Javascript Async and Promises

Nathan Lapinski

Excellent open-sourced resources

The You Don't Know JS titles are by far the best resources I have ever seen for learning the ins and outs of JS. And you can read them for free!
My experience has been that scope & closures and especially this & object prototypes are the topics that come up the most on JS interviews.

https://github.com/getify/You-Dont-Know-JS

This one is also really good if you are kind of new to the language. Again, completely free:
http://eloquentjavascript.net/

Agenda

Sync and Async

Agenda

Sync and Async
Callbacks

Agenda

Sync and Async
Callbacks
Promises

Agenda

Sync and Async

Callbacks

Promises

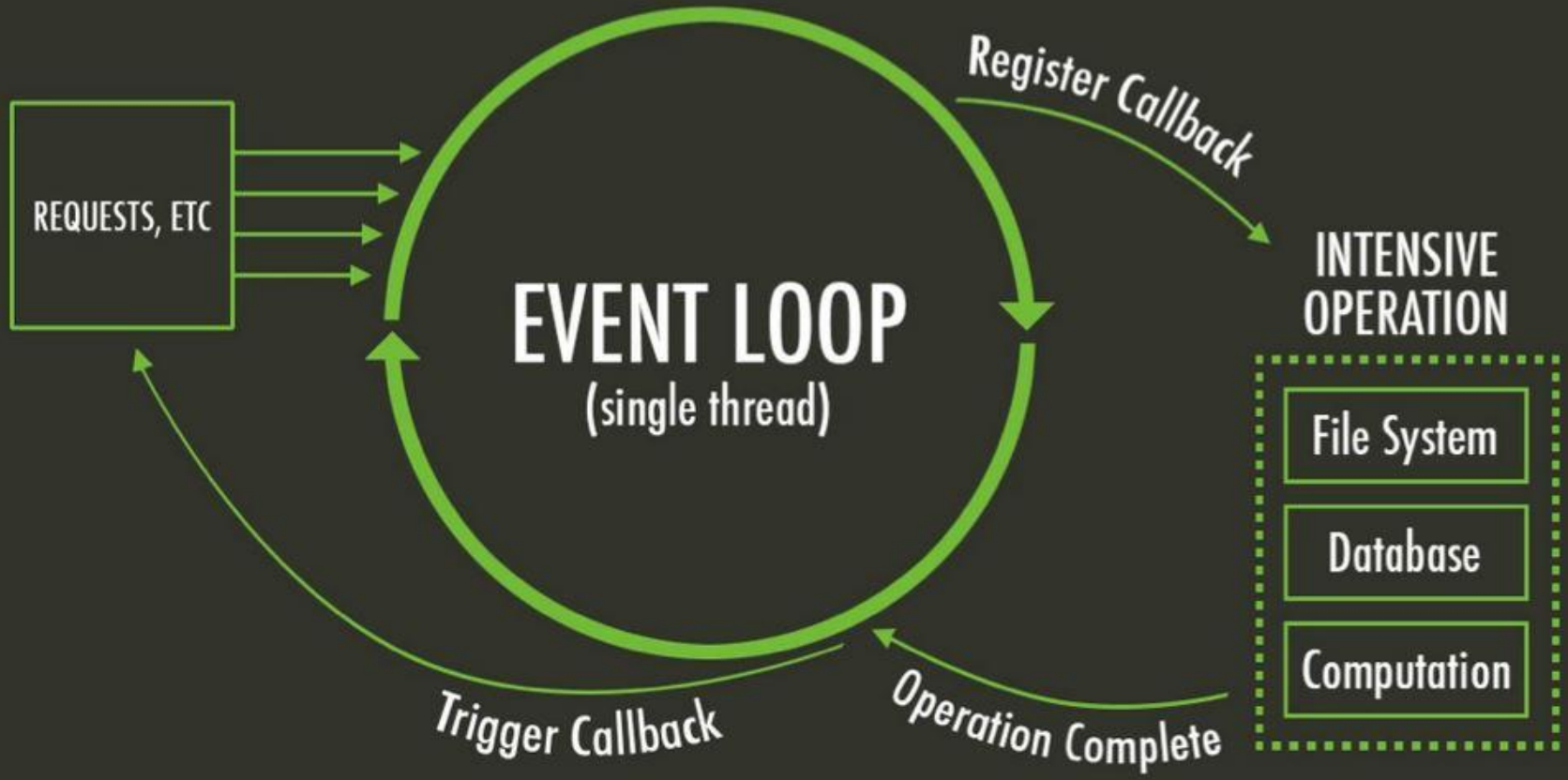
Generators and Iterators (ES6)

What is Async?

Async is the concept of now and later

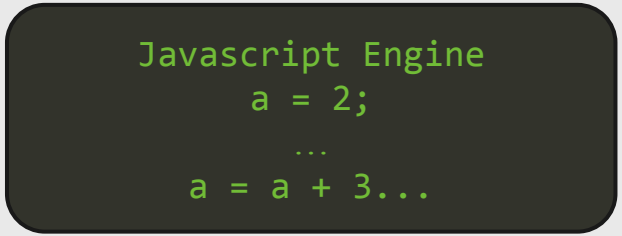
What is Async?

```
//async ajax call for data

var data = ajax( “http://google....” );

console.log(data);          //uh oh
```


REQUESTS, ETC
EVENT LOOP
(single thread)
Register Callback
INTENSIVE OPERATION
File System
Database
Computation
Trigger Callback
Operation Complete

Event Loop

Event Loop



Event Loop

Javascript Environment

```
Javascript Engine
a = 2;
...
a = a + 3...
```

Event Loop

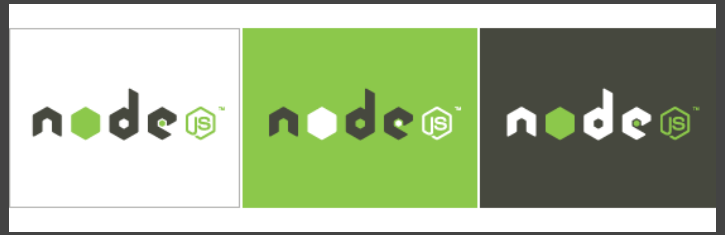
Javascript Environment

```
Javascript Engine
a = 2;
...
a = a + 3...
```

Event Loop

Javascript Environment

```
Javascript Engine
a = 2;
...
a = a + 3...
```



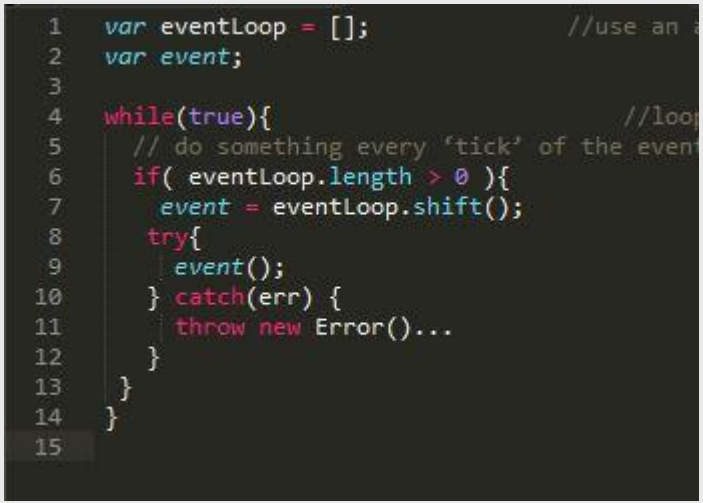
Event Loop

The environment is responsible for scheduling things.

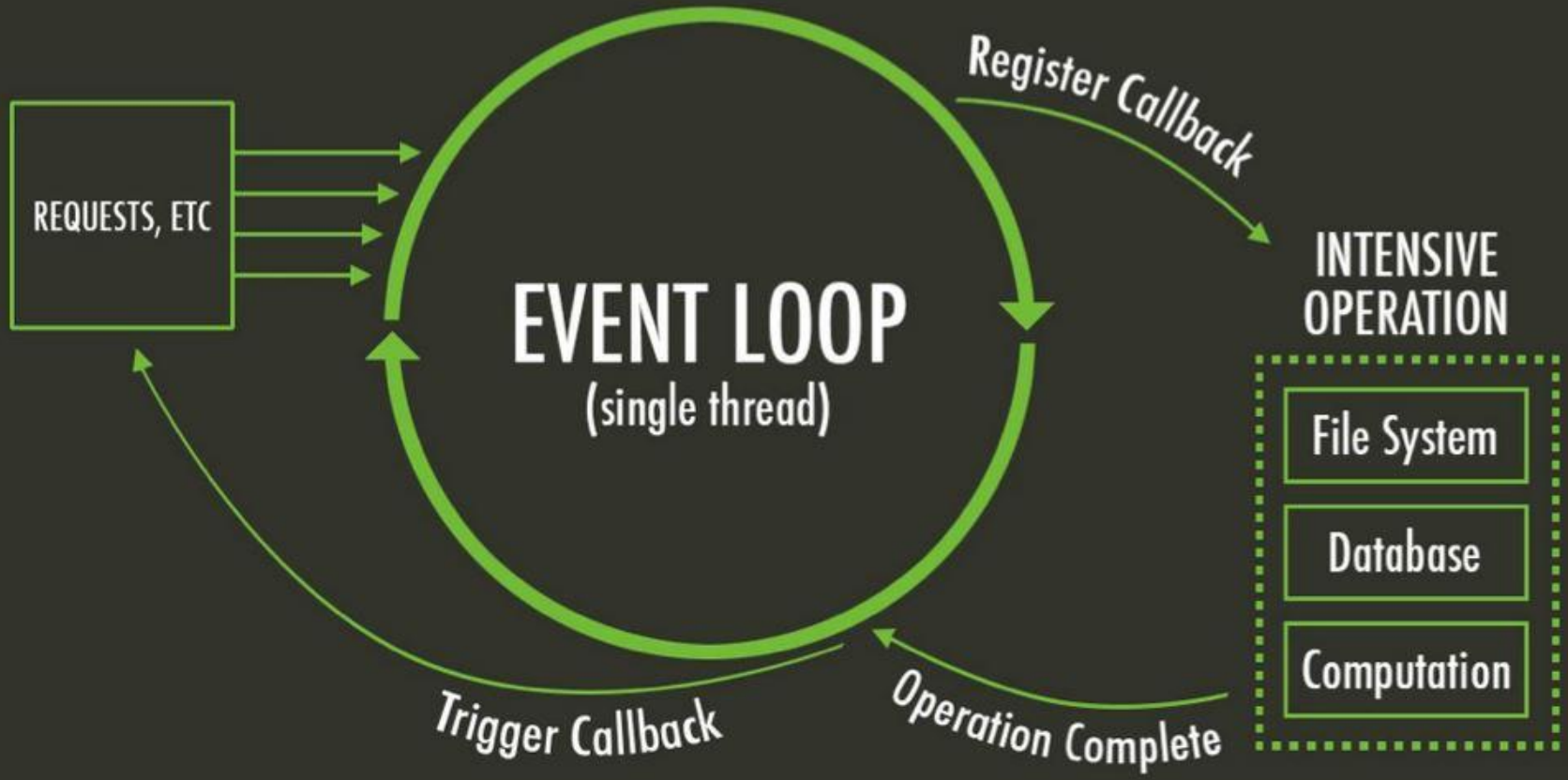
Event Loop

Uses an event loop to achieve this.

Event Loop



```
var eventLoop = [];              //use an
var event;

while(true){                                //loo
  // do something every 'tick' of the even
  if( eventLoop.length > 0 ){
    event = eventLoop.shift();
   try{
     event();
   } catch(err) {
     throw new Error()...
   }
 }
}
```


REQUESTS, ETC
EVENT LOOP
(single thread)
Register Callback
INTENSIVE OPERATION
File System
Database
Computation
Trigger Callback
Operation Complete

Async

Asynchronicity

Loop operates on “ticks”

Asynchronicity

JS environments like the browser and Node have how many threads?

Asynchronicity

JS environments like the browser and Node have how many threads?

(one)

Asynchronicity

So how do you do more than one thing at once?

Callbacks

Most fundamental async pattern in Javascript

Callbacks

Not without its shortcomings, devs are looking towards *promises*

Callbacks

but you can’t really understand an abstraction without understanding what it abstracts

Callback Example

```
console.log(“A”);

setTimeout( function(){
  console.log(“B”);
},1000);

console.log(“C”);
```

Callback Example

```
console.log(“A”);

setTimeout( function(){
  console.log(“B”);
},1000);

console.log(“C”);

//A  C  B
```

Callback Example

```
doA( function() {
  doB();

  doC( function(){
      doD();
   });

   doE();
});

doF();
```

Callback Example

```
doA( function() {
  doB();

  doC( function(){
      doD();
   });

   doE();
});

doF();
```

// A F B C E D

Sequential Brain

We think synchronously

Async Planning

We’re really not very good at it.

Callback Hell

We like to think of it as just an indentation/readability issue

Callback Hell

but it’s far worse than that

Callback Example

```
doA( function() {
  doB();

  doC( function(){
      doD();
   });

   doE();
});

doF();
```

The Real Problem

Lack of sequentiality

The Real Problem

Lack of sequentiality

Lack of trustability

Promises Promises

Future Value

Promises Promises

Future Value
Completion Effect

Promises, Promises

Coffee Example

1. I place an order for a cup of coffee and pay $2.00.

Coffee Example

1. I place an order for a cup of coffee and pay $2.00.
2. I receive a receipt to claim my coffee when it’s ready.

Coffee Example

1. I place an order for a cup of coffee and pay $2.00.
2. I receive a receipt to claim my coffee when it’s ready.
3. I can act on this receipt.

Coffee Example

1. I place an order for a cup of coffee and pay $2.00.
2. I receive a receipt to claim my coffee when it’s ready.
3. I can act on this receipt.
4. Eventually, I either receive my coffee, or I get sad news about my coffee, or everyone leaves and I don’t know...

Immutable

Once a promise resolves, it is immutable

Immutable

It can be either fulfilled, rejected, or unresolved

Completion Event

A temporal this-then-that sequence for async tasks

Completion Event

With callbacks, the notification would be the callback invoked by `foo()`

Completion Event

With promises, we expect that `foo()` will notify us when it is done, and then we can proceed accordingly

Completion Event

```
var evt = foo(42);
//bar listens to foo()’s evt completion
bar(evt);
//as does baz
baz(evt);
```

Promise Event

In the previous example, evt was an analogy for a Promise object

Promise Event

```
function foo(x){
    //start a long task
    //construct and return a promise
    return new Promise(function(resolve,reject){
        //eventually call either resolve()
        //or reject() which are the resolution
        //callbacks for the promise
    });
}

var p = foo(42); //Assign the promise to p
bar(p);
baz(p);

function bar(fooPromise){
    fooPromise.then(
        function(){
            //success, so do bar's task
        },
        function(){
            //error, deal with it
        });
}

//...same for baz()
```

Promise Event

```
function foo(x){
    //start a long task
    //construct and return a promise
    return new Promise(function(resolve,reject){
        //eventually call either resolve()
        //or reject() which are the resolution
        //callbacks for the promise
    });
}

var p = foo(42); //Assign the promise to p
p.then(bar,oopsBar);
p.then(baz,oopsBaz);

function bar(){
    //foo has succeeded, so do
    //bar's task
}

function oopsBar(){
    //foo failed, so handle this
}

//...same for baz()
```

Promise Trust

Promises establish a notion of trust when writing async code

Promise Trust

Promises establish a notion of trust when writing async code

Promise Trust

Callbacks can be called too early

Promise Trust

Callbacks can be called too early
or too late (or never)

Promise Trust

Callbacks can be called too early
or too late (or never)
or too few/many times

Promise Trust

Callbacks can be called too early
or too late (or never)
or too few/many times
may not pass along proper parameters

Promise Trust

Callbacks can be called too early
or too late (or never)
or too few/many times
may not pass along proper parameters
swallow any errors/exceptions that may occur

Promise Trust

Callbacks can be called too early

or too late (or never)

or too few/many times

may not pass along proper parameters

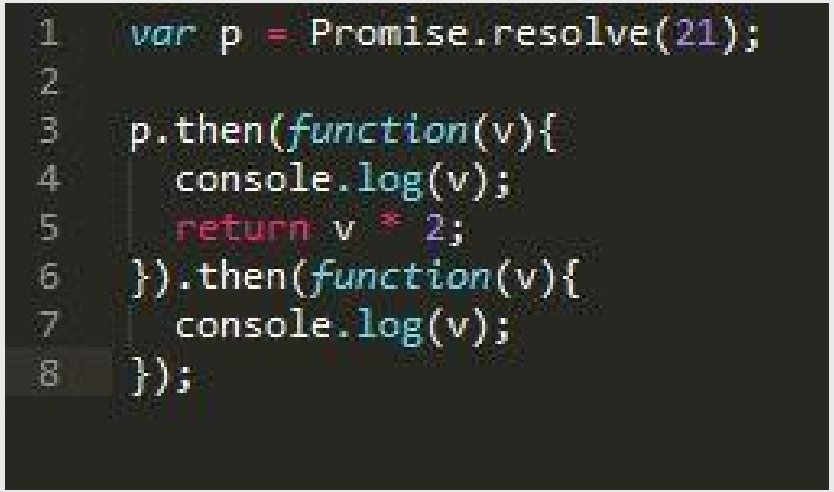
swallow any errors/exceptions that may occur

(Promises are engineered to solve all of these problems)

Chain Flow

.then() returns a Promise object, which can be used to chain promises together

Chain Flow


var p = Promise.resolve(21);
p.then(function(v){
console.log(v);
return v * 2;
}).then(function(v){
console.log(v);
});

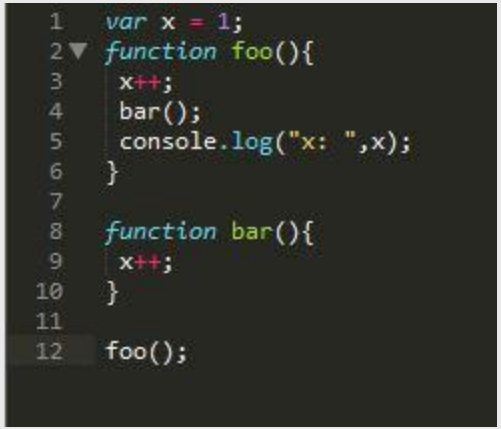
Is that it?

Are promises the only new async pattern we have?

Generators

ES6 gives us a new async patterns with function generators

Generators



```
1  var x = 1;
2  function foo(){
3   x++;
4   bar();
5   console.log("x: ",x);
6  }
7
8  function bar(){
9   x++;
10 }
11
12 foo();
```

Generators

```
1  var x = 1;
2  function foo(){
3   x++;
4   bar();
5   console.log("x: ",x);
6  }
7
8  function bar(){
9   x++;
10 }
11
12 foo();
```

```
1  var x = 1;
2  //declare a generator called foo
3  function * foo(){
4   x++;
5   yield;
6   console.log("x: ",x);
7  }
8
9  function bar(){
10  x++;
11 }
12
13 //assign the foo generator to an iterator called it
14 var it = foo();
15 it.next();
16 console.log("Before bar(), x is: ",x);
17 bar();
18 it.next();
```

iterators (var it in this case) control the execution of generators.

Why?

Javascript is seeing a steady rise in popularity

Why?

If it is to continue this success, Javascript needs to move away from callbacks