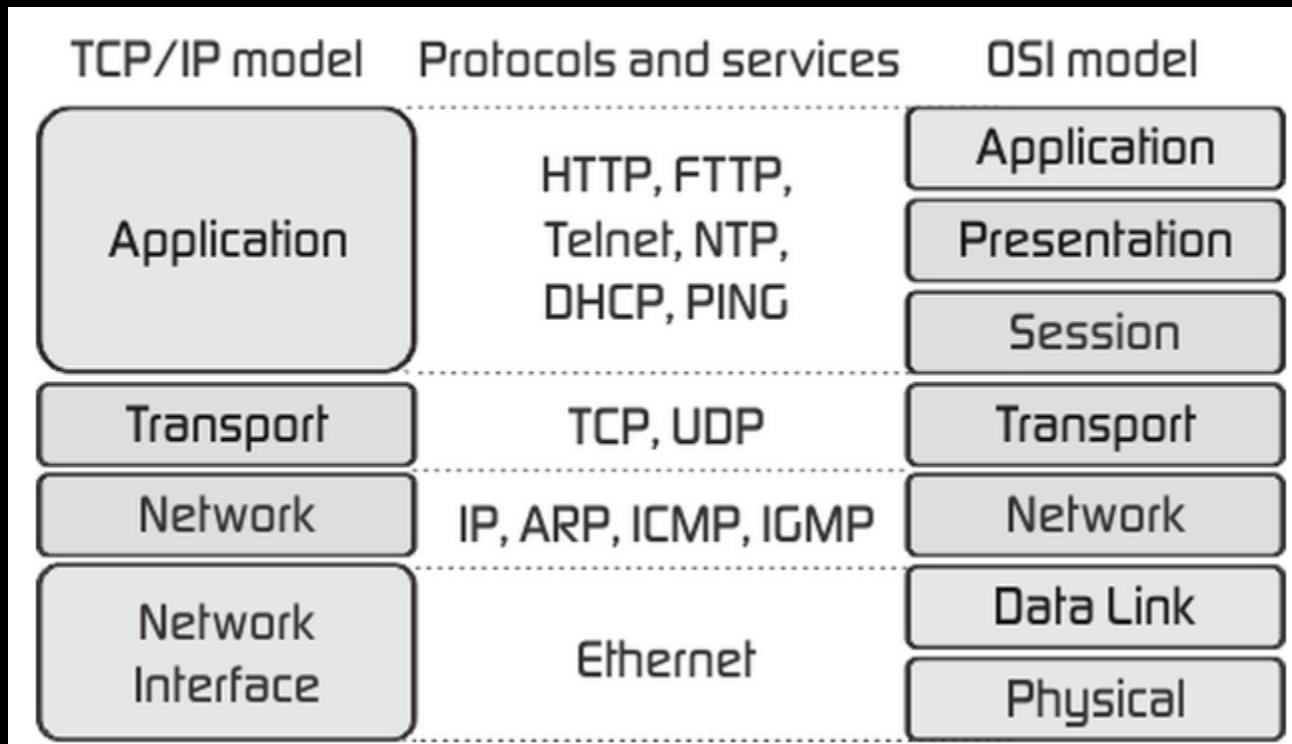


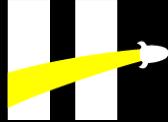
# Encapsulación



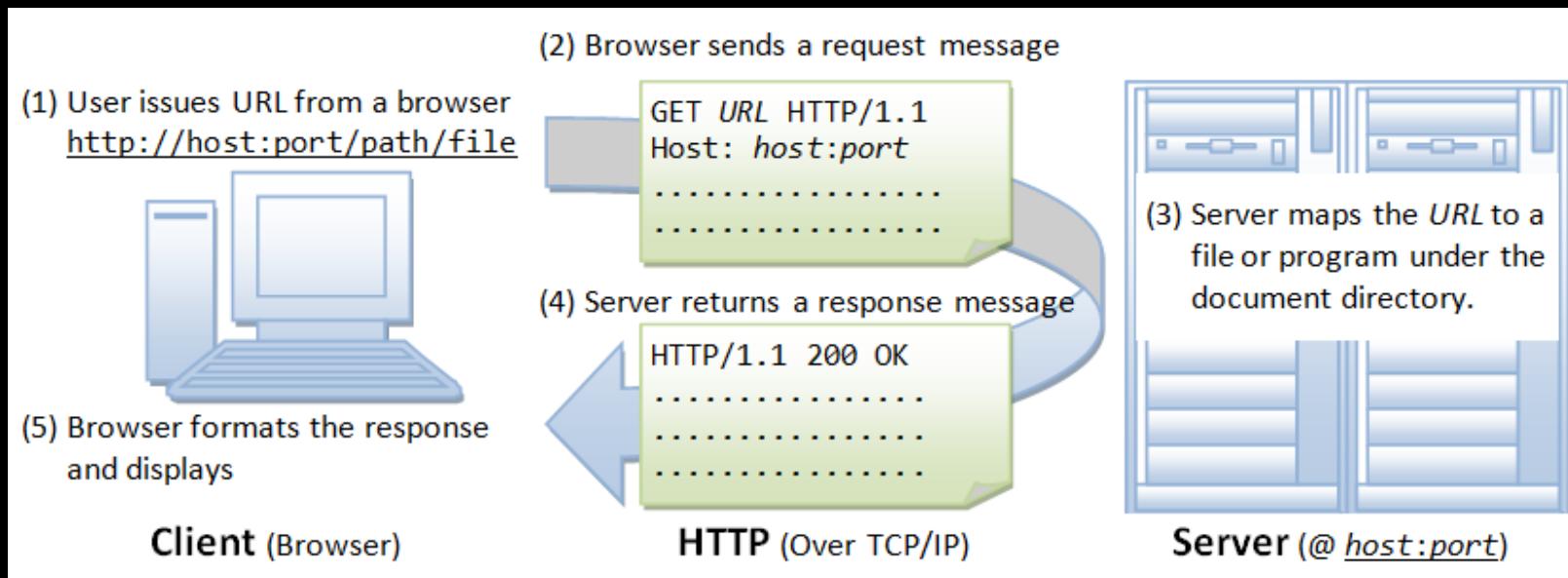


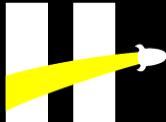
# Protocolos



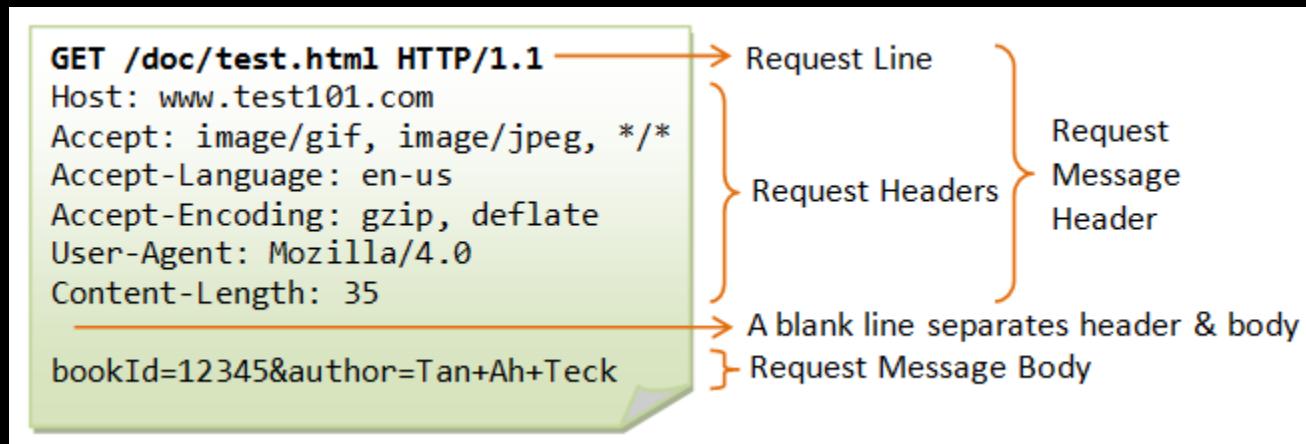


# HTTP





# HTTP



A *socket* is one endpoint of a two-way communication link between two programs running on the network.

A socket is bound to a port number so that the TCP layer can identify the application that data is destined to be sent to.



# Server Básico



```
1
2 var http = require('http'); // importamos el módulo http para poder trabajar con el protocolo
3
4
5 // Creamos una serie de events listener,
6 // que van a escuchar por requests que ocurren en este socket
7
8 http.createServer(function(req, res){
9
10 //Para crear un response empezamos escribiendo el header
11 res.writeHead(200, { 'Content-Type':'text/plain' })
12 //Le ponemos el status code y algunos pair-values en el header
13 res.end('Hola, Mundo!\n');
14
15 }).listen(1337, '127.0.0.1');
16 // Por último tenemos que especificar en que puerto y
17 // en qué dirección va a estar escuchando nuestro servidor
```

< DEMO />



# Enviando HTML

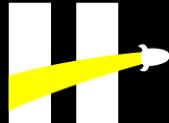


```
1 var http = require('http');
2 var fs = require('fs');
3
4 //Importamos el módulo fs que nos permite leer y escribir archivos del file system
5
6 http.createServer(function(req, res){
7
8 res.writeHead(200, { 'Content-Type':'text/html' })
9 var html = fs.readFileSync(__dirname +'/html/index.html');
10 res.end(html);
11
12
13 }).listen(1337, '127.0.0.1');
```



```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <title>Prueba!</title>
5 </head>
6 <body>
7 <h1>Hola, Mundo!</h1>
8 <p>Bienvenidos!</p>
9 </body>
10 </html>
```

< DEMO />



# Enviando HTML (Templates)

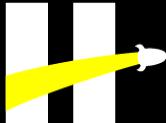


```
1 var http = require('http');
2 var fs = require('fs');
3 //Importamos el módulo fs que nos permite leer y escribir archivos del file system
4
5 http.createServer(function(req, res){
6
7 res.writeHead(200, { 'Content-Type':'text/html' })
8 var html = fs.readFileSync(__dirname +'/html/template.html', 'utf8');
9 //Codificamos el buffer para que sea una String
10 var nombre = 'Soy Henry';
11 //Esta es la variable con la que vamos a reemplazar el template
12 html = html.replace('{nombre}', nombre)
13 // Usamos el método replace es del objeto
14 res.end(html);
15
16
17 }).listen(1337, '127.0.0.1');
```



```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <title>Prueba!</title>
5 </head>
6 <body>
7 <h1>Hola, {nombre}!</h1>
8 <p>Bienvenidos!</p>
9 </body>
10 </html>
```

< DEMO />

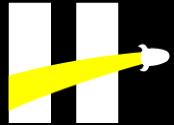


# JSON

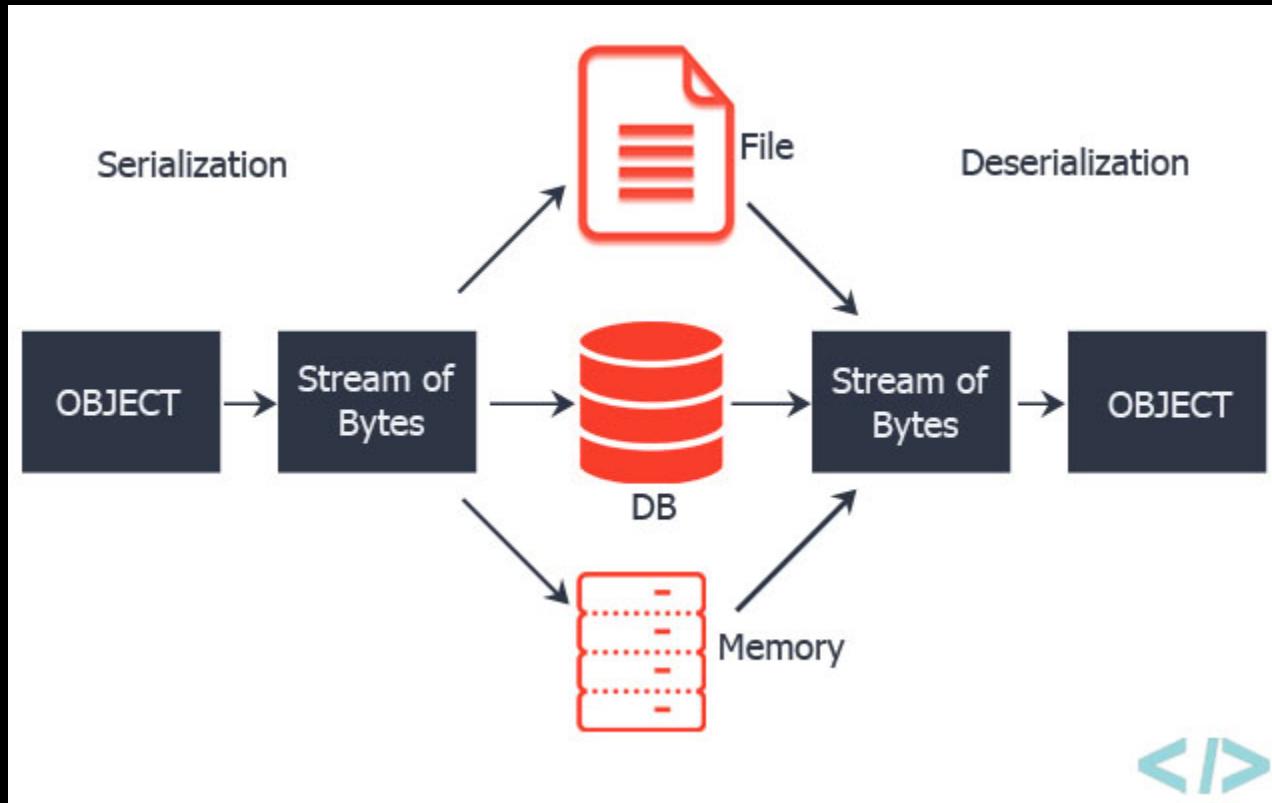


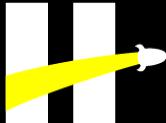
```
1 var http = require('http');
2 var fs = require('fs');
3
4 http.createServer(function(req, res){
5
6 res.writeHead(200, { 'Content-Type':'application/json' })
7 //Vamos a devolver texto en formato JSON
8 var obj = {
9 nombre: 'Juan',
10 apellido: 'Perez'
11 }; //Creamos un objeto de ejemplo para enviar como response
12
13 res.end(JSON.stringify(obj));
14 //Antes de enviar el objeto, debemos parsearlo y transformarlo a un string JSON
15
16 }).listen(1337, '127.0.0.1');
```

< DEMO />



# Serialize / DeSerialize



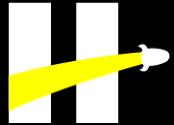


# Rutas

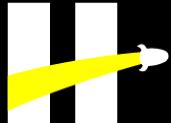


```
1 var http = require('http');
2 var fs = require('fs');
3
4 http.createServer(function(req, res){
5 if(req.url === '/') {
6 res.writeHead(200, { 'Content-Type':'text/html' })
7 var html = fs.readFileSync(__dirname +'/html/index.html');
8 res.end(html);
9 }else if(req.url === '/api'){
10 res.writeHead(200, { 'Content-Type':'application/json' })
11 var obj = {
12 nombre: 'Juan',
13 apellido: 'Perez'
14 };
15 res.end(JSON.stringify(obj));
16 } else{
17 res.writeHead(404); //Ponemos el status del response a 404: Not Found
18 res.end(); //No devolvemos nada más que el estado.
19 }
20
21 }).listen(1337, '127.0.0.1');
```

< DEMO />

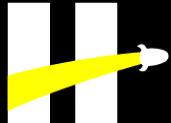


# RESTful API



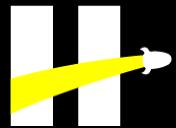
# RESTful API

*“ Rest es una arquitectura o forma de diseñar el backend de una aplicación que viva en internet. REST viene de "REpresentational State Transfer" y está basado fuertemente en cómo trabaja HTTP, que es el protocolo que usamos comúnmente en la web.*



# RESTful API

| Task                    | Method | Path        |
|-------------------------|--------|-------------|
| Create a new task       | POST   | /tasks      |
| Delete an existing task | DELETE | /tasks/{id} |
| Get a specific task     | GET    | /tasks/{id} |
| Search for tasks        | GET    | /tasks      |
| Update an existing task | PUT    | /tasks/{id} |



# RESTful API

- Cliente - Servidor
- Stateless
- Cacheable
- Sistema de capas

# Henry

---



Hacé click acá para dejar tu feedback sobre esta clase.

Hacé click acá completar el quiz teórico de esta lectura.

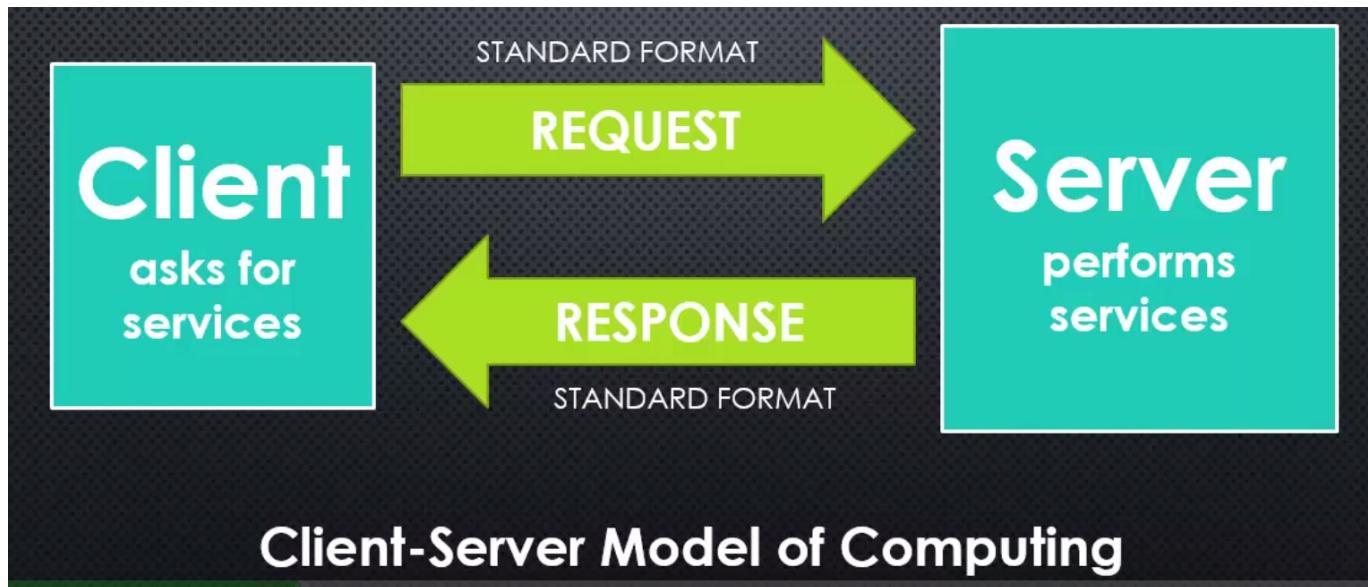
## Armando un Servidor Web con Nodejs

---

### Qué es un servidor web, exactamente?

Un servidor web es cualquier computadora o sistema que procese solicitudes (requests) y que devuelva una respuesta (response) a través de un protocolo de red.

Modelo cliente servidor



Este es el modelo simplificado de cualquier consulta a un servidor web.

El cliente pide un recurso o servicio a un servidor, usando un mensaje en un formato standart (HTTP). El servidor recibe el mensaje, lo decodifica, realiza el servicio que se le pidió y devuelve el resultado en el mismo formato.

### Qué necesitamos que tenga un servidor web en Node

- Maneras de organizar nuestro código para que sea reusable
  - Nodejs Modules
- Poder leer y escribir archivos ( input/output)
  - streams y pipes

- Leer y escribir en Bases de Datos
- Poder enviar y recibir datos de internet
- Poder interpretar los formatos estándares
  - http\_parser
- Alguna forma de manejar procesos que lleven mucho tiempo
  - Naturaleza asincrónica de javascript, callbacks

## Entendiendo el protocolo HTTP

Nodejs viene preparado para poder leer e interpretar el contenido de un mensaje HTTP. Lo logra con la librería [http\\_parser](#), qué es un programa hecho en C y que está embebido en Node.

Usando esta librería de C, Nodejs tiene módulos que nos ayudan a manejar de manera fácil request y reponses HTTP.

## Empezemos a construir un server básico

---

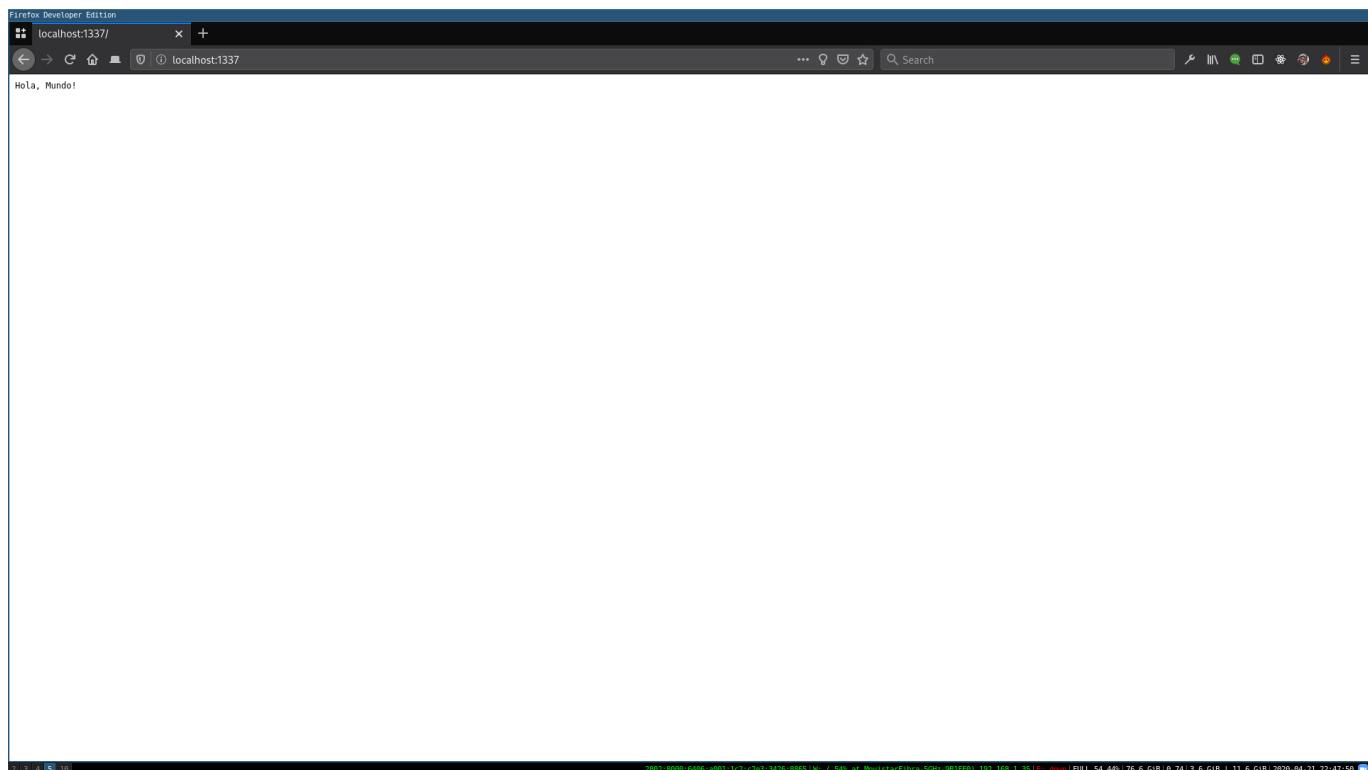
```
var http = require('http'); // importamos el módulo http para poder trabajar con el protocolo

http.createServer(function(req, res){ // Creamos una serie de events listener, que van a escuchar por requests que ocurren en este socket
 //Para crear un response empezamos escribiendo el header
 res.writeHead(200, { 'Content-Type':'text/plain' }) //Le ponemos el status code y algunos pair-values en el header
 res.end('Hola, Mundo!\n');

}).listen(1337, '127.0.0.1'); //Por último tenemos que especificar en que puerto y en qué dirección va a estar escuchando nuestro servidor
```

Para probar el código, corremos el código usando [node app.js](#). Primero notamos que a diferencia de otro código que podamos haber corrido, este programa no termina su ejecución, esto sucede porque el servidor está hecho de manera que se quede escuchando indefinidamente por requests en el puerto y dirección especificadas.

Ahora, para ver si está funcionando como esperábamos vamos a ir al browser y en la barra de dirección escribimos [localhost:1337](#), de esta forma el browser va a ser un request al socket donde dejamos escuchando nuestro server.



Como vemos el servidor respondió con el texto que especificamos en nuestro código!

De hecho, usando las developer tools, podemos ver el request HTTP que hizo el server:

A screenshot of the Firefox Developer Edition Network tab. It shows two requests: a GET request for the document at / (status 200 OK) and a GET request for the favicon.ico file (status 200 OK). The Headers section is expanded, showing standard HTTP headers like Content-Type, Date, and User-Agent. The Request Headers section also lists several client-side headers.

¿Qué más podemos hacer, ahora que tenemos un server corriendo?

Enviando HTML

Ahora vamos a mejorar el servidor para que no sólo devuelva texto plano, sino que devuelva HTML. Podríamos incluir el html como string dentro del código del servidor, pero eso sería engorroso de manejar.

Por lo que vamos a crear un nuevo archivo HTML muy simple llamado `index.html` al que vamos a leer con el servidor.

```
<!DOCTYPE html>
<html>
<head>
 <title>Prueba!</title>
</head>
<body>
 <h1>Hola, Mundo!</h1>
 <p>Bienvenidos!</p>
</body>
</html>
```

Como dijimos que vamos a tener que **leer** el archivo, vamos a necesitar el módulo `fs` de nodejs.

```
var fs = require('fs');
```

Además ahora tenemos que cambiar la propiedad `Content-type` de `text/text` a `text/html` para que el browser sepa que le estamos enviando un archivo que contiene html y no sólo texto plano. **¿Qué pasaría si no cambiaramos esta propiedad?**

Ahora vamos a usar el método `readFileSync` de `fs` para leer el archivo `index.html` que habíamos creado y lo guardamos en una nueva variable que llamaremos `html`.

**En este caso particular usaremos el método de lectura sincrónico para no complicar el código**

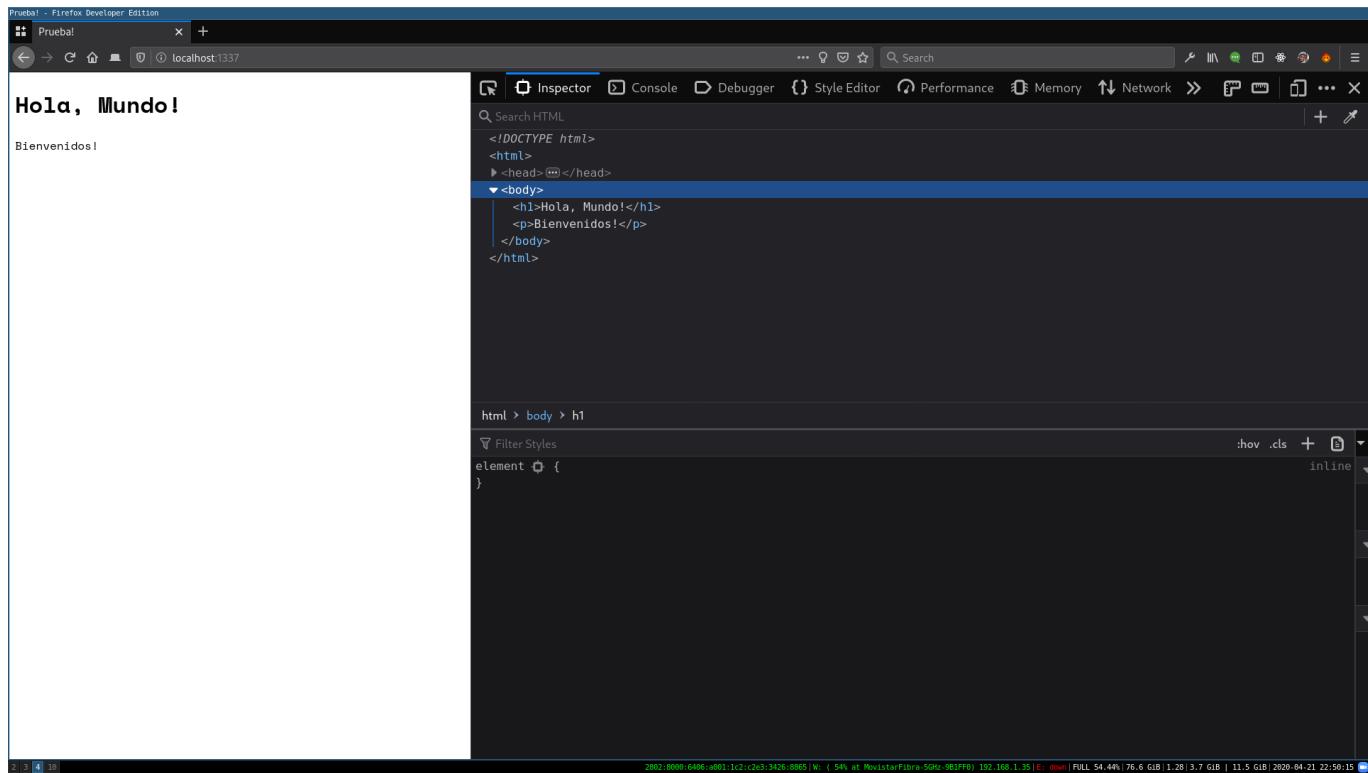
Una vez leído el archivo, vamos a poder enviarlo como argumento del método `end`.

```
var http = require('http');
var fs = require('fs'); //Importamos el módulo fs que nos permite leer y escribir archivos del file system

http.createServer(function(req, res){
 res.writeHead(200, { 'Content-Type':'text/html' })
 var html = fs.readFileSync(__dirname + '/html/index.html');
 res.end(html);

}).listen(1337, '127.0.0.1');
```

Corremos el servidor con `node` y vemos el nuevo resultado:



Como vemos, cuando le llegó el request al server, este leyó el archivo html y lo envió. El browser interpretó que el contenido era **text/html** y lo renderizó como tal.

## Contenido Dinámico: **Templates**

Si quisieramos que el contenido del html no sea estático podríamos, por ejemplo, hacer que el html varie según una variable. Veamos como podemos lograr eso.

Primero vamos a crear un nuevo html al que llamaremos **template.html**. Y vamos a agregar un placeholder (en este caso particular encerramos una palabra con **{}** ), lo que haremos luego será buscar en el archivo lo que esté encerrado en **{}** y reemplazarlo por el contenido la variable que elijamos.

```
<!DOCTYPE html>
<html>
<head>
 <title>Prueba!</title>
</head>
<body>
 <h1>Hola, {nombre}!</h1>
 <p>Bienvenidos!</p>
</body>
</html>
```

Esta idea, de tener placeholders que luego serán reemplazados por contenido que esté en una variable es conocido como **Templates**. Existen varios *lenguajes* de templating que trabajan con este concepto. Más adelante veremos algunos de ellos.

Ahora volvamos al código del servidor. Ahora, antes de enviar el html leído del archivo lo vamos a tener que parsear, por eso vamos a tener que tratar el archivo como una **String** y no como un **Buffer**, por lo que

agregaremos el argumento '`utf-8`' a la función `readFileSync`, para que el buffer sea codificado a una String.

Luego, vamos a crear la variable que va a tener el texto que queremos que se reemplace en nuestro template. Por ejemplo: `var nombre = 'Soy Henry'`.

Ahora vamos a usar el método `replace` del objeto `String`, para que cuando encuentre el placeholder antes definido (`{nombre}`) lo reemplace por el contenido de nuestra variable `nombre`.

Finalmente, enviamos lo que está en la variable `html` en el response.

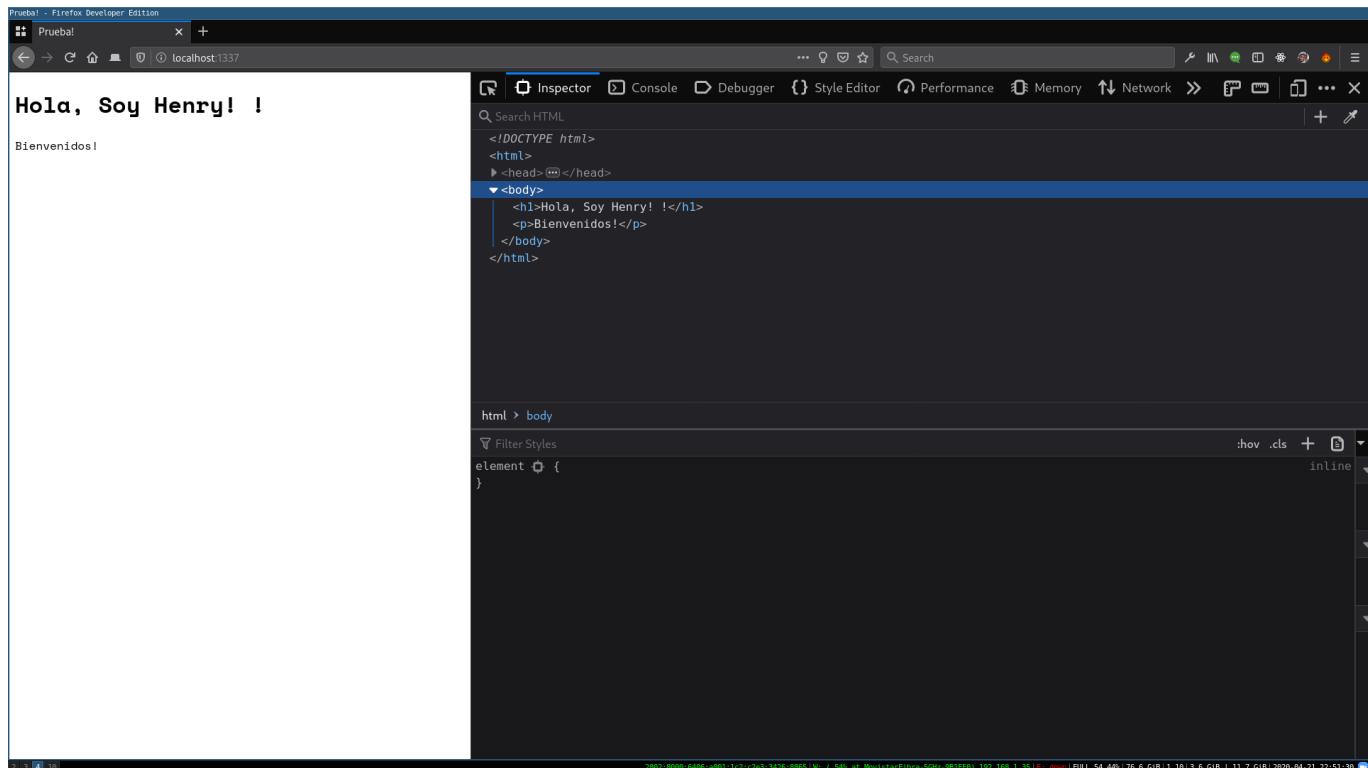
```
var http = require('http');
var fs = require('fs'); //Importamos el módulo fs que nos permite leer y
escribir archivos del file system

http.createServer(function(req, res){

 res.writeHead(200, { 'Content-Type':'text/html' })
 var html = fs.readFileSync(__dirname +'/html/template.html', 'utf8');
 //Codificamos el buffer para que sea una String
 var nombre = 'Soy Henry'; //Esta es la variable con la que vamos a reemplazar
 el template
 html = html.replace('{nombre}', nombre); // Usamos el método replace es del
 objeto String
 res.end(html);

}).listen(1337, '127.0.0.1');
```

Si todo funcionó bien, al recibir el request el server leerá el archivo html, reemplazará el contenido de la variable dentro del placeholder que habíamos definido. Luego enviará el resultado al browser por lo que deberíamos ver lo siguiente:



## Devolviendo JSON

Como sabemos, existen servidores cuyas URLs no nos devuelven un archivo HTML, si no que nos devuelven datos en formato JSON. Comunmente estos *endpoints* son parte de un **API**.

Veamos como podríamos modificar nuestro servidor para que se comporte como lo haría un endpoint de un API.

Como vimos antes, lo primero que debemos cambiar es el header que indica el tipo de contenido que estamos devolviendo, ahora vamos a cambiar el **content-type** a **application/json**, que es el **MIME TYPE** para JSON.

Ahora, en vez de leer de un archivo, vamos a crear algo de datos. Por ejemplo, podemos crear un objeto:

```
var obj = {
 nombre: 'Juan',
 apellido: 'Perez'
};
```

Antes de enviar el objeto, debemos convertir el contenido a un String con formato JSON. Para esto vamos a usar la función **stringify** del objeto **JSON**, el cuál transforma un objeto a un string con notación JSON.

El proceso de transformar un objeto a un formato que pueda ser transferido o guardado se conoce como **SERIALIZE**. En este caso transformamos un objeto que estaba en memoria en un string con formato JSON que puede ser enviado por internet. Otros ejemplo de formatos pueden ser CSV, XML, etc. El proceso inverso (el de transformar de un formato a un objeto en memoria) se conoce como **DESERIALIZE**.

```
var http = require('http');
var fs = require('fs');

http.createServer(function(req, res){

 res.writeHead(200, { 'Content-Type':'application/json' }) //Vamos a devolver
 texto en formato JSON
 var obj = {
 nombre: 'Juan',
 apellido: 'Perez'
 }; //Creamos un objeto de ejemplo para enviar como response

 res.end(JSON.stringify(obj)); //Antes de enviar el objeto, debemos parsearlo
 y transformarlo a un string JSON

}).listen(1337, '127.0.0.1');
```

De nuevo, corremos el servidor y vamos al browser a probar nuestro nuevo endpoint.

The screenshot shows a browser window titled "Pruebal" with the URL "localhost:1337". In the developer tools Network tab, a request to "localhost" is listed. The response is a JSON object with the keys "nombre" and "apellido", both set to "Juan" and "Perez" respectively. The response headers include "Content-Type: application/json", "Date: Sun, 31 Jul 2016 20:28:03 GMT", and "Transfer-Encoding: chunked". The network timeline shows a single request taking approximately 200ms.

Como vemos, nuestro servidor nos devolvió correctamente el objeto que habíamos creado.

Creando más de un Endpoint: **Routing**

Todos los ejemplos anteriores de servidores mantiene una sola ruta, es decir, que cualquier request que llegue al servidor va a ser servida de la misma forma (*Prueben desde el browser hacer request a un path distinto, por ejemplo: <http://localhost:1337/index/> o <http://localhost:1337/hola.jpg>*). Esto se debe a que en ningún momento en nuestro código nos fijamos qué URL está pidiendo el request, simplemente le devolvemos lo mismo a cualquier request!

Ahora veremos como podemos mapear distintos requests HTML a distintos contenidos en el servidor, este proceso de mapeo es conocido como **ROUTING**.

Para hacerlo vamos a tener que examinar en cada request que llega la URL a la que quiere acceder, para eso vamos a hacer uso del objeto `req` que vive dentro de la función `createServer`. Ese objeto tiene toda la información del request que llegó, en este caso nos interesa la URL, ese dato lo encontraremos en `req.url`.

Para este ejemplo vamos a agregar dos rutas, la primera '/' en la que devolveremos el html leyendo del archivo como hicimos al principio, y la segunda '/api' en la que devolveremos el objeto JSON.

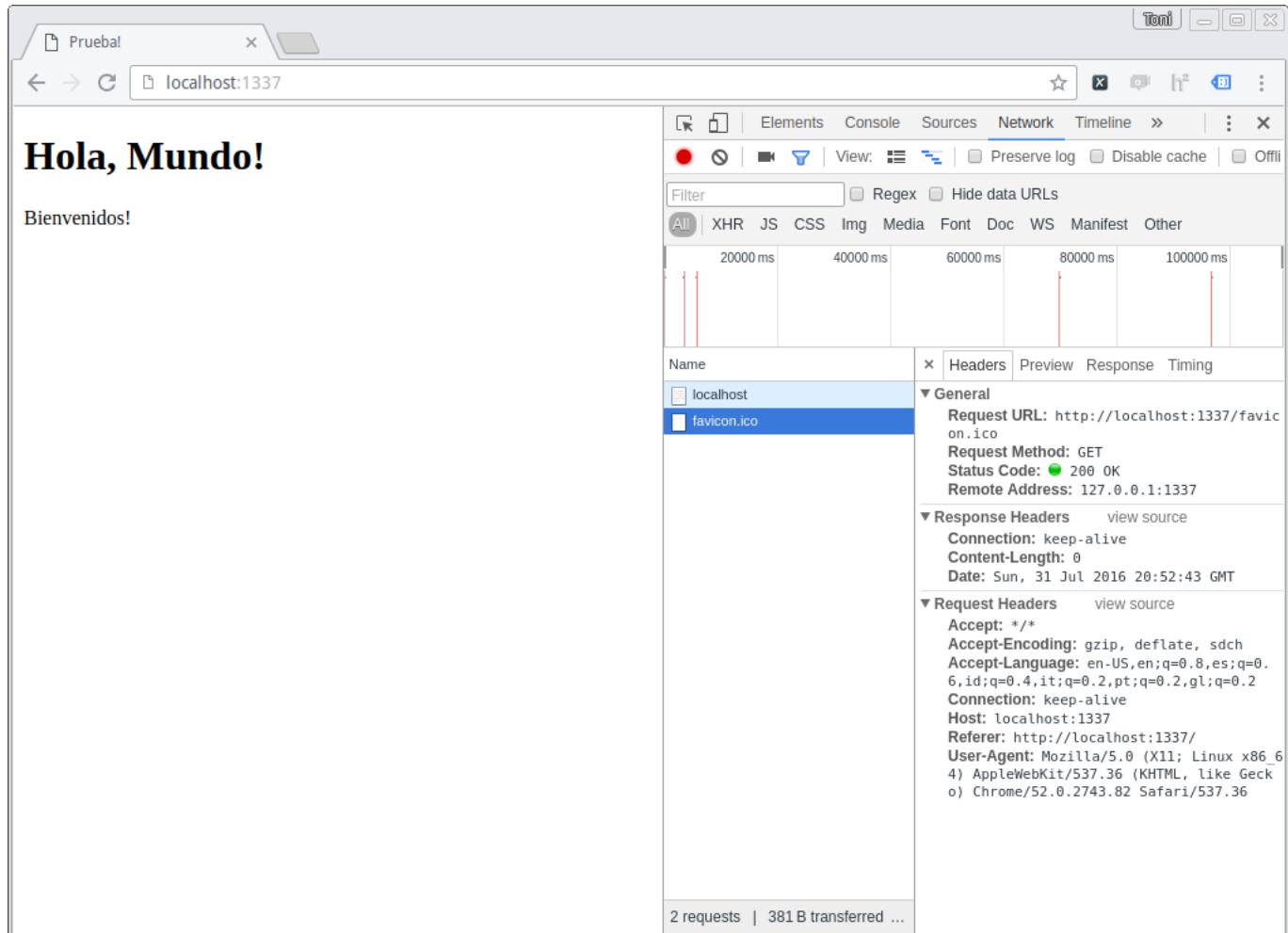
Como dijimos la propiedad `req.url` contiene la URL del request, por lo tanto nos fijaremos en su contenido para decidir qué queremos devolver:

```
var http = require('http');
var fs = require('fs');

http.createServer(function(req, res){
 if(req.url === '/') { //Si la URL es / devolvemos el HTML
 res.writeHead(200, { 'Content-Type':'text/html' })
 var html = fs.readFileSync(__dirname + '/html/index.html');
 res.end(html);
 }
 if(req.url === '/api'){ //Si la URL es /api devolvemos el objeto
 res.writeHead(200, { 'Content-Type':'application/json' })
 var obj = {
 nombre: 'Juan',
 apellido: 'Perez'
 };
 res.end(JSON.stringify(obj));
 }
}).listen(1337, '127.0.0.1');
```

Ahora corramos el servidor, y probemos los distintos endpoints en el browser.

Vamos a 'localhost:1337':



Nos devolvió el HTML, como esperábamos!

Ahora a 'localhost:1337':

The screenshot shows a browser window with the URL `localhost:1337/api`. The Network tab of the developer tools is open, displaying a single request for the endpoint `/api`. The response status is `200 OK` with a `Content-Type: application/json`. The response body contains the JSON object `{"nombre": "Juan", "apellido": "Perez"}`.



**¿Qué pasa ahora con los demás endpoints, por ejemplo 'localhost:1337/hola/como/va'? Por qué el servidor se comporta de esa forma?**

### Manejando las URLs que no existen

Todos sabemos que cuando ingresamos una URL que no existe en la web terminamos obteniendo el error **404**, que es el código que define el error 'Not Found' dentro del standart HTTP.

Entonces, agreguemos a nuestro server la funcionalidad que si la URL del request no coincide con ninguna de las que habiamos ruteado que devuelva el error 404. Vamos a agregar un `else if` y luego un `else` para asegurarnos que sólo se ejecute la porción de código que queremos en cada caso:

```
var http = require('http');
var fs = require('fs');

http.createServer(function(req, res){
 if(req.url === '/') {
 res.writeHead(200, { 'Content-Type': 'text/html' })
 var html = fs.readFileSync(__dirname + '/html/index.html');
 res.end(html);
 }else if(req.url === '/api'){
 res.writeHead(200, { 'Content-Type': 'application/json' })
 var obj = {
```

```

 nombre: 'Juan',
 apellido: 'Perez'
);
 res.end(JSON.stringify(obj));
} else{
 res.writeHead(404); //Ponemos el status del response a 404: Not Found
 res.end(); //No devolvemos nada más que el estado.
}

}).listen(1337, '127.0.0.1');

```

Ahora, si probamos desde el browser una URL que *no existe*, por ejemplo

<http://localhost:1337/hola/como/va> obtenemos el siguiente resultado:

The screenshot shows a browser window with the address bar containing "localhost:1337/hola/como/va". To the right of the browser is the developer tools Network tab. The Network tab displays a single request labeled "va" which failed with a status code of 404 Not Found. The request details show the URL, method (GET), and remote address (127.0.0.1:1337). The response headers include Connection: keep-alive, Date: Sun, 31 Jul 2016 21:00:31 GMT, Transfer-Encoding: chunked, and a User-Agent string for Chrome 52.0.2743.82. The timing section indicates the request took approximately 1000ms.

Copado, no?

Ya pudimos armar nuestro propio servidor con algunos endpoints. Pero no les parece que si nuestro servidor tuviera muchas endpoints el código se haría muy engorroso y difícil de mantener? Por suerte hay gente que ya se topó con estos problemas y escribieron código que les ayuda a simplificar esta tarea.

## RESTful API

---

API (Application Programming Interface)

Una API es a un software o aplicación lo que una interfaz gráfica es al manejo de una computadora, es decir, una API nos facilita la comunicación con el software. Lo hace **abstrayendo** la implementación subyacente y mostrando solamente las acciones que son útiles para el desarrollador. De esta forma, al reducir el nivel de complejidad aparente de un software, bajan la carga de *entendimiento* que tienen que tener los desarrolladores antes de utilizar ciertas tecnologías.

Ahora están de moda las API webs, pero existen de todo tipo y en casi todos los tipos de software:

## En Librerías y Frameworks

Cada vez que usemos una librería o un framework y vamos a su documentación es muy probable que nos encontremos un link a su API (si la librería o framework está bien documentado). En ella vamos a encontrar la descripción, la forma de usar y el comportamiento esperado de las tareas que realiza esa librería. Según el tipo de lenguaje que usemos, la documentación de las API puede variar, por ejemplo: para lenguajes de scripting como **Lua** las api pueden consistir en describir funciones y rutinas con fines específicos, pero en lenguajes orientas a objetos como **java**, la api puede describir una serie de *clases* y sus respectivos *métodos*.

## Sistemas Operativos

Un API tambien puede especificar la interfaz entre una aplicación y el sistema operativo. De hecho, puede servir para mantener compatibilidad hacia atrás, ya que podemos especificar distintas versiones del API y mantener varias implementaciones. Por ejemplo, Microsoft Windows mantiene API de compatibilidad para ejecutar programas viejos sobre los nuevos sistemas operativos Windows, le llaman *Modo de Compatibilidad*.

## Apis Remotas

Una API remota les permite a los desarrolladores manipular recursos remotos a traves de un *protocolo* (no necesariamente HTTP). Por ejemplo, existen API que nos dejan hacer queries a distintos tipos de bases de datos que en general se encuentran en hosts remotos.

## Web APIS

Cuando la usamos en la web, van a estar montadas sobre el protocolo **http**, por lo tanto podríamos decir que una API web es un set de tipos de mensajes HTTP posibles, junto con la descripción el formato de la eventual respuesta de esos mensajes ( en general la respuesta es en JSON o XML ). Históricamente el término Web api es similar a *web services*. La forma de implementar y de usar estos web services fue cambiando de paradigma, actualmente se usa el diseño tipo REST, y en general en una web se consumen más de un API y se combinan sus resultados en una sola página.

## Endpoints

Un endpoint en un web api especifica donde están los recursos que pueden ser accedidos desde afuera de la API. Por ejemplo: <https://api.punkapi.com/v2/beers> especifica un endpoint de la API de **Punk Api**.

Noten que esta API tiene la versión, por lo tanto ofrece compatibilidad hacia atrás.

## REST

Rest es una estilox arquitectura o forma de diseñar el backend de una aplicación que viva en internet. REST viene de "REpresentational State Transfer" y está basado fuertemente en cómo trabaja HTTP, que es el

protocolo que usamos comúnmente en la web.

Como sabemos, HTTP es un protocolo basado en el modelo cliente servidor, quienes intercambian mensajes basados en ciertas acciones. Por ejemplo, un mensaje HTTP tipo GET realizado a la URL <http://example.com/>.

está basado en recursos y no en acciones.

Vamos a ver

## Concepto

---

### Características

---

#### Cliente - Servidor

La arquitectura REST utiliza los conceptos del modelo cliente servidor en el sentido de separar las inquietudes de la interfaz de usuario con las inquietudes del manejo y guardado de datos. Esto ayuda a que cada componente pueda evolucionar de manera separada, acomodarse a como cambian hoy en día las interfaces en internet.

#### Stateless

La comunicación entre el cliente y el servidor se logra sin que el servidor tenga guardado ningún contexto del cliente entre requests. Cada request del cliente contiene toda la información necesaria para que el servidor pueda contestar correctamente al request.

#### Cacheable

En esta arquitectura, cada recurso tiene que estar marcado como cacheable o no. En el primer caso para ayudar al servidor a realizar menos trabajo y aumentar la performance y en el segundo para que no lleguen al cliente recursos con datos inapropiados.

#### Sistema de capas

Un cliente no debería ser capaz de distinguir si se está conectado directamente con el servidor, o está pasando por un intermediario antes de llegar a él. Estos servidores intermedios pueden ayudar a aumentar la performance implementando servicios de load-balancing, shared caches, etc...

#### Interface Uniforme

El diseño de una interfaz uniforme es fundamental para la arquitectura, hacerlo bien simplifica mucho la arquitectura y la hace modular, logrando así que pueda evolucionar o escalar para parte por separado. Las características que deben tener las interfaces uniformes son:

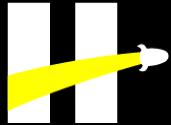
- **Identificación de recursos:** Cada recurso tiene que ser identificado en el request, por ejemplo a través de un *URI*. El recurso per se también está separado de su representación, la que es enviada al cliente; esta puede ser, por ejemplo, el mismo recurso representado en: JSON, XML, HTML, etc...

- **Manipulación de recursos a través de representaciones:** Cuando un cliente tiene la *representación* de un recursos, deberá tener la suficiente información para *modificar* o *eliminar* ese **recurso**.
- **Mensajes descriptivos:** Cada mensaje deberá contener suficiente información para describir cómo procesar el mensaje. En una API web, esto se traduce a mapear rutas con los verbos *HTTP*.
- *Hypermedia as the engine of application state(HATEOAS)*: Un cliente REST debería ser capaz de *navegar* y *descubrir* todas las acciones posibles de hacer a un recurso luego de interactuar con él. Es similar a cuando entramos a una web desde una *URL* y la web misma nos provee los *links* para que sigamos navegando. En la arquitectura REST debería ocurrir lo mismo, cada respuesta debería contener *links* o información a las siguientes acciones que se pueden tomar.

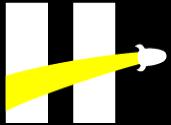
## Otras Arquitecturas

REST no es la única forma de diseñar tu API, existen otras. Cada una será mejor o peor según el problema a resolver y un poco por el gusto del programador. Veamos algunas que están pisando fuerte pero todavía no son las más usadas:

- **JSON API**: Es una especificación, al igual que REST. Está pensada para minimizar el número de requests y la cantidad de datos que se transmiten por la red.
- **GraphQL**: Es en realidad una librería que ofrece un nuevo **lenguaje** para hacer consultas a nuestra API. Cambia un poco el concepto de endpoints, y los embebe en esta nueva forma de hacer queries. Todavía es nuevo, pero está tomando tracción rápidamente.



# Promesas



# Promesas

.then retorna una nueva promesa!



```
1
2 promiseB = promiseA.then(successHanlder, FailureHandler);
```



# Promesas

Por esto podemos encadenarlas!

```
1
2 promiseA
3 .then(hacerAlgo)
4 .then(hacerAlgoMas)
5 .then(masCosas)
6 .catch(errorHandler);
7
8
9 /**
10 .catch(errorHandler)
11
12 // es equivalente a
13
14 .then(null, errorHandler)
```

# Henry



Hacé click acá para dejar tu feedback sobre esta clase.



Hacé click acá completar el quiz teórico de esta lecture.

## Advanced Promises

Las promesas pueden ser algo complejas, sobre todo cuando queremos hacer *chaining* de promesas. Veamos los siguientes casos y qué sucede en cada uno de ellos:

```
doSomething().then(function () {
 return doSomethingElse();
}).then(finalHandler);

doSomething().then(function () {
 doSomethingElse();
}).then(finalHandler);

doSomething().then(doSomethingElse())
 .then(finalHandler);

doSomething().then(doSomethingElse)
 .then(finalHandler);
```

### Caso I

Código:

```
doSomething().then(function () {
 return doSomethingElse();
}).then(finalHandler);
```

Solución:

```
doSomething
|-----|
 doSomethingElse(undefined)
|-----|
 finalHandler(resultOfDoSomethingElse)
```

```
| ----- |
```

## Caso II

Código:

```
doSomething().then(function () {
 doSomethingElse();
}).then(finalHandler);
```

Solución:

```
doSomething
| ----- |
 doSomethingElse(undefined)
 | ----- |
 finalHandler(undefined)
 | ----- |
```

## Caso III

Código:

```
doSomething().then(doSomethingElse())
.then(finalHandler);
```

Solución:

```
doSomething
| ----- |
doSomethingElse(undefined)
| ----- |
 finalHandler(resultOfDoSomething)
 | ----- |
```

## Caso IV

Código:

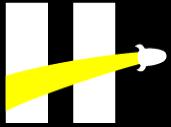
```
doSomething().then(doSomethingElse)
 .then(finalHandler);
```

Solución:

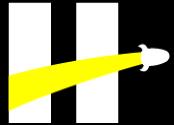
```
doSomething
|-----|
 doSomethingElse(resultOfDoSomething)
|-----|
 finalHandler(resultOfDoSomethingElse)
|-----|
```

Material Recomendado:

- [We have a problem with promises](#)



# Express

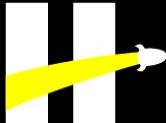


# Express

Es un framework web rápido, minimalista y  
flexible para [Node.js](#)



```
1 npm install express --save
```



# Express

```
var express = require('express');
var app = express();
app.get('/', function(req, res, next) {
 next();
})
app.listen(3000);
```

HTTP method for which the middleware function applies.

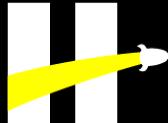
Path (route) for which the middleware function applies.

The middleware function.

Callback argument to the middleware function, called "next" by convention.

HTTP response argument to the middleware function, called "res" by convention.

HTTP request argument to the middleware function, called "req" by convention.

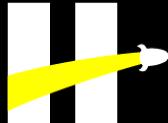


# Express - Routing



```
1 // Esta ruta matcheará acd y abcd
2 app.get('/ab?cd', function(req, res) {
3 res.send('ab?cd');
4 });
5
6 // Esta ruta matcheará abcd, abbcd, abbbcd, y así sucesivamente
7 app.get('/ab*cd', function(req, res) {
8 res.send('ab*cd');
9 });
10
11 // pasando parámetros
12 app.get('/api/:id', function(req, res) {
13 res.json({ parametro: req.params.id });
14 });
```

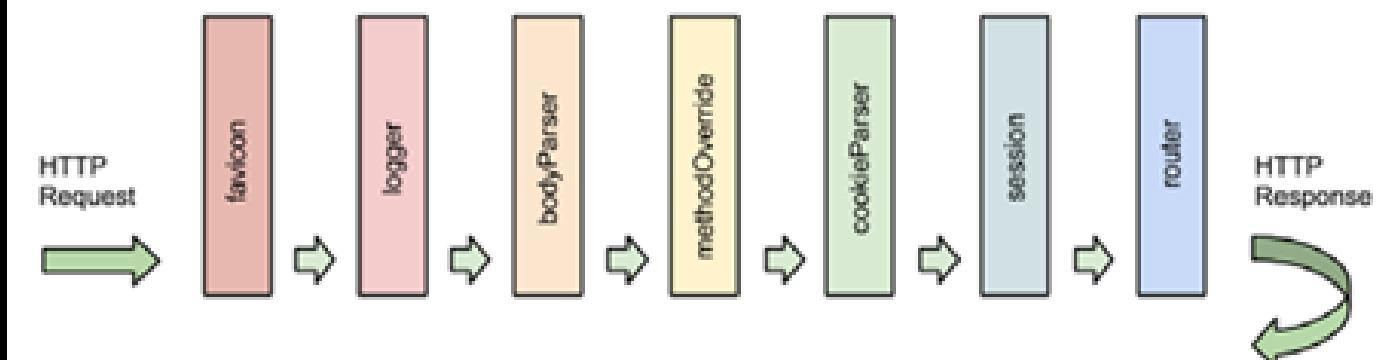
Express nos ofrece muchísimas soluciones para varios problemas, es por eso que vamos a tener que aprender a leer y buscar en la documentación (que por cierto es muy buena) de *express*.

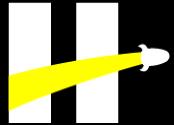


# Express -MiddleWares

## The Express Middleware Stack

An HTTP request may pass through a series of Express middlewares before one of the middlewares or route handlers returns the HTTP response.





# Express -MiddleWares



```
1 app.use('/', function(req,res, next){
2 console.log('Hicieron un Request a '+ req.url);
3 next();
4});
```



# Express - Enviando Datos al server

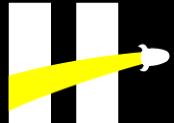
## Query String

Una forma de enviar datos es hacerlo en la URL a la que apuntamos el request. Para ello nos valemos de una serie de parámetros o datos que se incluyen en la URL. Normalmente distinguimos en la URL por un nombre y un valor separados por el signo igual, y se separan del endpoint por el carácter ?, y entre cada variable por el signo &. Por ejemplo:

```
GET /?id=4&page=3 HTTP/1.1
Host: www.learnwebdev.net
Cookie: username=abc;name=Tony
```



```
1 app.get('/datos/', function(req, res) {
2 res.json(req.query);
3 });
```



# Express - Enviando Datos al server

Otras formas de enviar datos al servidor es a través de Formularios. Ahora usaremos otro verbo HTTP, el **POST**.

POST / HTTP/1.1

Host: www.learnwebdev.net

Content-Type: application/x-www-form-urlencoded

Cookie: num=4;page=2

username=Tony&password=pwd



```
1 var urlencodedParser = express.urlencoded({ extended: false })
2 app.post('/form', urlencodedParser, function (req, res) {
3 res.json(req.body)
4 });
```



# Express - Enviando Datos al server

También podemos enviar datos al servidor en formato JSON, donde también usamos POST, pero el content-type es ahora application/json. Por ejemplo en un request generado por AJAX. En este caso los datos también estarán en el body del request.

POST / HTTP/1.1  
Host: www.learnwebdev.net  
Content-Type: application/json  
Cookie: num=4;page=2

```
{
 "username": "Tony",
 "password": "pwd"
}
```

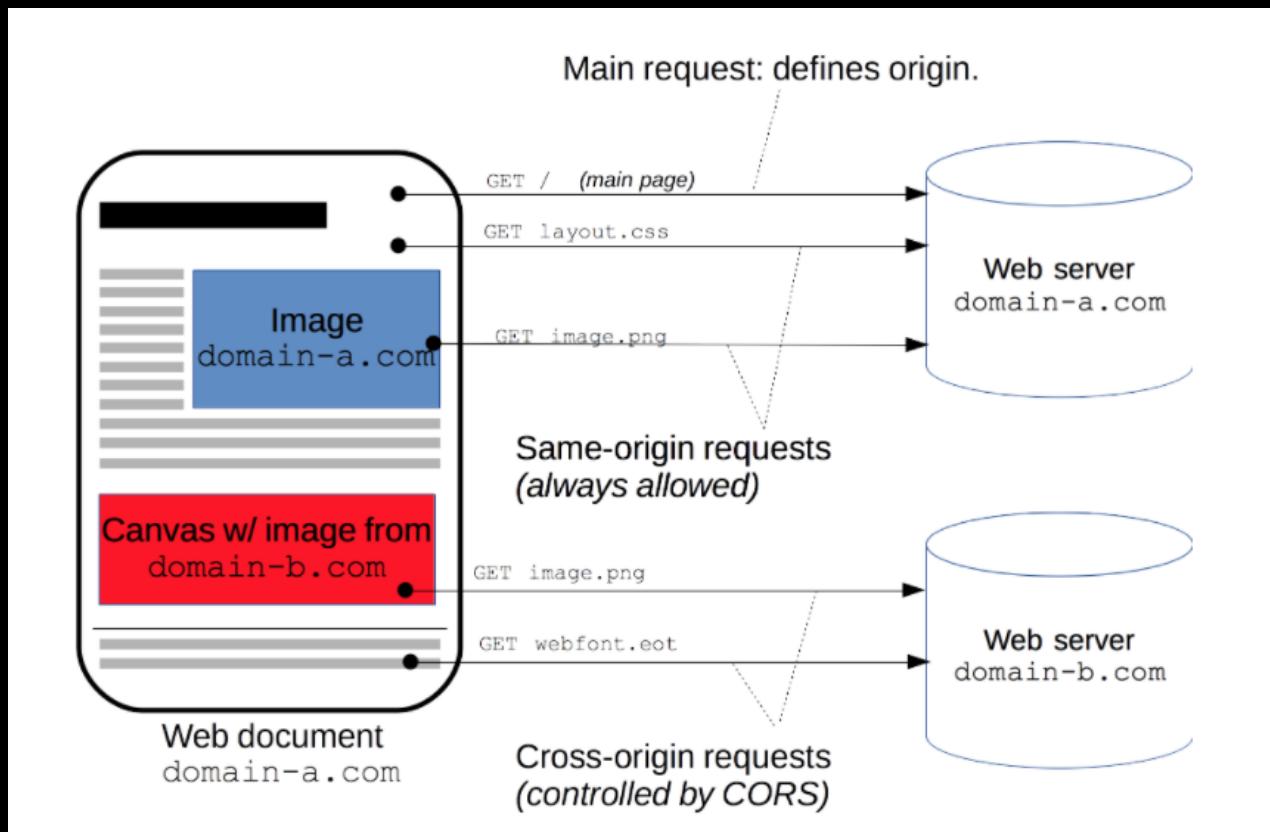


```
1 var jsonParser = express.json()
2 app.post('/formjson', jsonParser, function (req, res) {
3 res.json(req.body)
4 });
```



# CORS

## Cross-Origin Resource Sharing



# Henry

---



Hacé click acá para dejar tu feedback sobre esta clase.

Hacé click acá completar el quiz teórico de esta lecture.

## Express

---

*Express.js* o simplemente *Express* es un framework diseñado para crear aplicaciones web y APIS. Está bajo la licencia [MIT](#), y es, tal vez, el framework web más usado en el ecosistema de Nodejs.

### Instalación

Vamos a iniciar una app nueva con `npm init` a la que vamos a llamar `express-test` y luego vamos a instalar express usando `npm`:

`npm install express --save` --save para que se guarde en `package.json`

Ahora vamos a crear un archivo `index.js` (*o con el nombre que hayan definido como entry point*) y vamos a requerir 'express'.

```
var express = require("express");
```

Cuando requerimos express, lo que nos devuelve la librería es una función, que envuelve toda la funcionalidad de express. Por eso, para inicializar una nueva aplicación, vamos a crear una variable (comúnmente llamada `app`) y guardar en ella la ejecución de express.

```
var express = require("express");
var app = express();
```

Dentro de nuestra nueva variable `app`, vamos a tener varias funciones. Una de ella es `listen()`, que tiene envuelto adentro a la función `http.createServer()` que habíamos usado para crear nuestro propio webserver. Ahora en vez de llamar esa función directamente llamamos a `listen()` y le pasamos un puerto.

```
app.listen(3000);
```

Noten, que todo lo que hace express, lo pueden hacer ustedes también escribiendo el código, sólo es un montón de código preescrito que resuelve problemas muy comunes de una manera muy buena, justamente por eso es que es tan popular.

## Creando rutas

Ahora la variable `app` tiene asociado varios métodos que mapean a [métodos HTTP](#), también llamados 'verbos Http'. Con esto especificamos qué queremos que haga el servidor según el tipo de método http del request y la URL.

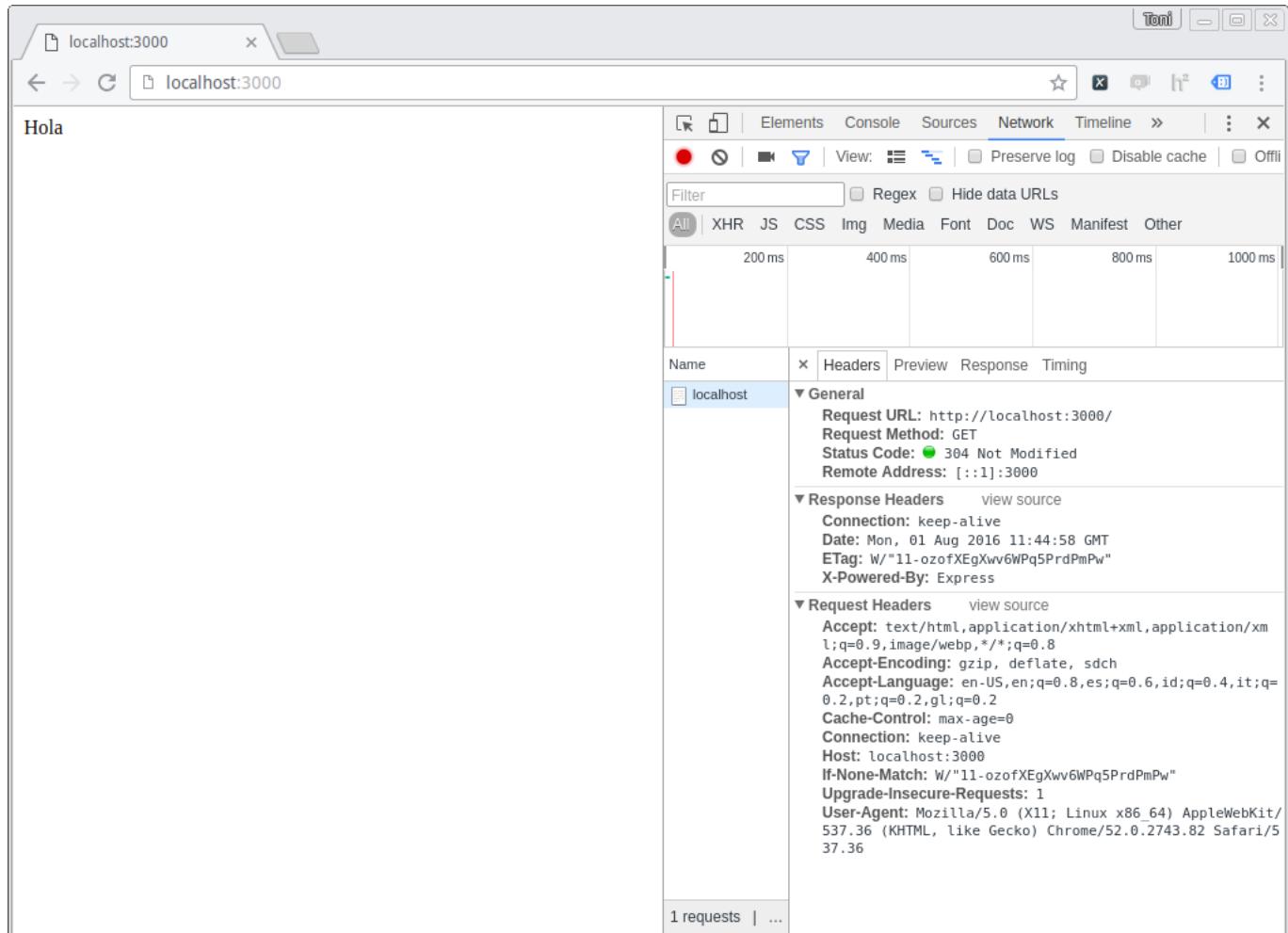
Agreguemos una ruta para `'/'` con el método GET. (*El método GET es el que hace el browser por defecto cuando escribimos algo en la barra de direcciones*):

```
app.get("/", function (req, res) {
 //Ruta para un GET a /
});
```

Como vemos, esto reemplaza la serie de `ifs` que habíamos escrito para nuestro webserver antes. Además nos agrega la discriminación del método HTTP. Estos métodos reciben como parámetro un callback, el cual correrán cuando llegue un request de este tipo al URL definido (`/` en nuestro ejemplo). Noten que ese callback tiene siempre como parámetro dos variables, `req` por **request** y `res` por **response**, que envuelven los objetos `http.request` y `http.response` respectivamente, agregándole más funcionalidad.

```
app.get("/", function (req, res) {
 res.send("Hola");
});
```

Como ejemplo, vamos a usar `res.send()` para enviar texto como respuesta. No fué necesario explicitar el **Content-type**, ya que *express* se encarga de eso por nosotros tambien!

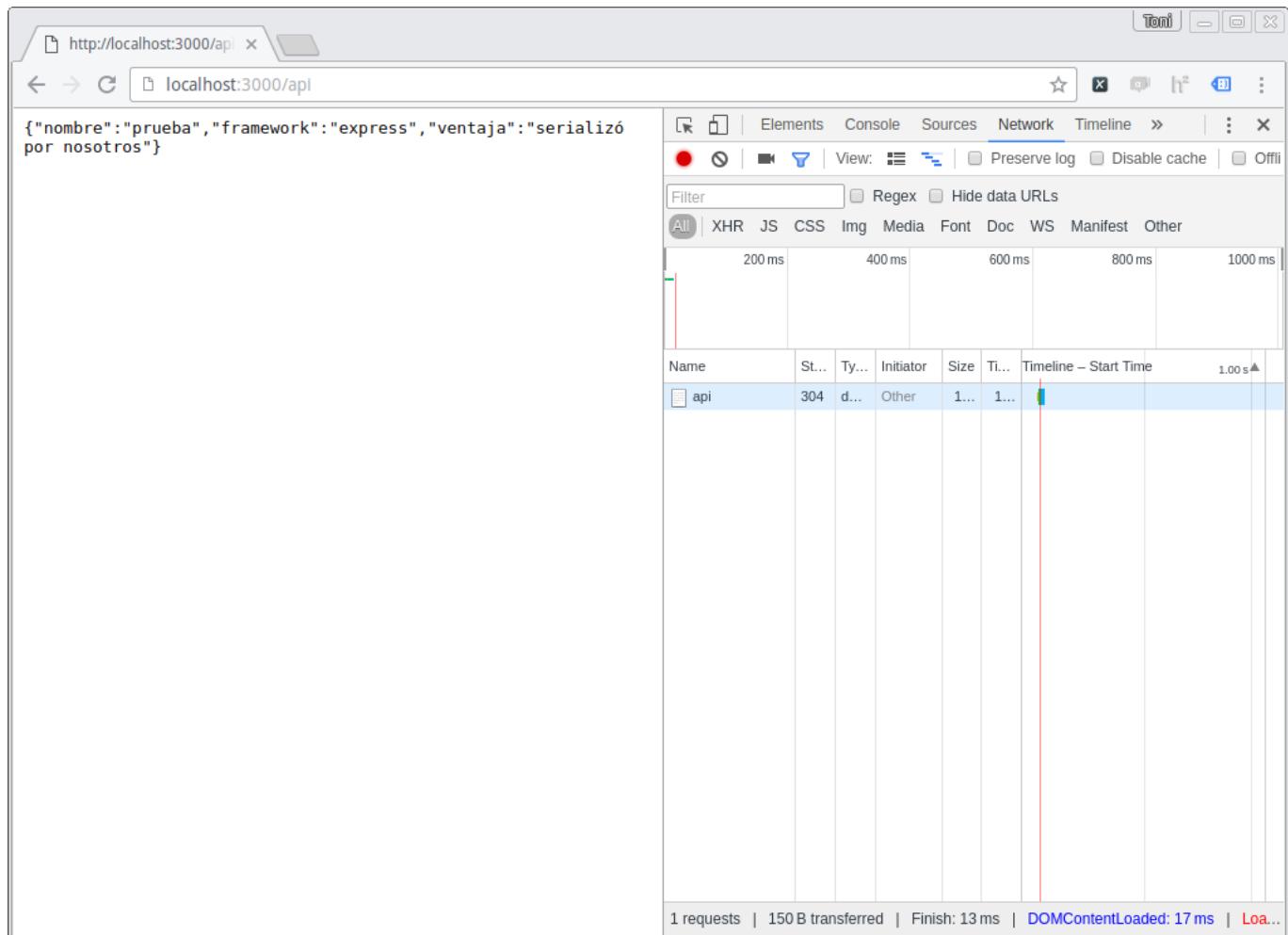


Podemos ejecutar el server con `nodemon` y probar la URL '`/`' en el browser.

Agreguemos una nueva ruta `/api` y retornemos un objeto JSON. Para esto, express viene preparado con la función `json()` que recibe un objeto y lo pasa como response. Noten que no tuvimos que *Serializar* el objeto! De esta forma *express* nos ahorra mucho tiempo.

```
app.get("/api", function (req, res) {
 var obj = {
 nombre: "prueba",
 framework: "express",
 ventaja: "serializó por nosotros",
 };
 res.json(obj);
});
```

Probemos el nuevo *endpoint* en el browser:



Veamos que más podemos hacer con *express*!

## Routing

Express nos ofrece muchísimas soluciones para varios problemas, es por eso que vamos a tener que aprender a leer y buscar en la documentación (que por cierto es muy buena) de *express*.

Veamos la documentación sobre [routing](#).

Como vemos, hay varias formas de hacer las rutas usando las URL: Podemos hacerlo con un nombre fijo como `veniamos` haciendo o de hecho también podemos usar `strings patterns` y `regular expressions` tal que matcheen múltiples rutas, por ejemplo:

### Basadas en String Patterns

Esta ruta matcheará `acd` y `abcd`:

```
app.get("/ab?cd", function (req, res) {
 res.send("ab?cd");
});
```

Esta ruta matcheará `abcd`, `abbcd`, `abbbcd`, y así sucesivamente:

```
app.get("/ab*cd", function (req, res) {
 res.send("ab*cd");
});
```

## Pasando parámetros en las rutas

Veamos como nos ayuda *express* a capturar parámetros embebidos en la URL.

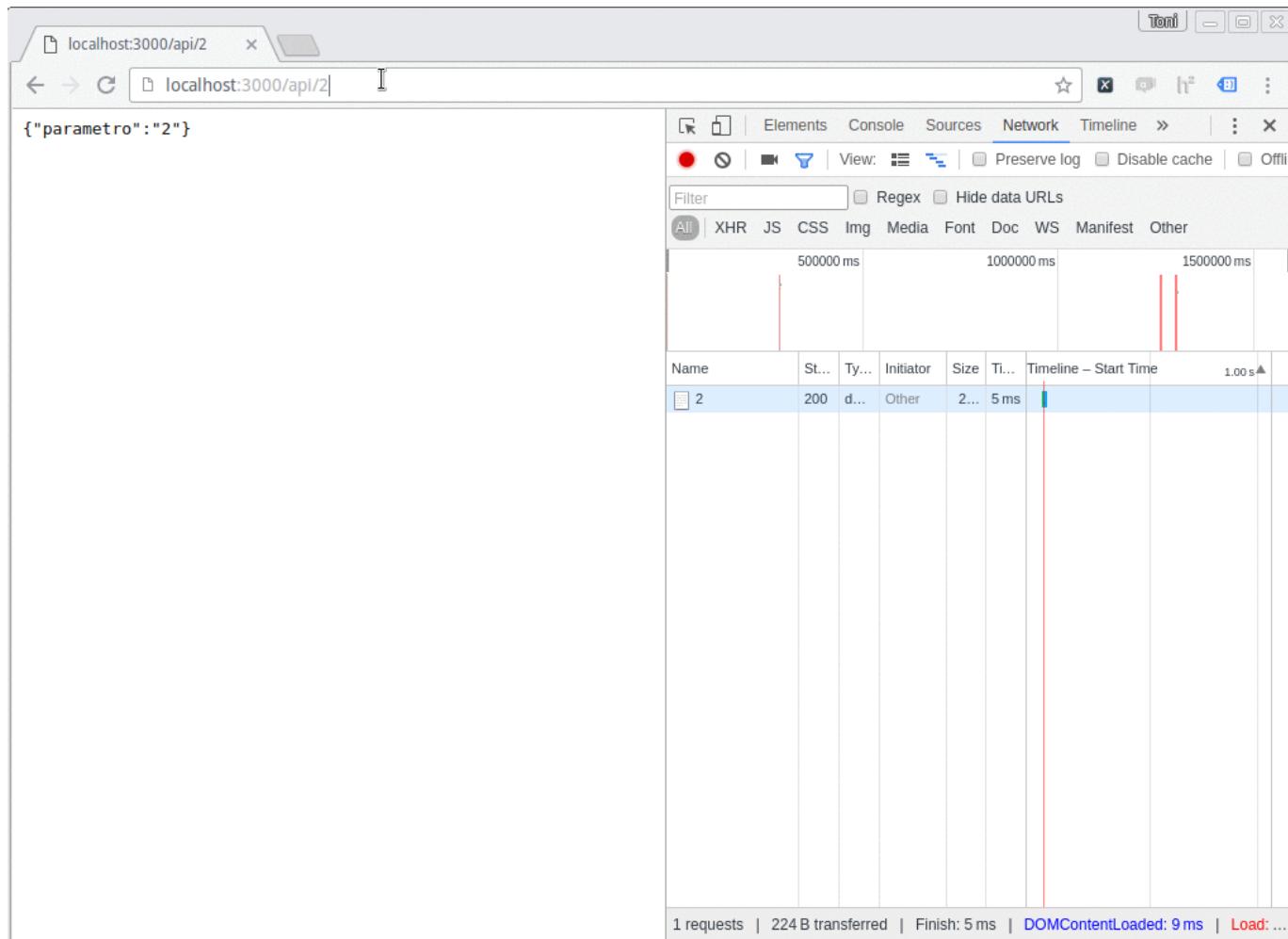
Veamos el siguiente ejemplo:

```
app.get("/api/:id", function (req, res) {
 res.json({ parametro: req.params.id });
});
```

A la ruta `/api` le hemos agregado `/:id`, ahora *express* va a parsear esa ruta, y va a tomar como parámetro lo que esté después de la primera `/` y lo vamos a poder acceder a través del nombre `id`. Por ejemplo en `/api/2` `id` va a tomar el valor `2`. En `/api/hola`, `id` va a tomar el valor `hola`.

Para mostrar el comportamiento, esa ruta va a devolver un objeto json con la propiedad `parametro` y cuyo valor es el contenido de `id`. Los parámetros parseados por *express* los vamos a encontrar en `req.params` y el nombre que pusimos después de `:` en la ruta, en este caso `req.params.id`.

Probando en el browser:



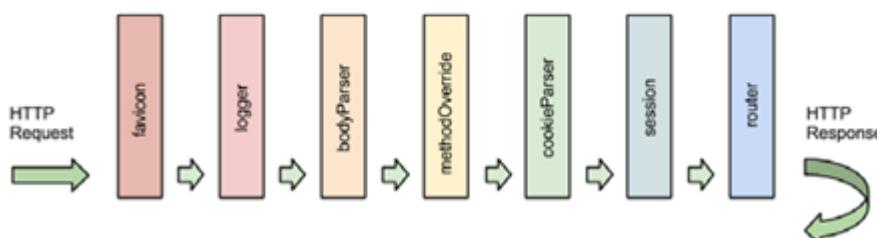
Podemos agregar más de un parámetro por URL. Prueben armar una ruta que maneje la siguiente URL: `/api/:id/:nombre/:valor`.

## Archivos Estáticos y Middleware

**Middleware:** hace referencia a código que existe entre dos capas de software, en el caso de *express* a código que este entre el `request` y el `response`.

### The Express Middleware Stack

An HTTP request may pass through a series of Express middlewares before one of the middlewares or route handlers returns the HTTP response.



Veamos un ejemplo muy común de middleware en *express*. En casi todas las aplicaciones web vamos a tener archivos que queremos que se bajen siempre, por ejemplo: imágenes, archivos `.css` o `.js`, etc. Como sabemos, para poder acceder a estos archivos vamos a tener que crear para cada archivo una ruta tal que sean accesibles a través de una URL. Esto puede llegar a complicarse fácilmente, no? Por suerte, *express* ya

pensó en esto y nos da un middleware para manejar estos *static files* (se llaman estáticos porque no dependen de ningún input y no deben ser procesados de ninguna manera, siempre son iguales).

Digamos que queremos tener una carpeta donde guardamos todos los *archivo estáticos*, comúnmente esta carpeta tiene el nombre de `public`. Dentro de ella vamos a crear un archivo `.css` y guardar ahí una imagen.

\_Vamos a crear una ruta que devuelva un html que utilice el archivo `.css` que acabamos de crear y también que contenga la imagen.

```
app.get("/static", function (req, res) {
 res.send(
 '<html><head> \
 <link href="/assets/style.css" rel="stylesheet"> \
 </head><body> \
 <p>Archivos estaticos rapido y facil!!</p> \
 \
 </body></html>'
);
});
```

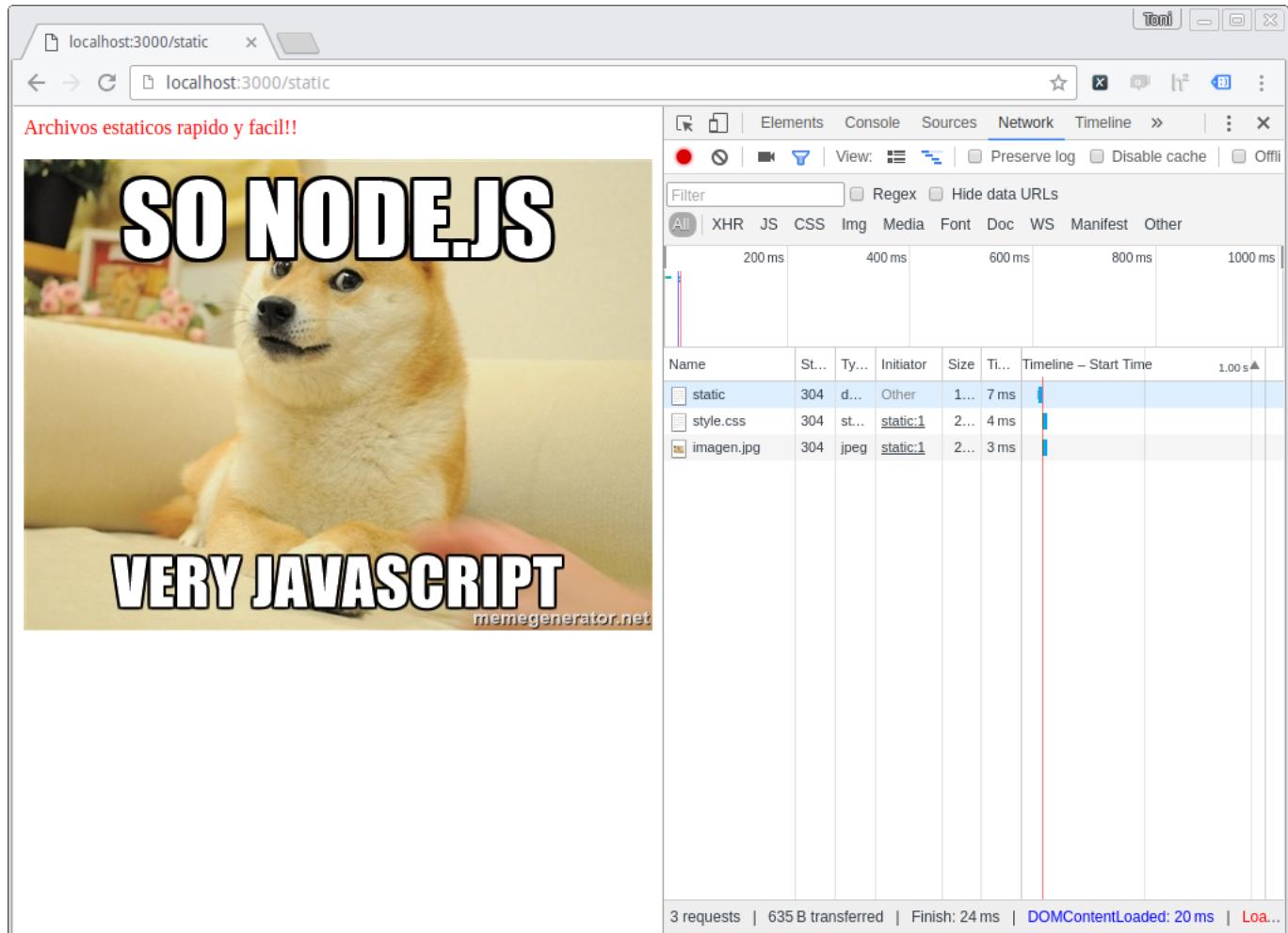
En el ejemplo requerimos los archivos estáticos en `/assets/`. Ahora para mapear todos los archivos que estén en esa carpeta a una ruta dentro de `/assets` (por ejemplo: `/assets/image.jpg`) de forma automática, vamos a usar el middleware `express.static` que viene por defecto con `express`.

Para agregarlo, vamos a usar la función `use()`, que le indica a `express` para que use el middleware que le pasamos en el segundo parámetro, en las URL que le pasamos en el primero. Así que con lo siguiente le estamos diciendo que mapee las rutas de `/assets/` con el middleware `static`.

```
app.use("/assets/", express.static(__dirname + "/public"));
```

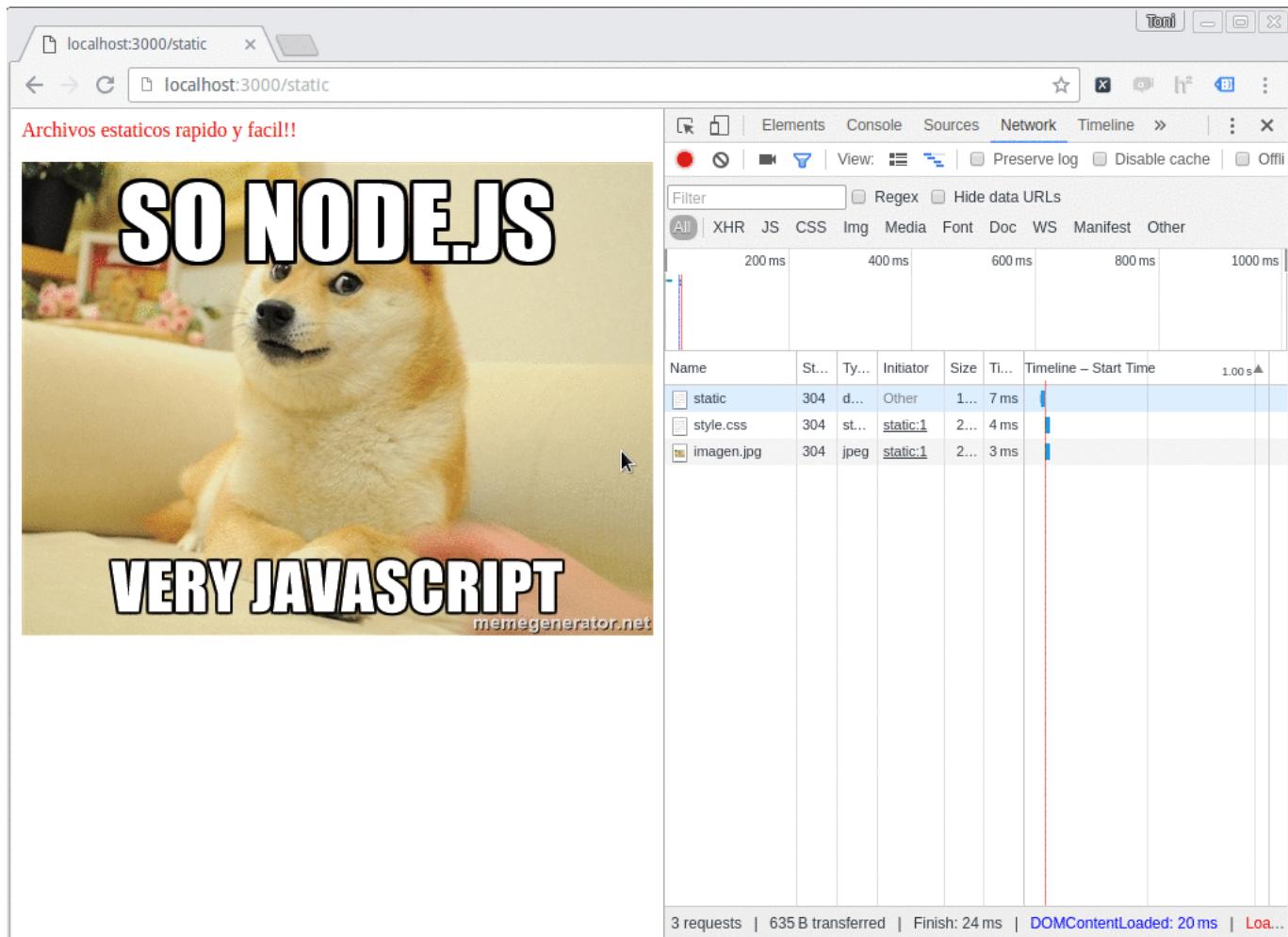
El único parámetro que recibe `static` es el nombre del directorio donde están los archivos estáticos, en nuestro ejemplo están en `/public`.

Ahora probemos la nueva ruta `/static/` y probemos si carga los archivos estáticos:



Como vemos la imagen, y vemos el párrafo de color rojo, sabemos que se cargaron correctamente los archivos estáticos!

Si accedemos a cada uno individualmente:



Vemos que cada uno tiene su propia ruta dentro de `/assets/` y que fue mapeada automáticamente por `express.static` a cada archivo dentro de la carpeta `public`. Muy potente, no? Such express!

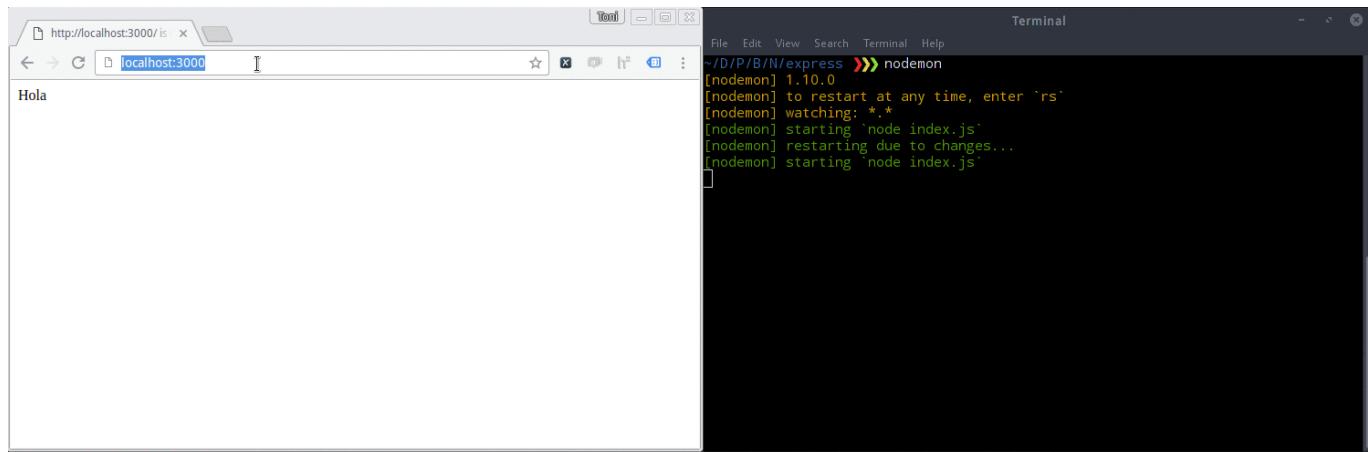
## Nuestro propio Middleware con `app.use()`

Vimos que los middlewares pueden ser muy potentes. Pero como hacemos para crear uno propio?

Express nos facilita esta tarea, ya que el callback pasamos en `app.use` tiene tres parámetros: los que ya conociamos `req` y `res` y uno nuevo, `next`. Lo que esto quiere decir es que cuando el request vaya a la ruta que especificamos en el primer argumento, en este caso `'/'`, express va a ejecutar el callback, cuando encuentre la función `next()` le estamos indicando que corra el siguiente middleware, podemos pensar que todos los `app.get` que veniamos programando también son middleware, por lo tanto son esas funciones las que se ejecutaran luego.

```
app.use("/", function (req, res, next) {
 console.log("Hicieron un Request a " + req.url);
 next();
});
```

Por lo tanto al hacer un request al servidor, primero se pasará por ese middleware (veremos el console log en la terminal) y luego se creará el response según lo que habíamos definido para esa ruta:



Como se podrán imaginar, esta forma de trabajar de *express* hace que sea super potente. Además hay muchísimos middlewares ya creados por otros desarrolladores que podremos usar. ¿Dónde buscarlos? Sí, en npm! También pueden empezar por esta [lista de middleware creados por el equipo de \*express\*](#).

## Enviado datos al servidor:

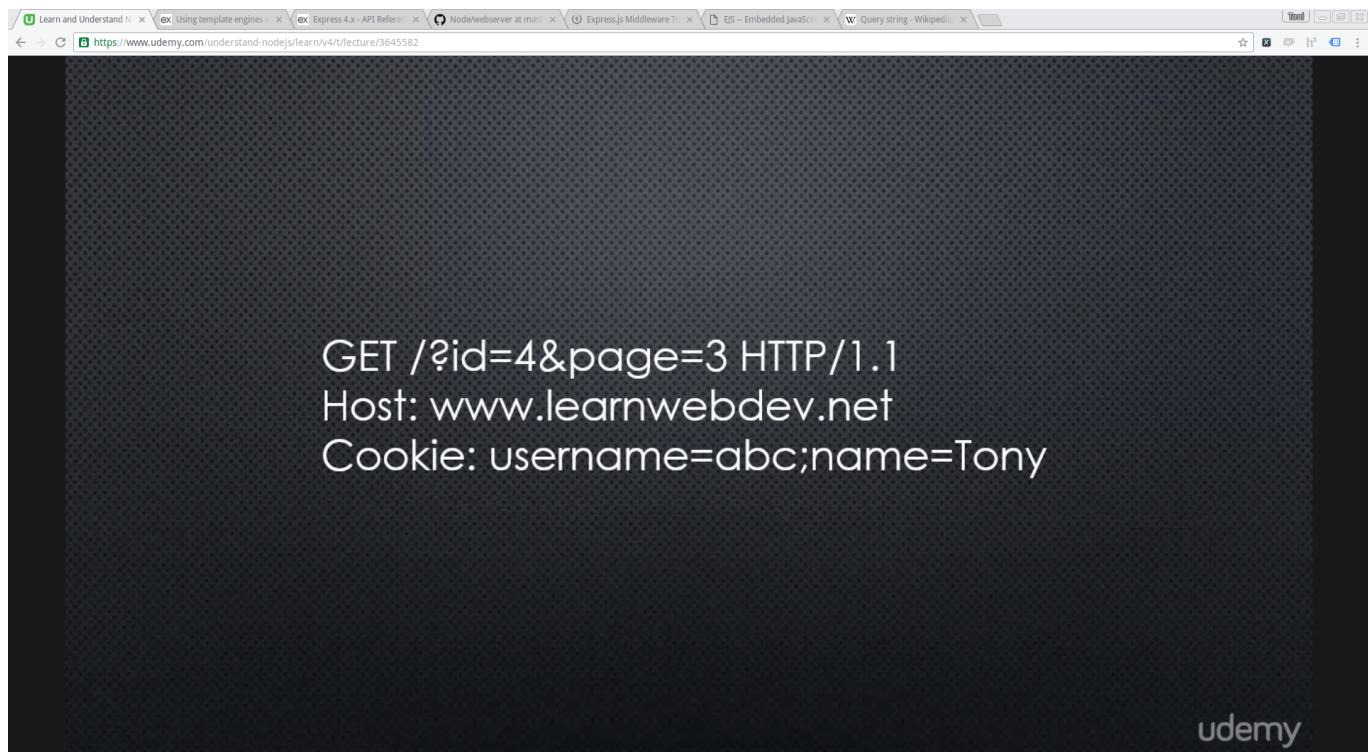
Por ahora nos habíamos concentrado sólo en obtener contenido de nuestra app usando requests tipo **GET**, sin mandar datos nosotros (de hecho el único dato que enviamos fue como un parámetro en la URL). Ahora veremos que hay distintas formas de enviar datos al servidor.

### Query String

Una forma de enviar datos es hacerlo en la URL a la que apuntamos el request. Para ello nos valemos de una serie de parámetros o datos que se incluyen en la URL. Normalmente distinguimos en la URL por un nombre y un valor separados por el signo igual, y se separan del endpoint por el carácter ?, y entre cada variable por el signo &. Por ejemplo:

`www.com/index?nombredevalor1=valor1&nombredevalor2=valor2`

o



## POST y Forms

Otras formas de enviar datos al servidor es a través de Formularios. Ahora usaremos otro verbo HTTP, el **POST**.



En este caso, los datos irán dentro del **body** del request, y lo que nos indica que es datos de un formulario es el **content-type** que estará seteado a 'x-www-form-urlencoded'.

## POST y Ajax

También podemos enviar datos al servidor en formato JSON, donde también usamos POST, pero el **content-type** es ahora **application/json**. Por ejemplo en un request generado por AJAX. En este caso los datos también estarán en el **body** del request.

```
POST / HTTP/1.1
Host: www.learnwebdev.net
Content-Type: application/json
Cookie: num=4;page=2
```

```
{
 "username": "Tony",
 "password": "pwd"
}
```

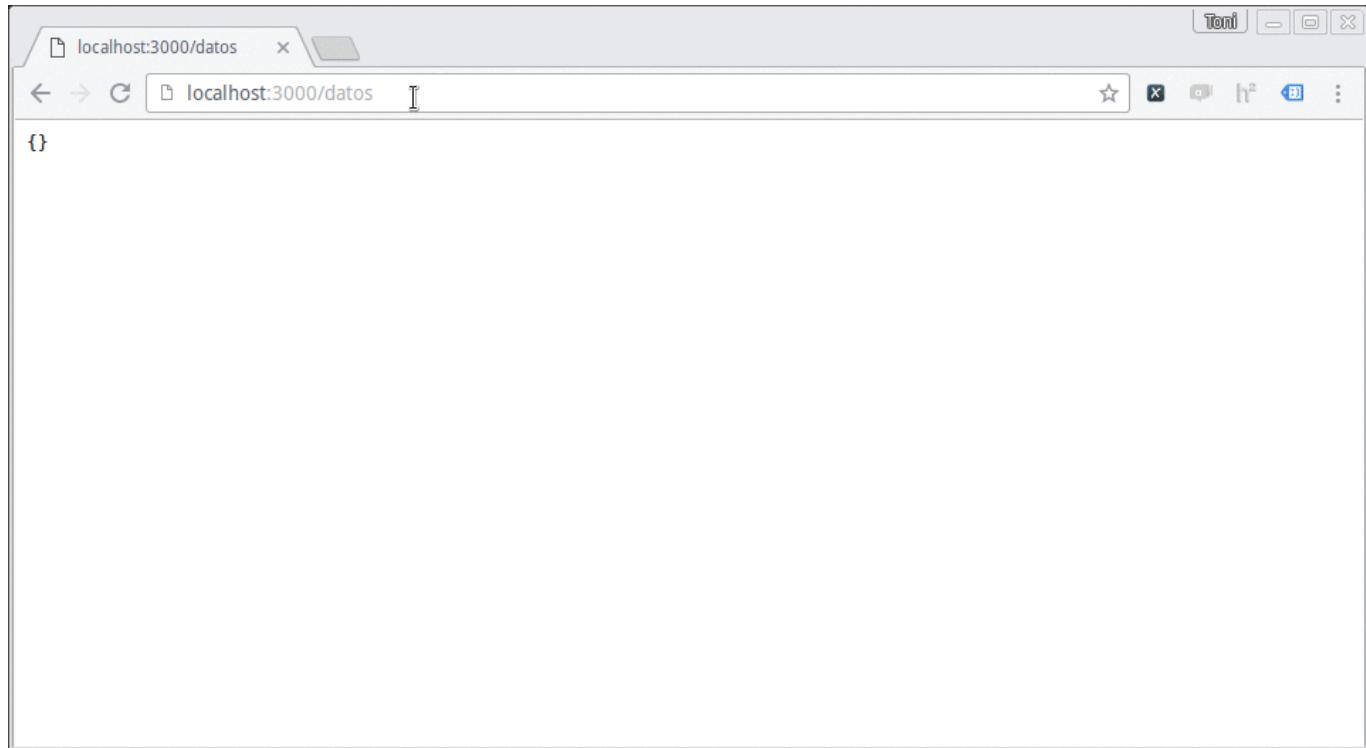
En fin, como vemos vamos a necesitar middleware para lograr procesar cada una de estas formas de recibir datos (pensando desde el lado del servidor).

## Tomando los Datos

De las anteriores, la forma más simple es tomar los parámetros enviados por el query string. Esto lo hacemos usando en nuestra ruta el objeto `req.query`. Express busca y parsea el query string por si sólo y guarda los resultados en ese objeto. Veamos un ejemplo:

```
app.get("/datos/", function (req, res) {
 res.json(req.query);
});
```

Corramos el servidor y probemos nuestra nueva ruta:



Como vemos, *Express* toma las variables del query string y al parsearla las guarda en un objeto cuyas propiedades son el nombre de esas variables junto con sus respectivos valores.

## Forms

Ahora, si queremos tomar datos que vienen de un formulario vamos a tener que usar un middleware, porque no es algo que *express* haga *out of the box*.

Podríamos escribir el código nosotros mismos, pero ya alguien lo hizo por nosotros! El paquete que antes usábamos era [body-parser](#). Hay muchos otros paquetes que hacen lo mismo, pero este era el más común para usar con *Express*. Puedes ver su documentación [aquí](#) Pero ahora usamos [express.json\(\)](#), el cuál es un método ya incluído en *express* desde la versión 4.16.0.

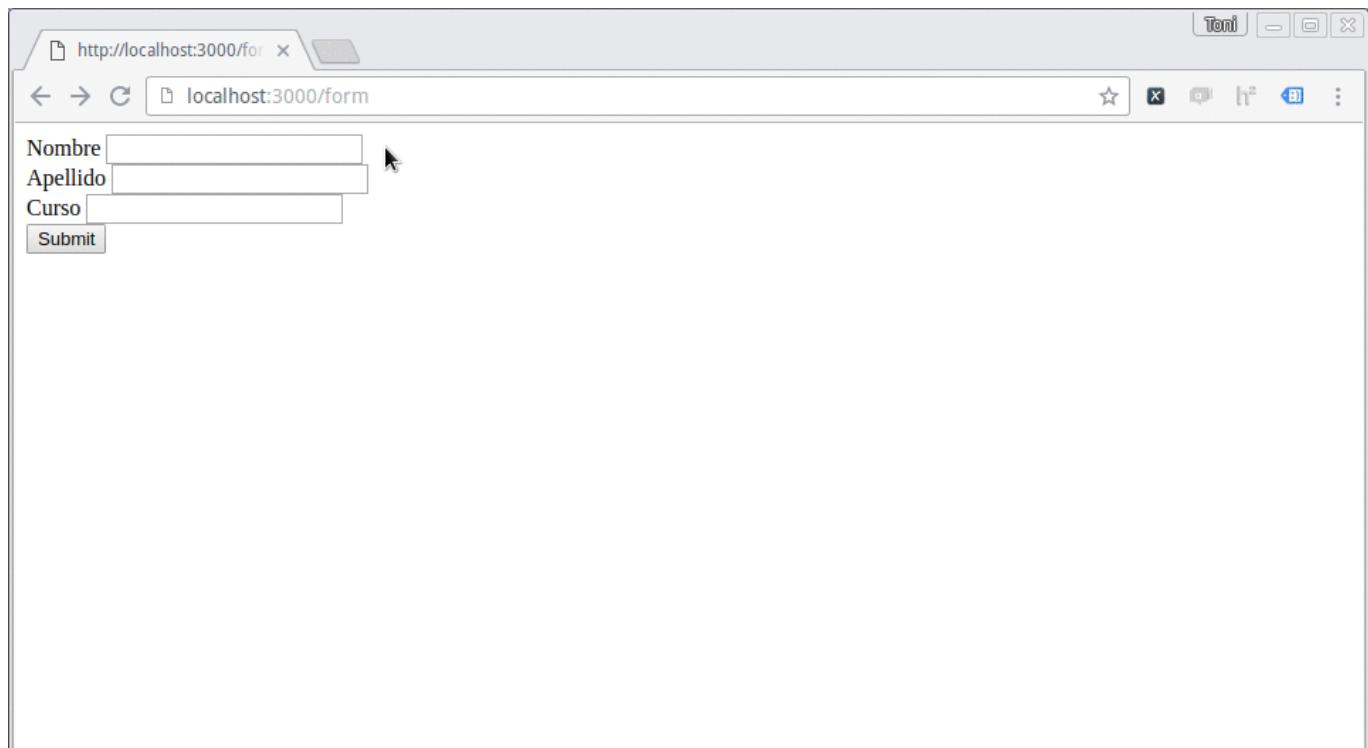
Vamos a encontrar la documentación sobre como usarlo [aquí](#). También vamos a necesitar crear un formulario en html. Haremos que un GET en [/form](#) devuelva un formulario HTML simple. También creamos una nueva ruta, que reciba un POST en la misma URL.

```
app.get("/form", function (req, res) {
 res.send(
 '<html><head> \
 <link href="/assets/style.css" rel="stylesheet"> \
 </head><body>\
 <form method="POST" action="/form">\
 Nombre <input name="nombre" type="text">
 \
 Apellido <input name="apellido" type="text">
 \
 Curso <input name="curso" type="text">
 \
 <input type="submit"> \
 </form>
 </body></html>'
);
});
```

```
app.use(express.urlencoded({ extended: false }));
app.post("/form", function (req, res) {
 res.json(req.body);
});
```

Le decimos a app que use el middleware `express.urlencoded`. Ahora vamos a ir a `/form` y vamos a probar submitear el formulario. Cuando hacemos el submit, el browser genera un request tipo POST, ese request será *capturado* en la nueva ruta `.post()` que definimos y luego enviará como response el objeto `req.body`, que es donde `express.json()` guarda los datos procesados.

Vamos a probarlo:



Excelente! con la ayuda del método `express.json()` que tiene express vamos a poder trabajar con las variables que recibiamos de un formulario.

## Ajax

Ok, ahora vemos el tercer escenario, en donde el cliente genera un request tipo POST que contenga datos en formato JSON.

En el escenario anterior le dijimos a app que use el método `express.urlencoded` y ahora vamos a decirle que use el método `express.json`

```
app.use(express.json());
app.post("/formjson", function (req, res) {
 res.json(req.body);
});
```

Ahora, para poder probar este *endpoint* y enviar datos, vamos a usar una herramienta muy útil: **POSTMAN**, que es una app (viene como extensión de Chrome y una app para Mac) que nos permite generar todo tipos de request HTTP, y nos va a ser de gran ayuda para probar nuestros endpoint y APIs.

También probá usando un request generado desde una página con AJAX

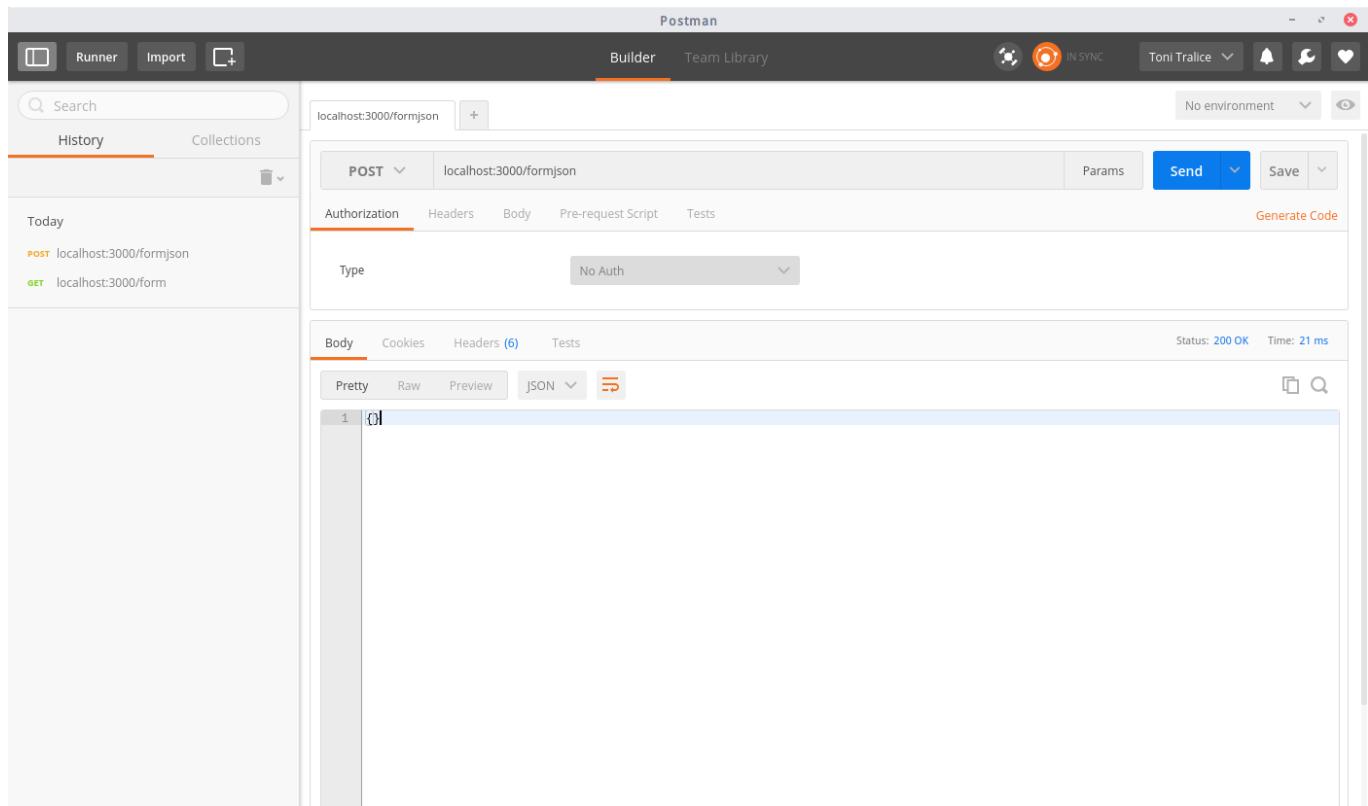
## POSTMAN

Veamos como usarlo para generar un post apuntado a `/formjson` y que envie data en formato JSON.

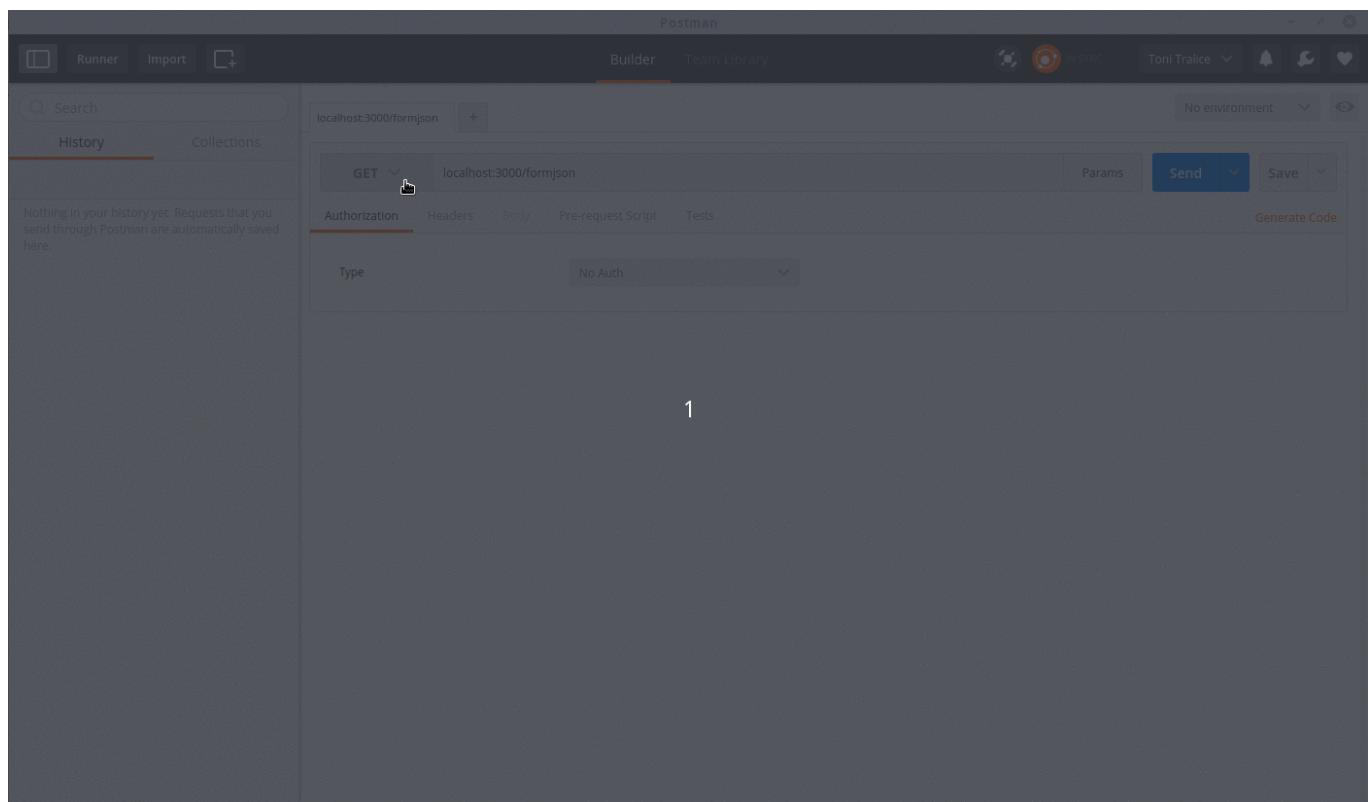
The screenshot shows the Postman application interface. On the left, there's a sidebar with 'History' and 'Collections'. In the main area, a 'GET' request is selected for 'localhost:3000/form'. The 'Body' tab shows an HTML form with fields for 'Nombre', 'Apellido', and 'Curso'. The 'Headers' tab shows '(6)' headers. The 'Tests' tab shows a snippet of JavaScript code. The 'Preview' tab shows the raw HTML of the response. The 'HTML' tab shows the rendered HTML of the response. The top right shows 'Status: 200 OK' and 'Time: 27 ms'. There are 'Send', 'Save', and 'Generate Code' buttons at the top right of the request panel.

Como vemos en la imagen, la interfaz es bastante intuitiva, a la izquierda seleccionamos qué tipo de http request queremos hacer (en el ejemplo hicimos un GET). Especificamos la URL ('/form') en este caso, y al apretar el botón SEND se genera el request. Abajo podremos ver el resultado del mismo. En este caso, el servidor nos devolvió el html del formulario que habíamos creado en ese endpoint. Estos son los datos que recibe el browser cuando escribimos la URL en la barra de direcciones, pero el browser al recibir los datos los parsea y los dibuja automáticamente, por eso vemos el formulario directamente.

Ahora, queremos probar hacer un **POST** a `/formjson`, por lo tanto vamos a cambiar la URL en postman y el tipo de request. Si probamos veremos el siguiente resultado: `{}`.



Lo primero que queremos saber es si el request fué procesado con éxito, sabemos que sí porque el **status** que trajo el response es el número **200**, que significa '**Todo ok**' en el standart HTTP. También sabemos que ese endpoint nos debería devolver un JSON con los datos que hayamos enviado en el POST, como no había datos en el request, era esperable que recibamos un objeto vacío. Por lo tanto, hasta acá venimos bien. Intentemos agregar datos al **POST** a ver que pasa:



Como vemos en la imagen de arriba, para agregar datos al post tenemos que ir a la pestaña **Body**, y allí seleccionar **raw**, y como tipo de datos usar **application/json**. Luego completamos dentro del cuadro de

texto con el objeto en formato JSON que queremos enviar (tengan cuidado con el formato, de hecho las dobles comillas son obligatorias para los nombres). Luego de cargar los datos, hacemos click en **Send** y abajo vemos la respuesta, qué como habíamos dicho, tiene que ser el mismo objeto que hemos enviado. Todo funciona sin problemas!

*Con POSTMAN podemos emular varios request, de hecho podríamos haber emulado el formulario con esto. Intenten hacerlo ustedes: en **Body** pueden empezar seleccionando 'x-www-form-urlencoded'. ¿Cuál es la diferencia con 'form-data'?*

## Estructurando nuestra App

Al hacer una app compleja, en donde van a existir muchos endpoints, se puede poner muy engorroso mantener todo en un mismo archivo. Por eso, vamos algunos patrones para mantener la aplicación lo más ordenada posible.

**Esto también es muy personal, cada uno puede estructurar su aplicación de la forma que le resulte más fácil de mantener y entender. Cuando trabajen en equipo tienen que estar de acuerdo en esto!!**

Express tiene un generador de proyectos (parecido al `npm init` pero más potente) llamado **express-generator**. Podemos ver su documentación [aquí](#). Vamos a instalarlo y crear un proyecto usándolo.

```
npm install express-generator -g express myapp -e
```

Para este ejemplo he creado una app con el nombre 'myapp' y le indiqué que el template engine a usar será **EJS**. Veamos todos los archivos que `express-generator` creó por nosotros:

```
create : myapp
create : myapp/package.json
create : myapp/app.js
create : myapp/views
create : myapp/views/index.ejs
create : myapp/views/error.ejs
create : myapp/public
create : myapp/routes
create : myapp/routes/index.js
create : myapp/routes/users.js
create : myapp/bin
create : myapp/bin/www
create : myapp/public/javascripts
create : myapp/public/stylesheets
create : myapp/public/stylesheets/style.css
```

Como vemos, se creó un **package.json** donde están listadas las dependencias de nuestro proyecto, por lo tanto lo primero que deberíamos hacer es entrar a la carpeta del proyecto y hacer un `npm install`.

Ahora veamos el archivo **app.js**. En ese archivo el generador incluyó a la app varios middleware de uso muy común (entre ellos están **express.json** y **express.static**).

```

var createError = require("http-errors");
var express = require("express");
var path = require("path");
var cookieParser = require("cookie-parser");
var logger = require("morgan");

var indexRouter = require("./routes/index");
var usersRouter = require("./routes/users");

var app = express();

// view engine setup
app.set("views", path.join(__dirname, "views"));
app.set("view engine", "jade");

app.use(logger("dev"));
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, "public")));

app.use("/", indexRouter);
app.use("/users", usersRouter);

```

Nos podemos imaginar que la carpeta `/public` es donde estará todo nuestro material estático (images, .css, js, etc..). En la carpeta `/views` estarán los templates del proyecto. Y también vemos una tercera carpeta de recursos que es la carpeta `/routes`, en ella guardaremos las rutas de cada endpoint de nuestra aplicación. Si investigamos un archivo autogenerados que están dentro de esa carpeta, por ejemplo `/routes/index.js` veremos que está pensado para ser importado ya que tiene el `module.exports` al final.

```

var express = require("express");
var router = express.Router();

/* GET home page. */
router.get("/", function (req, res, next) {
 res.render("index", { title: "Express" });
});

module.exports = router;

```

De hecho, si volvemos a mirar `app.js` vemos que importa como módulos a los archivos dentro de `/routes`.

```

var routes = require("./routes/index");
var users = require("./routes/users");

```

Un concepto nuevo que no habíamos visto, es la función `express.Router()`, que es un middleware de `express` para facilitarnos el uso de rutas. En vez de poner las rutas directamente en la `app` vamos a cargarlas de

la misma forma pero en el `router`, luego exportamos el objeto `router` en cada archivo y dentro de `app.js` vamos requerirlo y cargalos en nuestra app:

```
app.use("/", routes);
app.use("/users", users);
```

Hay que notar que cuando usamos `app.use` y le pasamos un `Router` también le pasamos un path, esto quiere decir que las rutas definidas dentro del `Router` que les pasamos serán accesibles desde el path origin que le pasamos. Por ejemplo:

Como en el archivo `routes/users.js` tenemos una ruta para el URL `/`, al cargarlo usando `app.use()` en `/users`, ese path será accesible en el path `/users/`. Para que se entienda mejor, creé una nueva ruta dentro de `users.js` con la URL `/api`, esa ruta será accesible desde `/users/api/`, ya que cargamos ese `Router` en `/users`. ;D

## Cross-Origin Resource Sharing (CORS)

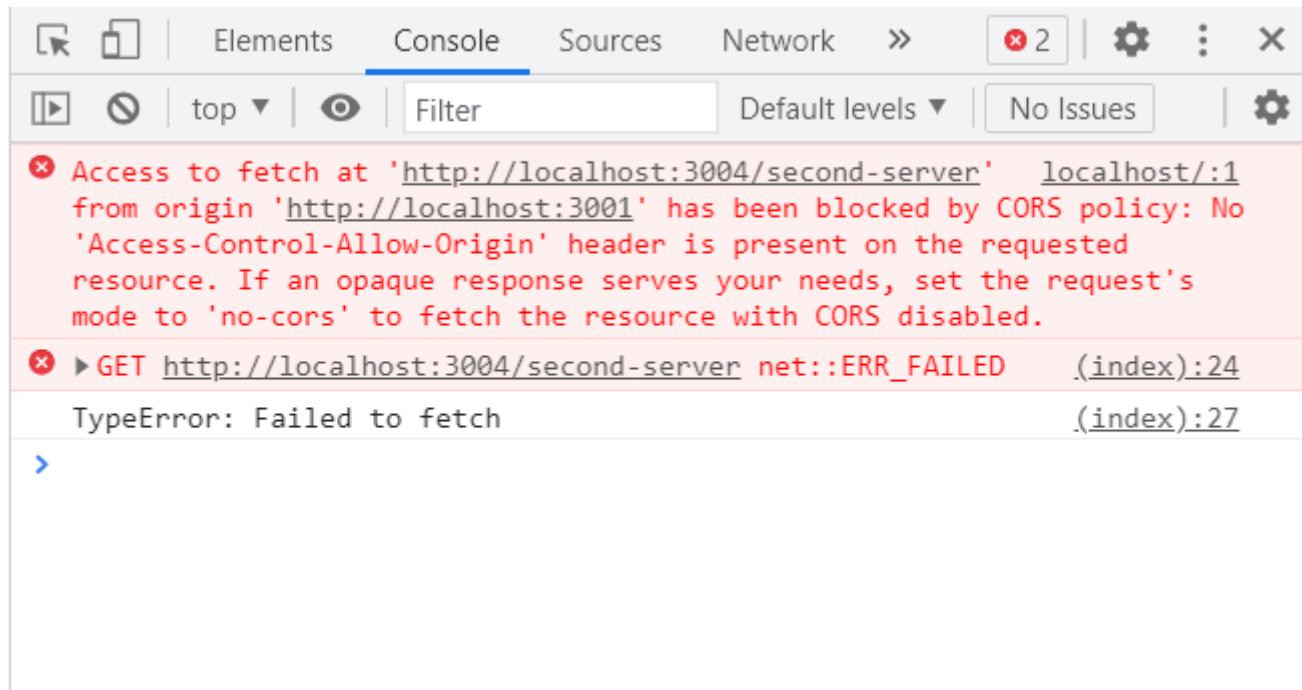
Por razones de seguridad, los navegadores sólo permiten que se carguen recursos que provengan del mismo origen. Dos URL tienen el mismo origen si el protocolo , el puerto y el host son los mismos para ambos. Esto es lo que llamamos Same-Origin Policy (SOP). Esta política ya viene por default en los navegadores y controla las interacciones entre orígenes diferentes ayudando así a aislar documentos potencialmente maliciosos y reduciendo posibles ataques.

Ejemplos de URL con mismo origen:

```
http://e-commerce.com/admin/orders //Mismo origen, Path diferente
```

```
http://e-commerce.com/user/me //Mismo origen, Path diferente
```

Sin embargo, muchas veces necesitamos cargar en nuestro sitio recursos provenientes de otro origen, por ejemplo, cuando utilizamos una fuente distinta o queremos mostrar una imagen. ¿Qué pasaría si quisieramos realizar una petición a un servidor con un dominio diferente? En ese caso, veríamos un error como este:



¿Qué está pasando? Desde 'http://localhost:3001' se realizó una petición a 'http://localhost:3004/second-server' y el navegador bloqueó la solicitud por política de CORS (Cross-Origin Resource Sharing). A diferencia de la política del mismo origen (SOP) este mecanismo nos permite, mediante el uso de cabeceras HTTP adicionales, acceder a recursos desde un dominio, un protocolo o un puerto diferente al del documento que lo generó. Este tipo de solicitudes se denominan de origen cruzado.

Ejemplos de URL con distinto origen:

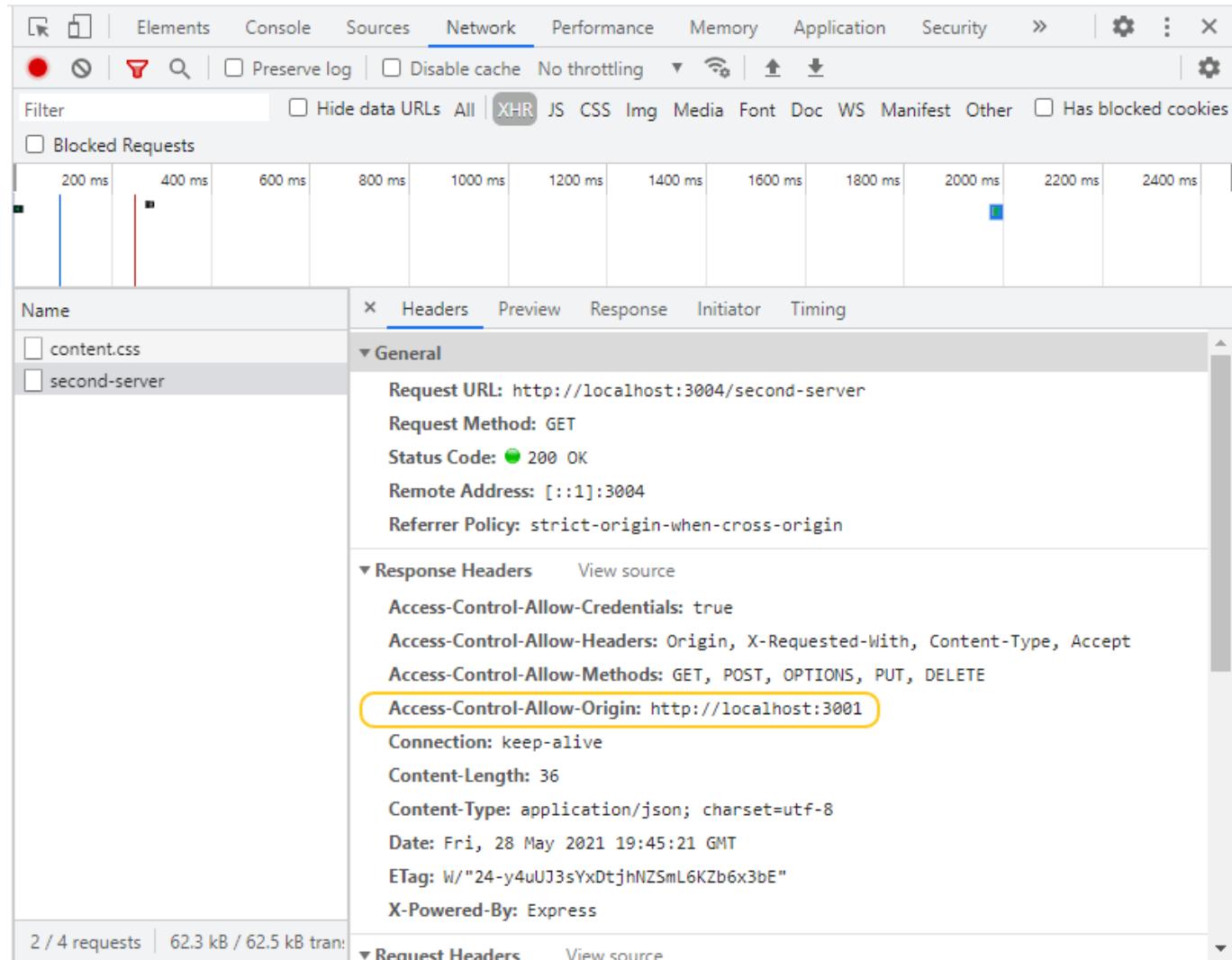
```
https://e-commerce.com/user/me //Diferente protocolo
http://api.e-commerce.com/user/me //Diferente host
```

### Access-Control-Allow-Origin

Para habilitar una petición de origen cruzado debemos incluir una cabecera denominada Access-Control-Allow-Origin en la respuesta de la petición, donde debe indicarse el dominio al que se le quiere dar permiso. Es decir, en nuestro ejemplo el servidor que está utilizando el puerto 3004 debería incluir en su respuesta a la solicitud de localhost:3001 un header Access-Control-Allow-Origin donde se indica el dominio al que se le quiere dar permiso.

```
Access-Control-Allow-Origin: http://localhost:3001
```

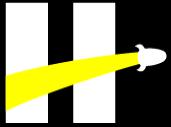
De esta forma, el navegador comprobará dichas cabeceras y si coinciden con el dominio de origen que realizó la petición, ésta se permitirá. En el ejemplo anterior, la cabecera tiene el valor http://localhost:3001, pero en algunos casos el valor puede ser un "\*". El asterisco indica que se permiten peticiones de origen cruzado a cualquier dominio.



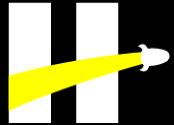
Además de ese, existen otros cors headers. Algunos de ellos son:

- Access-Control-Allow-Origin: ¿qué origen está permitido?
- Access-Control-Allow-Credentials: ¿también se aceptan solicitudes cuando el modo de credenciales es include?
- Access-Control-Allow-Headers: ¿qué cabeceras pueden utilizarse?
- Access-Control-Allow-Methods: ¿qué métodos de petición HTTP están permitidos?
- Access-Control-Expose-Headers: ¿qué cabeceras pueden mostrarse?
- Access-Control-Max-Age: ¿cuándo pierde su validez la solicitud preflight?
- Access-Control-Request-Headers: ¿qué header HTTP se indica en la solicitud preflight?
- Access-Control-Request-Method: ¿qué método de petición HTTP se indica en la solicitud preflight?

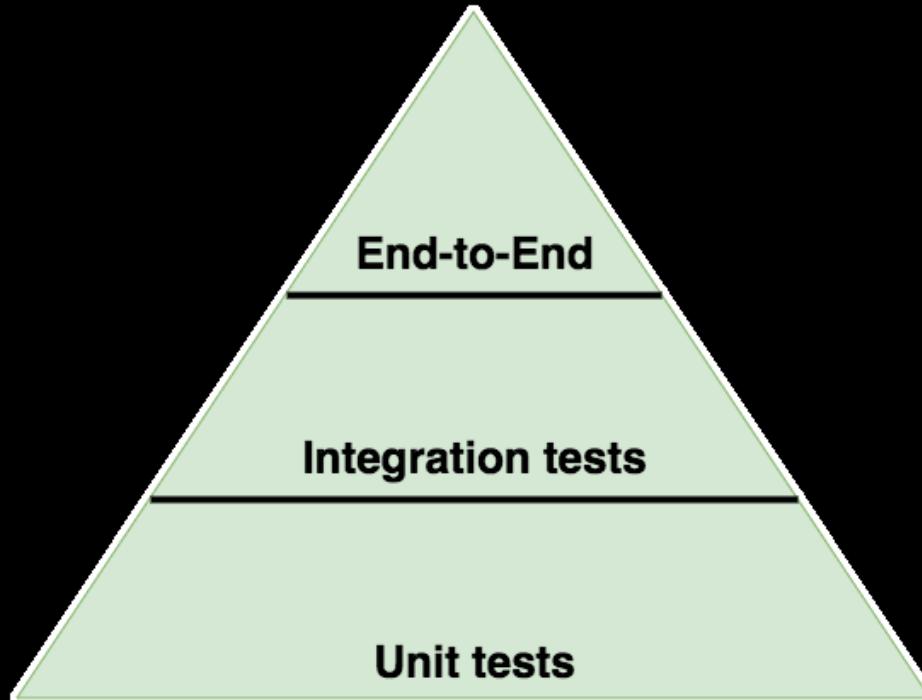
Otra forma que tenemos para habilitar las solicitudes CORS pueden ser librerías que manejen las autorizaciones por nosotros a modo de middleware. Por ejemplo, el módulo [cors](#).

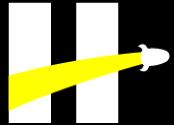


# Testing



# ¿Qué cosas testear? ¿Cuántos tests hacer?

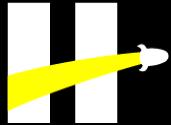




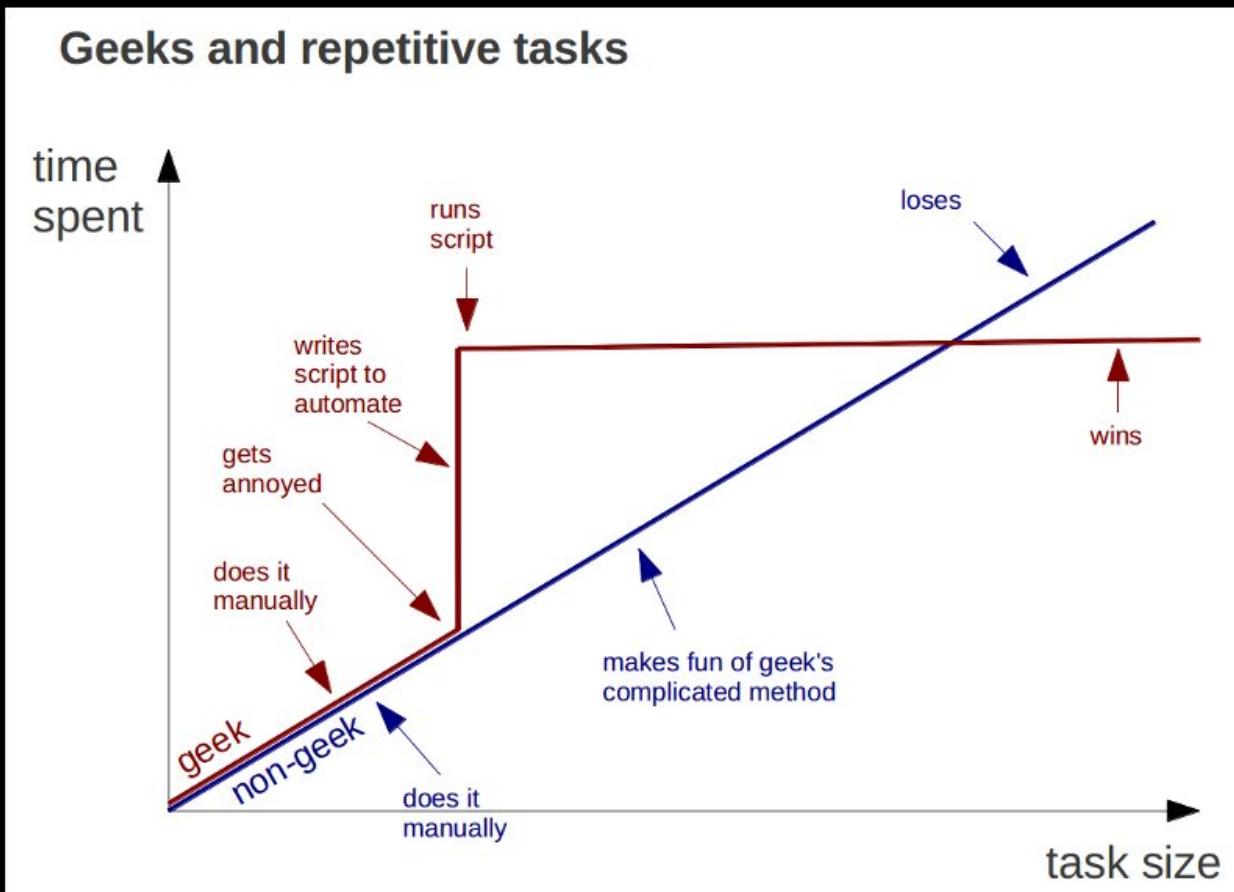
# Unit Testing

Un buen test unitario debería ser:

- Completamente automatizable
- Poder ejecutarse en cualquier orden en conjunto con otros tests.
- **Siempre** retorna el mismo resultado, no importa cuantas veces lo corra.
- Es rápido
- Testea un solo concepto lógico del sistema
- Es fácil de entender al leerlo
- Es fácil de mantener

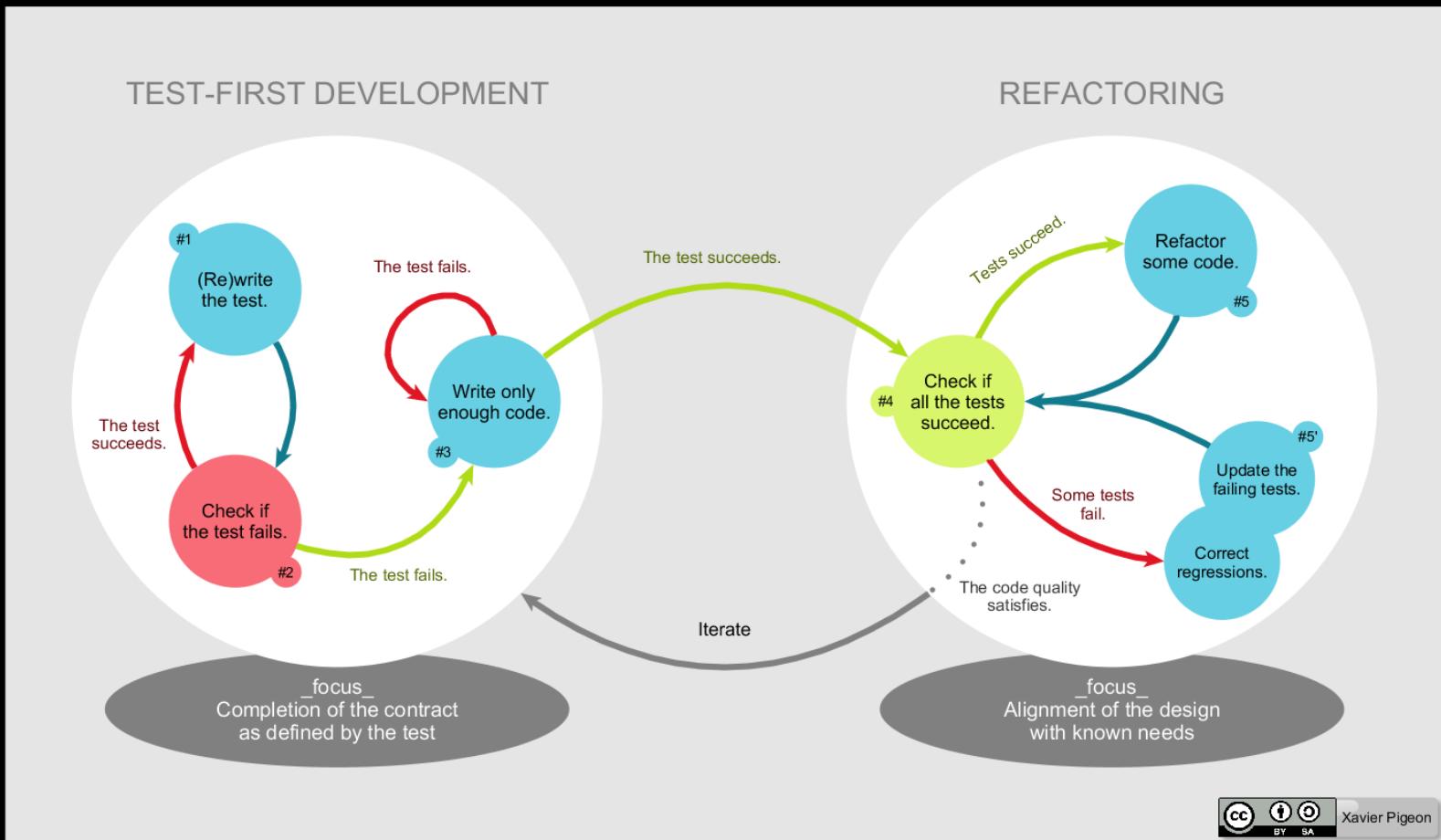


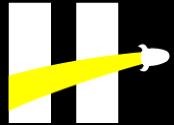
# Unit Testing





# TDD





# Testing Frameworks



VS



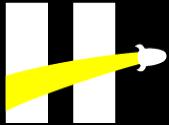
Assertion Library included



Needs other libraries  
(assertion, mocking)



More flexibility



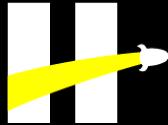
# Jest

## Configuration



```
1 // Installation
2 npm install --save-dev jest
3
4 // package.json configuration
5 ...
6
7 "scripts": {
8 "test": "jest"
9 }
10
11 ...
```



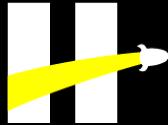


# Jest

## CLI Options

```
1 // Run only tests matching
2 jest test-pattern // pattern
3 jest path/to/test.js // filename
4 jest -t name-spec // describe or test name
5
6 // Run in watch mode
7 jest --watch // by default only changed files
8 jest --watchAll
9
10 // Show summary of each test file
11 // Running one file only --> automatically verbose
12 jest --verbose
```





# Jest

## First Example



```
1 // sum.js
2
3 function sum(a, b) {
4 return a + b;
5 }
6
7 module.exports = sum;
```



```
1 // sum.test.js
2
3 const sum = require('./sum');
4
5 // it === test
6 it('should return 8 if adding 3 and 5', () => {
7 expect(sum(3, 5)).toBe(8);
8 });
```





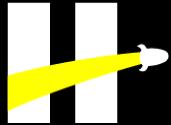
# Jest

## Matchers



`expect` devuelve un "expectation" object sobre el cual se pueden invocar los matchers

- **toBe**: igualdad exacta
- **toEqual**: verificación recursiva de cada propiedad del objeto o elemento del arreglo
- **toBeNull**, **toBeUndefined**, **toBeDefined**
- **toBeTruthy**: verifica que el valor de veracidad sea verdadero sin necesariamente ser literalmente **true**
- **toBeFalsy**: verifica que el valor de veracidad sea falso sin necesariamente ser literalmente **false**



# Jest

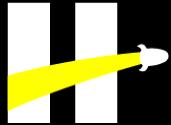
## Matchers



- **toBeGreaterThan, toBeGreaterThanOrEqual, toBeLessThan, toBeLessThanOrEqual** para números
- **toBeCloseTo** para números con decimales
- **toMatch**: compara contra una expresión regular
- **toContain**: verifica si dentro de un arreglo existe un elemento
- **toThrow**: verifica si la función arroja un error

Y más ...



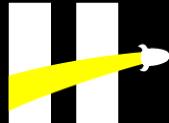


# Jest

## Running Options



- **xit**: evita que ese test en particular se ejecute
- **describe**: permite agrupar varios tests dentro de una misma temática o categoría (acepta mas de un nivel de anidación)
- **xdescribe**: evita que todos los tests de ese grupo se ejecuten
- **it.only**: hace que ese test sea el único en ejecutarse



# Asynchronous Code

## Resolved Promises



```
1 it('should resolve to Henry Promise', () => {
2 promisifiedFunction(false).then(data => {
3 expect(data).toBe('asd');
4 });
5 });
```



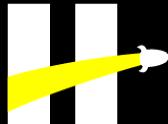
```
PASS ./promise.test.js
✓ should resolve to Henry Promise (1 ms)
```



```
1 it('should resolve to Henry Promise', () => {
2 return promisifiedFunction(true).then(data => {
3 expect(data).toBe('Henry Promise');
4 });
5 });
```



Si se omite el **return** el test va a completarse antes de que la promesa se complete



# Asynchronous Code

## Rejected Promises



```
1 it('should reject to Rejected Promise', () => {
2 return promisifiedFunction(true).catch(e => {
3 expect(e).toMatch('Rejected Promise')
4 });
5 });
```



PASS ./promise.test.js  
✓ should reject to Rejected Promise (2004 ms)



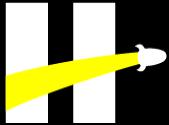
```
1 it('should reject to Rejected Promise', () => {
2 expect.assertions(1);
3 return promisifiedFunction(false).catch(e => {
4 expect(e).toMatch('Rejected Promise')
5 });
6 });
```



+ sobre  
assertions



Si se omite el **assertions** una promesa cumplida no hará que el test falle

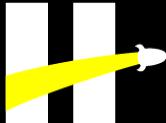


# Hooks



```
1 beforeAll(() => {
2 ...
3 });
4
5 beforeEach(() => {
6 ...
7 });
8
9 afterEach(() => {
10 ...
11 });
12
13 afterAll(() => {
14 ...
15 });
```





# Mock Functions

a.k.a "spies"



Permite testear el comportamiento de una función que indirectamente fue ejecutada por otra



```
1 const mockFunction = jest.fn(person => person.age > 18);
```

- **.mock.calls**: array con todas las invocaciones a la función donde cada elemento contiene otro array con los argumentos pasados
- **.mock.results**: array con todos los resultados devueltos por la función donde cada elemento contiene un objeto con el valor y el tipo de retorno

Pss, querés saber [más...](#) ?



<mock.test.js />



# Supertest

Permite testear los request a nuestro servidor de forma autocontenido sin necesidad de levantar nuestra app

- **statusCode**: podemos verificar si el código de respuesta es el adecuado
- **response**: podemos verificar si la respuesta del endpoint coincide con lo esperado (Puede ser por text o body)
- **type**: podemos verificar si el content type devuelto es el correcto



Si el **.listen** de express se encuentra en el archivo requerido en el testing va a generar que el test no termine de ejecutar nunca

# Mini Testing Workshop

Vamos a codear una mini app para hacer algo de testing.

Para testing vamos a usar **jest** y **supertest**.

**supertest** nos va a servir para levantar nuestra app cada vez que se ejecutan los tests, de tal modo que los tests sean autocontenidos.

## cheatsheet

En el repo tenemos una mini app de express con una serie de endpoints simples. Todo esto está en el archivo **index.js**. Los tests estan en la carpeta **/tests**. Para ejecutarlos hacer:

```
npm test
```

(no se olviden del **npm install**).

## Que hacemos?

Pasar todos los tests

Primero vamos a hacer que los test que están pasen. Es decir que vamos a agregar, o modificar nuestra app hasta que pasen todos los tests.

Agregar nuevos tests

El test de la ruta **sumArray** está incompleto. Falta testar por el caso que devuelva **false**. También falta testear que no sumen dos veces el mismo número para encontrar el resultado.

Agregar nueva funcionalidad

Ahora vamos a agregar una nueva funcionalidad.

## NumString

Vamos a crear un endpoint **/numString** que reciba un string y devuelva el número de caracteres que tiene ese string. Primero vamos a escribir los tests, y luego codear para que pasen: Nuestro nuevo endpoint debería:

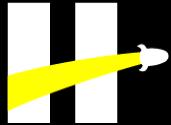
- Responder con status 200.
- Responder con 4 si enviamos 'hola'.
- Responder con un status 400 (bad request) si el string es un número.
- Responder con un status 400 (bad request) si el string esta vacio.

## Pluck

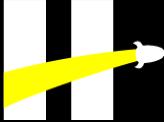
Vamos a crear un endpoint `/pluck` que reciba un arreglo de objetos y un nombre de una propiedad y devuelva un arreglo sólo con los valores de esa propiedad.

Nuestro nuevo endpoint deberia:

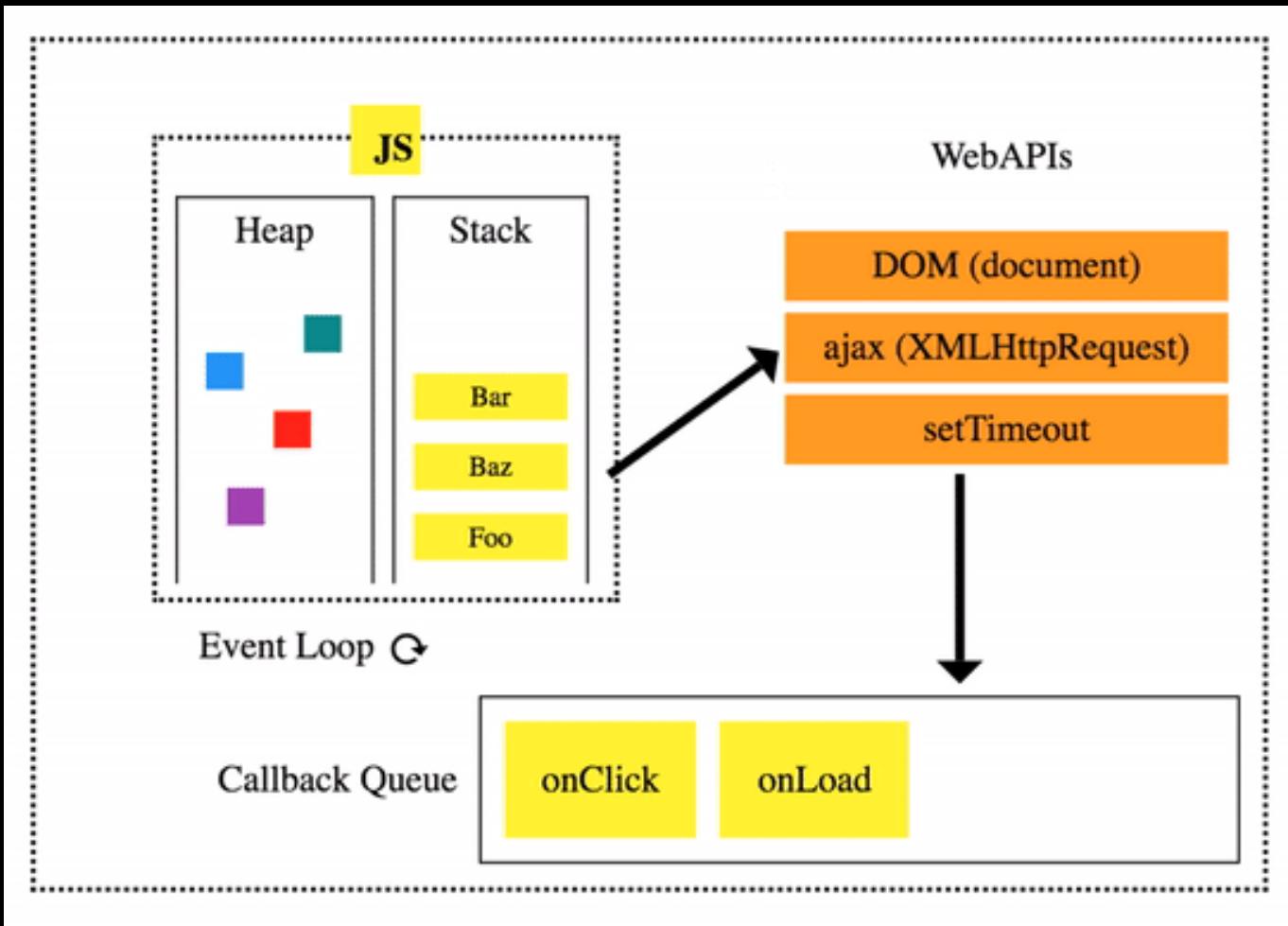
- Responder con status 200.
- Responder con al funcionalidad del pluck.
- Responder con un status 400 (bad request) si array no es un arreglo.
- Responder con un status 400 (bad request) si el string propiedad está vacio.

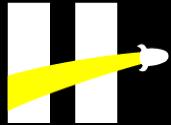


# Event Loop

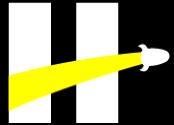


# Event Loop



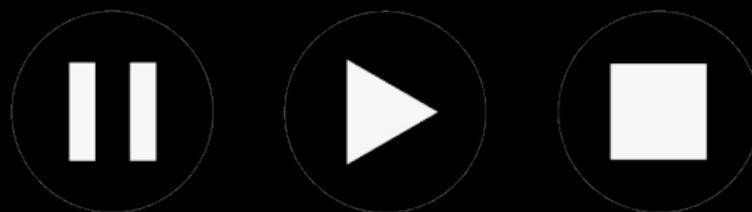


# Generators Functions



# Generator Function

"... are functions that can be exited and later re-entered. Their context (variable bindings) will be saved across re-entrances."

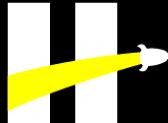




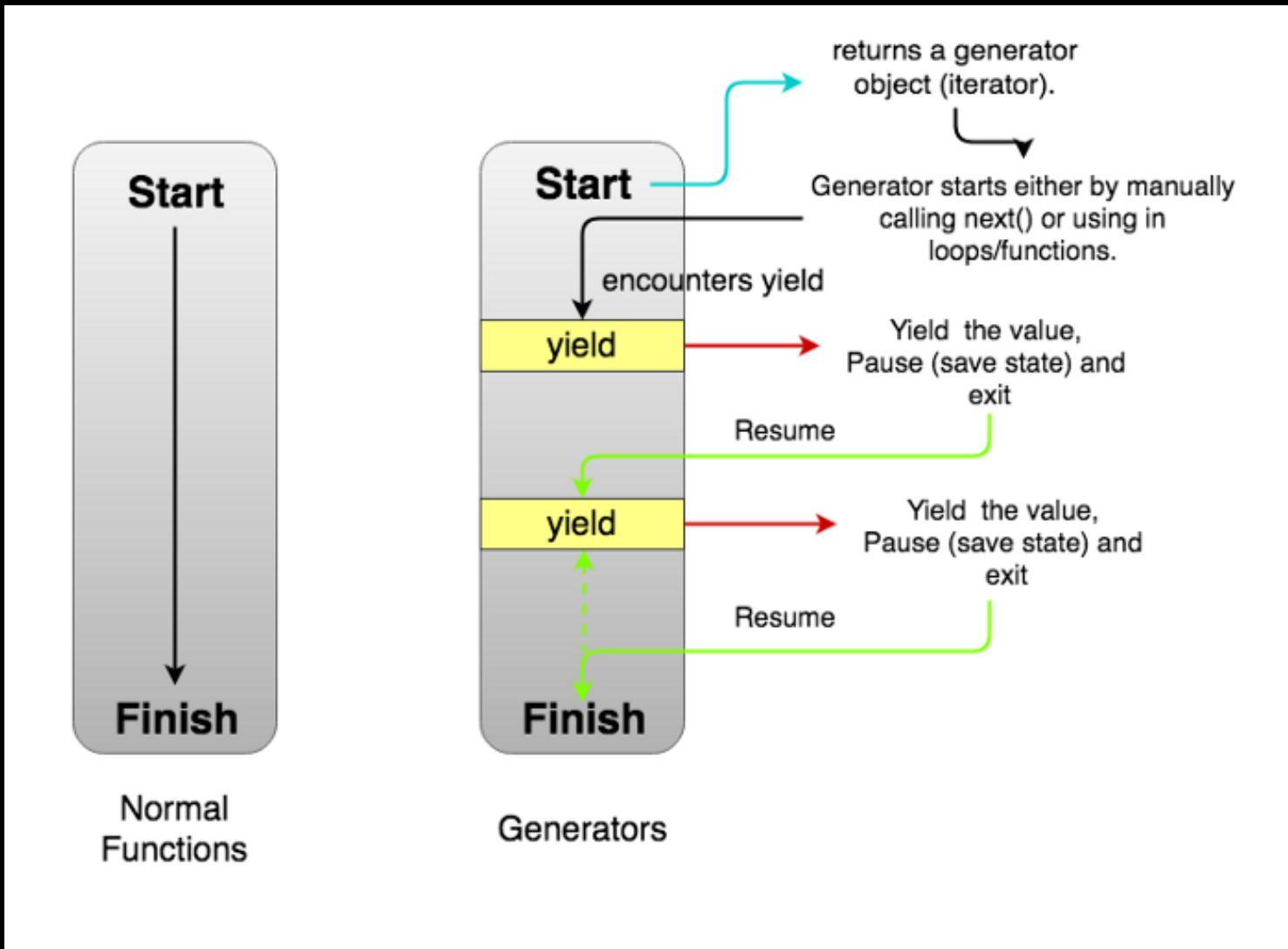
# run-to-completion model

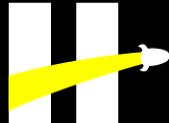
Las funciones que conocíamos hasta ahora en JS seguían este modelo donde la función va a ejecutarse por completo hasta completarse (return/error)

```
● ● ●
1 function normalFunction() {
2 console.log("Iniciando función");
3 console.log("Continuando función");
4 console.log("Finalizando función");
5 console.log("Fin!");
6 }
```



# Flow Differences





# Generator Function return value

`function\*` sirve para declarar un generator que retorna un *Generator object* sobre el cual se puede invocar el método `next()`



```
1 function* generatorShowInstructors() {
2 console.log("Iniciando generator function");
3 yield "Franco";
4 yield "Toni"
5 console.log("Generator function terminada");
6 }
7
8 var generatorObject = generatorShowInstructors();
9
10 generatorObject.next();
```

¡No se ejecuta el cuerpo de la función de forma instantánea!

<demoGF1.js />

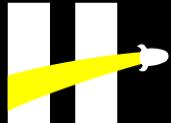


# yield vs return

- **Yield:** Pausa el generator y "retorna" el valor especificado
- **Return:** Finaliza el generator seteando el valor de done a true



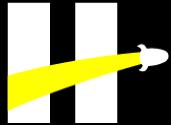
```
1 function* generatorUnreacheableValue() {
2 console.log("Iniciando generator function");
3 yield "First reacheable value";
4 yield "Second reacheable value";
5 return "Return executed";
6 yield "Unreacheable value"
7 }
8
9 var generatorObject = generatorUnreacheableValue();
10
11 generatorObject.next();
12 generatorObject.next();
13 generatorObject.next();
14 generatorObject.next();
```



# Infinite Generator



```
1 function* naturalNumbers() {
2 let number = 1;
3 while(true) {
4 yield number;
5 number = number + 1;
6 }
7 }
8
9 var generatorObject = naturalNumbers();
10
11 generatorObject.next();
12 generatorObject.next();
13 generatorObject.next();
14 generatorObject.next();
```



# Async/Await



# Async Function

Permiten código asíncrono basado en promesas  
sin necesidad de encadenar explícitamente  
promesas



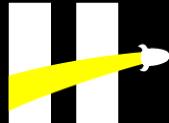
```
1 async function asyncCall() {
2 const result = await resolveAfter2Seconds();
3 }
```



# Basic Flow



```
1 function resolveAfter2Seconds() {
2 return new Promise(resolve => {
3 setTimeout(() => {
4 resolve('resolved');
5 }, 2000);
6 });
7 }
8
9 async function asyncCall() {
10 console.log('calling');
11 const result = await resolveAfter2Seconds();
12 console.log(result);
13 }
```



# Async function return value

¡Siempre retorna una promesa!

```
1 function resolveAfter2Seconds() {
2 return new Promise(resolve => {
3 setTimeout(() => {
4 resolve('Promesa resuelta!');
5 }, 2000);
6 });
7 }
8
9 async function asyncCall() {
10 console.log('Iniciando asyncCall');
11 const result = await resolveAfter2Seconds();
12 console.log(result);
13 }
14
15 var p1 = asyncCall(); // p1 --> Promise
```



## Success

La promesa retornada se va a resolver al valor returnedo por la función asíncrona

## Error

La promesa retornada se va a rechazar con la excepción lanzada por la función asíncrona

`<demoReturnValue.js />`



# Async Flow

"Yielding control"



```
1 async function showInstructors() {
2 const instructor1 = await new Promise((resolve) => setTimeout(() => resolve('Franco')));
3 console.log(instructor1);
4 const instructor2 = await new Promise((resolve) => setTimeout(() => resolve('Toni')));
5 console.log(instructor2);
6 }
7
8 function henryAwait() {
9 console.log("¿Quienes son los intstructores de Henry?");
10 showInstructors();
11 console.log("Gracias vuelvan pronto");
12 }
13
14 henryAwait()
15 console.log("FIN");
```



# Async Flow

"Yielding control"



```
1 async function showInstructors() {
2 const instructor1 = await new Promise((resolve) => setTimeout(() => resolve('Franco')));
3 console.log(instructor1);
4 const instructor2 = await new Promise((resolve) => setTimeout(() => resolve('Toni')));
5 console.log(instructor2);
6 }
7
8 async function henryAwait() {
9 console.log("¿Quienes son los intstructores de Henry?");
10 await showInstructors();
11 console.log("Gracias vuelvan pronto");
12 }
13
14 await henryAwait()
15 console.log("FIN");
```



# Async/Await in Loops

```
● ● ●
1 const instructores = ['Franco', 'Toni', 'Martu', 'Diego'];
2
3 const delay = 1000;
4
5 async function henryAwait() {
6 console.log("¿Quienes son los intstructores de Henry?");
7 for (let i = 0; i < instructores.length; i++) {
8 const instructor = await new Promise(resolve => setTimeout(
9 () => resolve(instructores[i]),
10 delay
11))
12);
13 console.log(instructor);
14 }
15 console.log("Gracias vuelvan pronto");
16 }
17
18 henryAwait();
```

.map



Si cada promesa tarda mucho en resolverse, al estar encadenandolas de forma secuencial, el tiempo total de ejecución será muy alto

<demoForLoop.js />



# Async/Await in Loops

## with callback



```
1 const instructores = ['Franco', 'Toni', 'Martu', 'Diego'];
2
3 async function henryAwait() {
4 console.log("¿Quienes son los intstructores de Henry?");
5 instructores.forEach(async instructor => {
6 const name = await new Promise(resolve => setTimeout(
7 () => resolve(instructor),
8 Math.random() * 1000
9)
10);
11 console.log(name);
12 });
13 console.log("Gracias vuelvan pronto");
14 }
15
16 henryAwait();
```

```
¿Quienes son los intstructores de Henry?
Gracias vuelvan pronto
← ▶ Promise { <state>: "fulfilled", <value>: undefined }
Martu
Toni
Franco
Diego
```





# Ventajas

- El código suele ser más prolíjo y similar a código sincrónico

```
 1 const readFilePromise = (archivo) => {
 2 promisifiedReadFile(archivo)
 3 .then(file => {
 4 console.log("Log promise file: ", file);
 5 return "Lectura exitosa";
 6 });
 7 }
 8
 9 const readFileAsync = async(archivo) => {
10 console.log("Log async file: ", await promisifiedReadFile(archivo));
11 return "Lectura exitosa";
12 }
```

<demoCleanCode.js />



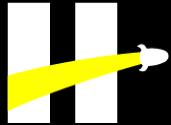
# Ventajas

- Permite manejar tanto errores de código sincrónico como asincrónico en un mismo lugar (try/catch)



```
1 const readFileAsync = async(archivo) => {
2 try {
3 console.log("Log async file: ", await promisifiedReadFile(archivo));
4 return "Lectura exitosa";
5 } catch (err) {
6 console.log("Error unificado: ", err);
7 }
8 }
```

<demoErrorHandler.js />



# Desventajas

El código parece sincrónico, pero en realidad se ejecuta de manera asíncrona.

Tener cuidado porque `async/await` nos **oculta la complejidad**.

# Async/Await = Generators + Promises

<demoAsyncAwaitWithGenerator.js />

# Henry

---



Hacé click acá para dejar tu feedback sobre esta clase.



Hacé click acá completar el quiz teórico de esta lecture.

## Generators

"Generator Functions are functions that can be exited and later re-entered. Their context (variable bindings) will be saved across re-entrances"

Los generators son funciones que a diferencia de las que conocíamos hasta el momento pueden detener su ejecución para luego retomar el control más adelante en el flujo del programa. Al volver a la función luego de una "pausa", el contexto sigue siendo el mismo, por lo que sus variables no se van a ver afectadas.

Podrían hacer la analogía de este tipo de funciones con un control remoto en el cual podemos pausar lo que estemos viendo y luego, cuando lo deseemos, volver a darle "play" o incluso dejar de ver lo que estábamos mirando (stop).

### Run-to-completion Model

Lo que usualmente conocemos como "Run-to-completion Model" es el paradigma sobre el cual se basan todas las funciones de JS, a excepción de los generators. La idea principal consiste en que toda función va a ejecutarse por completo hasta llegar a su última instrucción o punto de salida, ya sea por:

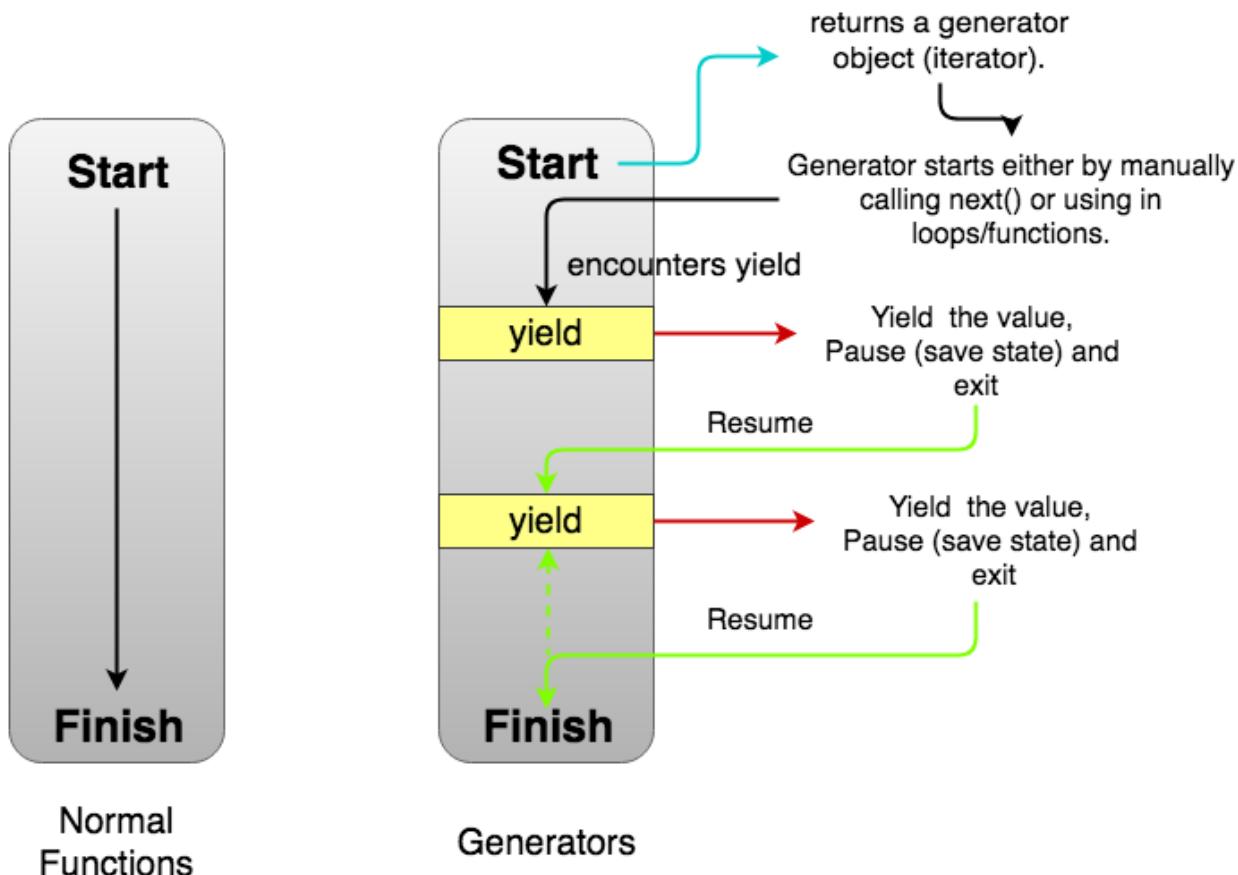
- Return statement
- Error thrown
- Fin de instrucciones (implicit undefined return)

### Generators Model

Por su parte el modelo en que se basan los generators previamente explicado consta de

- **Ejecución inicial:** retorna un "generator object" que va a ser nuestro iterador
- **Comienzo:** comienza a ejecutarse el generator cuando se invoca la instrucción `next` o a través de algún ciclo de iteración
- **"Pausas":** puede tener una cantidad infinita de puntos de pausa en los cuales se guarda su contexto y se cede nuevamente el control a la función o programa que invocó al generator
- **Resume:** luego de cada pausa es posible que el generator tome nuevamente el control para continuar con sus operaciones
- **Fin:** una vez completadas o pasados todos los puntos de pausa el generator llega a su fin

La siguiente imagen muestra en detalle las diferencias entre ambos modelos recién expuestas:



## Generators Syntax

```
function* generatorShowInstructors() {
 console.log("Iniciando generator function");
 yield "Franco";
 yield "Toni"
 console.log("Generator function terminada");
}

var generatorObject = generatorShowInstructors();

generatorObject.next();
```

En primer lugar para poder definir un generator debemos utilizar la sintaxis `function*` lo que va a indicar que la misma debe retornar un **Generator Object**.

Una vez definida dicha función podemos invocarla al igual que cualquiera de las funciones que ya conocíamos de antes. La particularidad que tienen los generators es que no se va a ejecutar el cuerpo de la función una vez que hagamos la invocación de la misma, en el ejemplo: `generatorShowInstructors()` sino que simplemente nos va a devolver como mencionamos antes un **Generator Object** que nos va a permitir luego "controlar" esta función.

Una vez obtenido nuestro iterador podemos invocar las veces que queramos al método `next()` que se va a encargar ahora si de ejecutar el cuerpo del generator previamente invocado hasta llegar a un "punto de pausa" que se van a identificar con la palabra reservada `yield`.

Para que quede más claro veamos como sería el flow de ejecución del ejemplo de arriba

```
var generatorObject = generatorShowInstructors();
// En este punto todavía no se ejecutó nada del cuerpo de la función
generatorShowInstructors
// Simplemente tenemos guardado en la variable generatorObject el justamente
Generator Object
// que nos va a servir como iterador y poder controlar el generator

var firstNext = generatorObject.next();
// La primera vez que ejecutamos el método next sobre el iterador vamos a ejecutar
las instrucciones
// de la función generatorShowInstructors hasta encontrar el primer yield

function* generatorShowInstructors() {
 console.log("Iniciando generator function"); // <-- Se ejecuta
 yield "Franco"; // <-- Se pausa la ejecución
 yield "Toni"
 console.log("Generator function terminada");
}

// Si observamos que quedó almacenado en firstNext obtendremos un objeto de la
siguiente forma:
{
 done: false,
 value: "Franco"
}
// Es decir tenemos información sobre el estado de nuestro generator.
// Por un lado la propiedad done nos indica que el generator aún no ha finalizado
// y por otro lado el value corresponde con el valor actual que tiene el generator
en
// este punto de pausa que se corresponde con el valor que se coloqué después de
la palabra
// yield

// Si volvemos a ejecutar next, ¿qué creen que sucederá?
var secondNext = generatorObject.next();

function* generatorShowInstructors() {
 console.log("Iniciando generator function");
 yield "Franco";
 yield "Toni" // <-- Avanza hasta acá y se pausa
 le ejecución
 console.log("Generator function terminada");
}

// Lo que sucede es que vuelve a tomar el control el generator y vuelve a avanzar
```

```
hasta el próximo
// punto de pausa o hasta su finalización (lo que ocurra primero)
// Por lo que si observamos ahora que contiene secondNext:
{
 done: false,
 value: "Toni"
}

// Otra vez :D
var thirdNext = generatorObject.next();

// Obtenemos:
{
 done: true,
 value: undefined
}

// Que nos está indicando que efectivamente el generator llegó a su fin ya que
done es igual a true
```

## Yield vs Return

Si bien a simple vista pueden parecer equivalentes tanto el `yield` como el `return`, no lo son:

- Yield: se encarga de establecer los "puntos de pausa" por lo que al llegar a un `yield` se pausa el generator y se retorna un objeto como el que vimos en el ejemplo de arriba que indica el estado actual del generator
- Return: una vez que se alcanza un `return` statement dentro de un generator, se finaliza su ejecución seteando el valor de "done" del objeto devuelto en `true`.

```
function* generatorUnreachableValue() {
 console.log("Iniciando generator function");
 yield "First reacheable value";
 yield "Second reacheable value";
 return "Return executed";
 yield "Unreacheable value"
}

var generatorObject = generatorUnreachableValue();

generatorObject.next(); // <-- {done: false, value: "First reacheable value"}
generatorObject.next(); // <-- {done: false, value: "Second reacheable value"}
generatorObject.next(); // <-- {done: true, value: "Return executed"}
generatorObject.next(); // <-- {done: true, value: undefined}

// Vamos a poder seguir ejecutando el método "next" sobre generatorObject pero
// como el generator
// ya finalizó vamos a seguir obteniendo siempre el mismo resultado: {done: true,
// value: undefined}
// En este ejemplo la instrucción `yield "Unreacheable value"` nunca va a ser
// ejecutada.
```

## Infinite Generator

Es posible también definir generators cuyo flow de ejecución sea infinito que no quiere decir que va a estar ejecutándose en background por tiempo indefinido sino que nosotros somos quienes tenemos el control y podríamos en caso de querer ejecutar el método `next` infinitas veces.

A continuación definiremos un generador de números naturales:

```
function* naturalNumbers() {
 let number = 1;
 while(true) {
 yield number;
 number = number + 1;
 }
}

var generatorObject = naturalNumbers();

generatorObject.next(); // <-- Retorna {done: false, value: 1}
generatorObject.next(); // <-- Retorna {done: false, value: 2}
generatorObject.next(); // <-- Retorna {done: false, value: 3}
generatorObject.next(); // <-- Retorna {done: false, value: 4}

// En value tendremos la secuencia de números naturales que queríamos
```

## Async/Await

Las funciones asíncronas o `async functions` nos van a permitir, como su nombre lo indica, definir código asíncrono con una sintaxis distinta a la que veníamos utilizando con las promesas por lo que no tendremos que encadenarlas nosotros mismos de forma explícita.

### Sintaxis

Para utilizar este tipo de funciones debemos definirlas con una sintaxis en particular:

```
async function asyncCall() {
 const result = await resolveAfter2Seconds();
}
```

La palabra `async` es la que va a informarle al intérprete que se trata de una `async function` y nos va a permitir hacer uso de la palabra reservada `await` en el cuerpo de dicha función. Como su nombre nos sugiere, lo que va a ocurrir cuando la ejecución del programa se tope con un `await` es que se detendrá la ejecución de esa función de forma momentánea hasta que la función o instrucciones que se encuentren a la derecha de dicha palabra finalicen.

### Basic Flow

A continuación analizaremos un pequeño ejemplo para comprender mejor el flow de estas nuevas funciones:

```
function resolveAfter2Seconds() {
 return new Promise(resolve => {
 setTimeout(() => {
 resolve('resolved');
 }, 2000);
 });
}

async function asyncCall() {
 console.log('calling'); // <-- Se ejecuta luego de la invocación de asyncCall()
 const result = await resolveAfter2Seconds(); // <-- Detiene la ejecución de
 asyncCall() hasta que finalice resolveAfter2Seconds()
 console.log(result); // <-- No se va a ejecutar hasta que la línea anterior
 finalice su ejecución
}

asyncCall()
```

## Return

Una particularidad de las `async` functions es que siempre retornan una promesa.

```
function resolveAfter2Seconds() {
 return new Promise(resolve => {
 setTimeout(() => {
 resolve('Promesa resuelta!');
 }, 2000);
 });
}

async function asyncCall() {
 console.log('Iniciando asyncCall');
 const result = await resolveAfter2Seconds();
 console.log(result); // <--- Va a loguear "Promesa resuelta!"
}

var p1 = asyncCall(); // p1 --> Va a ser una promesa la cual dependiendo el
momento en la cual la consultemos puede estar en estado pendiente o ya resulea si
asyncCall ya se terminó de ejecutar por completo
```

En el caso de que la `async` function no tenga un `return` statement, la promesa que devolverá tendrá un value `undefined`, por ejemplo el caso anterior almacena en `p1` lo siguiente:

```
Promise --> { state: "fulfilled", value: undefined }
```

En cambio si tuvieramos un return statement, por ejemplo algo así:

```
async function asyncCall() {
 console.log('Iniciando asyncCall');
 const result = await resolveAfter2Seconds();
 console.log(result);
 return "Franco";
}

var p2 = asyncCall();
```

Ahora p2 será una promesa cuyo valor de resolución será "Franco":

```
Promise --> { state: "fulfilled", value: "Franco" }
```

Como ya sabíamos de cuando estudiamos promesas, las mismas pueden resolverse o rechazarse por lo que en caso de `success` como vimos en el ejemplo de arriba, la promesa retornada se va a resolver al valor devuelto por la función asíncrona y en el caso de `error`, la promesa retornada se va a rechazar con la excepción lanzada por la función asíncrona.

Para más ejemplos que incluyen todos los distintos casos posibles ver la demo [demoRetunValue.js](#)

## Yielding Control

Algo importante de comprender es como continua el flow del resto del programa/aplicación una vez que se encuentra con un `await` dentro de una `async function`. Como ya hemos mencionado antes, lo que sucede es que en ese instante se le retorna el control a la función o punto del programa desde donde se había invocado a la `async function` y va a continuar su flow normalmente.

Para terminar de comprenderlo analicemos el siguiente ejemplo:

```
async function showInstructors() {
 const instructor1 = await new Promise((resolve) => setTimeout(() =>
 resolve('Franco')));
 console.log(instructor1);
 const instructor2 = await new Promise((resolve) => setTimeout(() =>
 resolve('Toni')));
 console.log(instructor2);
}

function henryAwait() {
 console.log("¿Quienes son los intstructores de Henry?");
 showInstructors();
 console.log("Gracias vuelvan pronto");
}
```

```
henryAwait()
console.log("FIN");
```

¿Cuál será el orden de ejecución de el código de arriba?

Luego de definir tanto ambas funciones, una asíncrona y la otra no, se ejecuta henryAwait que no es asíncrona. Hasta ahí todo normal:

```
function henryAwait() {
 console.log("¿Quienes son los intstructores de Henry?"); // <-- Loguea "¿Quienes
 son los intstructores de Henry?"
 showInstructors(); // <-- Invoca a showInstructors (Cede el control)
 console.log("Gracias vuelvan pronto"); // <-- Aún no se invocó...
}
```

Veamos ahora que sucede al ingresar a showInstructors:

```
async function showInstructors() {
 const instructor1 = await new Promise((resolve) => setTimeout(() =>
 resolve('Franco'))); // <-- Pausa la ejecución y retorna el control a quien se lo
 había cedido (henryAwait) hasta completar la promesa y poder avanzar con las
 siguientes instrucciones
 console.log(instructor1);
 const instructor2 = await new Promise((resolve) => setTimeout(() =>
 resolve('Toni')));
 console.log(instructor2);
}
```

Como ahora nuevamente el control lo tiene henryAwait, continua con su flow normal:

```
function henryAwait() {
 console.log("¿Quienes son los intstructores de Henry?");
 showInstructors(); // <-- Se había quedado acá, ahora avanza a la siguiente...
 console.log("Gracias vuelvan pronto"); // <-- Loguea "Gracias vuelvan pronto"
}
```

Y ahora como henryAwait finalizo continua hacia:

```
henryAwait() // <-- Ya finalizó, avanza...
console.log("FIN"); // <-- Logua "FIN"
```

Luego de todo esto recién ahí, y si la promesa donde se había pausado showInstructors ya finalizó, vuelve a tomar el control y continua con las sentencias que quedaron sin ejecutarse:

```
async function showInstructors() {
 const instructor1 = await new Promise((resolve) => setTimeout(() =>
 resolve('Franco')));
 console.log(instructor1); // <-- Loguea "Franco"
 const instructor2 = await new Promise((resolve) => setTimeout(() =>
 resolve('Toni'))); // <-- Pausa hasta finalizar la promesa
 console.log(instructor2); // <-- Una vez finalizada loguea "Toni"
}
```

Si quisieramos que el orden de ejecución seá:

1. ¿Quienes son los intstructores de Henry?
2. Franco
3. Toni
4. Gracias vuelvan pronto
5. FIN

¿Cómo deberíamos modificar el código previo?



```
async function showInstructors() {
 const instructor1 = await new Promise((resolve) => setTimeout(() =>
 resolve('Franco')));
 console.log(instructor1);
 const instructor2 = await new Promise((resolve) => setTimeout(() =>
 resolve('Toni')));
 console.log(instructor2);
}

async function henryAwait() {
 console.log("¿Quienes son los intstructores de Henry?");
 await showInstructors();
 console.log("Gracias vuelvan pronto");
}

await henryAwait()
console.log("FIN");
```

## Ventajas

- El código suele ser más prolíjo y similar a código sincrónico:

```
const readFilePromise = (archivo) => {
 promisifiedReadFile(archivo)
 .then(file => {
 console.log("Log promise file: ", file);
 return "Lectura exitosa";
 });
}

const readFileAsync = async(archivo) => {
 console.log("Log async file: ", await promisifiedReadFile(archivo));
 return "Lectura exitosa";
}
```

Para más detalle ver la demo [demoCleanCode.js](#)

- Permite manejar tanto errores de código sincrónico como asíncrono en un mismo lugar (try/catch)

```
const readFileAsync = async(archivo) => {
 try {
 console.log("Log async file: ", await promisifiedReadFile(archivo));
 return "Lectura exitosa";
 } catch (err) {
 console.log("Error unificado: ", err);
 }
}
```

Para más detalle ver la demo [demoErrorHandler.js](#)

## Desventajas

- El código suele ser más prolíjo y similar a código sincrónico. ¿¿Qué??? ¿No les habíamos dicho hace un par de líneas que era una ventaja esto?

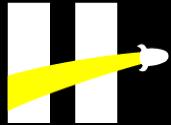
Si, no estamos locos, esto puede ser un arma de doble filo, porque al maquillar código asíncrono haciéndolo parecer como sincrónico muchas veces solemos utilizarlo de forma incorrecta y terminando sin entender el flow del programa. Recuerden el ejemplo que hicimos más arriba de showInstructors y verán que si no se comprende bien como funcionan `async` y `await` puede llevar a grandes confusiones.

Por eso mismo, ustedes que ya entienden a la perfección el funcionamiento de promesas si comprender en el fondo que es lo que está ocurriendo y no pensar que es simplemente 'magia'.

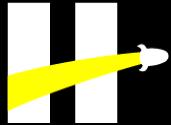
Por último podríamos pensar que Async/Await toma y combinó las ideas de Generators junto con Promesas:

$$\text{Async/Await} = \text{Generators} + \text{Promises}$$

HENRY



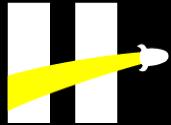
# DBMS



# DBMS

**Datos:** Información concreta sobre hechos, elementos, etc., que permite estudiarlos, analizarlos o conocerlos.

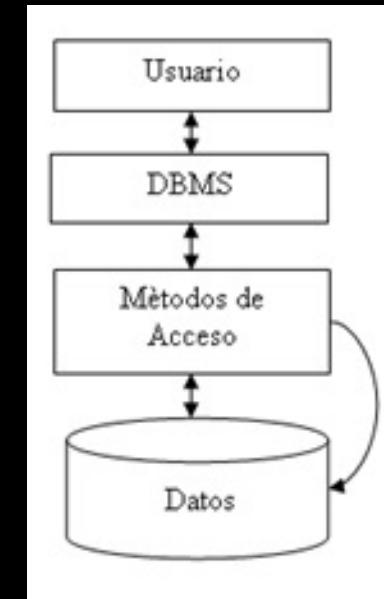
**Base de Datos:** Una base de datos es un conjunto de datos pertenecientes a un mismo contexto y almacenados sistemáticamente para su posterior uso.



# DBMS

DBMS ( DataBase Management System):

son una colección de programas que permiten al usuario acceder a una BD, manipular data, y hacer consultas.

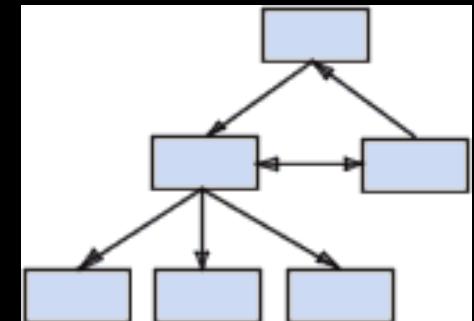


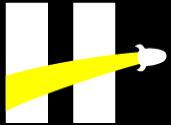


# Pre - SQL

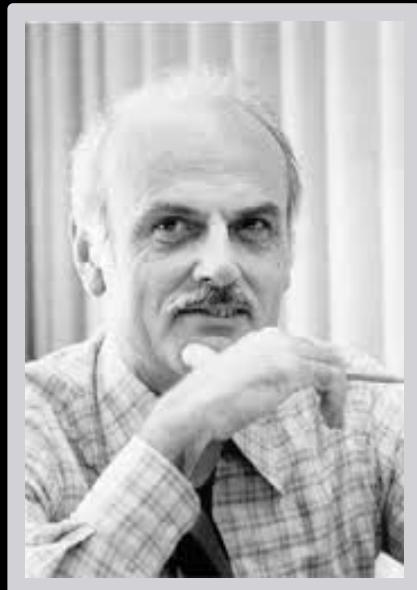
## CODASYL (1960s):

La estrategia de CODASYL estaba basada en la navegación manual por un conjunto de datos enlazados en red. Cuando se arrancaba la base de datos, el programa devolvía un enlace al primer registro de la base de datos, el cual a su vez contenía punteros a otros datos. Para encontrar un registro concreto el programador debía ir siguiendo punteros hasta llegar al registro buscado.





# Pre - SQL



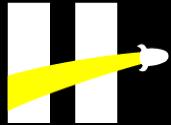
Edgar Frank Codd,

laboratorios IBM en San

José (California)

## Modelo Relacional (1970s):

El modelo relacional, para el modelado y la gestión de bases de datos, es un modelo de datos basado en la lógica de predicados y en la teoría de conjuntos.

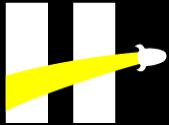


## Pre-SQL

### Prototipos (1970s):

Se crearon dos prototipos de DBSM basados en el modelo de Codd. **Ingres** creado en UBC, y **System R** de IBM. Ingres usaba un lenguaje de consultas llamado QUEL, y System R uno llamado SEQUEL.





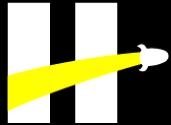
## Pre - SQL



Peter Chen

### Modelo E-R 1976:

Aparece un nuevo modelo de bases de datos llamado modelo de entidad-relación de P. Chen. Este modelo logró abstraer cómo se guardan los datos, y se comenzó a hablar de qué se guarda.



# SQL

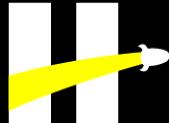
## 1980s

SQL (Structured Query Language) se convierte en el lenguaje estándar para hacer consultas a las BDs.

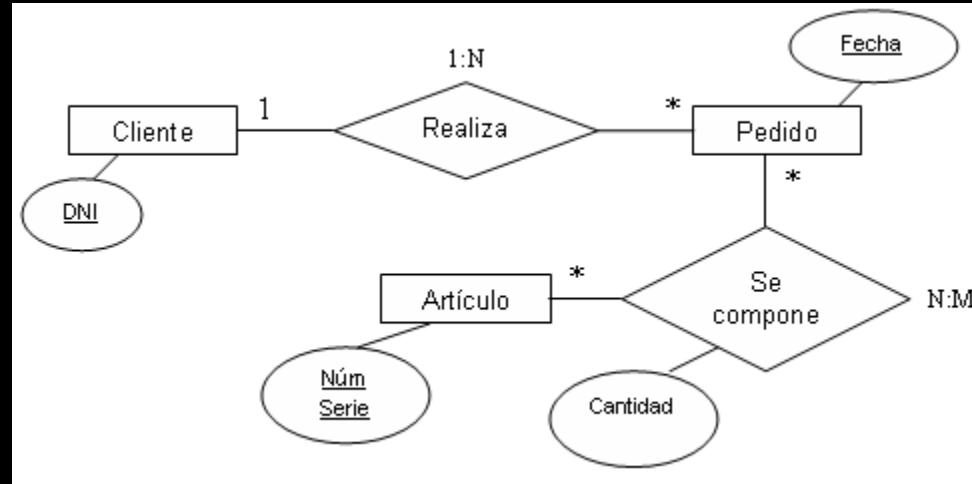


# Modelo Entidad Relacion

- **Entidad:** Representa una “cosa”, "objeto" o "concepto" del mundo real con existencia independiente
- **Atributos:** Los atributos son las características que definen o identifican a una entidad
- Conjunto de relaciones: Una **relación** es una asociación entre varias Entidades. Cada entidad interviene en una relación con una determinada cardinalidad.

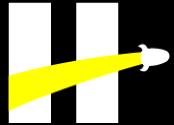


# Modelo Entidad Relacion

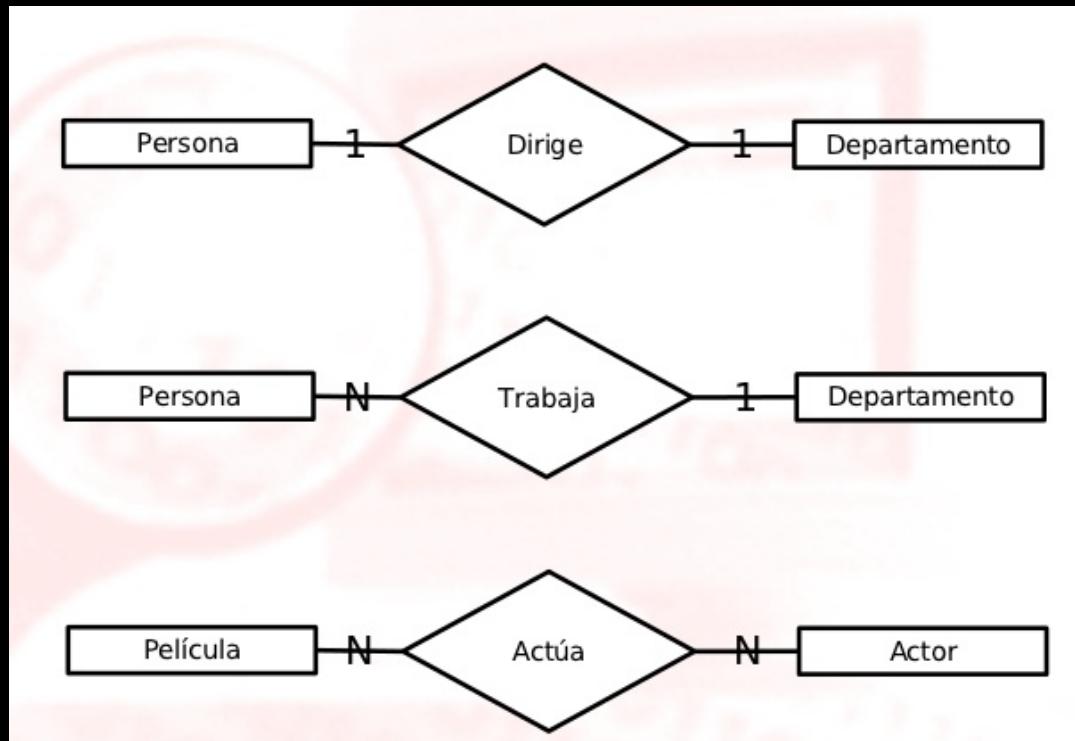


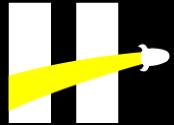
Hay standares para diagramar los modelos de entidad relacion, como por ejemplo: UML.

Pero no es necesario seguir los estandares, para hacerlo bien.

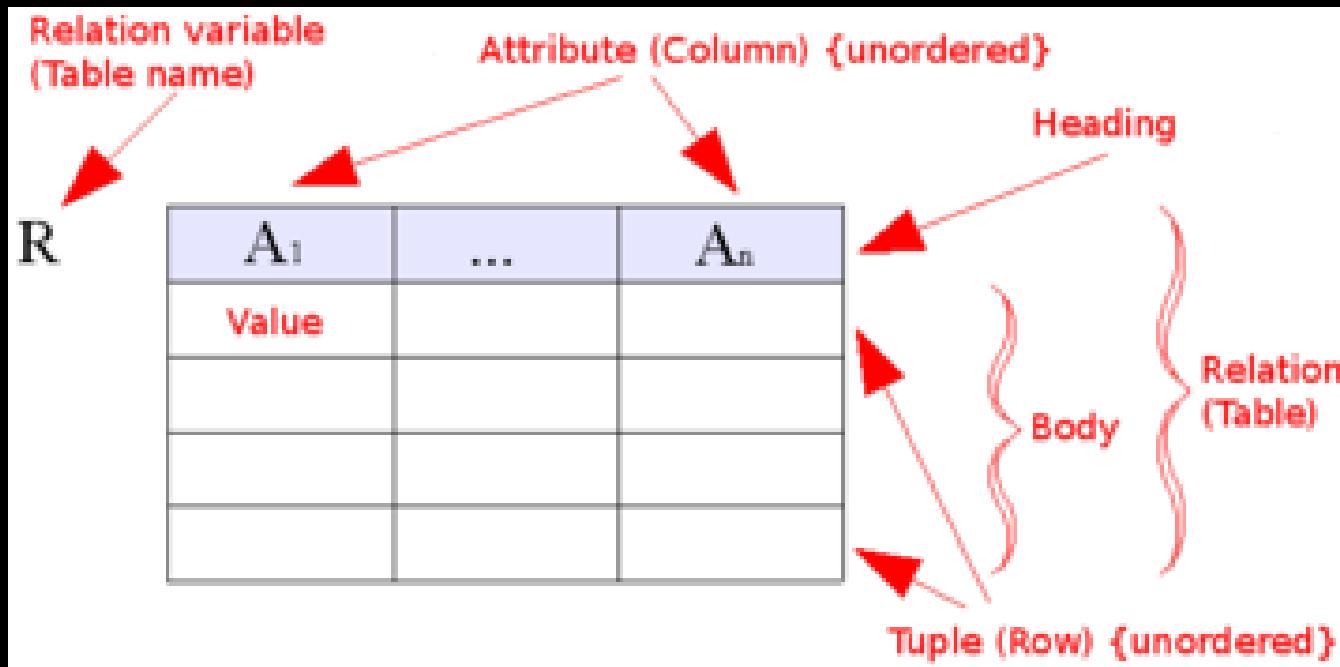


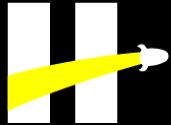
# Cardinalidad





# Modelo Relacional





# Modelo Relacional Normalización





# Modelo Relacional

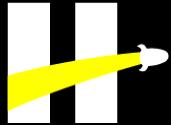
## Normalización

| TITULO              | AUTOR | NACIONALIDAD<br>AUTOR | FORMATO | TEMAS                                  |
|---------------------|-------|-----------------------|---------|----------------------------------------|
| Fullstack Developer | Toni  | Argentino             | Remoto  | Bases de Datos,<br>Node,<br>JavaScript |

Primera forma Normal (1NF):

Todos los atributos son atómicos. Un atributo es atómico si los elementos del dominio son simples e indivisibles.

Y no hay grupos de repetición.



# Modelo Relacional

## Normalización

| TITULO              | AUTOR | NACIONALIDAD AUTOR | FORMATO | TEMA1          | TEMA2 | TEMA3      |
|---------------------|-------|--------------------|---------|----------------|-------|------------|
| Fullstack Developer | Toni  | Argentino          | Remoto  | Bases de Datos | Node, | JavaScript |

Grupo de repetición

Primera forma Normal (1NF):

Todos los atributos son atómicos. Un atributo es atómico si los elementos del dominio son simples e indivisibles.

Y los grupos de repetición?



# Modelo Relacional Normalización

| TITULO              | AUTOR | NACIONALIDAD AUTOR | FORMATO | TEMA           |
|---------------------|-------|--------------------|---------|----------------|
| Fullstack Developer | Toni  | Argentino          | Remoto  | Bases de Datos |
| Fullstack Developer | Toni  | Argentino          | Remoto  | Node           |
| FullStack Developer | Toni  | Argentino          | Remoto  | JavaScript     |

Primera forma Normal (1NF):

De la forma anterior, no podríamos tener cursos que tengan más de tres temas sin cambiar la estructura de la tabla.



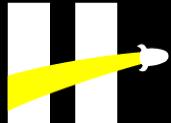
# Modelo Relacional Normalización

| TITULO              | AUTOR | NACIONALIDAD AUTOR | FORMATO | TEMA           |
|---------------------|-------|--------------------|---------|----------------|
| Fullstack Developer | Toni  | Argentino          | Remoto  | Bases de Datos |
| Fullstack Developer | Toni  | Argentino          | Remoto  | Node           |
| FullStack Developer | Toni  | Argentino          | Remoto  | JavaScript     |

Segunda forma Normal (2NF):

Cumple con la 1NF, y además no hay dependencia parcial entre los atributos.

En este ejemplo, el autor, la nacionalidad del autor y el formato dependen del curso, y NO  
del curso y del tema!

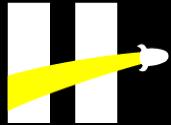


# Modelo Relacional

## Normalización

| TITULO              | AUTOR | NACIONALIDAD AUTOR | FORMATO | TEMA           |
|---------------------|-------|--------------------|---------|----------------|
| Fullstack Developer | Toni  | Argentino          | Remoto  | Bases de Datos |
| Fullstack Developer | Toni  | Argentino          | Remoto  | Node           |
| FullStack Developer | Toni  | Argentino          | Remoto  | JavaScript     |

Primary Key (PK): Es un atributo o conjunto de atributos que identifica únicamente a una fila.



# Modelo Relacional Normalización

| TITULO              | AUTOR | FORMATO | TEMA           |
|---------------------|-------|---------|----------------|
| Fullstack Developer | Toni  | Remoto  | Bases de Datos |
| Fullstack Developer | Toni  | Remoto  | Node           |
| FullStack Developer | Toni  | Remoto  | JavaScript     |

| AUTOR | NACIONALIDAD AUTOR |
|-------|--------------------|
| Toni  | Argentino          |

Segunda forma Normal (2NF):

Cumple con la 1NF, y además no hay dependencia parcial entre los atributos.



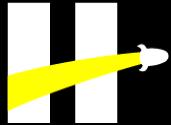
# Modelo Relacional Normalización

| TITULO              | AUTORID | FORMATO | TEMA           |
|---------------------|---------|---------|----------------|
| Fullstack Developer | 1       | Remoto  | Bases de Datos |
| Fullstack Developer | 1       | Remoto  | Node           |
| FullStack Developer | 1       | Remoto  | JavaScript     |

| ID | AUTOR | NACIONALIDAD<br>AUTOR |
|----|-------|-----------------------|
| 1  | Toni  | Argentino             |

Segunda forma Normal (2NF):

Cumple con la 1NF, y además no hay dependencia parcial entre los atributos.



# Modelo Relacional Normalización

TemasXCurso

| CursolD | TEMA           |
|---------|----------------|
| 1       | Bases de Datos |
| 1       | Node           |
| 1       | JavaScript     |

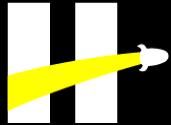
Autores

| ID | AUTOR | NACIONALIDA<br>D AUTOR |
|----|-------|------------------------|
| 1  | Toni  | Argentino              |

Cursos

| ID | TITULO              | AutorID | FORMATO |
|----|---------------------|---------|---------|
| 1  | Fullstack Developer | 1       | REMOTO  |

Foreign Key (FK): Es un atributo que identifica una fila en otra tabla.



# Modelo Relacional Normalización

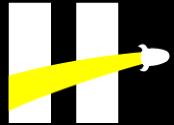
TemasXCurso

| CursolD | TEMA           |
|---------|----------------|
| 1       | Bases de Datos |
| 1       | Node           |
| 1       | JavaScript     |

Autores

| ID | AUTOR | NACIONALIDAD<br>AUTOR | CP   | Provincia/Estado | Ciudad |
|----|-------|-----------------------|------|------------------|--------|
| 1  | Toni  | Argentino             | 1425 | Buenos Aires     | Caba   |

La tabla se encuentra en 3FN si es 2FN y si no existe ninguna dependencia funcional transitiva en los atributos que no son clave.



# Modelo Relacional Normalización

## TemasXCurso

| CursolD | TEMA           |
|---------|----------------|
| 1       | Bases de Datos |
| 1       | Node           |
| 1       | JavaScript     |

## Autores

| ID | AUTOR | NACIONALIDA<br>D AUTOR | CP   |
|----|-------|------------------------|------|
| 1  | Toni  | Argentino              | 1425 |

| CP   | Provincia/Estado | Ciudad |
|------|------------------|--------|
| 1425 | Buenos Aires     | Caba   |

# Henry



Hacé click acá para dejar tu feedback sobre esta clase.



Hacé click acá completar el quiz teórico de esta lecture.

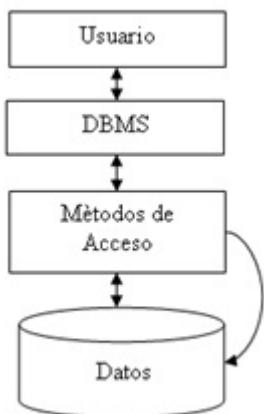
Henry

## SQL

### Necesidad de una base datos

En algún momento nuestra aplicación va a necesitar algún tipo de *persistencia* de datos, es decir que los datos queden guardados en el disco, y no importa si reinicio el servidor, o modiflico mi aplicación u otras cosas que puedan ocurrir. Ya aprendimos que con `nodejs` es fácil leer y escribir archivos. Así que una forma de lograr persistencia sería escribir en archivos de texto en el disco, no? sí! pero eso **no escala**. Si lo hicieramos, nos veríamos enredados en buscar lo que queríamos dentro de cada archivo, y en poco tiempo buscar algo tardaría mucho, y nos dejaría de servir. Obviamente, si sólo tenemos que guardar cosas para luego leerlas después, este sistema nos puede servir, como por ejemplo: los logs de las páginas. Cada entrada es guardada como una línea en un archivo de texto. Otra solución, más escalable para lograr persistencia de datos es usar una *base de datos*.

- **Base de datos:** Es una colección de datos de un mismo dominio y organizada sistemáticamente para su posterior uso. Esta organización, en general, está construida de tal manera que *modele* el problema de la mejor forma.
- **DBMS (Database Management System):** es una aplicación que interactúa con el usuario, otras aplicaciones y la base de datos misma, de tal forma que pueda definir, crear, borrar, modificar, consultar y administrar bases de datos y datos en sí.



Lo que vamos a hacer entonces, es usar un *DBMS* para que nos ayude a guardar los datos. Y como todo el mundo utiliza estas aplicaciones, ya hay escritas muchas librerías de `nodejs` para que nos sirven como interfaz y que son fáciles de usar. Como se imaginan hay muchos sabores de BDMS para elegir. Lo primero es elegir si

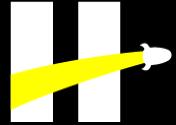
queremos uno que sea relacional (SQL) o uno no relacional (noSQL). Ahora vamos a empezar a ver como guardar datos en una base de datos no relacional, en particular [MongoDB](#).

## Bases de Datos Relacionales

Como habíamos visto, la alternativa a las bases de datos [NoSQL](#) son las bases de datos relacionales. En estas bases de datos las tablas tienen (no es obligatorio pero fuertemente recomendado) que estar normalizadas (3era forma normal). Y antes de empezar a cargar datos, tenemos que definir el modelo de datos de manera detallada, como en mongoose pero obligatoriamente!

Las ventajas de usar una base de datos SQL son:

- Cómo nos obliga a definir un modelo de antemano, la aplicación va a ser muy estable y difícilmente llegué un dato no deseado a la BD. El problema es que es poco flexible y hacer cambios una vez arrancado el proyecto puede ser muy costoso.
- Estas bases de datos son transaccionales, es decir que el motor de DB nos asegura que las operaciones que hagamos van a hacerse atómicamente, es decir que jamás vamos a tener datos corruptos.
- Es una tecnología muy estudiada, hace años que ya está estable, en contrapartida con las bases de datos NoSQL que son relativamente nuevas.



# SQL

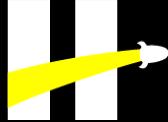


# SQL



# SQL

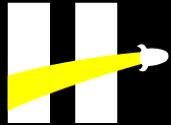
(Structured Query Language) es el lenguaje más utilizado para hacer consultas a una Base de Datos.



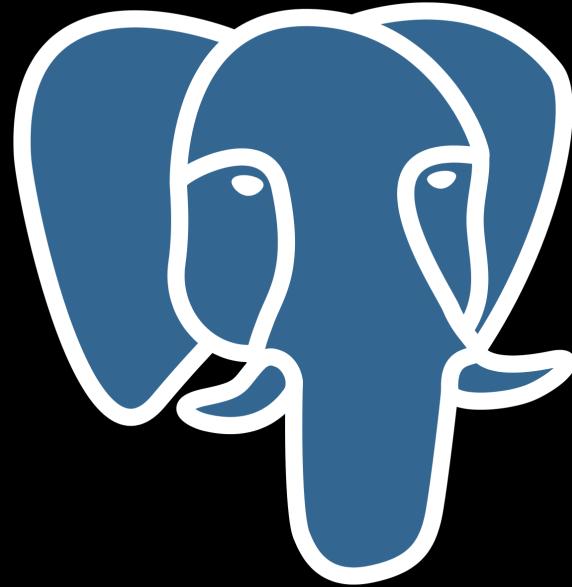
# SQL

```
UPDATE clause {UPDATE country
SET clause {SET population = $\overbrace{\text{population} + 1}$
WHERE clause {WHERE $\overbrace{\text{name} = \text{'USA'}}$;}}}} statement
```

SQL es un lenguaje de consultas, tiene expresiones y statements igual que JS.  
Está compuesto por Cláusulas.



# PostgreSQL



Existe muchos motores de SQL. Algunos gratis y otros pagos.  
PostgresSQL es un proyecto Open Source, que está siempre a  
la vanguardia.

Instalar PostgreSQL



# SQL



```
1 // Cada Statement SQL termina en ;
2
3 CREATE DATABASE prueba;
4
5
6 // Con Create vamos a poder crear Bases de Datos, Tablas, índices, etc..
7
8 CREATE TABLE table_name
9 (
10 column_name1 data_type(size),
11 column_name2 data_type(size),
12 column_name3 data_type(size),
13
14);
```



# SQL

```
1 CREATE TABLE ciudades
2 (
3 id serial PRIMARY KEY,
4 nombre varchar(255) UNIQUE
5);
6
7 CREATE TABLE personas
8 (
9 id serial PRIMARY KEY,
10 apellido varchar(255) NOT NULL,
11 nombre varchar(255) UNIQUE,
12 ciudad integer REFERENCES ciudades (id)
13);
```

En SQL los tipos de datos están bien definidos, veamos una lista [acá](#).



# SQL



```
1 CREATE TABLE ciudades
2 (
3 id serial PRIMARY KEY,
4 nombre varchar(255) UNIQUE
5);
6
7 CREATE TABLE personas
8 (
9 id serial PRIMARY KEY,
10 apellido varchar(255) NOT NULL,
11 nombre varchar(255) UNIQUE,
12 ciudad integer REFERENCES ciudades (id)
13);
```

Además del tipo de datos, podemos definir **Constraints**. Que son reglas forzadas sobre un atributo.

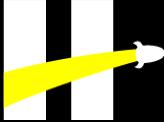
- **NOT NULL Constraint** – El atributo no puede ser nulo.
- **UNIQUE Constraint** – No puede haber dos atributos iguales en esta tabla.
- **PRIMARY Key** – Identifica únicamente una fila en una tabla.
- **FOREIGN Key** – La key debería existir sí o sí en otra tabla.



# SQL



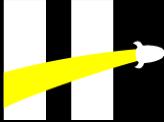
```
1 // Insertar filas en una tabla
2
3 INSERT INTO table_name (column1,column2,column3,...)
4 VALUES (value1,value2,value3,...);
5
6
7 // Ejemplo
8
9 INSERT INTO ciudades (nombre)
10 VALUES ('Tucuman');
11
12 INSERT INTO ciudades (nombre)
13 VALUES ('Buenos Aires');
14
15 INSERT INTO ciudades (nombre)
16 VALUES ('New York');
17
18 INSERT INTO personas (nombre, apellido, ciudad)
19 VALUES ('Toni', 'Tralice', 1);
20
21 INSERT INTO personas (nombre, apellido, ciudad)
22 VALUES ('Emi', 'Chequer', 3);
23
24 INSERT INTO personas (nombre, apellido, ciudad)
25 VALUES ('Fran', 'Etcheverri', 2);
```



# SQL

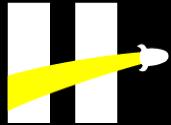


```
1 // El Select nos permite recuperar filas de una tabla.
2
3 SELECT column_name,column_name
4 FROM table_name;
5
6 // Ejemplo
7
8 SELECT * FROM personas;
9
10 SELECT * FROM ciudades;
11
12 // el * indica que traiga todas las columnas disponibles
13
14
15 // ORDER BY CLAUSE
16
17 SELECT * FROM ciudades
18 ORDER BY nombre, id DESC;
19
20 // WHERE CLAUSE
21
22 SELECT * FROM personas
23 WHERE nombre = 'Toni'
24 AND apellido = 'Tralice';
```



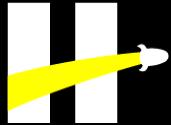
# SQL

```
1 // GROUP BY CLAUSE
2
3 # select * from COMPANY;
4 id | name | age | address | salary
5 ----+-----+-----+-----+-----
6 1 | Paul | 32 | California | 20000
7 2 | Allen | 25 | Texas | 15000
8 3 | Teddy | 23 | Norway | 20000
9 4 | Mark | 25 | Rich-Mond | 65000
10 5 | David | 27 | Texas | 85000
11 6 | Kim | 22 | South-Hall | 45000
12 7 | James | 24 | Houston | 10000
13
14
15 SELECT NAME, SUM(SALARY) FROM COMPANY GROUP BY NAME;
16
17 name | sum
18 -----
19 Teddy | 20000
20 Paul | 20000
21 Mark | 65000
22 David | 85000
23 Allen | 15000
24 Kim | 45000
25 James | 10000
```



# SQL

```
1 // SUB-QUERIES
2
3 SELECT column_name [, column_name]
4 FROM table1 [, table2]
5 WHERE column_name OPERATOR
6 (SELECT column_name [, column_name]
7 FROM table1 [, table2]
8 [WHERE])
```

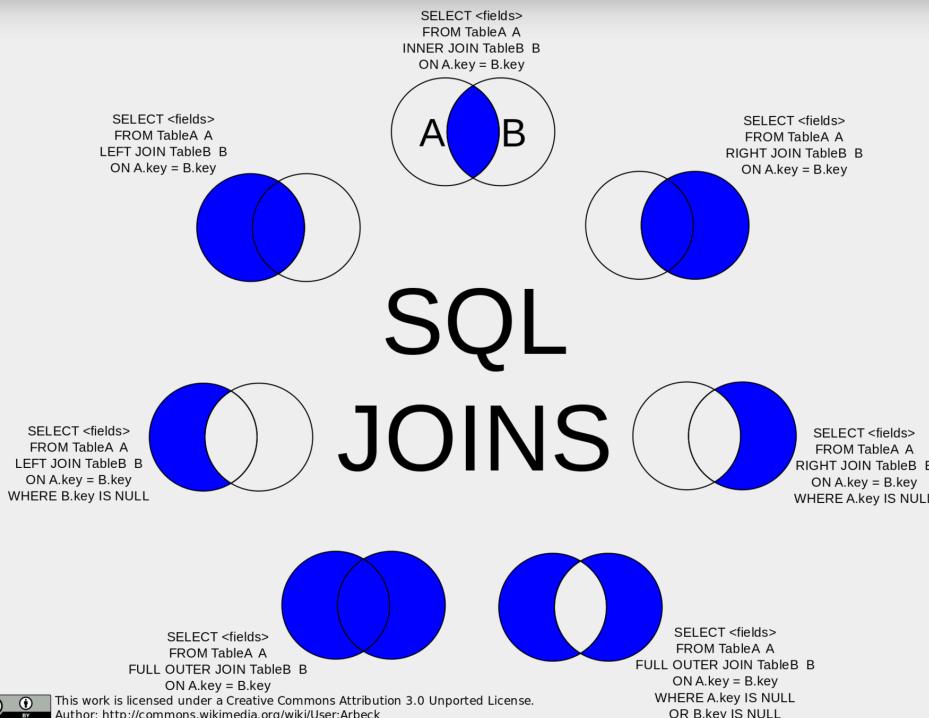


# SQL



# SQL

```
1 // EJoin nos permite unir datos de diferentes tablas según una condición.
2
3 SELECT * FROM personas
4 JOIN ciudades
5 ON ciudades.id = personas.ciudad;
```



# Henry

---



Hacé click acá para dejar tu feedback sobre esta clase.

Hacé click acá completar el quiz teórico de esta lecture.

# Henry

---

## SQL - Postgres

Como en todo en este mundo, hay muchas opciones de bases de datos SQL. De hecho las hay pagas y gratis. Podríamos usar: MySQL, ORACLE, IBM DB2, SQL server, access, etc. Todas utilizan el lenguaje SQL, así que son muy parecidas, el 80% de las cosas se puede hacer con cualquier motor. Nosotros vamos a ver en nuestro ejemplo **PostgreSQL**, que es una motor gratis de código abierto que tiene una comunidad muy activa. De hecho, postgres logra sacar funcionalidades antes que los motores pagos!



Para instalar Postgre sigan las instrucciones para su sistema operativo [acá](#).

### SQL

Como dijimos, vamos a interactuar con la base de datos a través de SQL. Este es un lenguaje especialmente diseñado para hacer consultas a las bases de datos relacionales. SQL es el acrónimo de Structured Query Language y es un standart mantenido por el ANSI (American National Standards Institute). Usando SQL vamos a poder crear tablas, buscar datos, insertar filas, borrarlas, etc..

### Creando una BD y una tabla

Lo primero que tenemos que hacer es crear una base de dato (Un motor de SQL puede manejar muchas Bases de Datos). Para eso vamos a usar el Statement **Create database**.

```
CREATE DATABASE prueba;
```

Cada Statement SQL termina en ; (punto y coma). En algunas interfaces es obligatorio.

Una vez ejecutado el comando vamos a ver listada la nueva base de datos, yo estoy usando la interfaz CLI de postgres, también pueden usar alguna interfaz gráfica.

Ahora vamos a crear una tabla. Usamos el statement **CREATE TABLE** que tiene la siguiente forma:

```
CREATE TABLE table_name
(
 column_name1 data_type(size),
 column_name2 data_type(size),
 column_name3 data_type(size),
 ...
);
```

Básicamente ponemos el nombre de la columna y luego el tipo de datos de esa columna. Podemos ver algunos tipos de datos comunes [aquí](#). También vamos a poder agregar **CONSTRAINTS** o restricciones por ejemplo:

```
CREATE TABLE ciudades
(
 id serial PRIMARY KEY,
 nombre varchar(255) UNIQUE
);

CREATE TABLE personas
(
 id serial PRIMARY KEY,
 apellido varchar(255) NOT NULL,
 nombre varchar(255) UNIQUE,
 ciudad integer references ciudades (id)
);
```

Con este statement hemos creado dos tablas. La primera es ciudad, que cuenta con ID que tiene la constraint que es PRIMARY KEY, es decir que tiene que ser única y no puede ser nula, y ademas tiene una columna nombre que es de tipo texto (varchar) y tiene la constraint UNIQUE, por lo tanto no puede haber dos iguales en la misma tabla.

La segunda es la tabla personas, vemos que tambien tenemos un id que es PRIMARY KEY (esta práctica es muy común y muy recomendable), tenemos nombre y apellido, con la condición que nombre no se repita (es sólo para el ejemplo) y pusimos una columna que se llama ciudad, que hace referencia a un ID de la tabla *ciudades*. Esto último denota una *relación* entre las tablas, y lo hicimos de esta forma para respetar la normalización y de esa forma reducir el tamaño final de la base de datos. Más adelante veremos como hacer queries y obtener los datos de una relación.

Ahora agregamos algunos datos en las tablas. Tenemos que empezar por ciudades, ya que para cargar una persona luego (y mantener la **integridad referencial**) vamos a tener que tener algunas ciudades y sus ids. Para insertar datos vamos a usar el statement **INSERT INTO**:

```
INSERT INTO table_name (column1,column2,column3,...)
VALUES (value1,value2,value3,...);
```

Insertemos tres ciudades:

```
INSERT INTO ciudades (nombre)
VALUES ('Tucuman');
```

```
INSERT INTO ciudades (nombre)
VALUES ('Buenos Aires');
```

```
INSERT INTO ciudades (nombre)
VALUES ('New York');
```

El tipo de datos SERIAL (el id) es un entero AUTOINCREMENTAL, es decir que no tengo que especificar el ID, si no que se va generando solo. El primero es 1, el segundo 2 y así sucesivamente.

Perfecto ahora tenemos ciudades! Insertemos algunas personas que sean de esas ciudades!

```
INSERT INTO personas (nombre, apellido, ciudad)
VALUES ('Toni', 'Tralice', 1);
```

```
INSERT INTO personas (nombre, apellido, ciudad)
VALUES ('Emi', 'Chequer', 3);
```

```
INSERT INTO personas (nombre, apellido, ciudad)
VALUES ('Fran', 'Etcheverri', 2);
```

😊 Ahora tenemos tablas con datos, pero cómo los consultamos?

## SELECT, WHERE, ORDER BY

Para recuperar datos usamos el statement **SELECT** de esta forma:

```
SELECT column_name, column_name
FROM table_name;
```

Para recuperar todas las personas:

```
SELECT * FROM personas;
```

y todas las ciudades:

```
SELECT * FROM ciudades;
```

El asterisco quiere decir 'todas las columnas'.

Genial, esto sería equivalente al `db.prueba.find()` de mongo! Las filas en SQL no tiene un orden dado, ni siquiere por el orden en el que fueron creadas (muchas veces coincide pero no es necesariamente así), así que si queremos tener los resultados ordenados vamos a usar la cláusula `ORDER BY`, esta va al final del query y le especificamos en qué columna se tiene que fijar para ordenar:

```
SELECT * FROM ciudades
ORDER BY nombre;
```

El motor se da cuenta el tipo de datos de la columna y los ordena en base a eso. también podemos especificar que ordene por más de un campo, y si queremos que sea en orden ascendente o descendiente:

```
SELECT * FROM ciudades
ORDER BY nombre, id DESC;
```

Ahora veamos como buscar o filtrar filas: para eso vamos a usar la cláusula `WHERE`:

```
SELECT column_name,column_name
FROM table_name
WHERE column_name operator value;
```

Por ejemplo, busquemos todas las personas que se llaman 'Toni':

```
SELECT * FROM personas
WHERE nombre = 'Toni';
```

Casi que podemos leerlo en lenguaje natural: 'Seleccioná todas las columnas de la tabla personas donde el nombre sea Toni'. 😊

Podemos agregar más de una condición:

```
SELECT * FROM personas
WHERE nombre = 'Toni'
AND apellido = 'Tralice';
```

Ahora, si vemos el output de consultar la tabla 'personas', vemos que tenemos el *código* de ciudad, pero no el nombre. Y probablemente en nuestra aplicación querramos mostrar el nombre y no el código, no? Bien, entonces para hacerlo podríamos consultar ambas tablas, y luego buscar el código de cada fila de *personas* en la tabla *ciudades*. De esa forma tendríamos el nombre asociado... por suerte SQL ya viene preparado para eso! con el statement `JOIN`.

## JOINS

La cláusula **JOIN** nos sirve para combinar filas de una tabla con otras de otra tabla, basándonos en un campo que tengan en común. Es el caso de ciudad y personas, vamos a unir las filas de cada tabla basados en el ID de la ciudad.

Para definir un JOIN tenemos que decir qué tablas queremos unir y en base a qué campos, para lo primero ponemos el nombre de la tabla a unir después del **JOIN** y lo segundo lo hacemos con el parámetro **ON**:

```
SELECT * FROM personas
JOIN ciudades
ON ciudades.id = personas.ciudad;
```

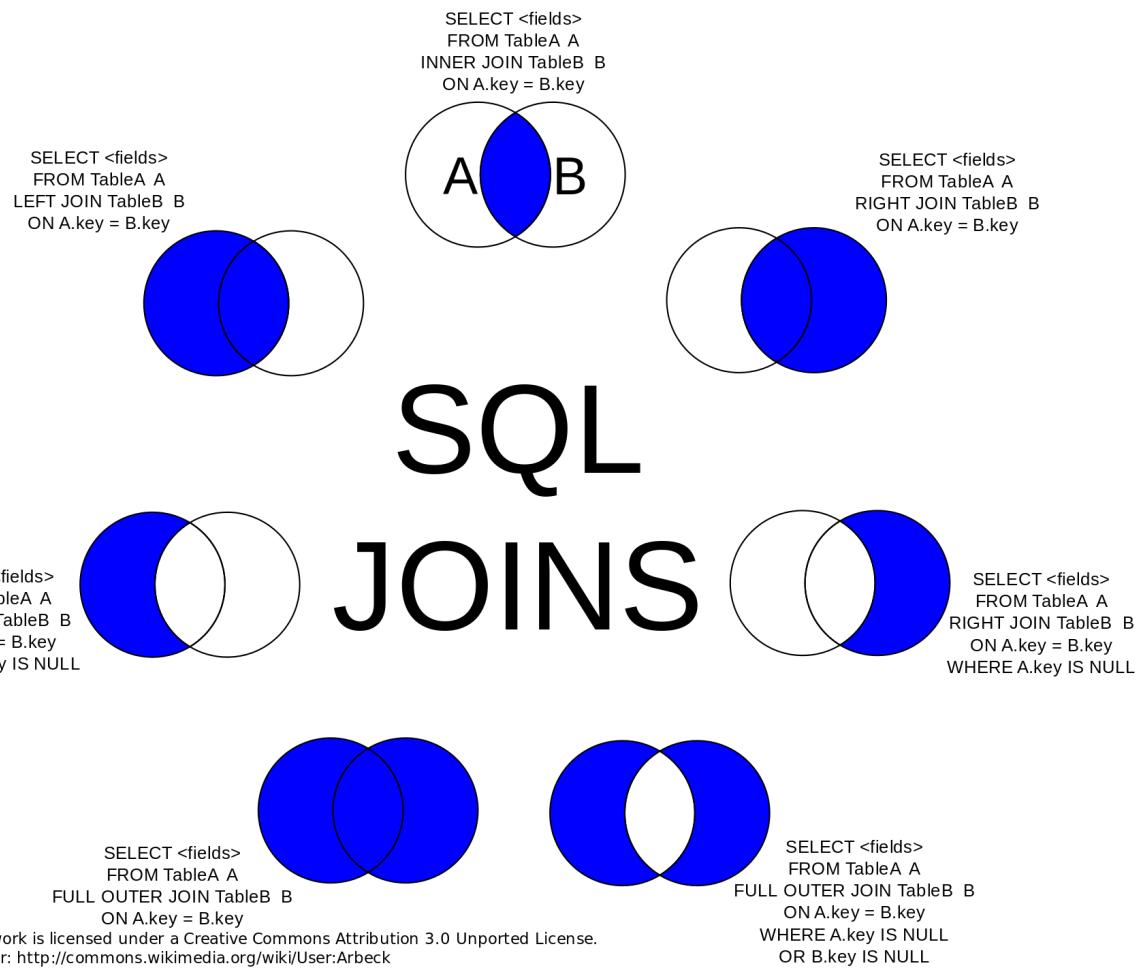
Básicamente estamos diciendo: 'Seleccioná todas las columnas de la tabla personas y uní todas las filas con la tabla ciudades donde el id de ciudades sea igual al campo ciudad de personas.'

Podemos reescribir la consulta de esta forma:

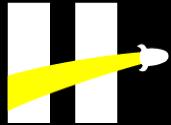
```
SELECT p.nombre, p.apellido, c.nombre FROM personas p
JOIN ciudades c
ON c.id = p.ciudad;
```

Ahora sólo pedimos el nombre, apellido de las personas y el nombre de la ciudad. Como **nombre** está en las dos tablas, tenemos que especificar de qué tabla es la columna. Para no escribir todo el nombre completo, podemos definir un ALIAS en la consulta, en este caso a **personas** le dimos el alias **p** y a **ciudades** el alias **c**.

Según el tipo de unión que queremos hacer vamos a usar alguno de estos tipos de **JOINS**:

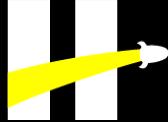


Los joins pueden ser operaciones muy costosas, de hecho, las bases de datos no relacionales suelen ser tan performantes porque esquivan los JOINS, logran ser más rápidas, pero ocupando más espacio.

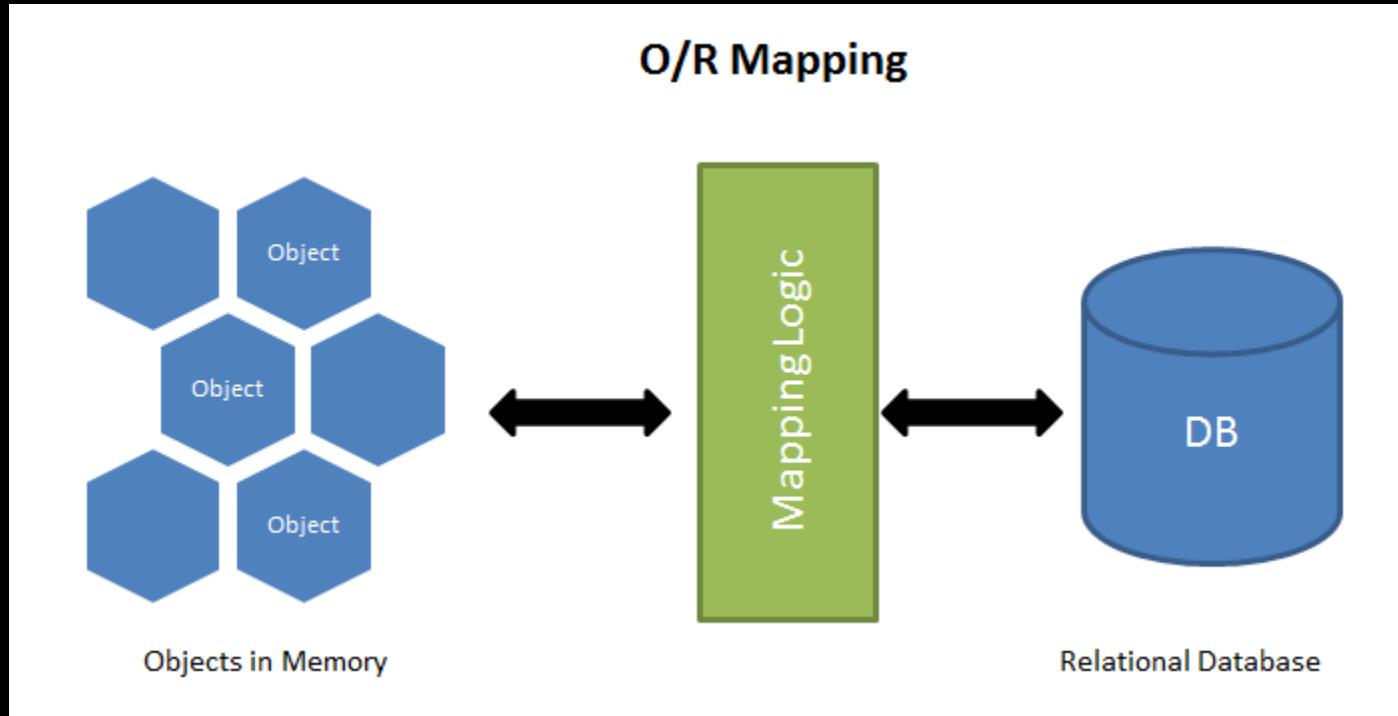


# ORMs

## Object-Relational Mapping



# ORMs





# Sequelize



"**Promise-based** Node.js ORM for Postgres, MySQL, MariaDB,  
SQLite and Microsoft SQL Server"



La mayoría de los métodos son asíncronos por lo que devuelven promesas



# Sequelize

## Installing

```
● ● ●
1 npm install --save sequelize
2 npm install --save pg pg-hstore # Postgres
```

## Connecting

```
● ● ●
1 const { Sequelize } = require('sequelize');
2
3 // Opción 1: Connection URI
4 const sequelize = new Sequelize('postgres://user:pass@example.com:5432/dbname')
5
6 // Opción 2: Parámetros separados
7 const sequelize = new Sequelize('database', 'username', 'password', {
8 host: 'localhost',
9 dialect: /* one of 'mysql' | 'mariadb' | 'postgres' | 'mssql' */
10});
```



# Data Types

```
1 // TEXTO
2 DataTypes.STRING // VARCHAR(255)
3 DataTypes.STRING(1234) // VARCHAR(1234)
4 DataTypes.TEXT // TEXT
5
6 // NUMEROS
7 DataTypes.INTEGER // INTEGER
8 DataTypes.FLOAT // FLOAT
9
10 // FECHAS
11 DataTypes.DATE // TIMESTAMP WITH TIME ZONE
12 DataTypes.DATEONLY // DATE without time
13
14 // OTROS
15 DataTypes.ENUM('foo', 'bar') // An ENUM with allowed values 'foo' and 'bar'
```

Listado Completo



# Model

## Definition

Abstracción que representa una tabla de nuestra base de datos

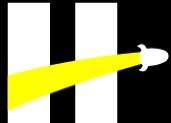
```
1 const User = sequelize.define('User', {
2 firstName: {
3 type: DataTypes.STRING
4 },
5 lastName: {
6 type: DataTypes.STRING
7 }
8 });
```



```
1 class User extends Model {}
2
3 User.init({
4 firstName: {
5 type: DataTypes.STRING
6 },
7 lastName: {
8 type: DataTypes.STRING
9 }
10 }, {
11 sequelize, // Connection instance
12 modelName: 'User' // Model name
13 });
```



User === `sequelize.models.User`



# Model

## Synchronization

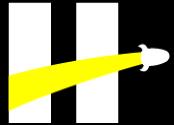
Definir el modelo arma el esqueleto pero no aplica los cambios en la base de datos, para eso debemos sincronizar los modelos

- **Model.sync()**: crea la tabla si no existe o no hace nada si ya existe
- **Model.sync({force: true})**: elimina (drop) la tabla y luego la vuelve a crear
- **Model.sync({alter: true})**: aplica los cambios necesarios a la tabla actual para que coincida con el modelo



```
1 // Un modelo a la vez
2 await Model.sync({ force: true });
3
4 // Todos los modelos juntos
5 await sequelize.sync({ force: true });
```





# Sequelize

## Logging



```
1 const sequelize = new Sequelize('postgres://user:pass@example.com:5432/dbname', {
2 // Opciones:
3
4 // Default
5 logging: console.log,
6
7 // Muestra información adicional más allá de la Query SQL
8 logging: (...msg) => console.log(msg),
9
10 // Deshabilita el logging
11 logging: false,
12});
```



# Model

## Automatic Timestamps

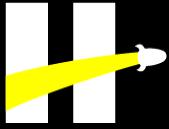
Se agregan automáticamente los campos `createdAt` y `updatedAt` que a su vez se actualizan sólo al crear o modificar una instancia del modelo



```
1 sequelize.define('User', {
2 // ... (attributes)
3 }, {
4 timestamps: false
5 });
```



```
1 sequelize.define('User', {
2 // ... (attributes)
3 }, {
4 timestamps: true,
5 createdAt: false,
6 updatedAt: 'actualizado'
7 });
```



# Model

## Column Options



```
1 class Foo extends Model {}
2 Foo.init({
3 flag: { type: DataTypes.BOOLEAN, allowNull: false, defaultValue: true },
4 myDate: { type: DataTypes.DATE, defaultValue: DataTypes.NOW },
5 someUnique: { type: DataTypes.STRING, unique: true },
6
7 // Composite unique key.
8 uniqueOne: { type: DataTypes.STRING, unique: 'compositeIndex' },
9 uniqueTwo: { type: DataTypes.INTEGER, unique: 'compositeIndex' },
10
11
12 identifier: { type: DataTypes.STRING, primaryKey: true },
13 incrementMe: { type: DataTypes.INTEGER, autoIncrement: true },
14
15 });
```



# Instances

Representa un objeto con la estructura definida en el modelo que va a mapearse con una fila de la tabla asociada

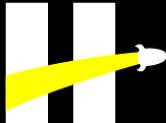


```
1 const jane = User.build({ name: "Jane" });
2 await jane.save();
```



```
1 const jane = await User.create({ name: "Jane" });
```





# Instances

## Modifications



```
1 const jane = await User.create({ name: "Jane" });
2 jane.name = "Ada";
3 await jane.save();
```



```
1 const jane = await User.create({ name: "Jane" });
2 await jane.destroy();
```



El método `save` está optimizado para sólo actualizar los campos que efectivamente fueron modificados. Si se invoca sin ningún cambio ni siquiera ejecutará una query



```
1 const jane = await User.create({ name: "Jane" });
2 console.log(jane); // Mal! (En principio para lo que queremos)
3 console.log(jane.toJSON()); // Bien!
```

# Queries

## SELECT



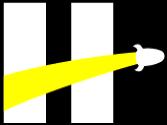
```
1 // SELECT * FROM ...
2 const instancias = await Model.findAll();
3
4 // SELECT foo, bar FROM ...
5 Model.findAll({
6 attributes: ['foo', 'bar']
7 });
8
9 // SELECT foo, bar as baz FROM ...
10 Model.findAll({
11 attributes: ['foo', ['bar', 'baz']]
12 });
13
14 // Exclude some attribute
15 Model.findAll({
16 attributes: { exclude: ['baz'] }
17 });
```



```
1 const instances = Model.findAll({
2 where: {
3 clothe: 'orange'
4 status: 'good'
5 }
6 });
```



```
1 const { Op } = require("sequelize");
2 Model.findAll({
3 where: {
4 [Op.and]: [
5 { clothe: 'orange' },
6 { status: 'good' }
7]
8 }
9 });
```



# Queries

## Finders



```
1 const instance = await Model.findByPk(4); // null si no lo encuentra
```



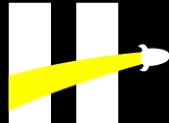
```
1 const instance = await Model.findOne({
2 where: { name: 'Goku' }
3 }); // null si no lo encuentra
```



```
1 const [instance, created] = await Model.findOrCreate({
2 where: { name: 'Goku' },
3 defaults: {
4 gender: 'M',
5 race: 'Saiyan'
6 }
7 });
```

findOrCreate busca si existe un registro según las condiciones de búsqueda y si no encuentra ninguno procede a crear uno nuevo. Luego retorna la instancia creada o encontrada y un booleano indicando cual de los dos caminos tomó





# Queries

## Operators

```
1 const { Op } = require("sequelize");
2 Model.findAll({
3 where: {
4 [Op.and]: [{ a: 5 }, { b: 6 }], // (a = 5) AND (b = 6)
5 [Op.or]: [{ a: 5 }, { b: 6 }], // (a = 5) OR (b = 6)
6 someAttribute: {
7 // Basics
8 [Op.eq]: 3, // = 3
9 [Op.ne]: 20, // != 20
10 [Op.is]: null, // IS NULL
11 [Op.not]: true, // IS NOT TRUE
12
13 // Number comparisons
14 [Op.gt]: 6, // > 6
15 [Op.lt]: 10, // < 10
16 [Op.between]: [6, 10], // BETWEEN 6 AND 10
17 [Op.notBetween]: [11, 15], // NOT BETWEEN 11 AND 15
18
19 // Other operators
20 [Op.in]: [1, 2], // IN [1, 2]
21 [Op.notIn]: [1, 2], // NOT IN [1, 2]
22
23 ...
24 }
25 }
26});
```

Listado Completo



# Queries

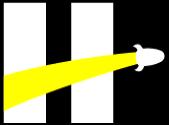
## UPDATE



```
1 // UPDATE table
2 // SET transformation = 'SS1'
3 // WHERE name = 'Goku'
4
5 await User.update({ transformation: 'SS1' }, {
6 where: {
7 name: 'Goku'
8 }
9 });
```



- Actualizará todas las instancias que coincidan con la cláusula where indicada. Si no se coloca ninguna condición actualizará todos los registros



# Queries

## DELETE

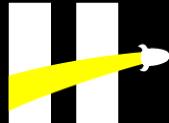
```
1 // DELETE FROM table
2 // WHERE race = 'Android'
3
4 await Model.destroy({
5 where: {
6 race: 'Android'
7 }
8 });
9
10 // Truncate
11 await Model.destroy({
12 truncate: true
13 });
```



Borrará todas las instancias que coincidan con la cláusula where indicada. Si no se coloca ninguna condición borrará todos los registros



¿TRUNCATE = DESTROY? Truncate no acepta condiciones y elimina todos los registros de una mientras que destroy va revisando registro a registro



# Getters



```
1 const instance = sequelize.define('model', {
2 attribute: {
3 type: DataTypes.STRING,
4 get() {
5 const rawValue = this.getDataValue(attribute);
6 return rawValue ? rawValue.toUpperCase() : null;
7 }
8 }
9 });
```

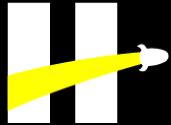
Se llama de forma automática cuando se intenta acceder al atributo



El valor NO se va a modificar en la base de datos



this.attribute generaría un loop infinito (Si accedemos a otro attributo que no sea donde estamos definiendo el getter funcionaria ok el this)



# Setters

```
1 const User = sequelize.define('user', {
2 password: {
3 type: DataTypes.STRING,
4 set(value) {
5 this.setDataValue('password', hash(this.username + value));
6 }
7 }
8});
```

Se llama de forma automática previo al almacenamiento de los datos en la base de datos



# Virtual Fields

```
1 const User = sequelize.define('user', {
2 firstName: DataTypes.TEXT,
3 lastName: DataTypes.TEXT,
4 fullName: {
5 type: DataTypes.VIRTUAL,
6 get() {
7 return `${this.firstName} ${this.lastName}`;
8 },
9 set(value) {
10 throw new Error('Do not try to set the `fullName` value!');
11 }
12 }
13});
```



El valor NO se va a guardar en la base de datos



# Validators



```
1 sequelize.define('foo', {
2 bar: {
3 type: DataTypes.STRING,
4 validate: {
5 is: /^[a-z]+$/i,
6 isEmail: true,
7 isUrl: true,
8 isAlpha: true,
9 isAlphanumeric: true,
10 isNumeric: true,
11 isLowercase: true,
12 notNull: true,
13 notEmpty: true,
14 equals: 'specific value',
15 contains: 'foo',
16 isIn: [['foo', 'bar']],
17 notContains: 'bar',
18 len: [2,10],
19 isAfter: "2011-11-05",
20 max: 23,
21
22 // Custom validators:
23 isEven(value) {
24 if (parseInt(value) % 2 !== 0) {
25 throw new Error('Only even values are allowed!');
26 }
27 }
28 }
29 }
30});
```

Listado Completo



# Associations

## One-To-One

```
1 FoohasOne(Bar);
2 Bar.belongsTo(Foo);
```

1:1

## One-To-Many

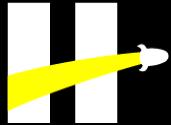
```
1 TeamhasMany(Player);
2 Player.belongsTo(Team);
```

1:N

## Many-To-Many

```
1 Movie.belongsToMany(Actor, { through: 'ActorMovies' });
2 Actor.belongsToMany(Movie, { through: 'ActorMovies' });
```

N:N



# Mixins

FoohasOne(Bar)

Foo.belongsTo(Bar)



```
1 fooInstance.getBar()
2 fooInstance.setBar()
3 fooInstance.createBar()
```



```
1 fooInstance.getBars()
2 fooInstance.countBars()
3 fooInstance.hasBar()
4 fooInstance.hasBars()
5 fooInstance.setBars()
6 fooInstance.addBar()
7 fooInstance.addBars()
8 fooInstance.removeBar()
9 fooInstance.removeBars()
10 fooInstance.createBar()
```

FoohasMany(Bar)



# Mixins

Foo.belongsToMany(Bar, {through: Baz})



```
1 fooInstance.getBars()
2 fooInstance.countBars()
3 fooInstance.hasBar()
4 fooInstance.hasBars()
5 fooInstance.setBars()
6 fooInstance.addBar()
7 fooInstance.addBars()
8 fooInstance.removeBar()
9 fooInstance.removeBars()
10 fooInstance.createBar()
```



# Fetching Associations

## Lazy Loading

```
1 const awesomeCaptain = await Captain.findOne({
2 where: {
3 name: "Jack Sparrow"
4 }
5 });
6
7 console.log('Name:', awesomeCaptain.name);
8 console.log('Skill Level:', awesomeCaptain.skillLevel);
9
10 const hisShip = await awesomeCaptain.getShip();
11
12 console.log('Ship Name:', hisShip.name);
13 console.log('Amount of Sails:', hisShip.amountOfSails);
```





# Fetching Associations

## Eager Loading

```
1 const awesomeCaptain = await Captain.findOne({
2 where: {
3 name: "Jack Sparrow"
4 },
5 include: Ship
6 });
7
8 console.log('Name:', awesomeCaptain.name);
9 console.log('Skill Level:', awesomeCaptain.skillLevel);
10 console.log('Ship Name:', awesomeCaptain.ship.name);
11 console.log('Amount of Sails:', awesomeCaptain.ship.amountOfSails);
```



# Hooks

## Lifecycle Events



```
1 (1)
2 beforeValidate(instance, options)
3
4 [... validation happens ...]
5
6 (2)
7 afterValidate(instance, options)
8 validationFailed(instance, options, error)
9 (3)
10 beforeCreate(instance, options)
11 beforeDestroy(instance, options)
12 beforeUpdate(instance, options)
13
14 [... creation/update/destruction happens ...]
15
16 (4)
17 afterCreate(instance, options)
18 afterDestroy(instance, options)
19 afterUpdate(instance, options)
```



```
1 User.beforeCreate((user, options) => {
2 const hashedPassword = hashPassword(user.password);
3 user.password = hashedPassword;
4 });
```

# Henry

---



Hacé click acá para dejar tu feedback sobre esta clase.



Hacé click acá completar el quiz teórico de esta lecture.

## ORM

Un problema con las bases de datos relacionales (puede ser un gran problema en proyectos complejos) es que las cosas que guardamos en ella no mapean uno a uno a los objetos que tenemos en nuestra aplicación. De hecho, es probable que en nuestra App tengamos la clase *persona*, pero difícilmente tengamos la clase *ciudad* y que ambas estén relacionadas. Simplemente tendríamos una propiedad *ciudad* dentro de *persona*. Por lo tanto vamos a necesitar alguna capa de abstracción que nos oculte la complejidad de las tablas y sus relaciones y nosotros sólo veamos objetos desde la app. Para eso existen los **ORM** (Object relation mapping), que son librerías o frameworks que hacen este trabajo por nosotros. Sería lo mismo que [mongoose](#), pero un poco al revés!

### SEQUELIZE

Obviamente existen un montón de ORMs (de estos de verdad hay miles porque se vienen usando hace mucho). Nosotros vamos a utilizar [sequelize](#), en particular. Este es un ORM para nodejs que soporta varios motores de bases de datos:

- PostgreSQL
- MySQL
- MariaDB
- SQLite
- MSSQL (Microsoft)

Por supuesto que ustedes deben probar y jugar con varios de ellos hasta que encuentren alguno que se acomode a su filosofía de programación, no hay *uno* que sea el mejor de todos.

Hay otras librerías que no llegan a ser ORMs pero nos ayudan a hacer queries a la base de datos, si les gusta tener el control al 100% de su base de datos le recomiendo probar con algunos de estos, por ejemplo: [MassiveJS](#)

### Instalación

Como [sequelize](#) soporta varias bases de datos, vamos a necesitar primero instalar el módulo de [sequelize](#) per ser, y luego el módulo del conector a la base de datos que vayamos a usar. [sequelize](#) hace uso de estos últimos para conectarse a la base de datos.

```
$ npm install --save sequelize

y uno de los siguientes
$ npm install --save pg pg-hstore
$ npm install --save mysql // Para MySQL y MariaDB
$ npm install --save sqlite3
$ npm install --save tedious // MSSQL
```

Igual que con Mongoose, primero vamos a importar el módulo y vamos a crear un objeto **Sequelize** pasandole como parámetro la string que indica a qué base de datos conectarse, con qué usuario y qué password:

```
var Sequelize = require("sequelize"); //requerimos el modulo
var sequelize = new Sequelize('postgres://user:pass@example.com:5432/dbname');
```

## Modelos

Sequelize es muy parecido a Mongoose, primero vamos a tener que definir un modelo (las constrains y tipos de datos pueden diferir, pero el concepto es el mismo), los modelos se definen de la forma:

`sequelize.define('name', {attributes}, {options}).`, veamos un ejemplo:

```
var User = sequelize.define('user', {
 firstName: {
 type: Sequelize.STRING,
 field: 'first_name' //el nombre en la base de datos va a ser first_name
 },
 lastName: {
 type: Sequelize.STRING
 }
});

User.sync({force: true}).then(function () { //tenemos que usar este método porque
 // Tabla creadas // antes de guardar datos en las tablas
 return User.create({ // estas tienen que estar definidas
 firstName: 'Guille',
 lastName: 'Aszyn'
 });
});
```

A diferencia de Mongoose, con sequelize vamos a tener que preocuparnos un poco por las tablas que haya en nuestra base de datos, ya que sin no hay tablas creadas, no vamos a poder guardar datos (en mongodb nos creaba las colecciones solo, y no importaba la estructura para guardar documentos). Por eso usamos el método **sync()** que justamente sincroniza el modelo que tenemos en nuestra app con la BD (crea la tabla en la db si no existe), como vemos puede recibir un *callback*, en el cual usamos el método **create()** para crear un nuevo user y guardarlo en la tabla. Podemos ver más métodos y cómo funcionan en la [Documentación](#).

Fíjense que en Sequelize para pasar un callback lo hacemos en la función `.then()`, eso es una promesa, por ahora usenlo como una callback normal, más adelante las veremos en detalle.

## CRUD ( Create, Read, Update, Delete)

Para insertar datos en la base de datos, vamos a usar la función `create()`, que básicamente lo que hace es crear una nueva instancia del modelo y lo guarda en la base de datos:

```
User.create({
 firstName: 'Juan',
 lastName: 'Prueba'
}).then(function(user) {
 console.log(user);
})
```

La función `create()` en realidad, encapsula dos comportamiento, como dijimos arriba: instanciar el modelo y guardar en la bd. De hecho, podríamos hacer ambas cosas por separado, usando la función `build()` y `save()`:

```
var user = User.build({
 firstName: 'Juan',
 lastName: 'Prueba'
})

user.save().then(function(user) {
 console.log(user);
})
```

Para buscar registros usamos la función `find`, que viene en distintos sabores: `findAll()`: Sirve para buscar múltiples registros, `findOne()`: sirve para buscar un sólo registro y `findById()`: es igual a `findOne()` pero podemos buscar sólo por el ID del registro.

```
User.findAll({ where: ["id > ?", 25] }).then(function(users) {
 console.log(users) //busca TODOS los usuarios
});

User.findOne({
 where: {firstname: 'Toni'},
 attributes: ['id', ['firstname', 'lastname']]
}).then(function(user) {
 console.log(user)
});

User.findById(2).then(function(user) {
 console.log(user) //El user con ID 2
});
```

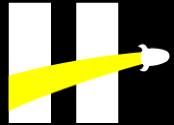
Las búsquedas pueden ser más complejas que en Mongo, por la naturaleza de las relaciones, por lo tanto es importante que leamos bien la [documentación](#).

Para modificar un registro, o varios, tenemos que pasarle los nuevos atributos que queremos modificar, y además una condición de búsqueda, en este caso voy a cambiarle el nombre a todos los registros que tengan `id = 1` (sólo puede haber uno 😅):

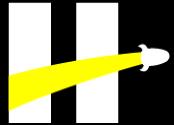
```
User.update({
 firstname: 'Martin' ,
}, {
 where: {
 id: '1'
 }
});
```

Para borrar un registro, vamos a usar el método `destroy()`, que tambien recibe un parámetro de búsqueda, en el ejemplo vamos a borrar todos los registros que tengan id 1, 2 ,3 o 4:

```
User.destroy(
 { where: { id: [1,2,3,4] }
});
```

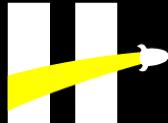


# Autenticación

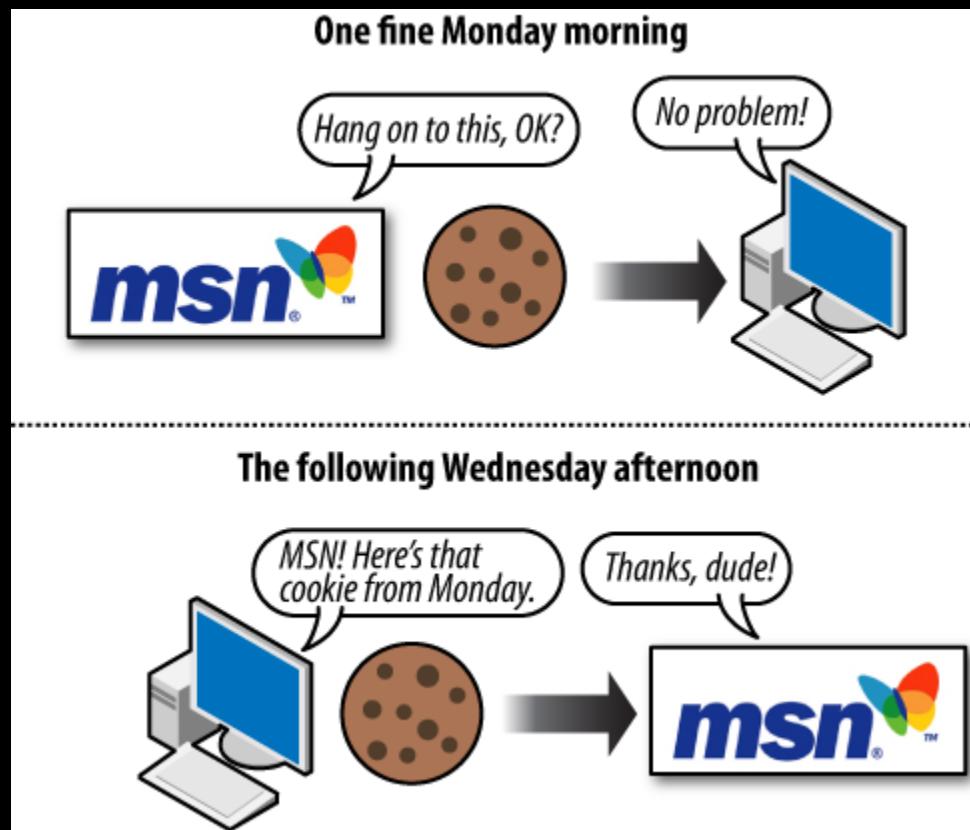


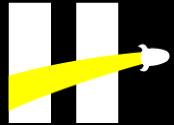
# Auth... entication? orization?





# Cookies



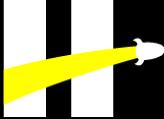


# Cookies

A screenshot of a Microsoft Edge browser window. The address bar shows the URL `localhost:3000/getuser`. The main content area displays a JSON object:

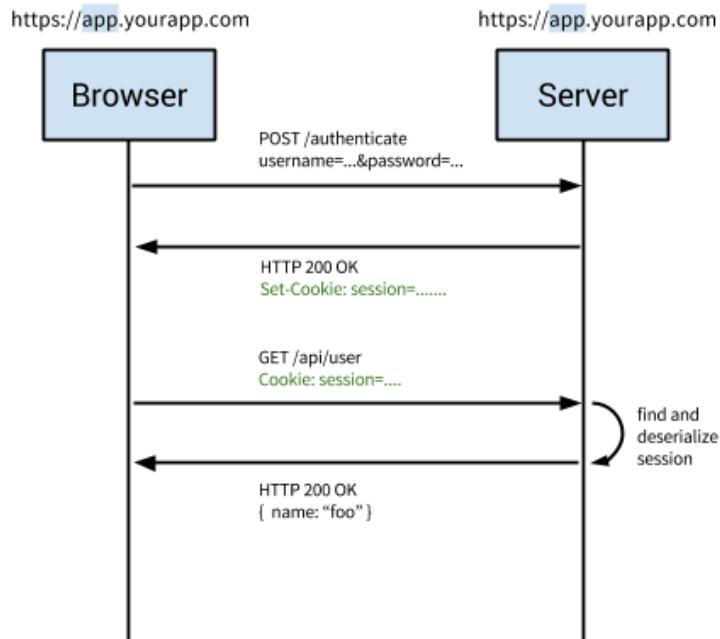
```
{"connect.sid": "s:hrwPi2vIQwHZxN_rciJUPuwJnw5-Qk6Z.fVjrasyYt/DgW98cIECtnNkdlib1zLeLpMrFieFFi3E"}
```

The browser interface includes a tab bar with multiple open tabs, a search bar at the bottom left, and a taskbar at the very bottom with various pinned icons.

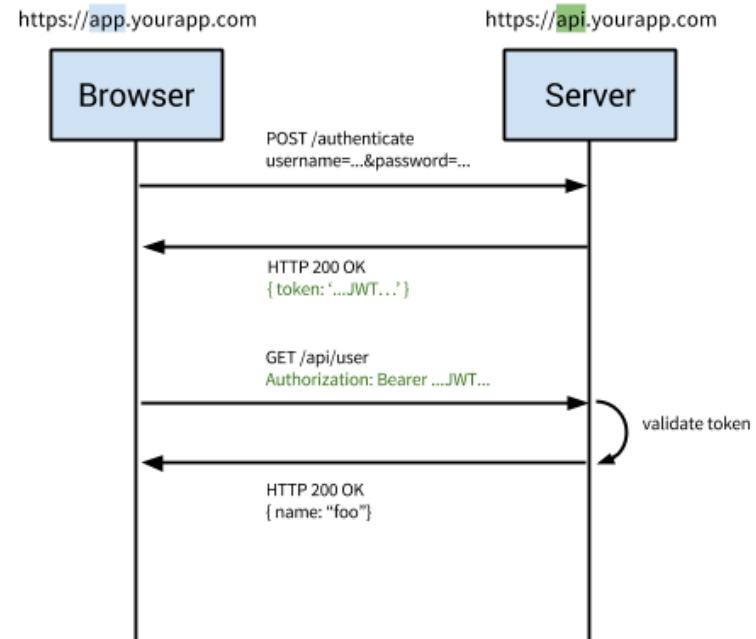


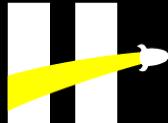
# Cookies

Traditional Cookie-Based Auth



Modern Token-Based Auth





# JWT

## Encoded

PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.e
yJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6Ikp
vaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.Sf
1KxwRJSMeKKF2QT4fwpMeJf36P0k6yJV_adQss
w5c
```

## Decoded

EDIT THE PAYLOAD AND SECRET

### HEADER: ALGORITHM & TOKEN TYPE

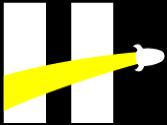
```
{
 "alg": "HS256",
 "typ": "JWT"
}
```

### PAYOUT: DATA

```
{
 "sub": "1234567890",
 "name": "John Doe",
 "iat": 1516239022
}
```

### VERIFY SIGNATURE

```
HMACSHA256(
 base64UrlEncode(header) + "." +
 base64UrlEncode(payload),
 your-256-bit-secret
) secret base64 encoded
```



# JWT

## Crypto Segment - Signature

```
eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVC
J9 .eyJzdWIiOiIxMjM0NTY3ODkwIiwibmF
tZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRyd
WUsImhlhdCI6MTUXNjIzOTAyMn0 .TCYt5Xs
ITJX1CxPCT8yAV-TVkIEq_PbCh0MqsLfRo
Psnsgr5WEuts01mq-pQy7UJiNSmgRx0-WU
cX16dUEMGlv50aqzpqh4Qktb3rk-BuQy72
IFLoqv0G_zS245-kxonKb78cPN25DGlcTw
LtjPAYuNzVBAh4vGH5rQyHudBBPh
```

# Henry

---

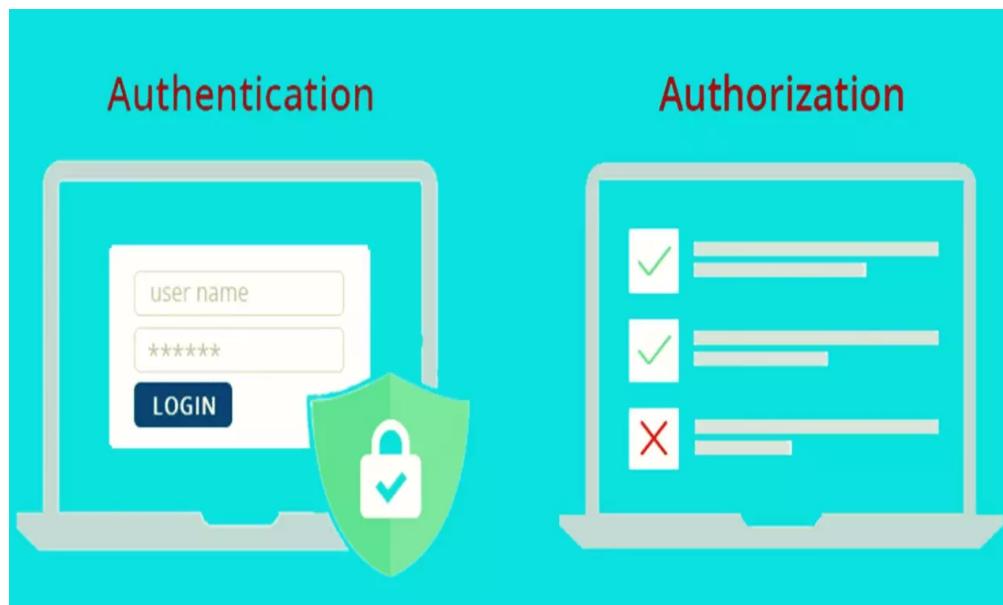


Hacé click acá para dejar tu feedback sobre esta clase.

Hacé click acá completar el quiz teórico de esta lecture.

## AUTENTICACIÓN VS AUTORIZACIÓN

Es normal pensar que autenticación y autorización son sinónimos. De hecho, ambos son procesos de seguridad, aunque tienen propósitos diferentes.



La autenticación es el proceso de identificar a los usuarios y garantizar que los mismos sean quienes dicen ser. ¿Cuál podría ser una prueba de autenticación? La más utilizada es la contraseña. A la hora de iniciar sesión, si conocen sus credenciales (nombre de usuario y contraseña), el sistema entenderá que su identidad es válida. En consecuencia, van a poder acceder al recurso o conjunto de recursos que quieran.

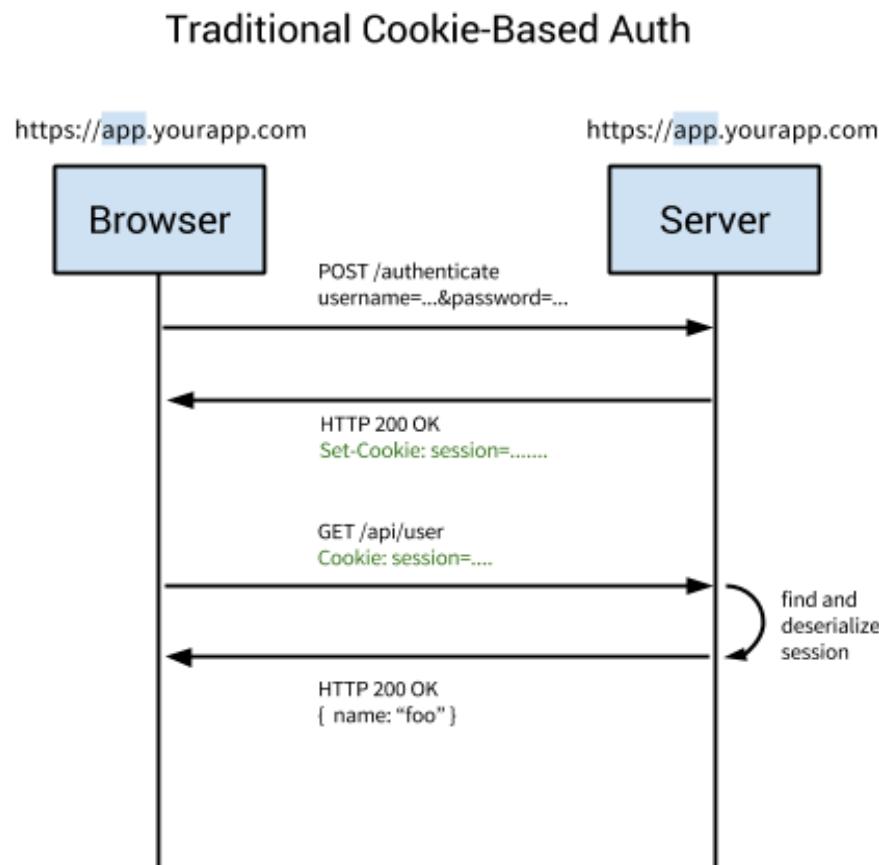
Por otro lado, la autorización es lo que define a qué recursos dentro de ese sistema autenticado van a poder acceder. Que hayan logrado pasar la instancia de la autenticación, no significa que van a poder utilizar el sistema por completo. Un ejemplo sería autenticarse en la página de un banco. Una autenticación exitosa no les va a otorgar la capacidad de ver las cuentas de otros clientes ni de retirar dinero de las mismas. En cambio, para un ejecutivo de cuentas, que inicia sesión como administrador, es habitual acceder a muchas cuentas y realizar acciones adicionales que un usuario, como cliente, no podría.

## AUTENTICACIÓN

Durante este tiempo, estuvimos trabajando con protocolos HTTP. Estos protocolos tienen una particularidad, y es que son stateless, osea que no tienen conocimiento sobre estados. Y eso qué significa? que cada vez que enviamos una request, esta se va a ejecutar, pero si refrescamos la página todos esos datos se van a perder. Es por esto que vamos a necesitar herramientas que **guarden** la sesión, es decir, que nos permitan como usuarios permanecer logueados una vez que enviamos nuestros datos.

## COOKIES

Una de las herramientas que nos van a permitir poder guardar esa información son las cookies. Una cookie (o galleta informática) es un pequeño archivo de datos creado por el sitio web visitado y almacenado en el navegador o dispositivo utilizado por los usuarios. Las cookies contienen pequeñas cantidades de información que se envían entre un emisor (generalmente el servidor de la web) y un receptor (el navegador del usuario) almacenando esta información en la memoria del navegador. El objetivo de este proceso es que la web visitada pueda comprobar esa cookie en una futura conexión del usuario y utilizarla.



- **Cookies de sesión (Session Cookies)**

Comprenden el tiempo que transcurre entre el inicio y el cierre de una sesión. Al iniciarla, el servidor genera un “ID de sesión” que se transmite al cliente. Este ID, también denominado identificador de sesión, es un número generado al azar que las cookies almacenan de manera temporal. Su único fin es asignar al usuario una sesión en particular. Este identificador tiene una gran ventaja: si nosotros como usuarios abrimos varias ventanas en el mismo sitio web, estas se van a asignar a una sola sesión. Esto nos va a permitir que iniciemos varias consultas de forma simultánea sin que se pierda información personal importante. Una vez cerrada esta sesión de navegación, tanto el identificador, como el resto de datos almacenados, se borran.

- **Cookies persistentes**

Opuestas a las cookies de sesión, las cookies persistentes son aquellas que siguen almacenadas en el navegador durante más tiempo incluso después de la sesión (pueden ser horas, meses o años).

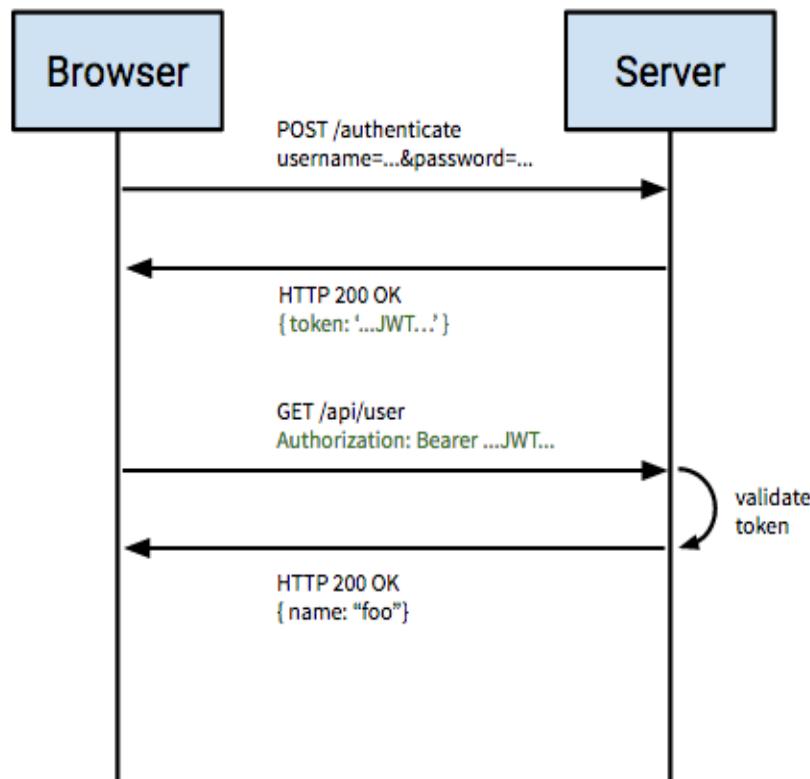
## TOKENS DE AUTENTICACIÓN

Por otro lado, la autenticación basada en tokens es un protocolo que nos va a permitir verificar nuestra identidad y, a cambio, recibir un token de acceso único. Durante la vida del token, los usuarios accedemos al sitio web o la aplicación para la que se emitió, en lugar de tener que volver a ingresar las credenciales cada vez que volvemos a ingresar a la misma página web, aplicación o cualquier recurso protegido con ese mismo token.

Lo pueden pensar como un boleto sellado. Básicamente, vamos a conservar el acceso mientras el token siga siendo válido. Una vez que cerramos sesión o salimos de una aplicación, el token se invalida.

La autenticación basada en tokens es diferente de las técnicas de autenticación tradicionales basadas en contraseña o en servidor. Los tokens ofrecen una segunda capa de seguridad y los administradores tienen un control detallado sobre cada acción y transacción.

## Modern Token-Based Auth



## JWT

Un JSON Web Token es un token de acceso estandarizado en el RFC 7519 que permite el intercambio seguro de datos entre dos partes. Contiene toda la información importante sobre una entidad, lo que implica que no hace falta consultar una base de datos ni que la sesión tenga que guardarse en el servidor (sesión sin estado).

Por este motivo, los JWT son especialmente populares en los procesos de autenticación. Con este estándar es posible cifrar mensajes cortos, enviarles información sobre el remitente y demostrar si este cuenta con los derechos de acceso requeridos. Los propios usuarios solo entran en contacto con el token de manera indirecta: por ejemplo, al introducir el nombre de usuario y la contraseña en una interfaz. La comunicación como tal entre las diferentes aplicaciones se lleva a cabo en el lado del cliente y del servidor.

## Token JWT

En la práctica, se trata de una cadena de texto que tiene tres partes codificadas en Base64, cada una de ellas separadas por un punto, como en el siguiente ejemplo:

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6I1NveSBIZW5yeSIsImhdCI6MTUxNjIzOTAyMn0.eyJ_6SpS1ppUImi6OPSOJGqW_bn31g5L0BydnnhDoM8
```

Podemos utilizar un [Debugger Online](#) para decodificar ese token y visualizar su contenido. Si accedemos al mismo y pegamos dentro el texto completo, se nos mostrará lo que contiene:

Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6I1NveSBIZW5yeSIsImhdCI6MTUxNjIzOTAyMn0.eyJ_6SpS1ppUImi6OPSOJGqW_bn31g5L0BydnnhDoM8
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
 "alg": "HS256",
 "typ": "JWT"
}
```

PAYOUT: DATA

```
{
 "sub": "1234567890",
 "name": "Soy Henry",
 "iat": 1516239022
}
```

VERIFY SIGNATURE

```
HMACSHA256(
 base64UrlEncode(header) + "." +
 base64UrlEncode(payload),
 your-256-bit-secret
) □ secret base64 encoded
```

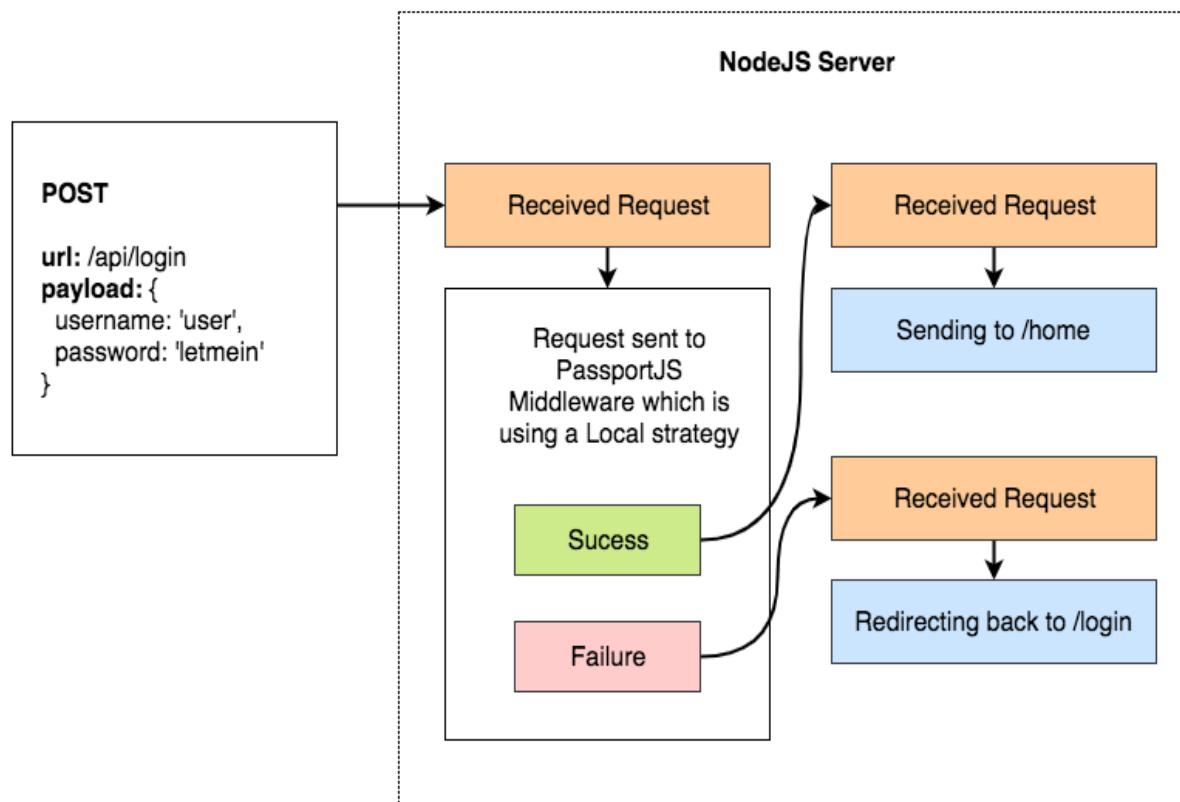
Como dijimos, un token tiene tres partes:

- **Header:** encabezado dónde se indica, al menos, el algoritmo y el tipo de token, que en el caso del ejemplo anterior era el algoritmo HS256 y un token JWT.

- **Payload:** donde aparecen los datos de usuario y privilegios, así como toda la información que queramos añadir, todos los datos que creamos convenientes.
- **Signature:** una firma que nos permite verificar si el token es válido. La firma se construye de tal forma que vamos a poder verificar que el remitente es quien dice ser, y que el mensaje no se ha modificado en el camino.

## Passport

Y cómo vamos a implementar entonces todo esto? Ahí es donde entra Passport! Passport es una librería de Node que nos va a dar la capacidad de autenticarnos. El único propósito de Passport es autenticar solicitudes, lo que hace a través de un conjunto extensible de complementos conocidos como estrategias. Y qué son las estrategias? Son diferentes mecanismos de autenticación que nos ofrece Passport para no tener que acudir a dependencias innecesarias. Por ejemplo, pueden autenticarse en una instancia de base de datos local/remota o utilizar el inicio de sesión único de proveedores de OAuth para Facebook, Twitter, Google, etc. La API es simple: le proporcionamos una solicitud de autenticación y Passport proporciona enlaces para controlar lo que ocurre cuando esta tiene éxito o falla.



## Modo de Uso

Primero que nada vamos a instalar Passport.

```
npm install passport
```

Lo seteamos en nuestro archivo raíz. Acá vamos a requerir la librería y la inicializamos junto con su middleware de autenticación de sesión.

```
const passport = require('passport');

app.use(passport.initialize());
app.use(passport.session());
```

Antes de autenticar las solicitudes, configuramos la estrategia (o estrategias) que vamos a usar.

```
passport.use(new LocalStrategy(
 function(username, password, done) {
 User.findOne({ username: username }, function (err, user) {
 if (err) { return done(err); }
 if (!user) {
 return done(null, false, { message: 'Incorrect username.' });
 }
 if (!user.validPassword(password)) {
 return done(null, false, { message: 'Incorrect password.' });
 }
 return done(null, user);
 });
 }
));
```

Para autenticar solicitudes vamos a usar `passport.authenticate()` y especificar qué estrategia queremos implementar. De forma predeterminada, si la autenticación falla, Passport nos va a responder con un estado 401 No autorizado y no va a invocar a ningún controlador de ruta adicional. Si la autenticación es exitosa, se invocará el siguiente controlador y la propiedad `req.user` se va a establecer en el usuario autenticado.

```
app.post('/login',
 passport.authenticate('local'),
 function(req, res) {
 // If this function gets called, authentication was successful.
 // `req.user` contains the authenticated user.
 res.redirect('/users/' + req.user.username);
});
```

Pueden encontrar más información sobre Passport y sus estrategias en [passportjs.org](http://passportjs.org)