Javascript Async Demystified

By Joe Sutton

Road to Demystifying Javascript Async

- Confusion/Problem
- Why
- Solutions

Confusion/Problem

```
console.log('before');
setTimeout(function() {
    console.log('async done');
}, 0);
console.log('after');
```

Confusion/Problem

```
console.log('before');
setTimeout(function() {
   console.log('async done');
}, 0);
console.log('after');


----------------------
before
after
async done
```

Confusion/Problem

So Why?

Why

- Javascript
  - Non-Blocking I/O [1]
  - Single Threaded [2]
  - Function Callbacks
  - Event Loop

1) NodeJS has blocking **Sync** functions
2) The **Worker** thread API allows code to run in another thread.

Why

Non-Blocking I/O

Why: What is Non-Blocking I/O?

- Non-Blocking
  - Refers to code that doesn't block execution
- I/O - Input/Output
  - HTTP Request (fetch, XMLHttpRequest...)
  - Read from and Writing to File(s)
  - localStore, sessionStorage, geolocation...
  - User Input
    - Keyboard, Mouse, Input Form...

Why

Non-Blocking

```
console.log('before');
nonBlocking_SetTimeout(function() {
   console.log('async done');
}, 1);
console.log('after');
----------------------
before
after
async done
```

Blocking

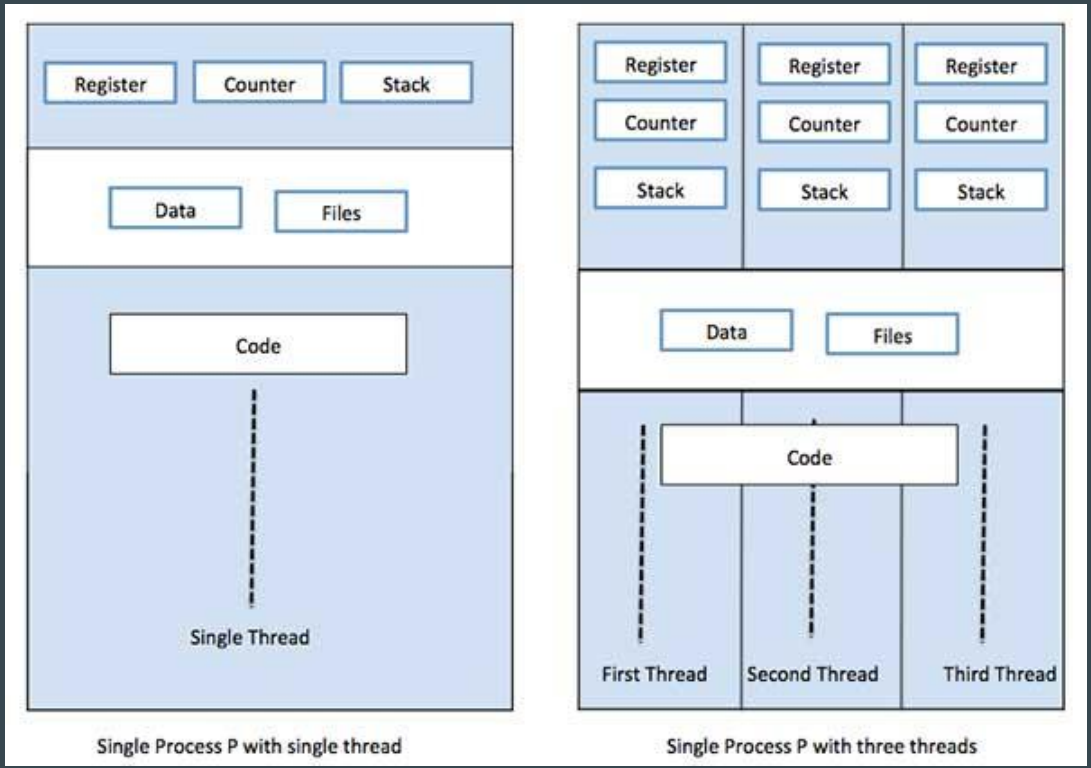
```
console.log('before');
blocking_SetTimeout(1);
console.log('async done');

console.log('after');
----------------------
before
async done
after
```

Why: What is Single-Threaded vs Multi?

- Single-threaded processes execute instructions in a single sequence. In other words, one command is processed at a time.
- Multi-threaded processes execute instructions on multiple sequences at the exact same time.

Why: Single-Thread vs Multi-Thread



Why: Multi-Thread vs Multi-Tasking

Why: Single vs Multi-Threaded

Single-Threaded

Multi-Threaded

Why

Function Callbacks

Why: Function Callbacks - Example

```
console.log('before');
myAsyncFunc(function() {
    console.log('async done');
});
console.log('after');

// ------------------------------------
function myAsyncFunc(callback, timeoutSeconds=1) {
    setTimeout(callback, timeoutSeconds * 1000);
}
```

Why

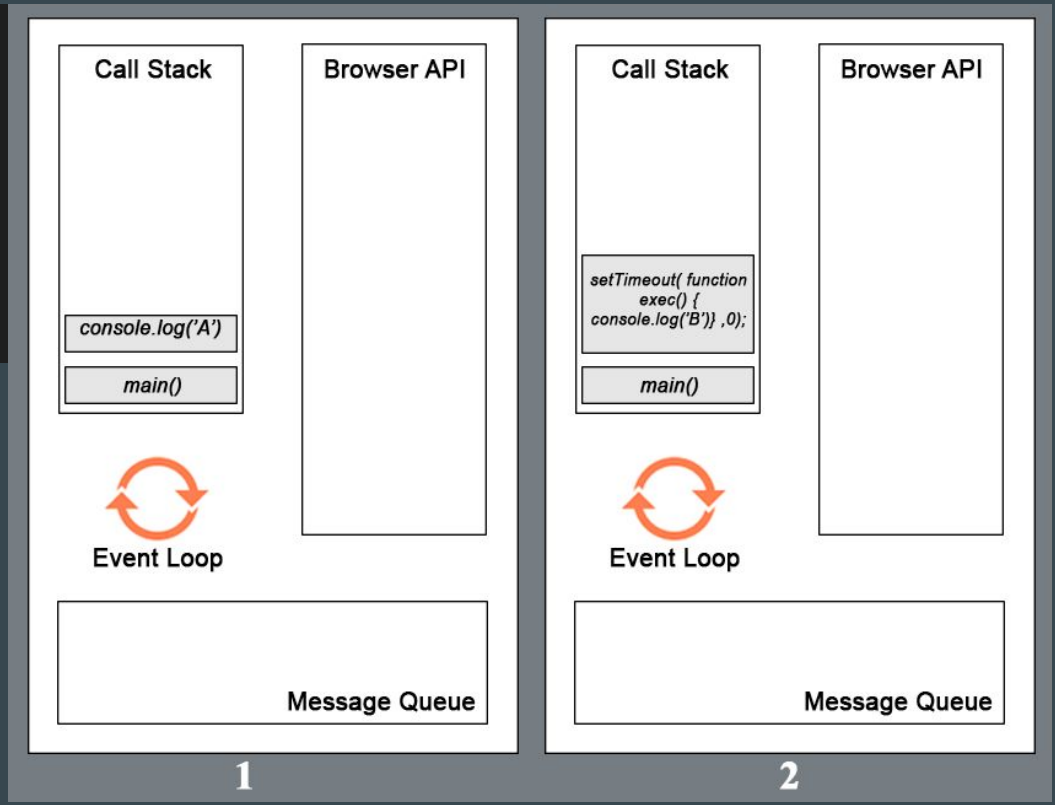
Event Loop

Why: What is the Event Loop?

- Technical
  - Executing code and queued sub-tasks
  - Collecting and processing events
  - https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop
- Simply
  - Orchestrator for async operations

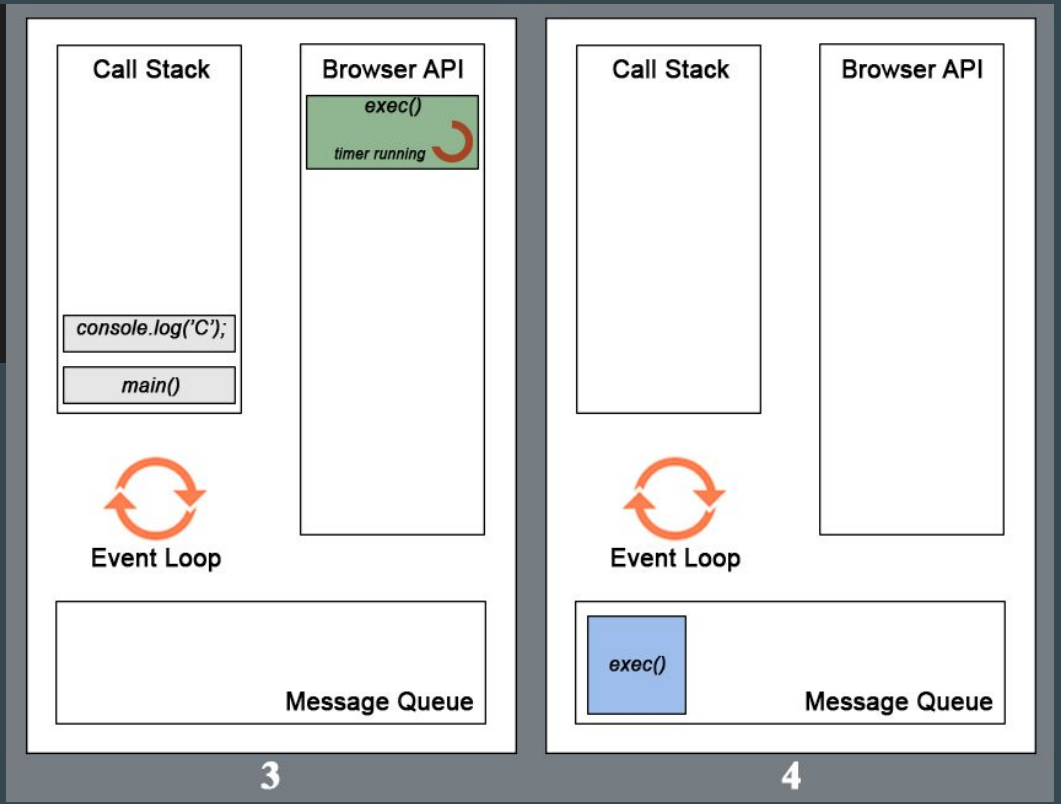
Why: Event Loop + Message Queue

```
1  console.log('A');
2  setTimeout(function() {
       console.log('B');
   }, 0);
   console.log('C');
```



Why: Event Loop + Message Queue

```
1   console.log('A');
2,4 setTimeout(function() {
        console.log('B');
    }, 0);
3   console.log('C');
```



Why: Event Loop + Message Queue

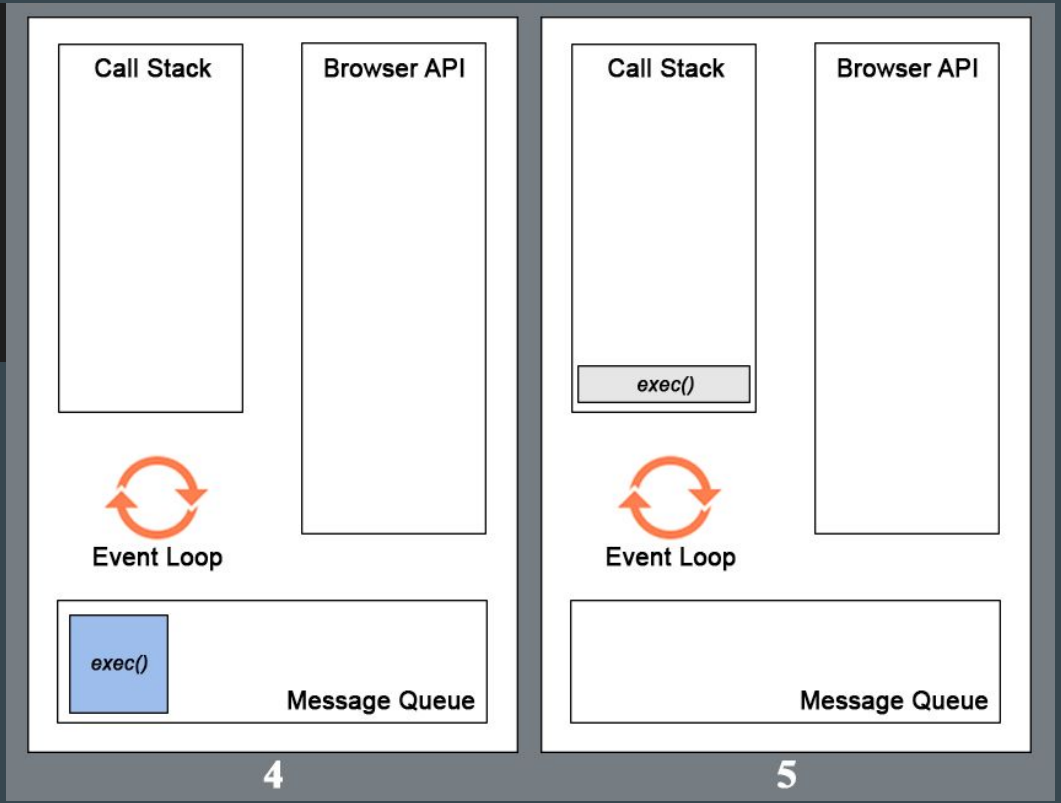
1 `console.log('A');`

2,4 `setTimeout(function() {`

5 `  console.log('B');`

`}, 0);`

3 `console.log('C');`



Why: Event Loop - Example 1

```
console.log('before');
myAsyncFunc(function() {
   console.log('async done 1');
}, 1 /* delay 1 seconds */);
myAsyncFunc(function() {
   console.log('async done 2');
}, 2 /* delay 2 seconds */);
console.log('after');
-----------------------
before
after
async done 1
async done 2
```

Why: Event Loop - Example 2

```
console.log('before');
myAsyncFunc(function() {
   console.log('async done 1');
}, 2 /* delay 2 seconds */);
myAsyncFunc(function() {
   console.log('async done 2');
}, 1 /* delay 1 seconds */);
console.log('after');
-----------------------
before
after
async done 2
async done 1
```

Why: Event Loop - Example 3

```
console.log('before');
myAsyncFunc(function() {
   console.log('async done 1');
   myAsyncFunc(function() {
       console.log('async done 2');
   }, 1 /* delay 1 seconds */);
}, 2 /* delay 2 seconds */);
console.log('after');
-----------------------
before
after
async done 1
async done 2
```

Why: Event Loop - Example 4

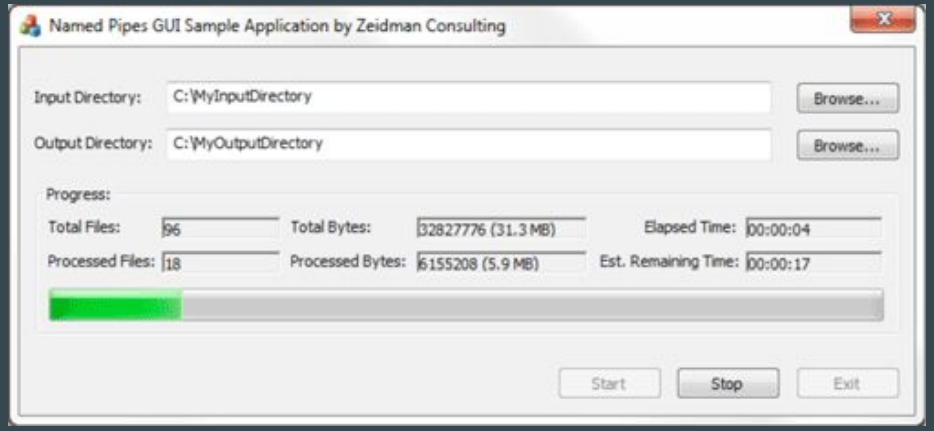
```
console.log('before');
myAsyncFunc(function() {
   console.log('async done 1');
}, 1 /* delay 1 seconds */);
myAsyncFunc(function() {
   console.log('async done 2');
}, 1 /* delay 1 seconds */);
console.log('after');
-----------------------
before
after
async done 1
async done 2
```

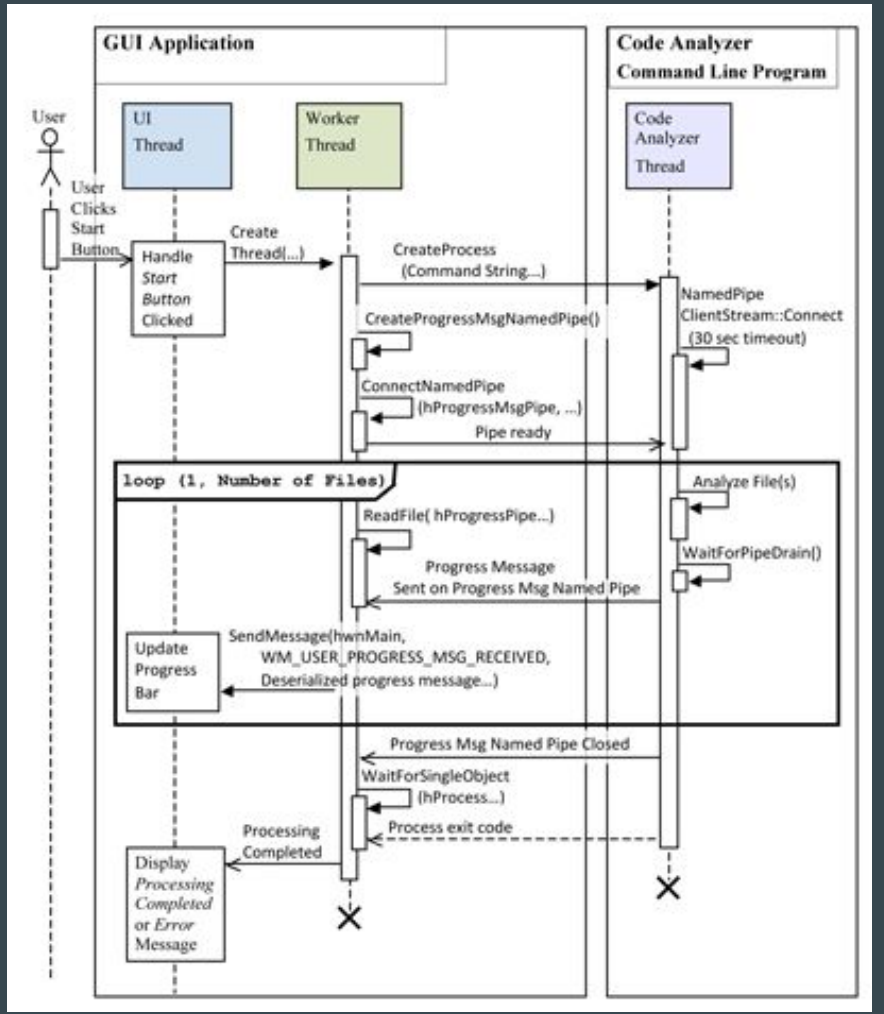
But Why?

- Other Languages, Example C++:
  - Blocking I/O
  - Single and Multi-Threaded
  - Don’t require use of Function Callbacks
  - Event Loop not built in

Why: GUI - Graphical User Interface

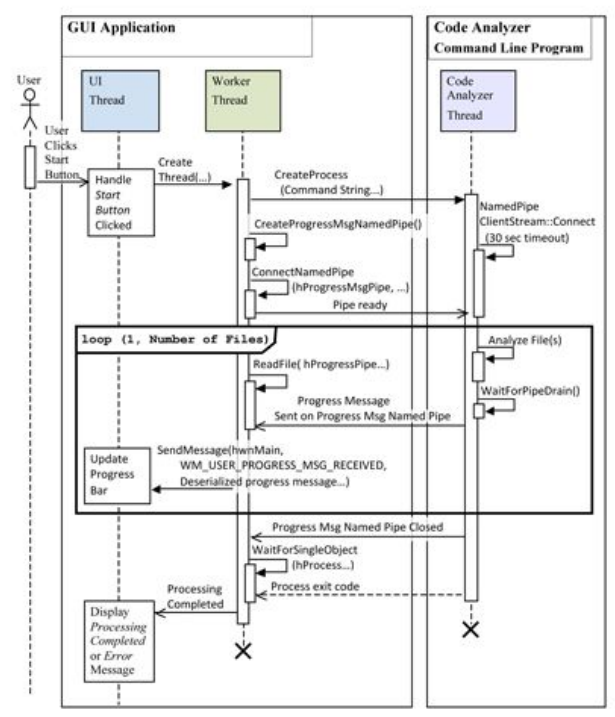
Why: GUI Complexity





https://docs.microsoft.com/en-us/cpp/windows/walkthrough-creating-windows-desktop-applications-cpp?view=vs-2019

Why: Complexity to Async (Confusion/Problem)




```
document.getElementById("orderBtn")
.onclick = function() {
  sendOrder(product)
  .then(function(result) {
     gotoConfirmPage(result);
  })
  .catch(function(err) {
     gotoErrorPage(err);
  })
};
```

Why: Solving Complexity

- Non-Blocking I/O
- Single Threaded
- Function Callbacks
- Event Loop

Why

Questions?

Solution

- Promise
  - What is it really?
- ES6 Async/Await
  - What is it and how is it different from Promises
  - When not to use Async/Await?

Solution: Promise

- Is a **Library**
- Code to streamline the callback nesting “tree” in larger web apps
- Normalizes asynchronous response (resolve, reject)
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

Solution: Promise - Example 1

```
console.log('before');
var promise1 = new Promise(function (resolve, reject){
   setTimeout(function() {
       resolve('async done 1');
   }, 1);
});
promise1.then(function (value){
   console.log(value);
});
console.log('after');
-----------------------
before
after
async done 1
```

Solution: Promise - Example 2

```
console.log('before');
myAsyncFunc(1).then(function (value){
  console.log('async', value, 'done');
});
console.log('after');

// ----------------------------------
function myAsyncFunc(timeoutSec=1) {
  return new Promise(function(resolve, reject) {
     setTimeout(function() {
        resolve(timeoutSec);
     }, timeoutSec * 1000);
  });
}
```

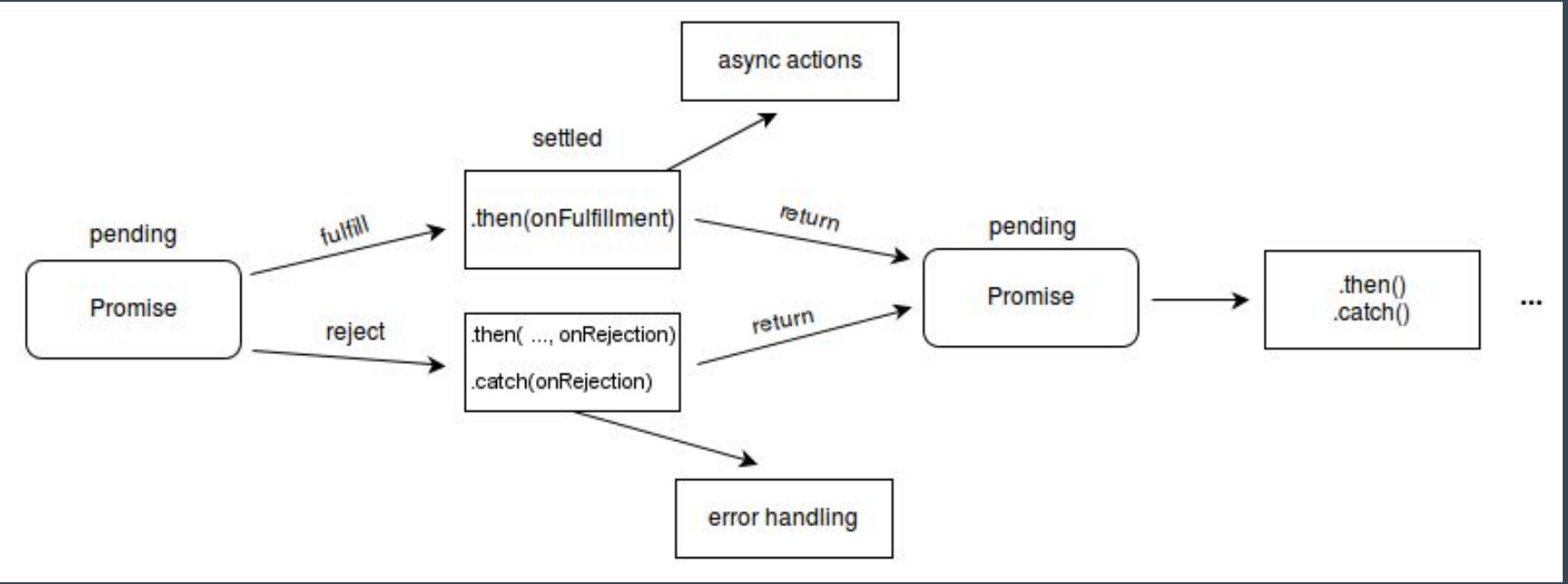
Solution: Convert to Promises

```
myAsyncFunc(function(){
  console.log('async 1 done');
  myAsyncFunc(function(){
    console.log('async 2 done');
    myAsyncFunc(function(){
      console.log('async 3 done');
      // ...
    }, 3);
  }, 2);
}, 1);


function myAsyncFunc(callback, timeoutSec=1) {
  setTimeout(callback, timeoutSec * 1000);
}
```

```
myAsyncFunc(1)
.then(function(){
  console.log('async 1 done');
  return myAsyncFunc(2);
})
.then(function(){
  console.log('async 2 done');
  return myAsyncFunc(3);
})
.then(function(){
  console.log('async 3 done');
  return myAsyncFunc(4);
});
function myAsyncFunc(timeoutSec=1) {
  return new Promise(function(resolve, reject) {
      setTimeout(resolve, timeoutSec * 1000);
  }); }
```

Solution: Promise Flow



https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

Solution: Promise - Building a mini Library!

- github.com/**jstty/async-promise/tree/master/promiselib**

Solution: Promise - Building a mini Library!

- Missing features
  - **.finally**
  - **.all**
  - **.any**

Solution

Async/Await

Solution: Async/Await

- Differences from Promise
  - Async/Await is language syntax, Promise is a library
  - Blocking like behavior
  - Improves code readability

Solution: ES6 Async/Await vs Promise

Promise

```
console.log('before');
myPromiseFunc().then(function (){
   console.log('promise done');
});
console.log('after');


----------------------
before
after
promise done
```

Async/Await

```
console.log('before');
await myAsyncFunc();
console.log('await done');
console.log('after');



----------------------
before
await done
after
```

Solution: ES6 Async/Await

Async/Await does not blocking all execution

```
console.log('before');
myAsyncFunc(1).then(function (){
   console.log('promise done');
});
await myAsyncFunc(2);
console.log('await done');
console.log('after');
----------------------
before
promise done
await done
after
```

Solution: ES6 Async/Await - Problem(s)

Nested callback functions, Example: forEach

```
let list = ['await done 1', 'await done 2'];
console.log('before');
list.forEach(async (item) => {
  await myAsyncFunc(1);
  console.log(item);
});
console.log('after');
-----------------------
before
after
await done 1
await done 2
```

Solution: ES6 Async/Await - Solution(s)

Nested callback functions, use **for loop** instead

```
let list = ['await done 1', 'await done 2'];
console.log('before');
for(let idx in list) {
  await myAsyncFunc(1);
  console.log(list[idx]);
}
console.log('after');
----------------------
before
await done 1
await done 2
after
```

Solution: ES6 Async/Await - Error Handling

Promise

```
myAsyncFunc()
.then(() => {
  console.log('promise done');
},
(err) => {
  console.log('promise error:', err);
})
.catch((err) => {
  console.log('catch error:', err);
});
----------------------
promise error: something bad happened
```

Async/Await

```
try {
  await myAsyncFunc();
}
catch(err) {
  console.log('catch error:', err);
}
----------------------
catch error: something bad happened
```

Solution: ES6 Async/Await

- When to use Promises over Async/Await
  - Control concurrency/optimizations
    - Request from multiple sources at the same time
- Async returns a Promise
  - Mix and match to your needs

Solution

Questions?

Conclusion

- Confusion/Problem
- Why
- Solutions

- You can find all the code/slides on github
  - github.com/**jstty/async-promise**

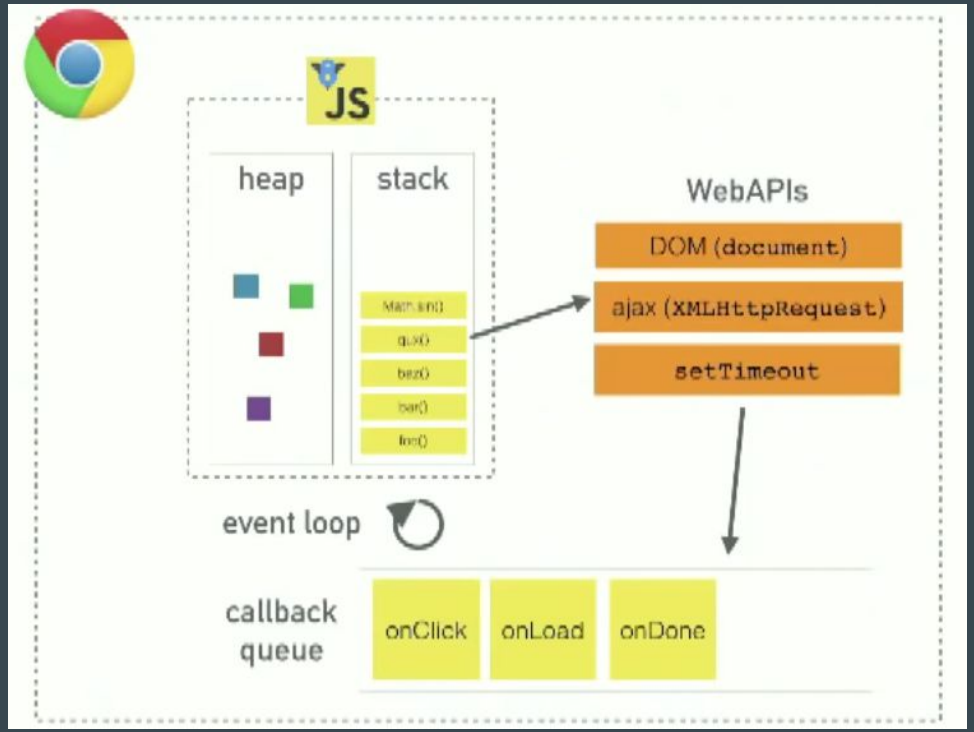
Thank you!

Joe Sutton

joe@jstty.com

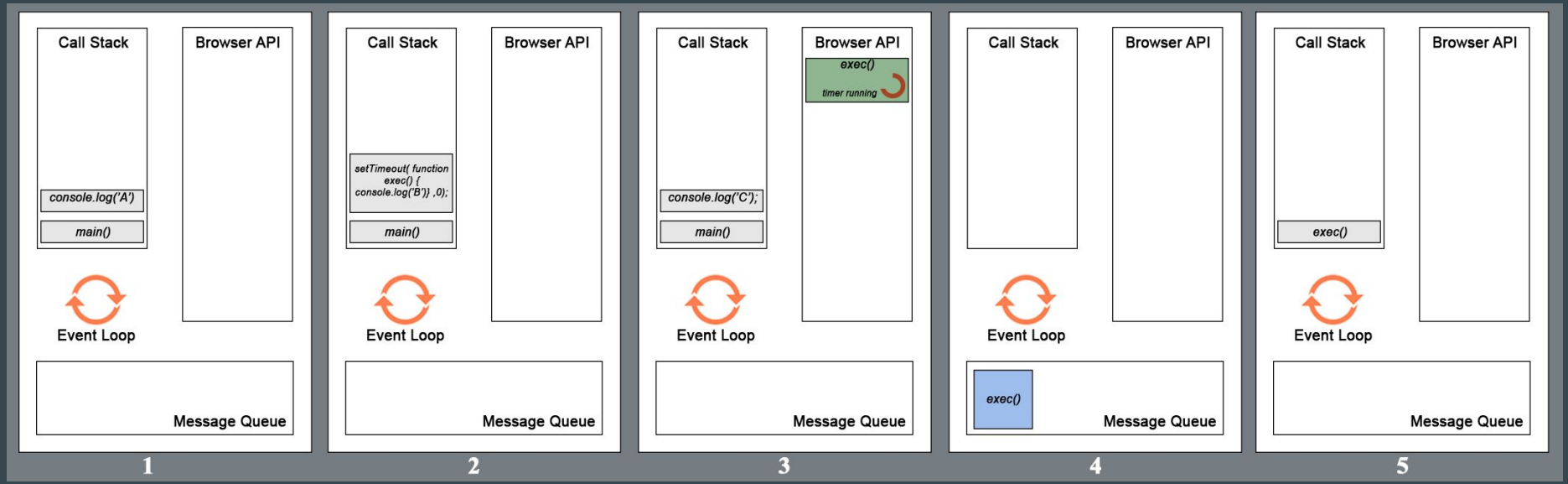
More...

Why: Event Loop + Message Queue



https://medium.com/front-end-weekly/javascript-event-loop-explained-4cd26af121d4

Why: Event Loop + Message Queue



https://medium.com/front-end-weekly/javascript-event-loop-explained-4cd26af121d4