

2. Ecosystème NodeJS

Année universitaire
2021-2022

4 TWIN



► Plan

1. Traitement Asynchrone
2. Modèle I/O non-bloquant (Non-blocking I/O model)
3. Modèle guidé par les événements (Event-driven Model)
4. Call Stack et Event loop
5. Création de serveurs en Node JS



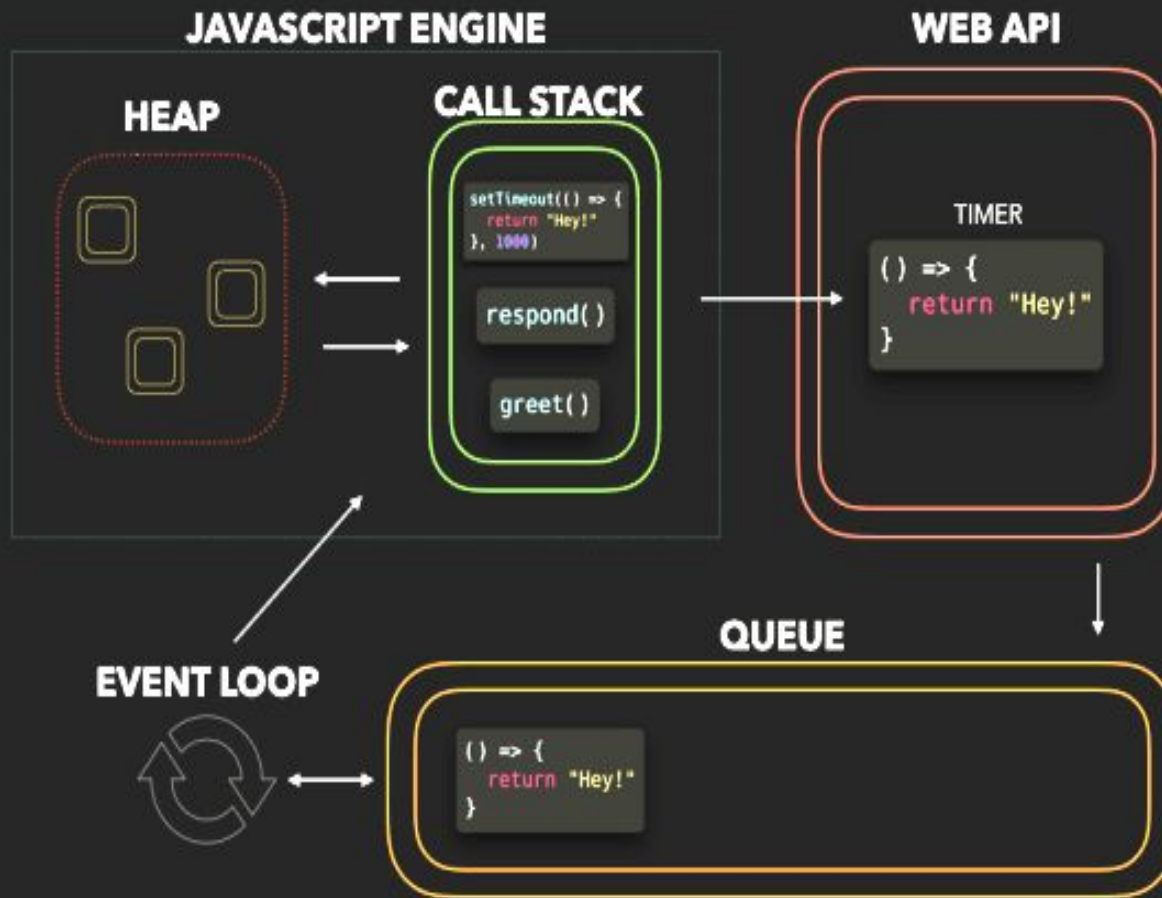
Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world.



▶ 1. Traitement Asynchrone

- En JavaScript une ligne du code est exécutée de façon synchrone.
- Il est possible d'exécuter du code de manière asynchrone.
- La fonction à exécuter en mode asynchrone sera placée dans une file d'attente qui contient les fonctions à exécuter.
- ☐ C'est ce qu'on appelle « **l'event loop** »

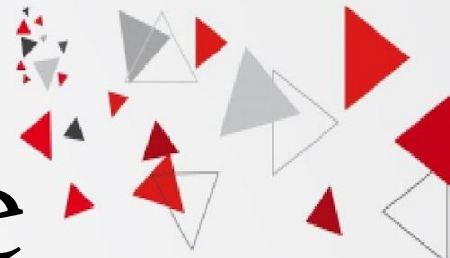
► 1. Traitement Asynchrone



```
function greet() {  
  return "Hello!"  
}  
  
function respond() {  
  return setTimeout(() => {  
    return "Hey!"  
  }, 1000)  
}  
  
greet()  
respond()
```

Exemple visualisé : <https://dev.to/lydiahallie/javascript-visualized-event-loop-3dif>

▶ 1. Traitement Asynchrone



- **setTimeout()** est la fonction la plus utilisée lorsqu'on cherche à exécuter du code asynchrone sans bloquer les autres fonctions en cours d'exécution.
- Cette fonction accepte 2 paramètres
 - La fonction à exécuter de manière asynchrone (qui sera ajoutée à la file d'attente de l'event loop) ;
 - Le délai, avant d'exécuter la fonction (en millisecondes).

```
console.log('Début')  
// Délai de 2 secondes avant de lancer l'application  
setTimeout(() => {  
  console.log('Voici 2 secondes d\'attente')  
}, 2000)  
console.log('Fin')
```

▶ 1. Traitement Asynchrone

- Il existe d'autres méthodes un peu moins répandues par rapport à la fonction `setTimeout()`.
- **setImmediate**: Prend un seul paramètre (la fonction à exécuter) et permet d'exécuter ce paramètre en mode synchrone.
- **setInterval** : elle fonctionne exactement comme `setTimeout`, elle exécute la fonction passé en paramètre en boucle pour une période. Pour arrêter l'exécution en boucle il suffit de passer le retour de cette fonction à la fonction « **clearInterval** »
- **clearInterval**: Permet d'arrêter l'exécution de la fonction **setInterval**.
- **clearTimeout**: Permet d'annuler l'exécution asynchrone de la fonction `setTimeout()`.



▶ 2. Non-blocking I/O model

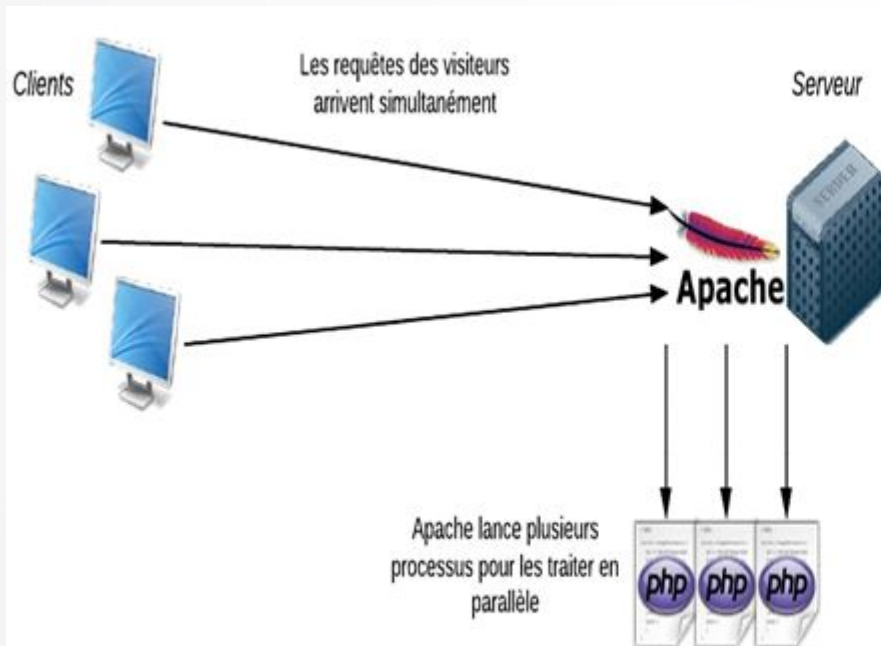
- Node JS est **Monothread**, mais il utilise un modèle **Non bloquant**

?

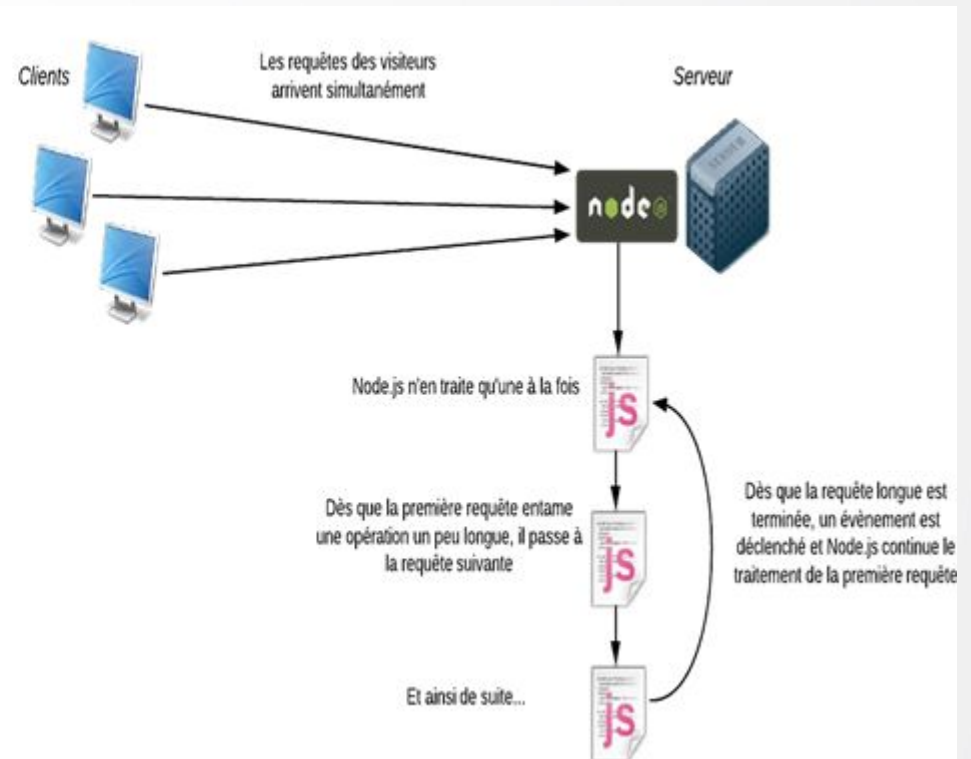
2. Non-blocking I/O model

Node.js :Single Thread

- Apache est multithread



- Node.js est mono-thread





▶ 2. Non-blocking I/O model

Simulation: Handling Requests

Mode Synchrone

2. Non-blocking I/O model

Simulation: Handling Requests

Mode Sychrone



Client 1



Server



Client 2



Client 3



DB

2. Non-blocking I/O model

Simulation: Handling Requests

Mode Sychrone



Client 1



Client 2



Client 3

Server



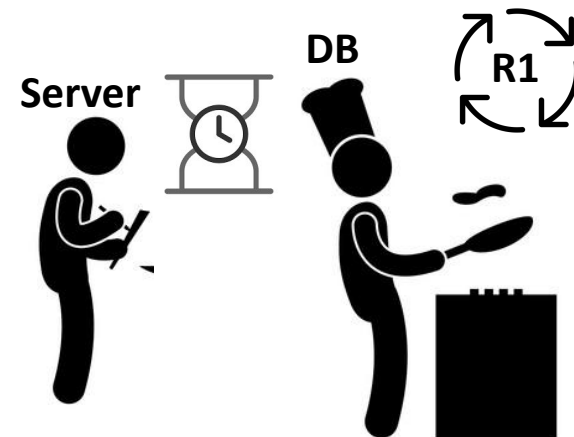
DB



2. Non-blocking I/O model

Simulation: Handling Requests

Mode Sychrone



2. Non-blocking I/O model

Simulation: Handling Requests

Mode Sychrone



Client 1



Client 2



Client 3



2. Non-blocking I/O model

Simulation: Handling Requests

Mode Sychrone



Client 1



Client 2



Client 3

Server



DB

2. Non-blocking I/O model

Simulation: Handling Requests

Mode Sychrone



Client 1



Client 2



Client 3

Server



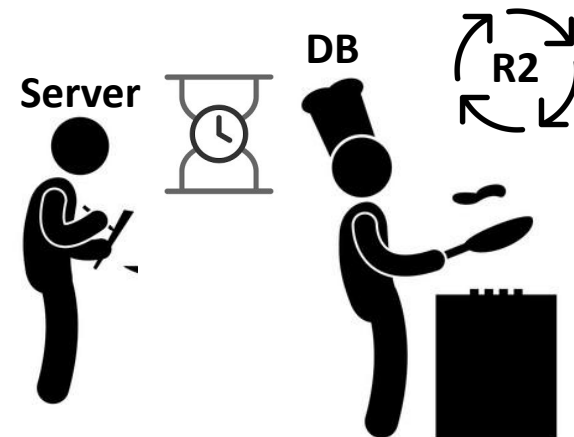
DB



2. Non-blocking I/O model

Simulation: Handling Requests

Mode Sychrone



2. Non-blocking I/O model

Simulation: Handling Requests

Mode Sychrone



Client 1



Client 2



Client 3



Server



DB



▶ 2. Non-blocking I/O model

Simulation: Handling Requests

Mode Asynchrone

2. Non-blocking I/O model

Simulation: Handling Requests

Mode Asynchrone



Client 1



Server



Client 2



Client 3



DB

2. Non-blocking I/O model

Simulation: Handling Requests

Mode Asynchrone



Client 1



Client 2



Client 3

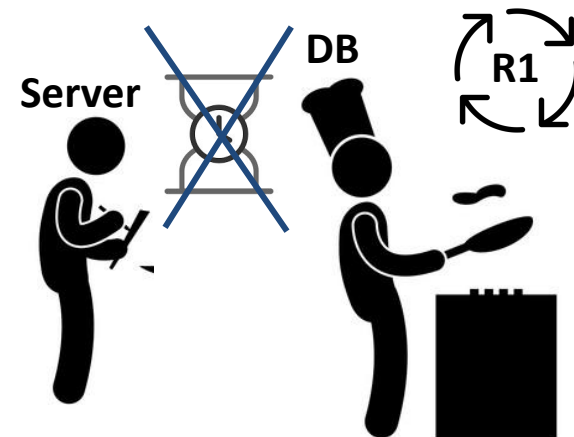
Server



2. Non-blocking I/O model

Simulation: Handling Requests

Mode Asynchrone



2. Non-blocking I/O model

Simulation: Handling Requests

Mode Asynchrone



Client 1



Client 2



Client 3

Server



2. Non-blocking I/O model

Simulation: Handling Requests

Mode Asynchrone



Client 1

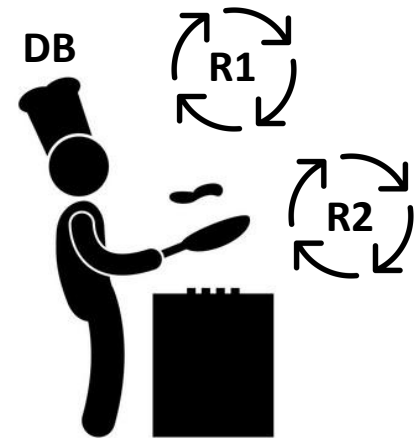


Client 2



Client 3

Server



2. Non-blocking I/O model

Simulation: Handling Requests

Mode Asynchrone



Client 1

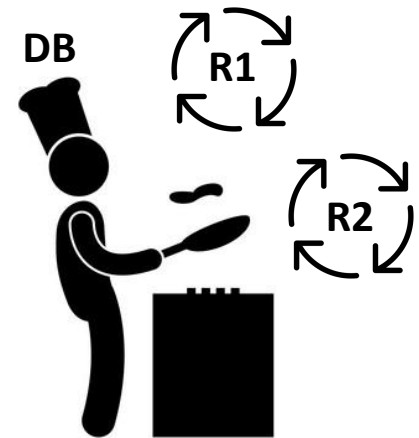


Client 2



Client 3

Server



2. Non-blocking I/O model

Simulation: Handling Requests

Mode Asynchrone



Client 1

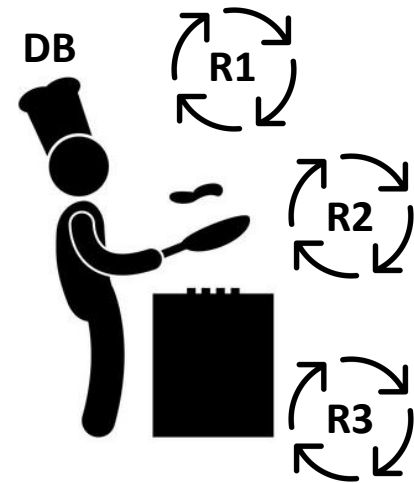


Client 2



Client 3

Server



2. Non-blocking I/O model

Simulation: Handling Requests

Mode Asynchrone



Client 1



Client 2

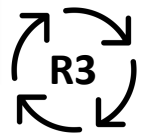
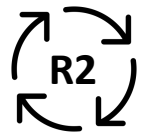


Client 3

Server



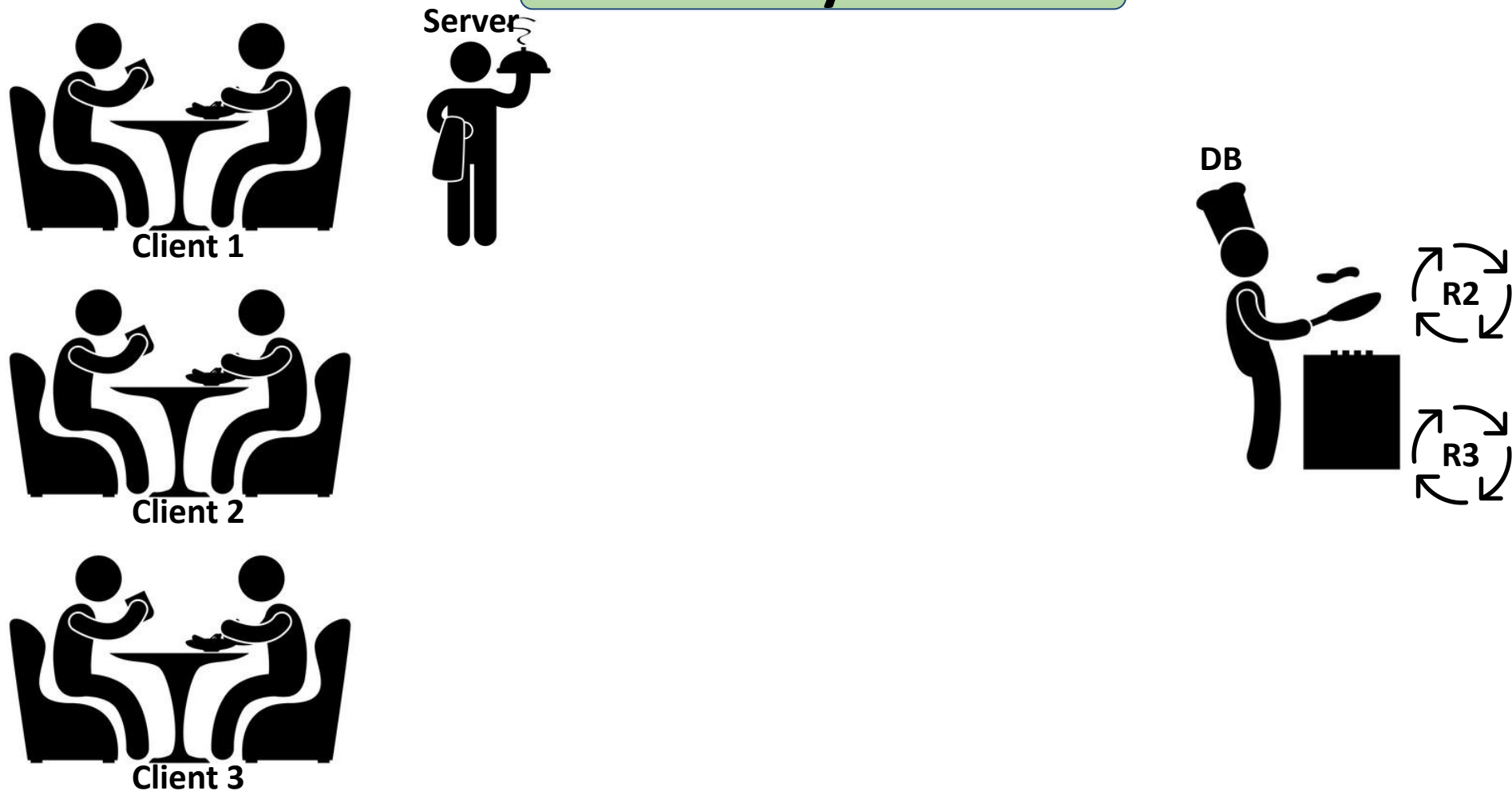
DB



2. Non-blocking I/O model

Simulation: Handling Requests

Mode Asynchrone



2. Non-blocking I/O model

Simulation: Handling Requests

Mode Asynchrone



Client 1



Client 2



Client 3

Server



DB



2. Non-blocking I/O model

Simulation: Handling Requests

Mode Asynchrone



Client 1



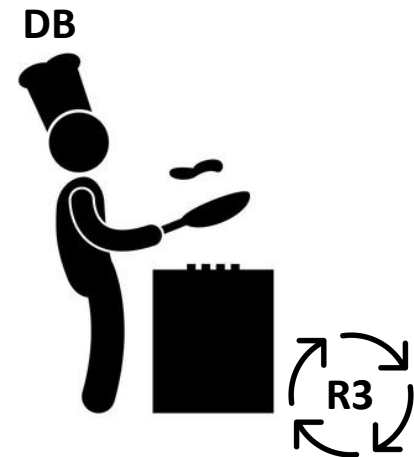
Client 2



Client 3



Server

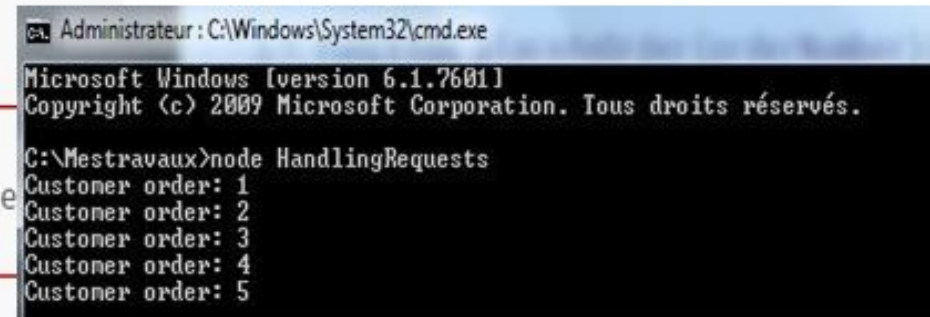


DB

2. Non-blocking I/O model

Simulation: Handling Requests

```
1 function placeAnOrder(orderNumber){  
2   console.log("Customer order:", orderNumber);  
3  
4   cookAndDeliverFood(function(){  
5     console.log("Delivered food order:", orderNumber);  
6   })  
7 }
```



```
Administrateur : C:\Windows\System32\cmd.exe  
Microsoft Windows [version 6.1.7601]  
Copyright (c) 2009 Microsoft Corporation. Tous droits réservés.  
  
C:\Mestravaux>node HandlingRequests  
Customer order: 1  
Customer order: 2  
Customer order: 3  
Customer order: 4  
Customer order: 5
```



```
13  
14 //Simulate a 5 second operation  
15 function cookAndDeliverFood(callback){  
16   setTimeout(callback, 5000);  
17 }
```

```
18  
19 //Simulate users web request  
20 placeAnOrder(1);  
21 placeAnOrder(2);  
22 placeAnOrder(3);  
23 placeAnOrder(4);  
24 placeAnOrder(5);
```



```
C:\Mestravaux>node HandlingRequests  
Customer order: 1  
Customer order: 2  
Customer order: 3  
Customer order: 4  
Customer order: 5  
Delivered food order: 1  
Delivered food order: 2  
Delivered food order: 3  
Delivered food order: 4  
Delivered food order: 5
```

2. Non-blocking I/O model

The image shows two side-by-side code editors comparing blocking and non-blocking I/O models. The left editor, titled 'blocking.js', contains synchronous code where the second user lookup and the sum calculation wait for the first lookup to complete. The right editor, titled 'non-blocking.js', contains asynchronous code using callbacks, allowing the second lookup and the sum calculation to proceed immediately after the first lookup. Handwritten labels 'Blocking' and 'Non-blocking' with arrows point to their respective code blocks.

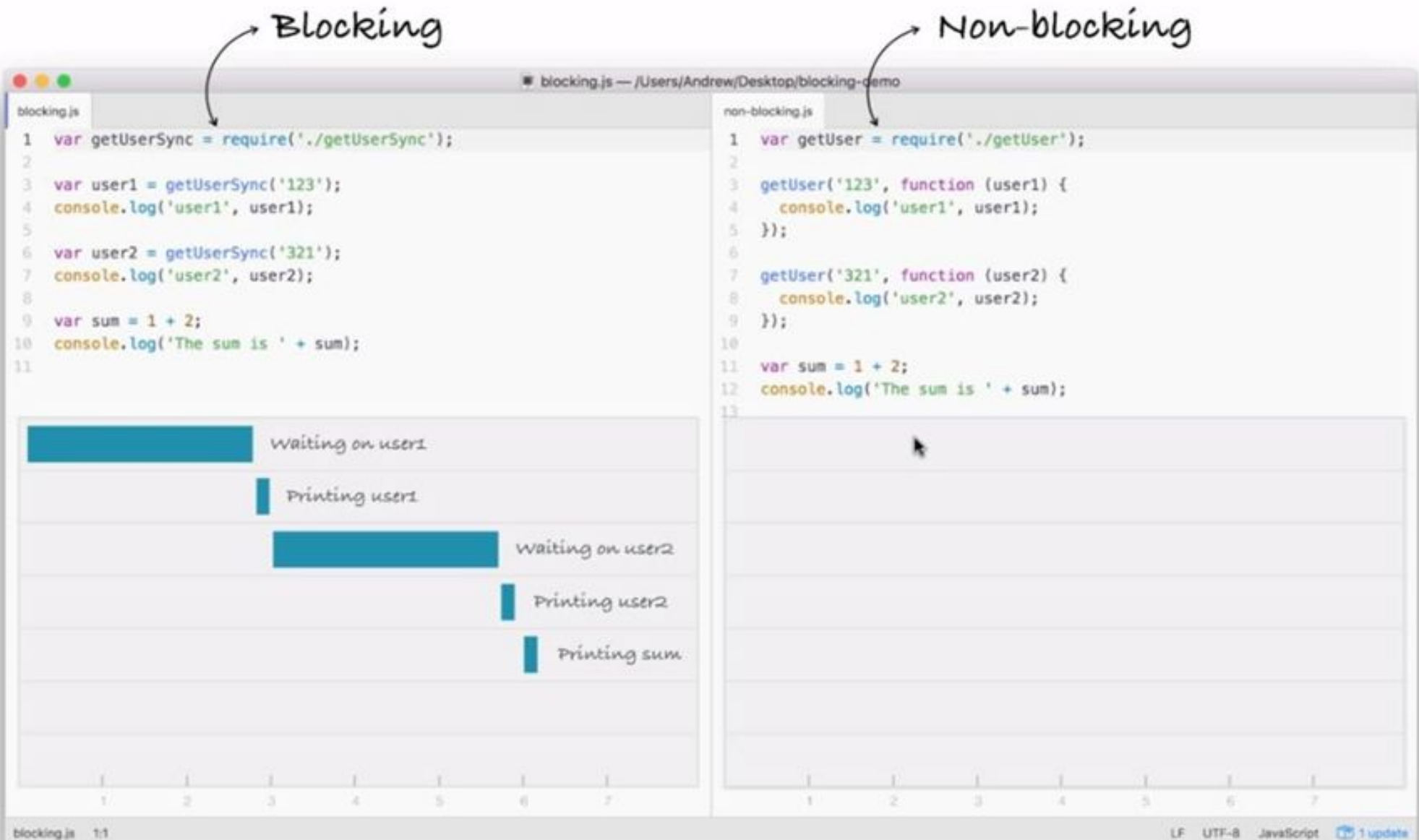
Blocking

```
1 var getUserSync = require('./getUserSync');
2
3 var user1 = getUserSync('123');
4 console.log('user1', user1);
5
6 var user2 = getUserSync('321');
7 console.log('user2', user2);
8
9 var sum = 1 + 2;
10 console.log('The sum is ' + sum);
11
```

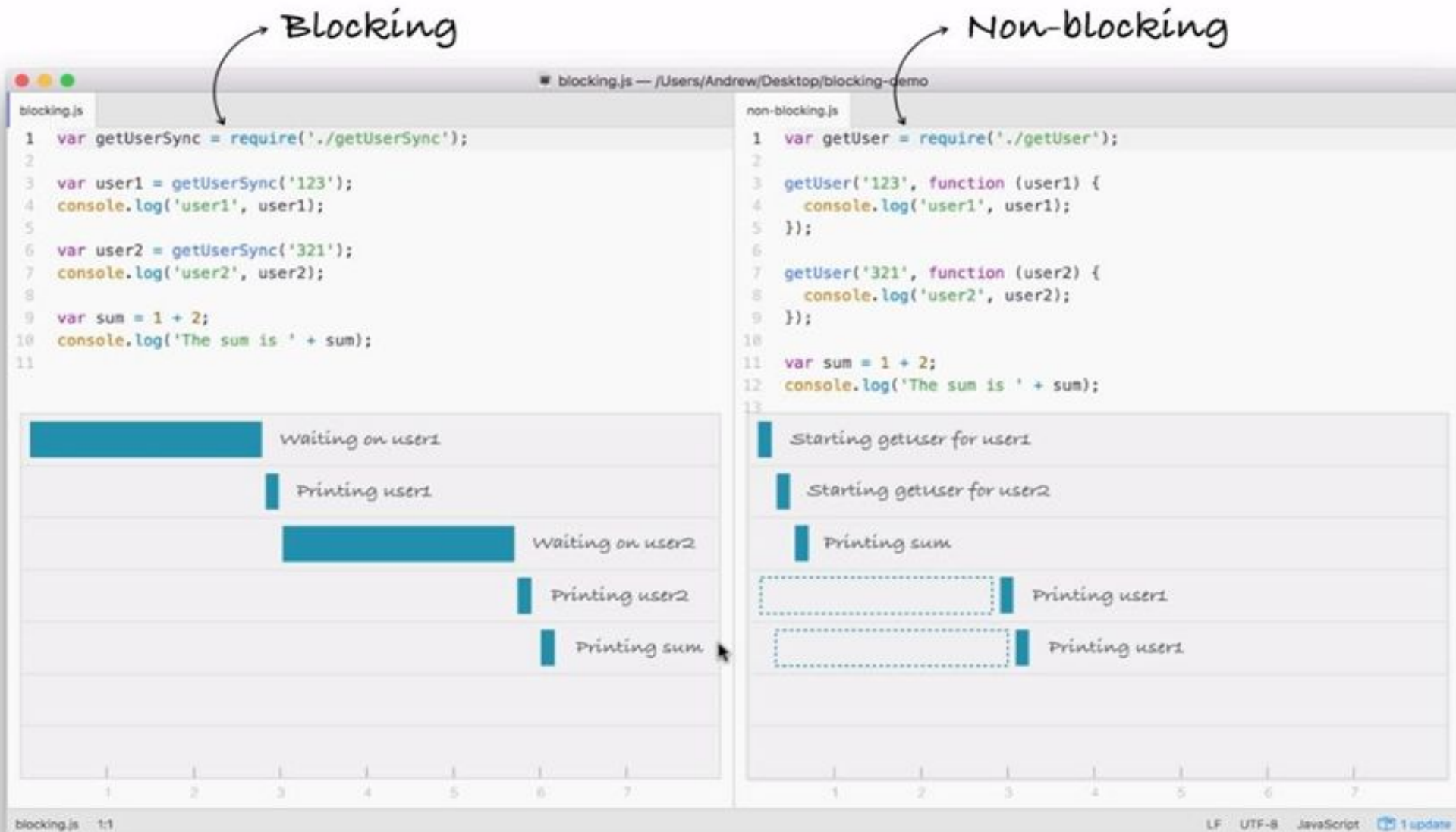
Non-blocking

```
1 var getUser = require('./getUser');
2
3 getUser('123', function (user1) {
4   console.log('user1', user1);
5 });
6
7 getUser('321', function (user2) {
8   console.log('user2', user2);
9 });
10
11 var sum = 1 + 2;
12 console.log('The sum is ' + sum);
13
```

2. Non-blocking I/O model

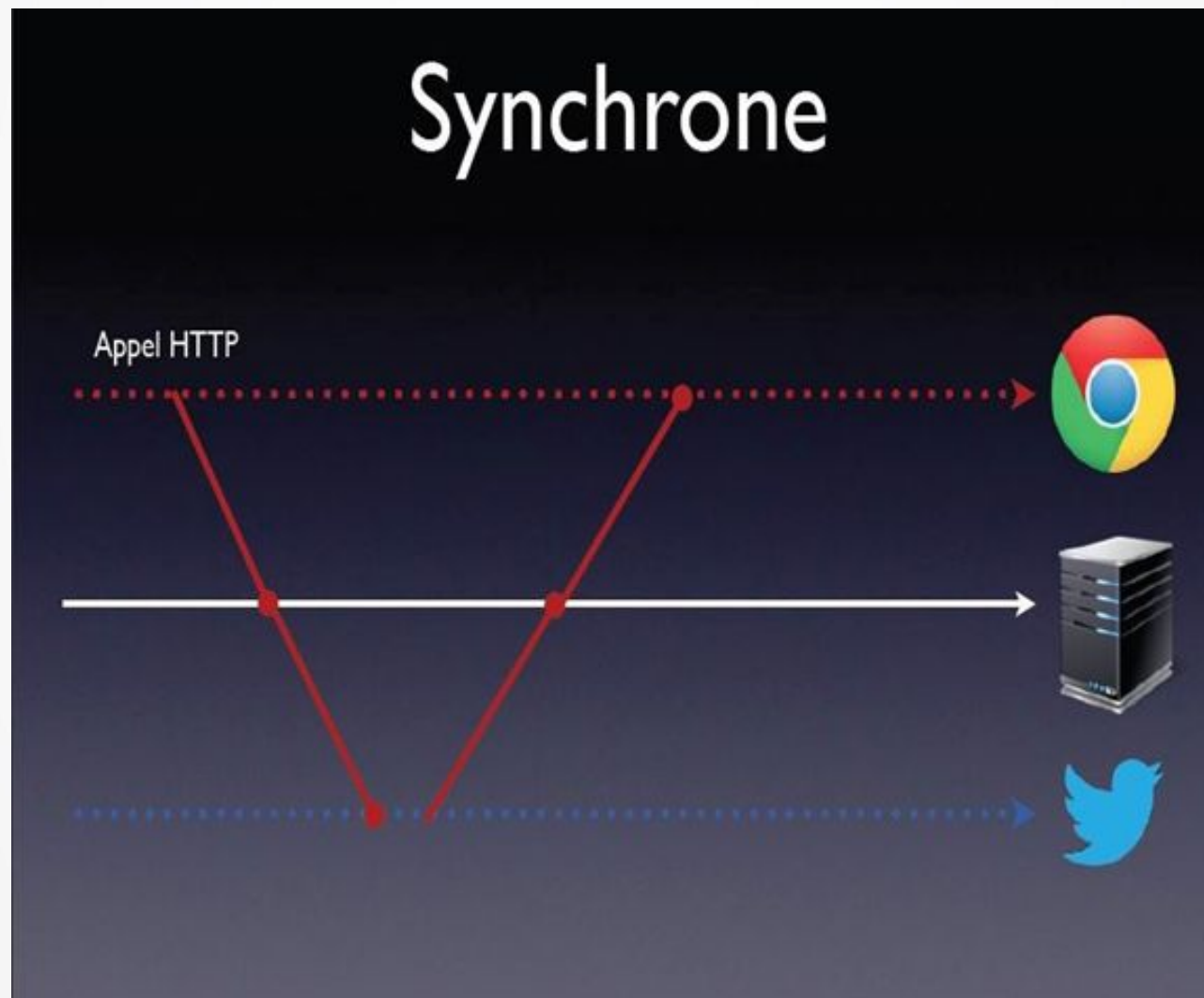


2. Non-blocking I/O model



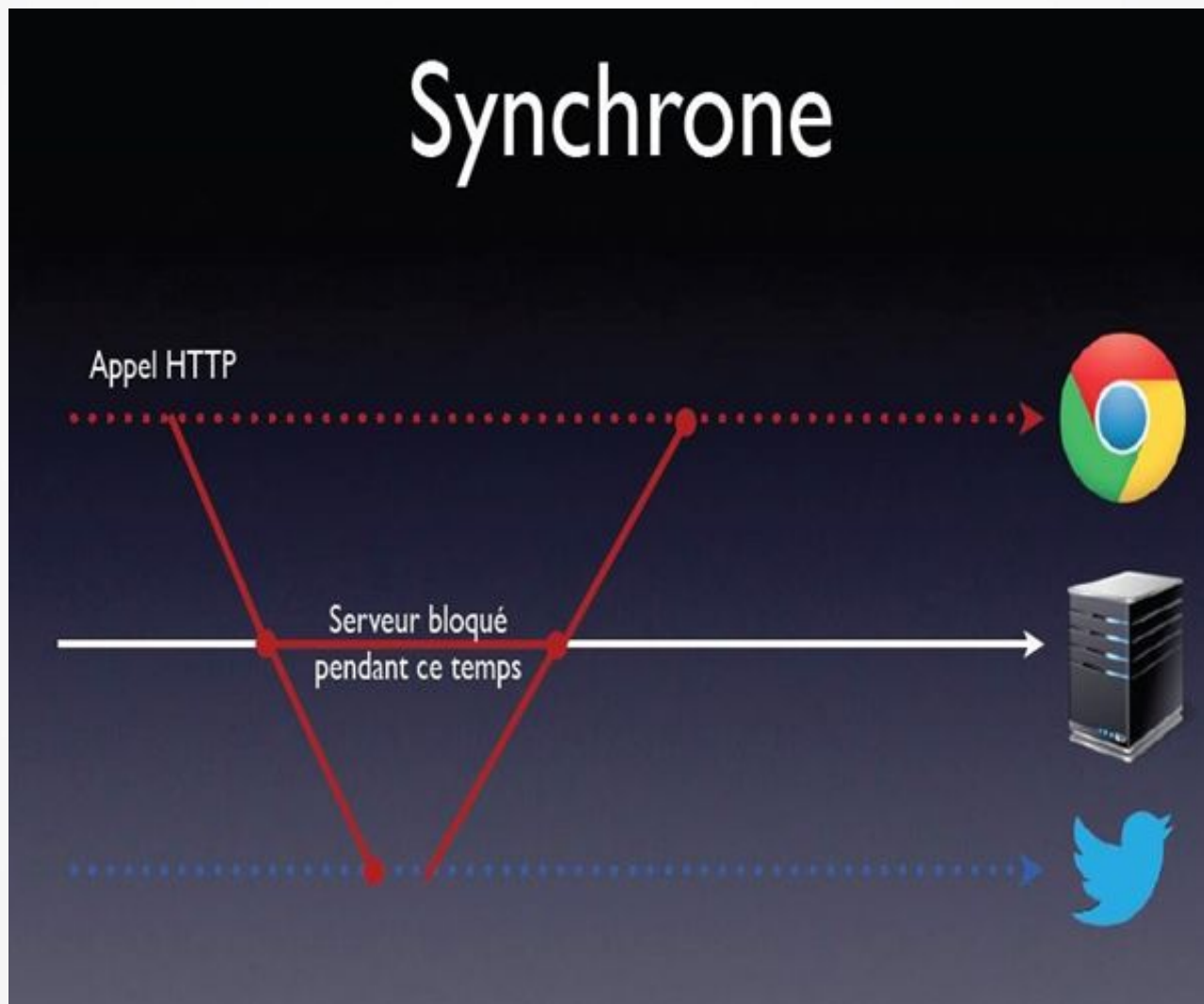
► 2. Non-blocking I/O model

Simulation : Asynchrone-Mode non-bloquant



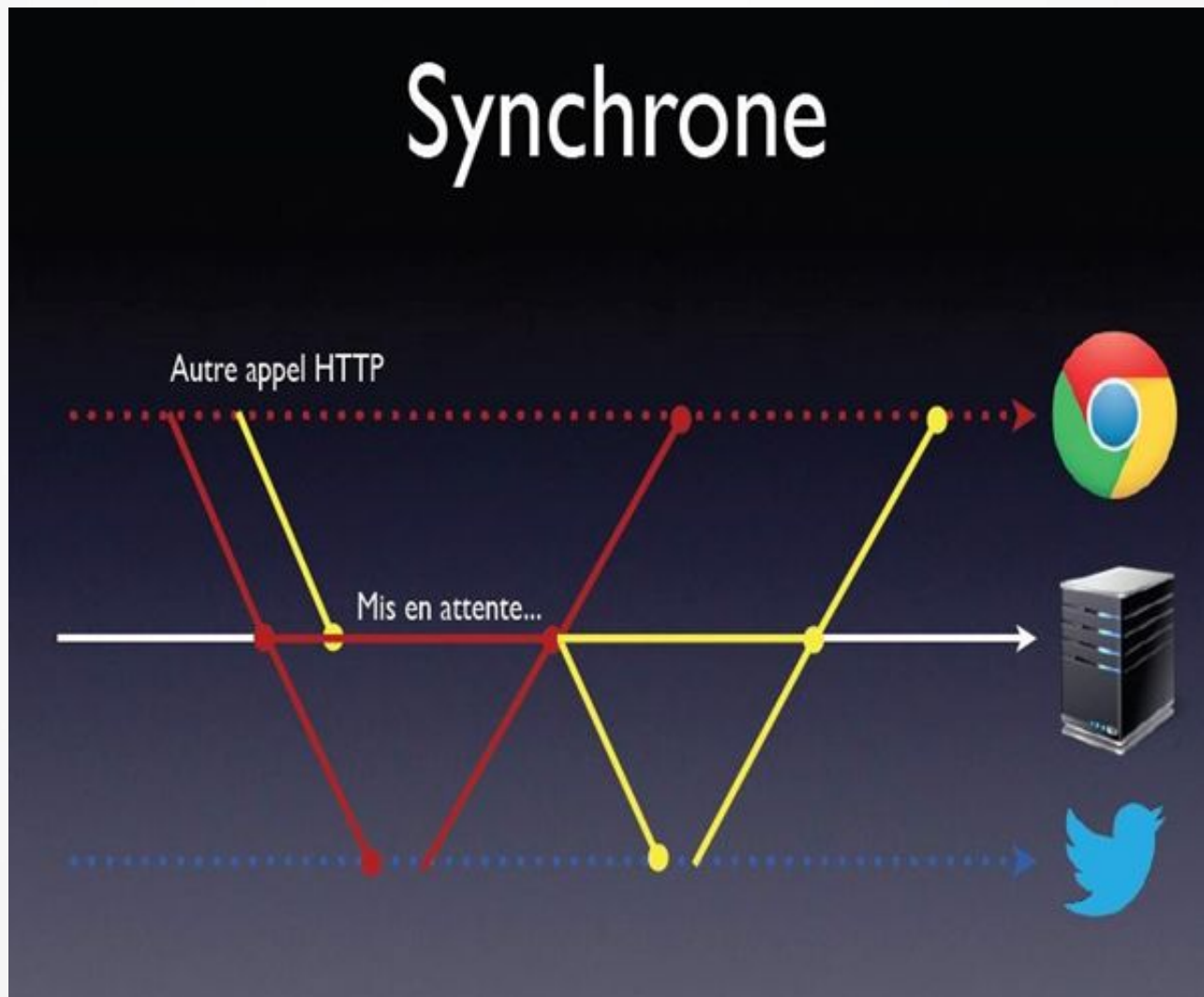
► 2. Non-blocking I/O model

Simulation : Asynchrone-Mode non-bloquant



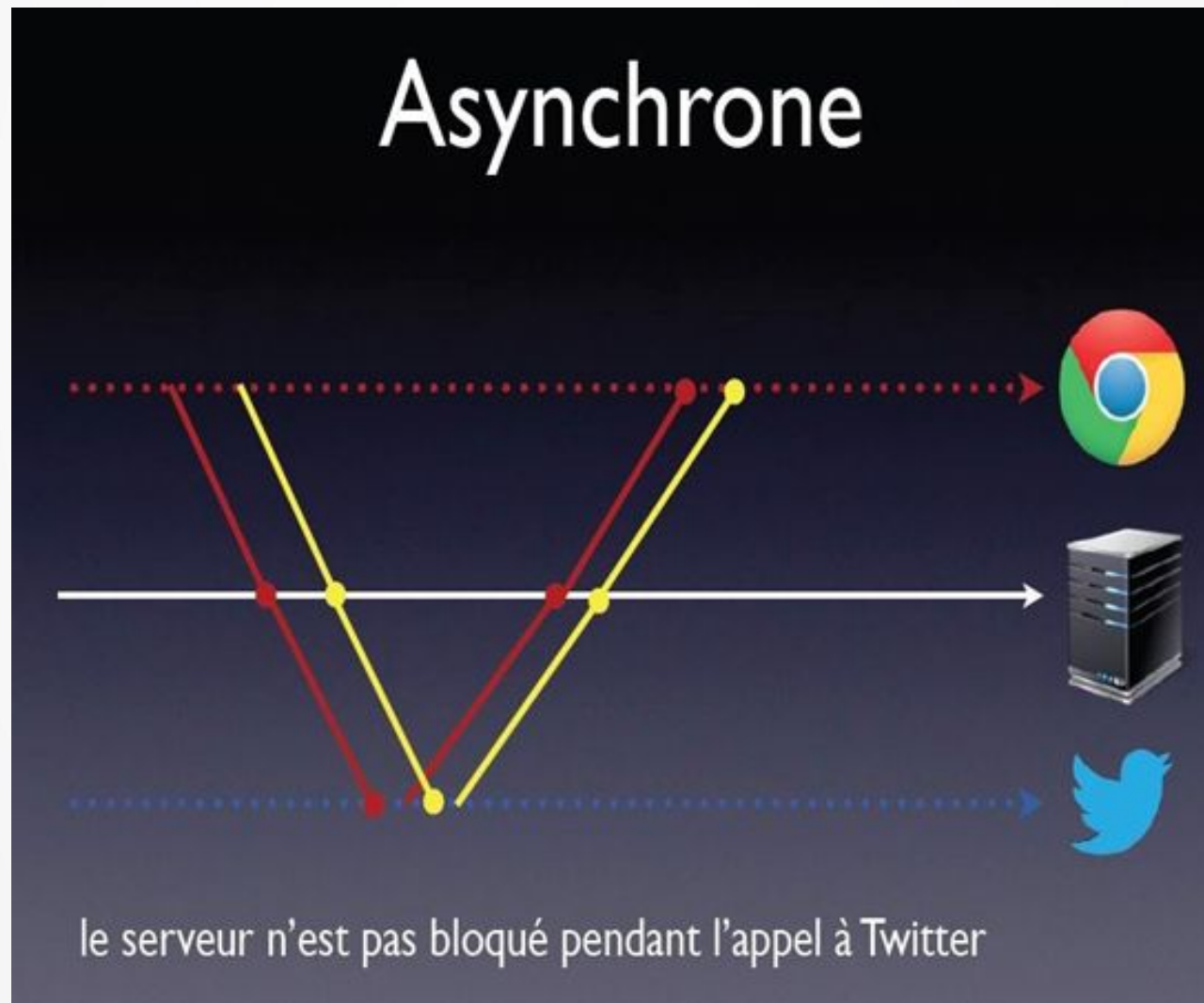
► 2. Non-blocking I/O model


Simulation : Asynchrone-Mode non-bloquant



► 2. Non-blocking I/O model


Simulation 2 : Asynchrone-Mode non-bloquant





▶ 3. Event-driven Model

- Plusieurs traitements en NodeJS sont basés sur les événements
- Les événements proviennent généralement des processus, opérations réseau, fichiers...
- La classe EventEmitter est au cœur de l'architecture événementielle asynchrone en NodeJS

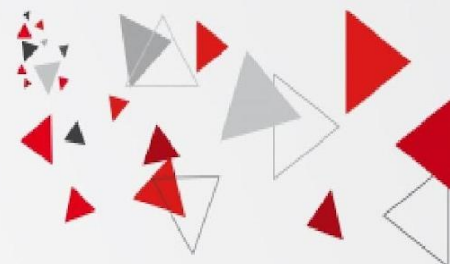


3. Event-driven Model

Exemple 1

```
const EventEmitter = require ('events');  
const emitter= new EventEmitter();  
  
//register a listener  
emitter.on('messageLogged', function(){  
    console.log('Listener called');  
});  
  
//raise an event  
emitter.emit('messageLogged');
```

3. Event-driven Model



Exemple 1


event
emitter

```
const EventEmitter = require ('events');  
const emitter= new EventEmitter();  
  
//register a listener  
emitter.on('messageLogged', function(){  
    console.log('Listener called');  
});  
  
//raise an event  
emitter.emit('messageLogged');
```

register
event
listener

event

callback
function



3. Event-driven Model

- Plusieurs classes dans NodeJS sont des 'Event Emitter' : net.Server, fs.readStream ...

event
emitter

Exemple 2

```
server.on('connection', (stream) => {  
  console.log('someone connected!');  
});
```

register
event
listener

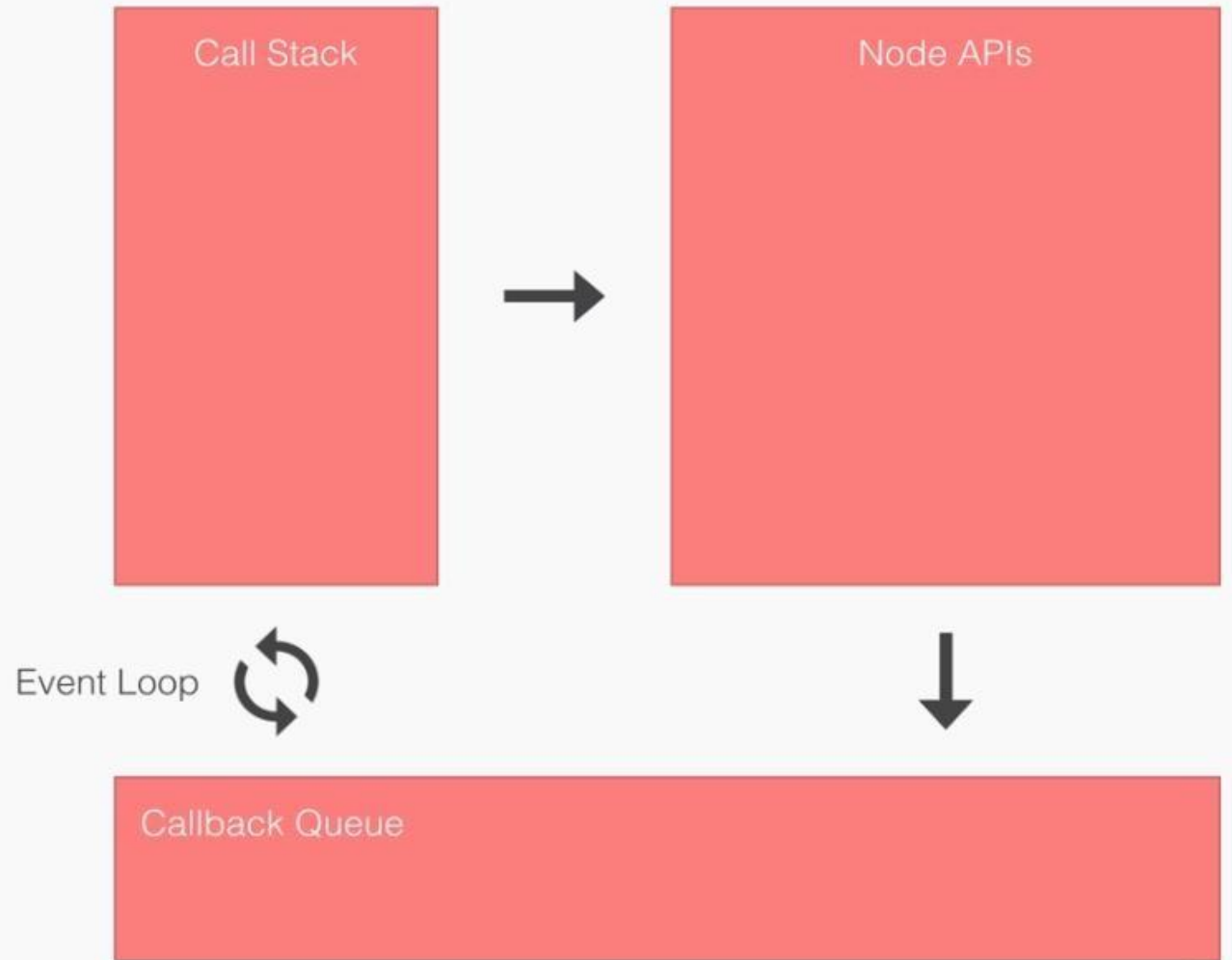
event

callback
function

▶ 4. Call Stack et Event loop

Exemple 1

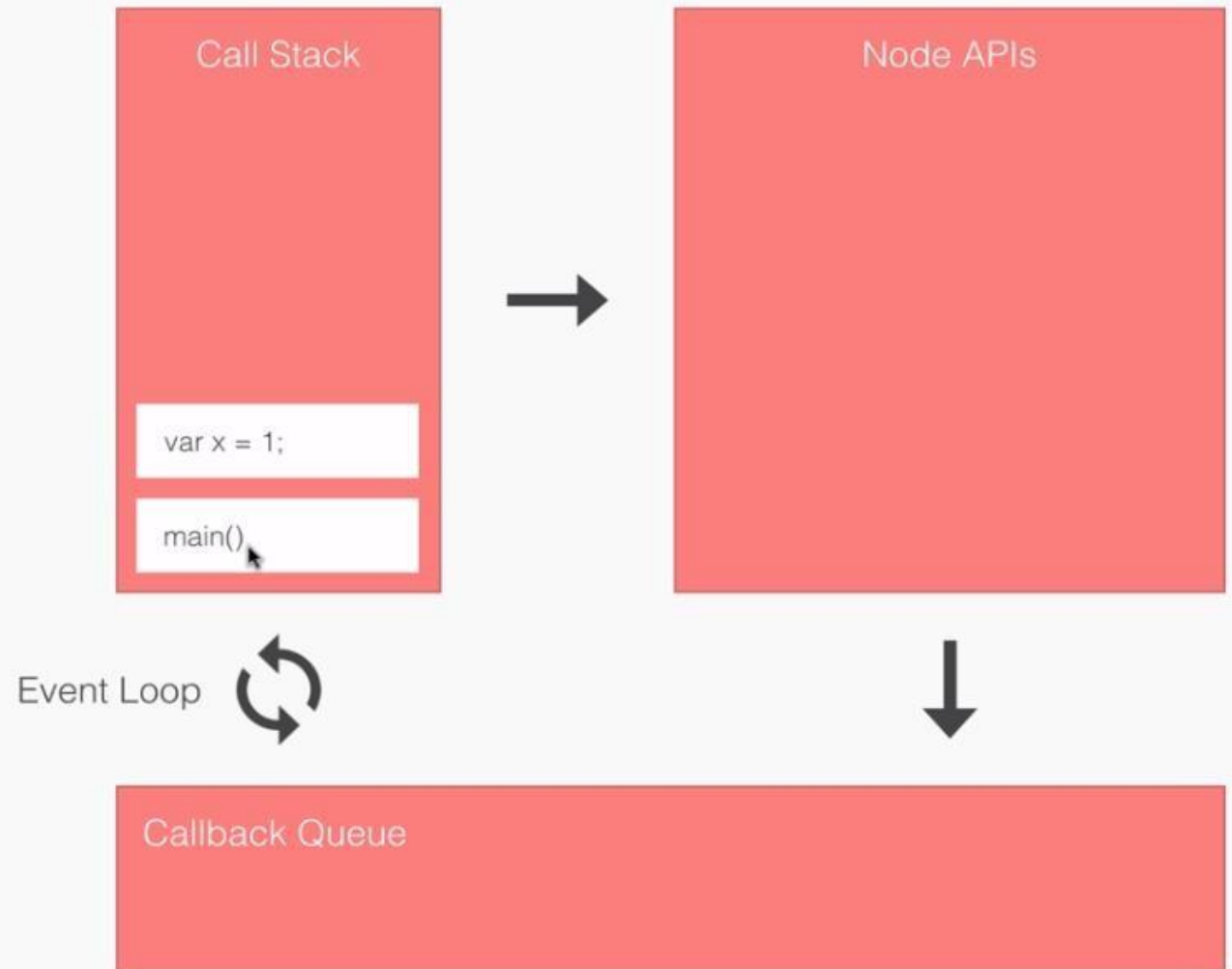
```
1 var x = 1;  
2  
3 var y = x + 9;  
4  
5 console.log(`y is ${y}`);  
6
```



▶ 4. Call Stack et Event loop

Exemple 1

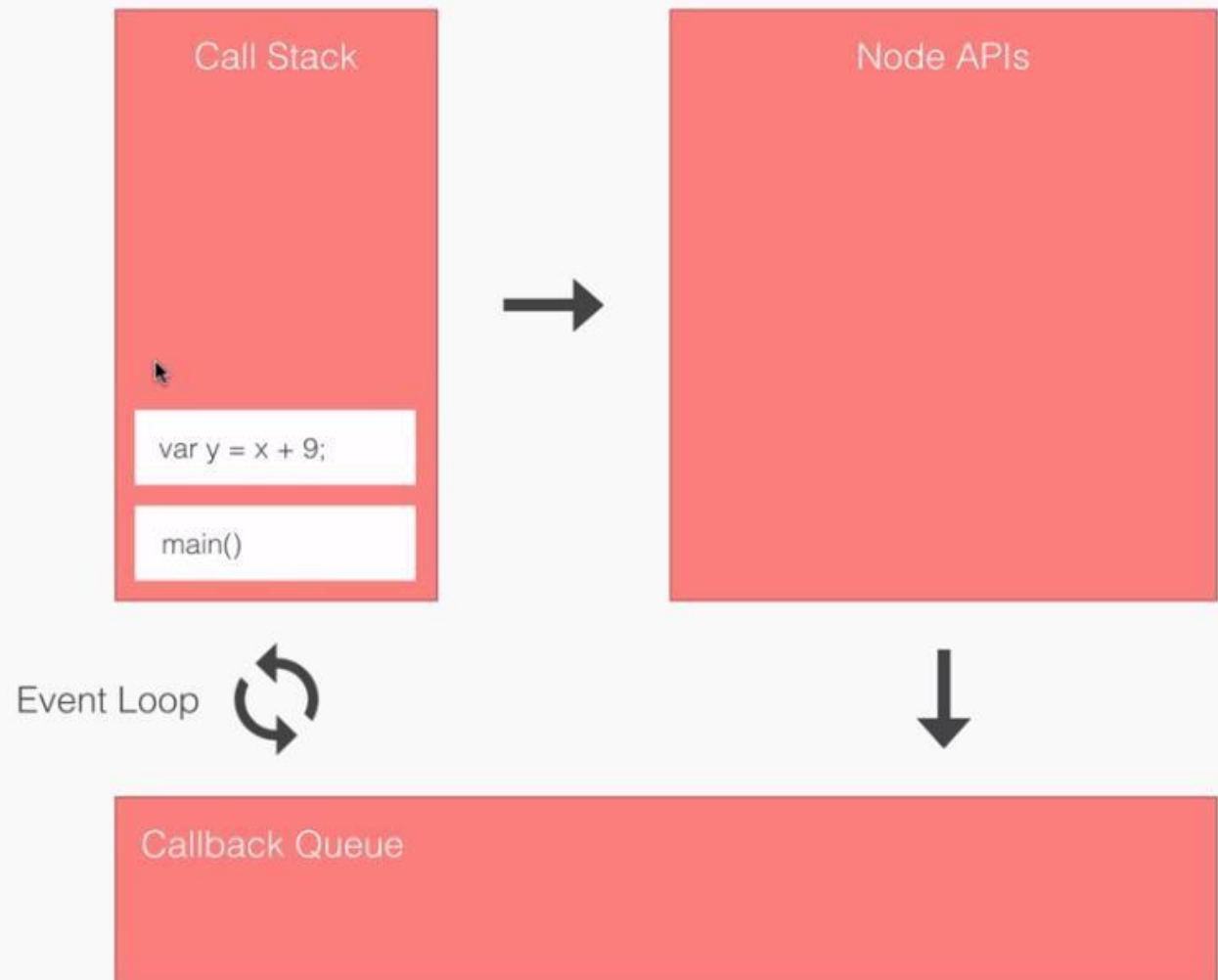
```
1  var x = 1;  
2  
3  var y = x + 9;  
4  
5  console.log(`y is ${y}`);  
6
```



▶ 4. Call Stack et Event loop

Exemple 1

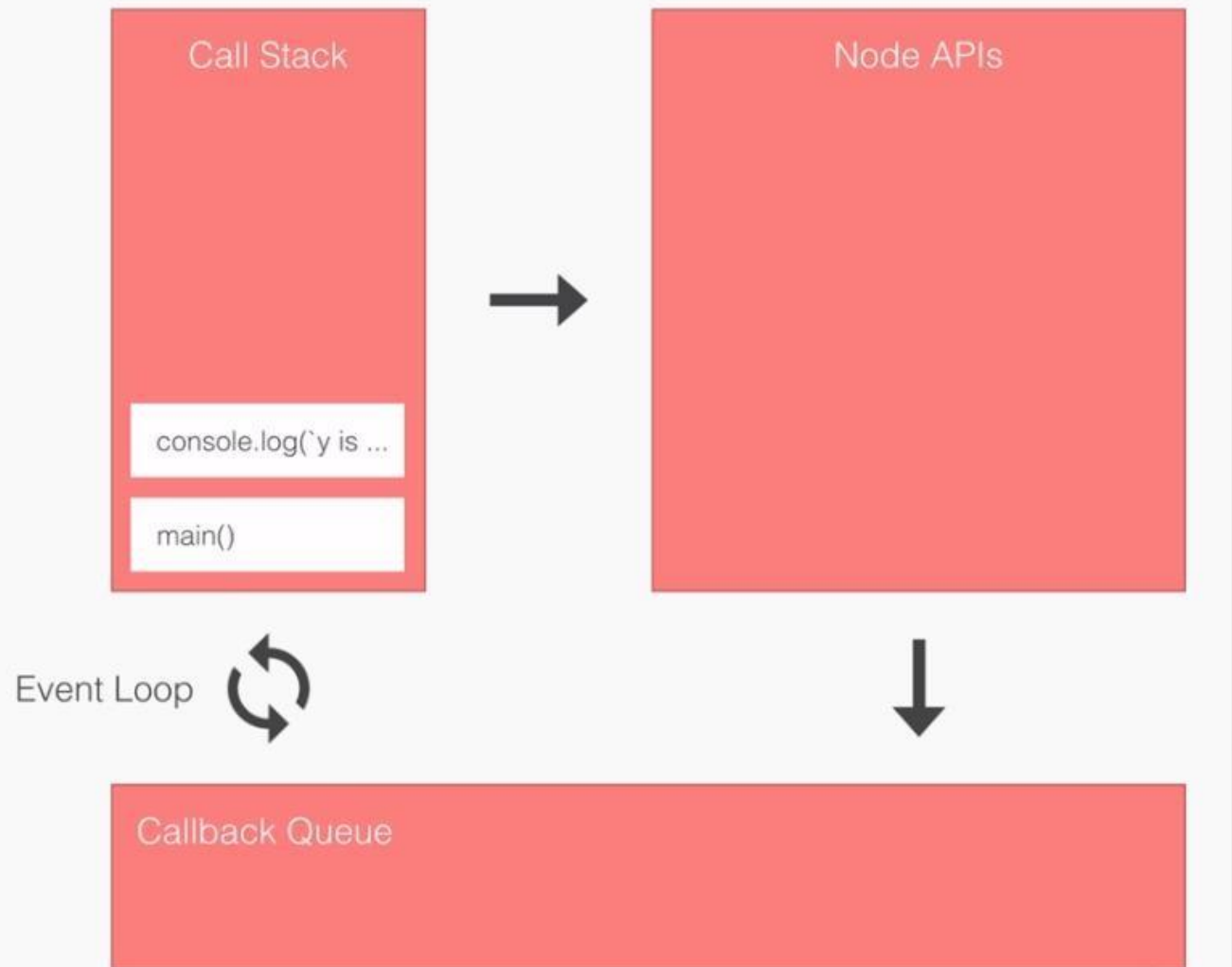
```
1 var x = 1;  
2  
3 var y = x + 9;  
4  
5 console.log(`y is ${y}`);  
6
```



► 4. Call Stack et Event loop

Exemple 1

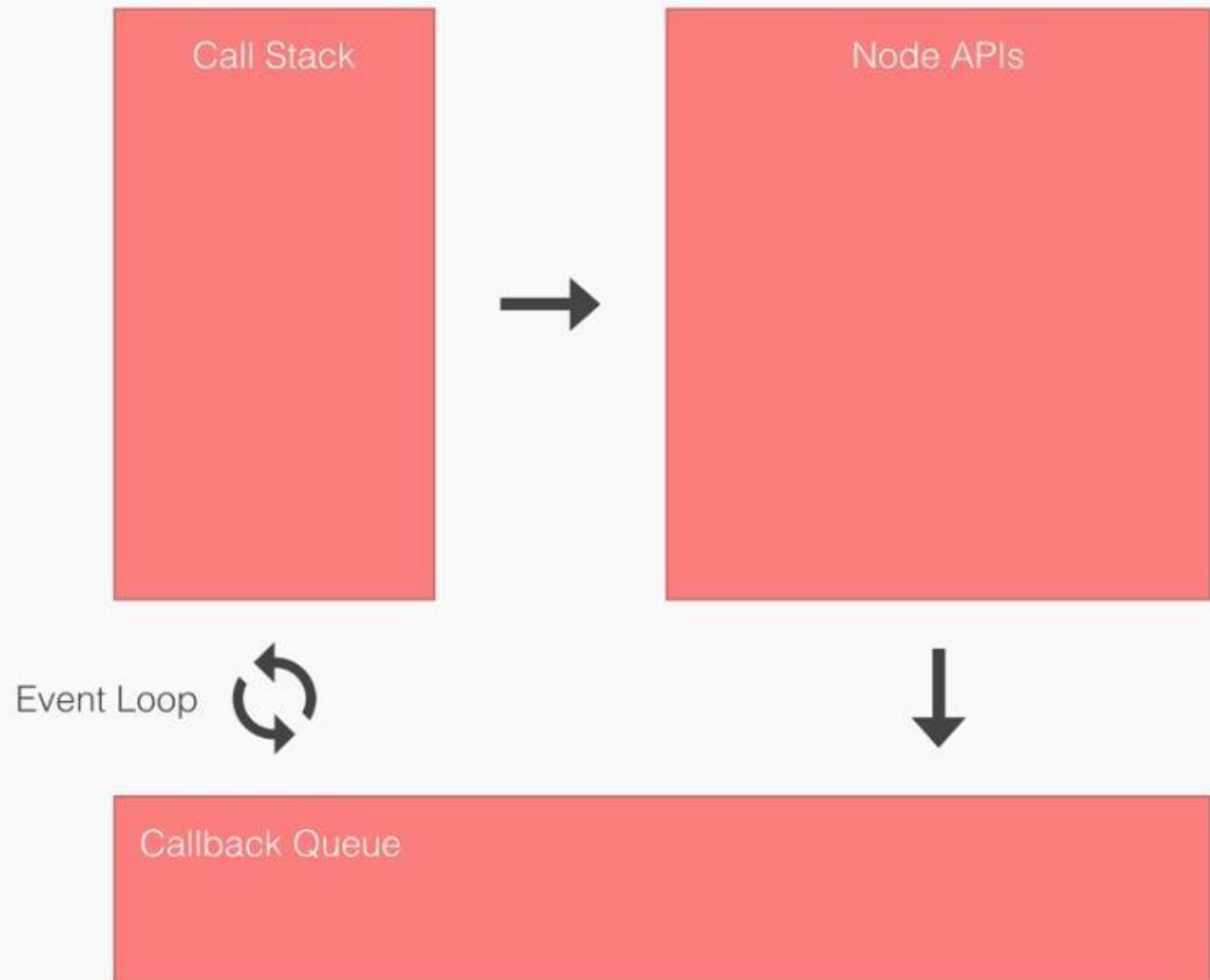
```
1 var x = 1;  
2  
3 var y = x + 9;  
4  
5 console.log(`y is ${y}`);  
6
```



▶ 4. Call Stack et Event loop

Exemple 1

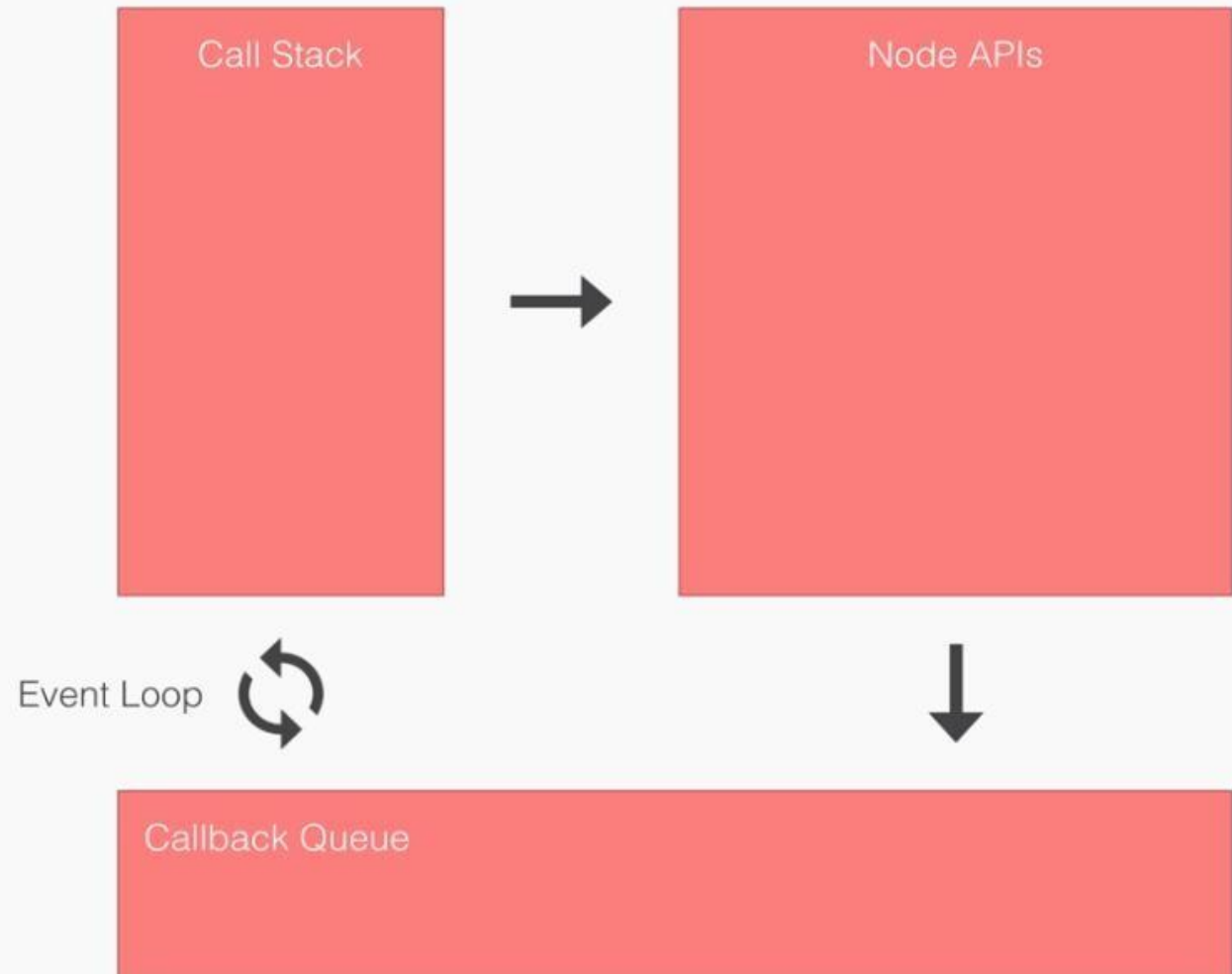
```
1 var x = 1;  
2  
3 var y = x + 9;  
4  
5 console.log(`y is ${y}`);  
6
```



► 4. Call Stack et Event loop

Exemple 2

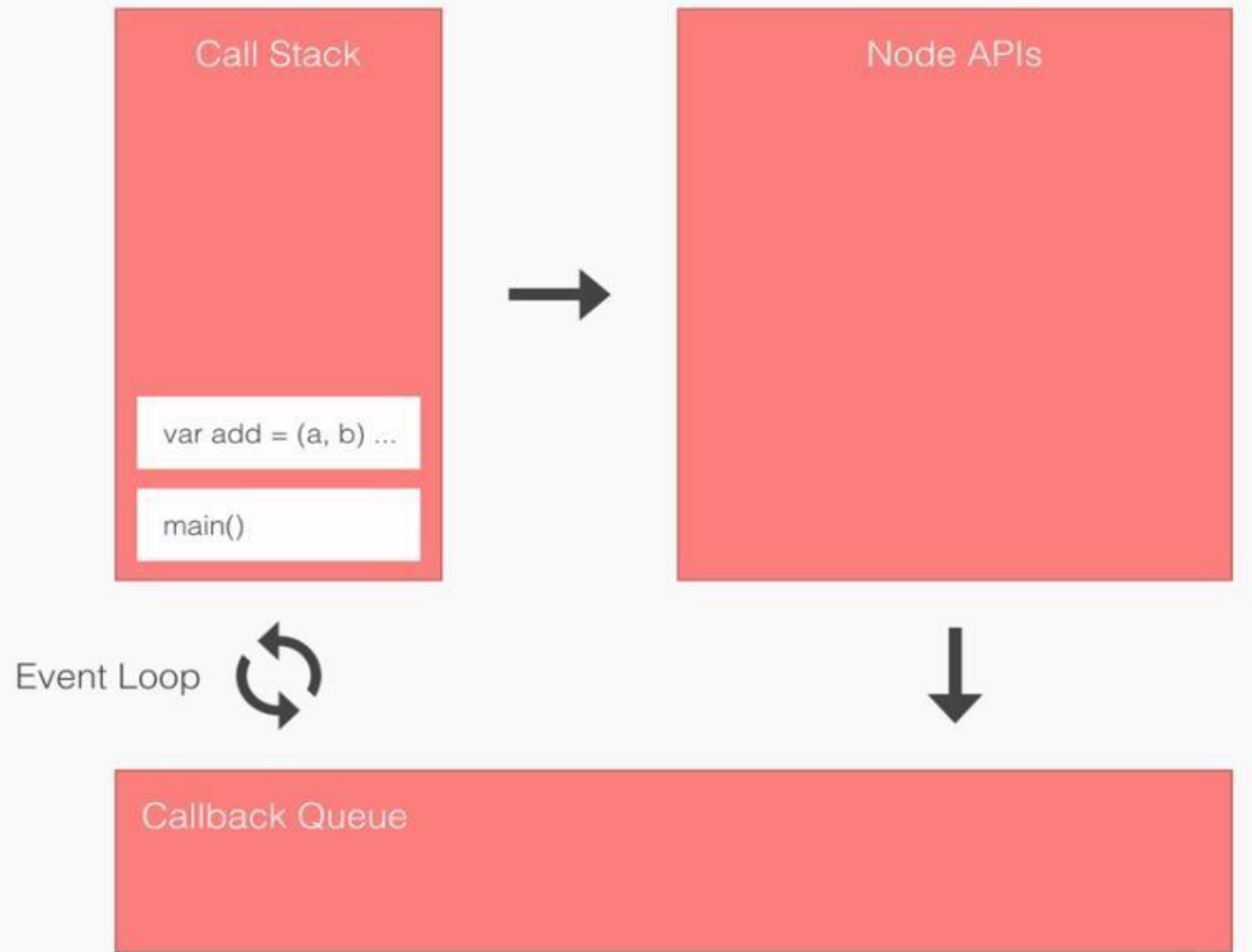
```
1  var add = (a, b) => {  
2    var total = a + b;  
3  
4    return total;  
5  };  
6  
7  var res = add(3, 8);  
8  
9  console.log(res);  
10
```



► 4. Call Stack et Event loop

Exemple 2

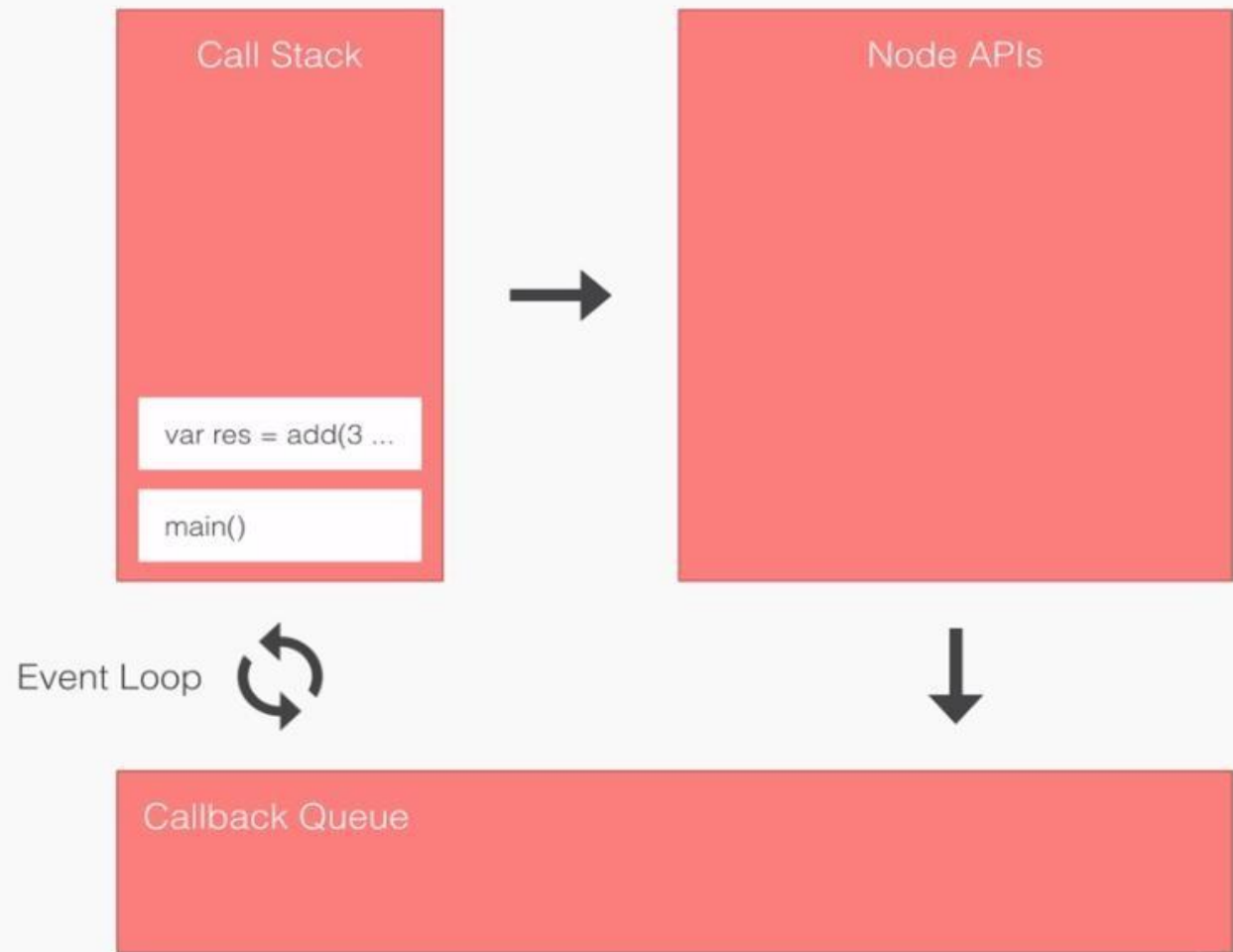
```
1  var add = (a, b) => {  
2    var total = a + b;  
3  
4    return total;  
5  };  
6  
7  var res = add(3, 8);  
8  
9  console.log(res);  
10
```



► 4. Call Stack et Event loop

Exemple 2

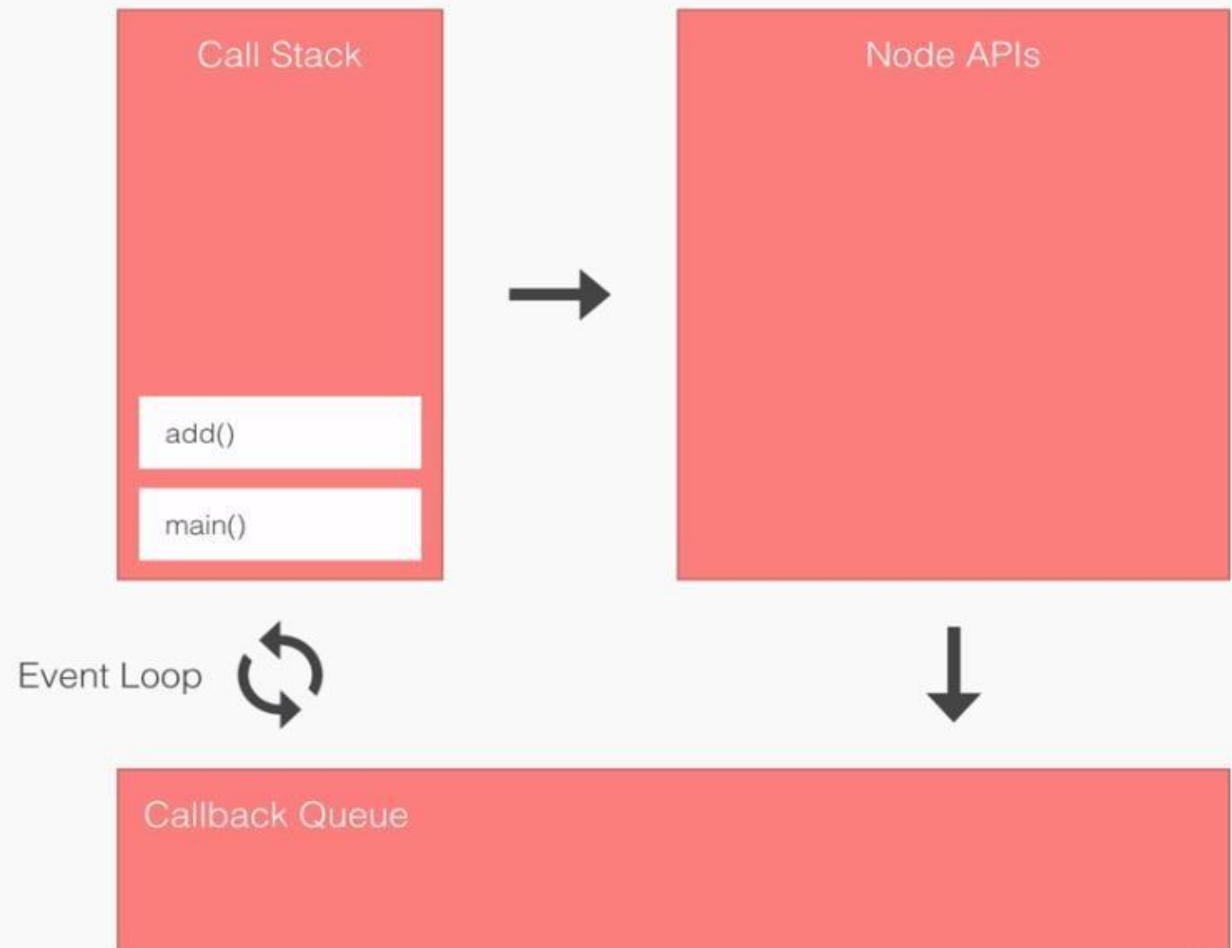
```
1  var add = (a, b) => {  
2    var total = a + b;  
3  
4    return total;  
5  };  
6  
7  var res = add(3, 8);  
8  
9  console.log(res);  
10
```



► 4. Call Stack et Event loop

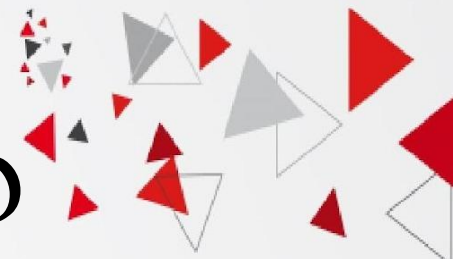
Exemple 2

```
1  var add = (a, b) => {  
2    var total = a + b;  
3  
4    return total;  
5  };  
6  
7  var res = add(3, 8);  
8  
9  console.log(res);  
10
```

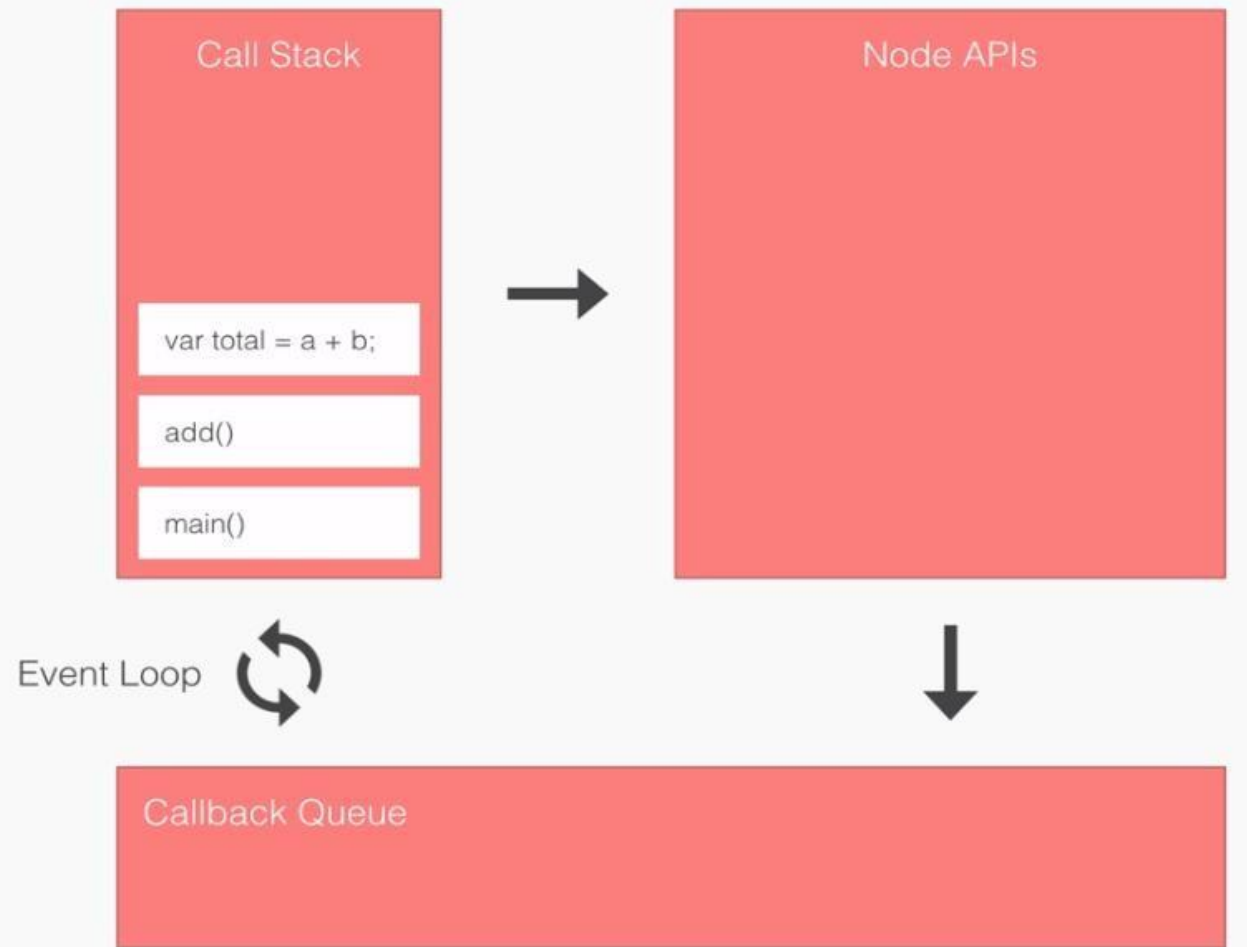


► 4. Call Stack et Event loop

Exemple 2



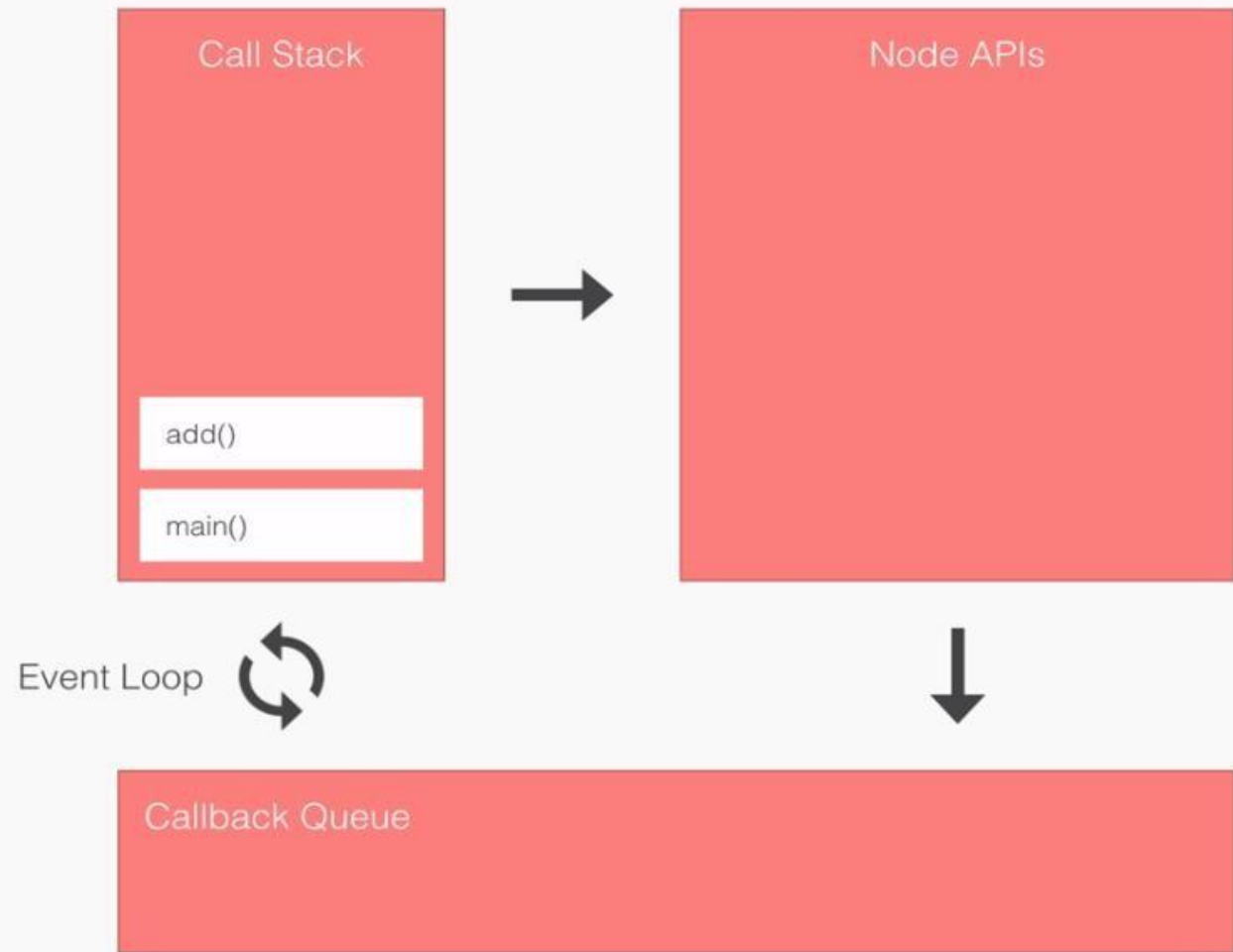
```
1  var add = (a, b) => {  
2    var total = a + b;  
3  
4    return total;  
5  };  
6  
7  var res = add(3, 8);  
8  
9  console.log(res);  
10
```



► 4. Call Stack et Event loop

Exemple 2

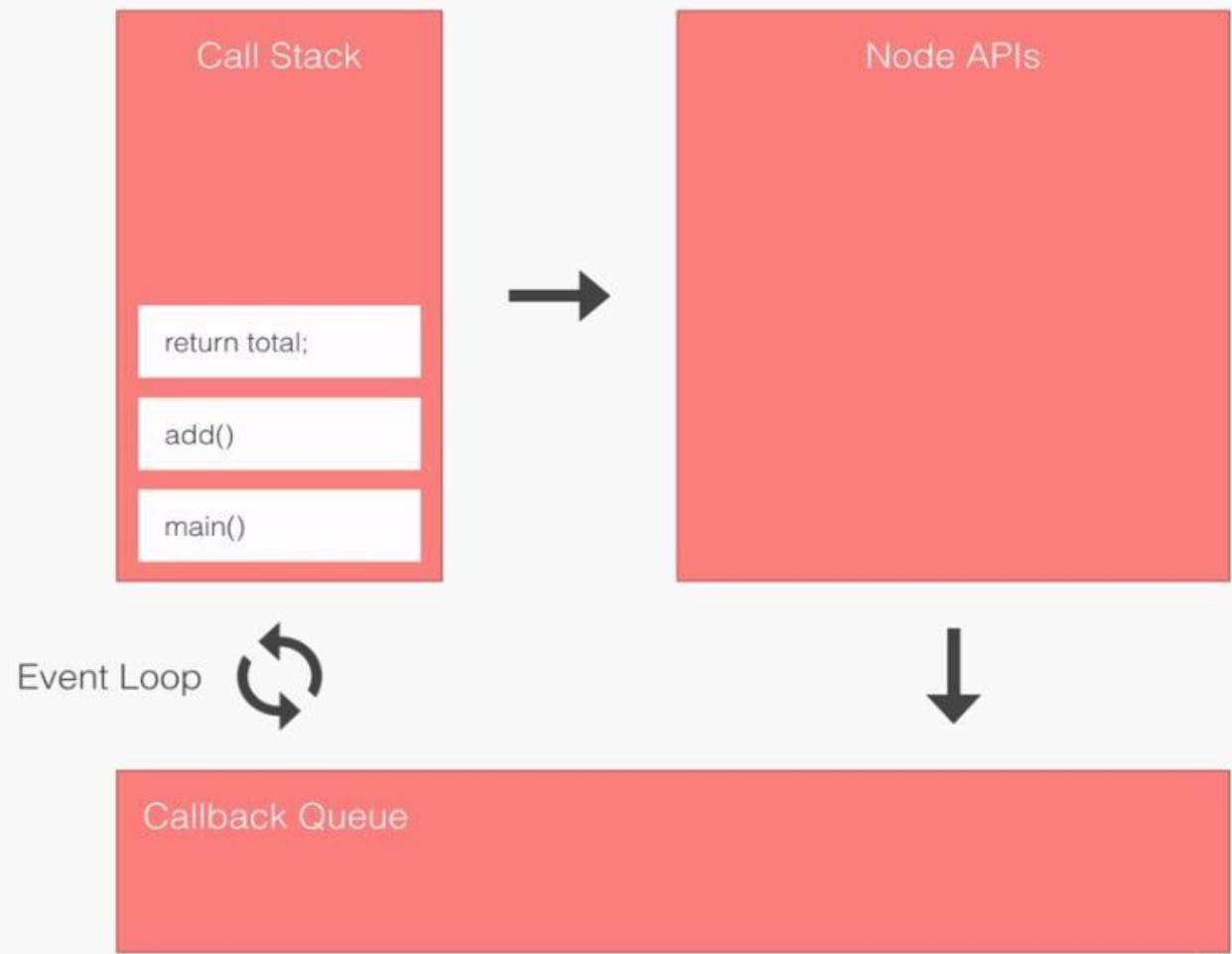
```
1  var add = (a, b) => {  
2    var total = a + b;  
3  
4    return total;  
5  };  
6  
7  var res = add(3, 8);  
8  
9  console.log(res);  
10
```



► 4. Call Stack et Event loop

Exemple 2

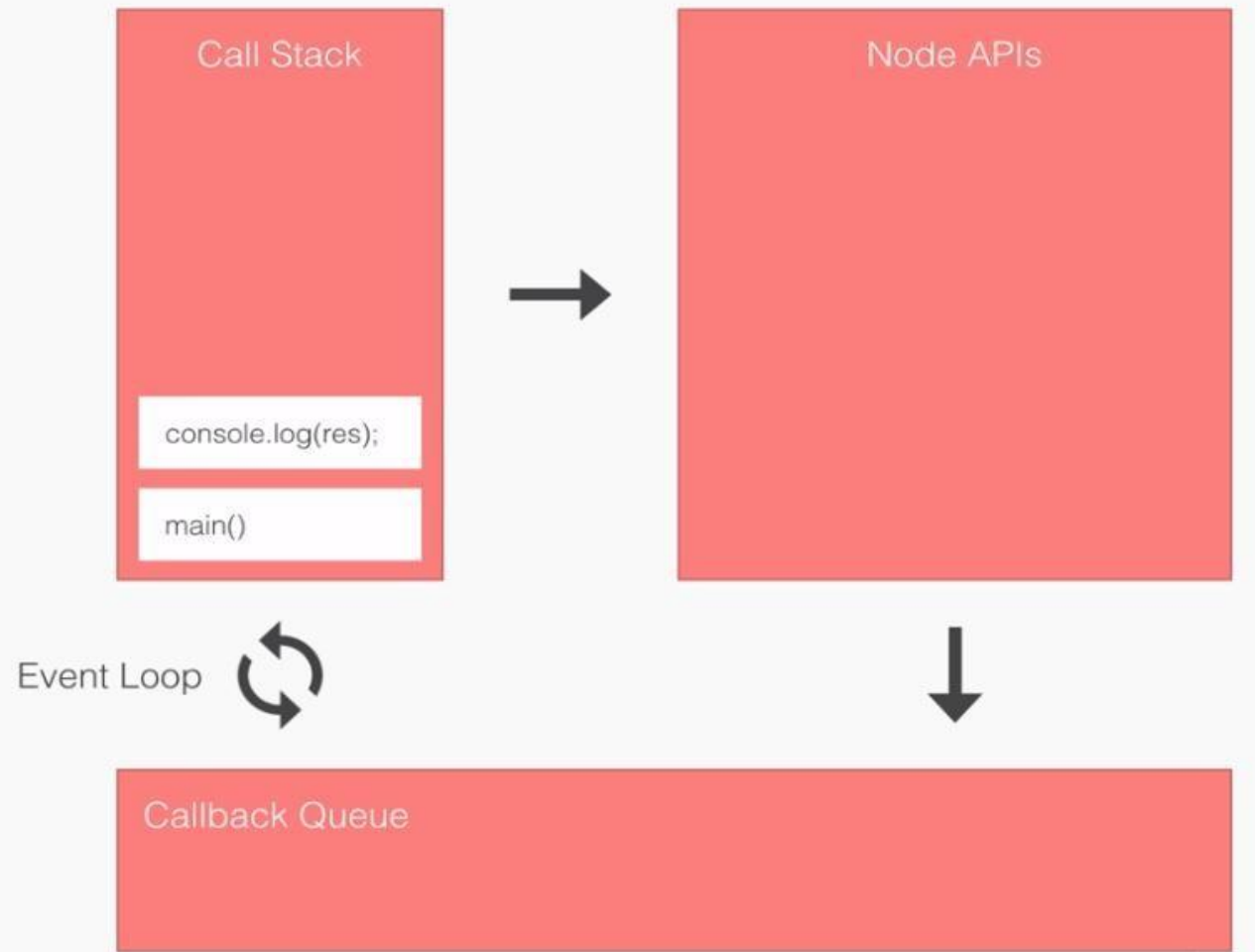
```
1  var add = (a, b) => {  
2    var total = a + b;  
3  
4    return total;  
5  };  
6  
7  var res = add(3, 8);  
8  
9  console.log(res);  
10
```



► 4. Call Stack et Event loop

Exemple 2

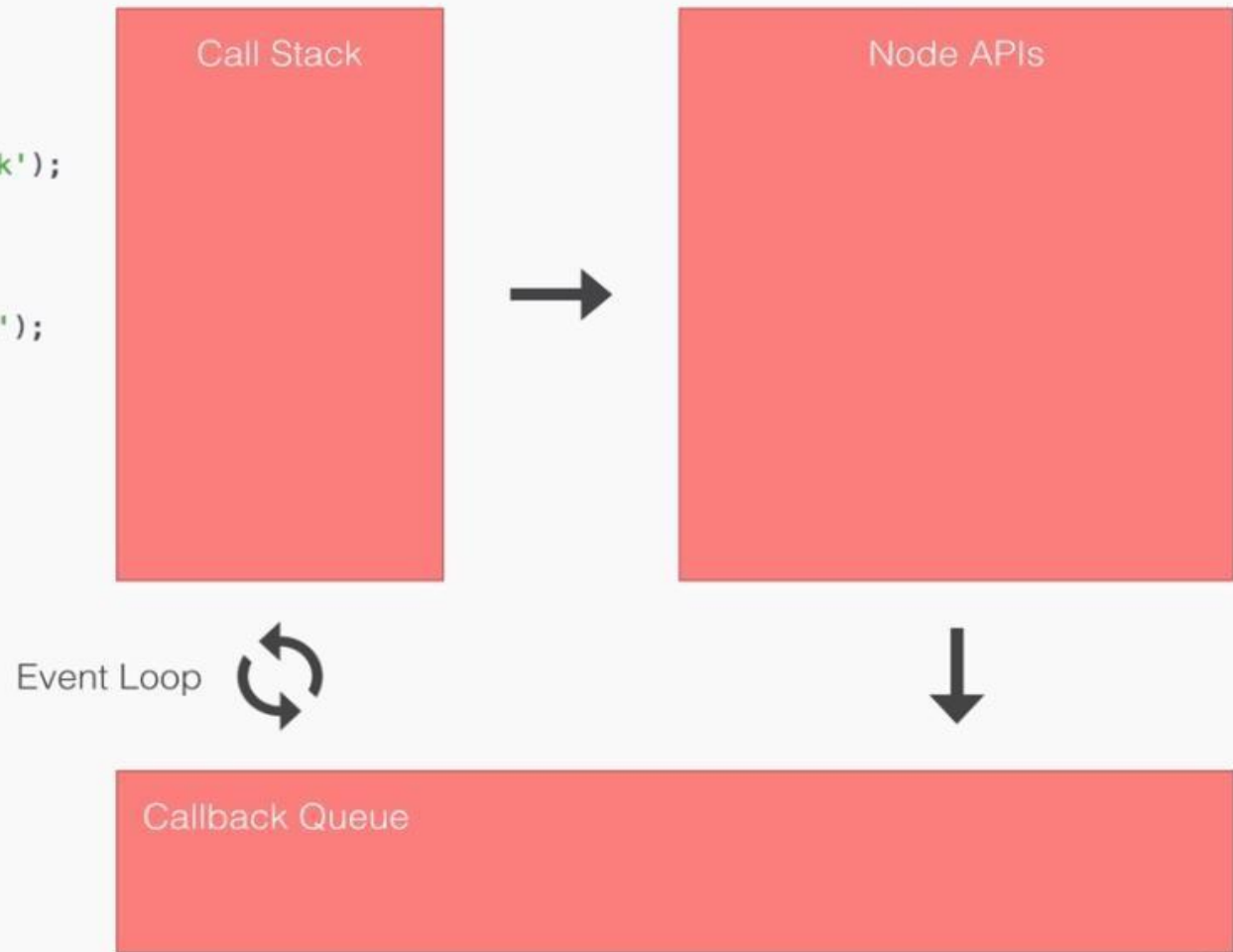
```
1  var add = (a, b) => {  
2    var total = a + b;  
3  
4    return total;  
5  };  
6  
7  var res = add(3, 8);  
8  
9  console.log(res);  
10
```



► 4. Call Stack et Event loop

Exemple 3

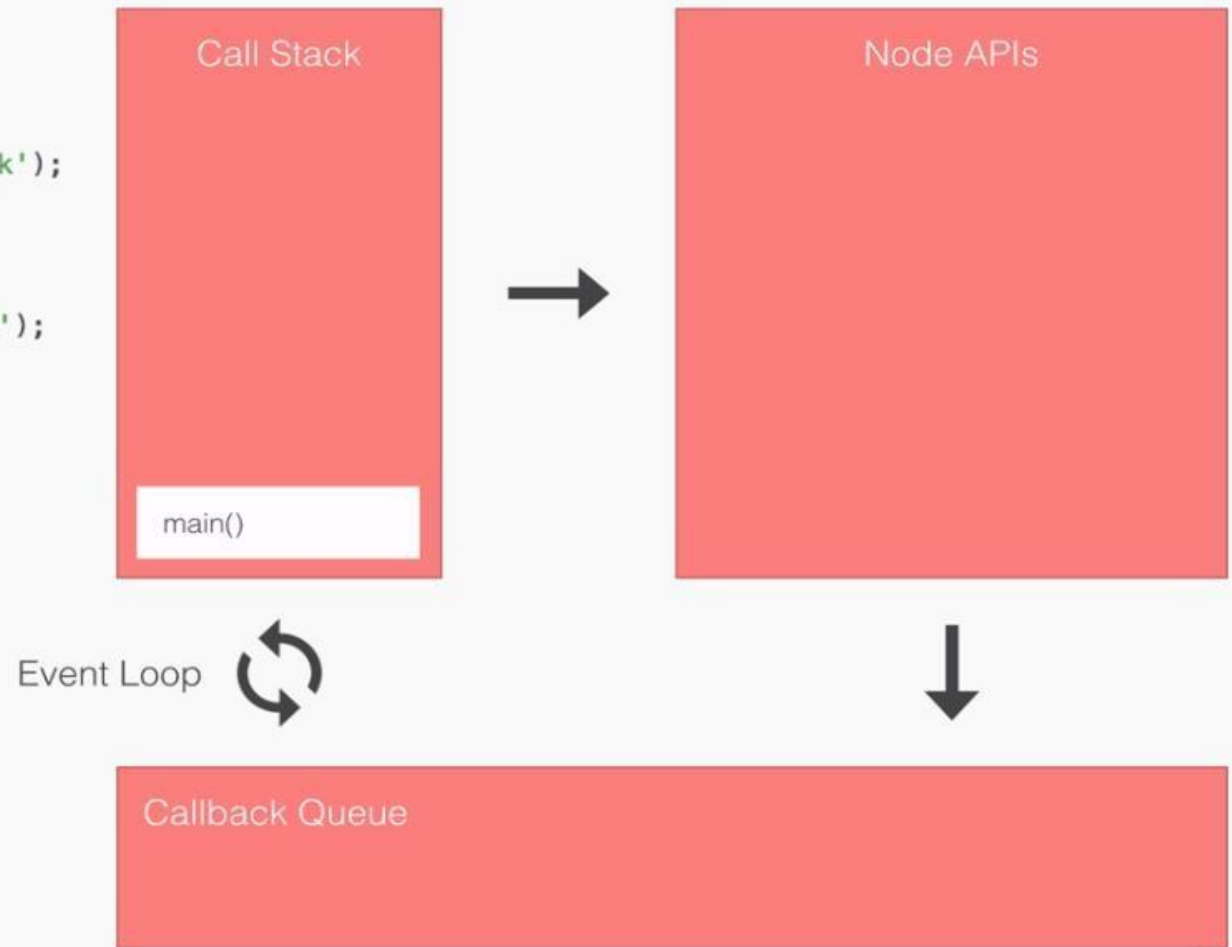
```
1 console.log('Starting app');
2
3 setTimeout(() => {
4   console.log('Inside of callback');
5 }, 2000);
6
7 setTimeout(() => {
8   console.log('Second setTimeout');
9 }, 0);
10
11 console.log('Finishing up');
12
```



▶ 4. Call Stack et Event loop

Exemple 3

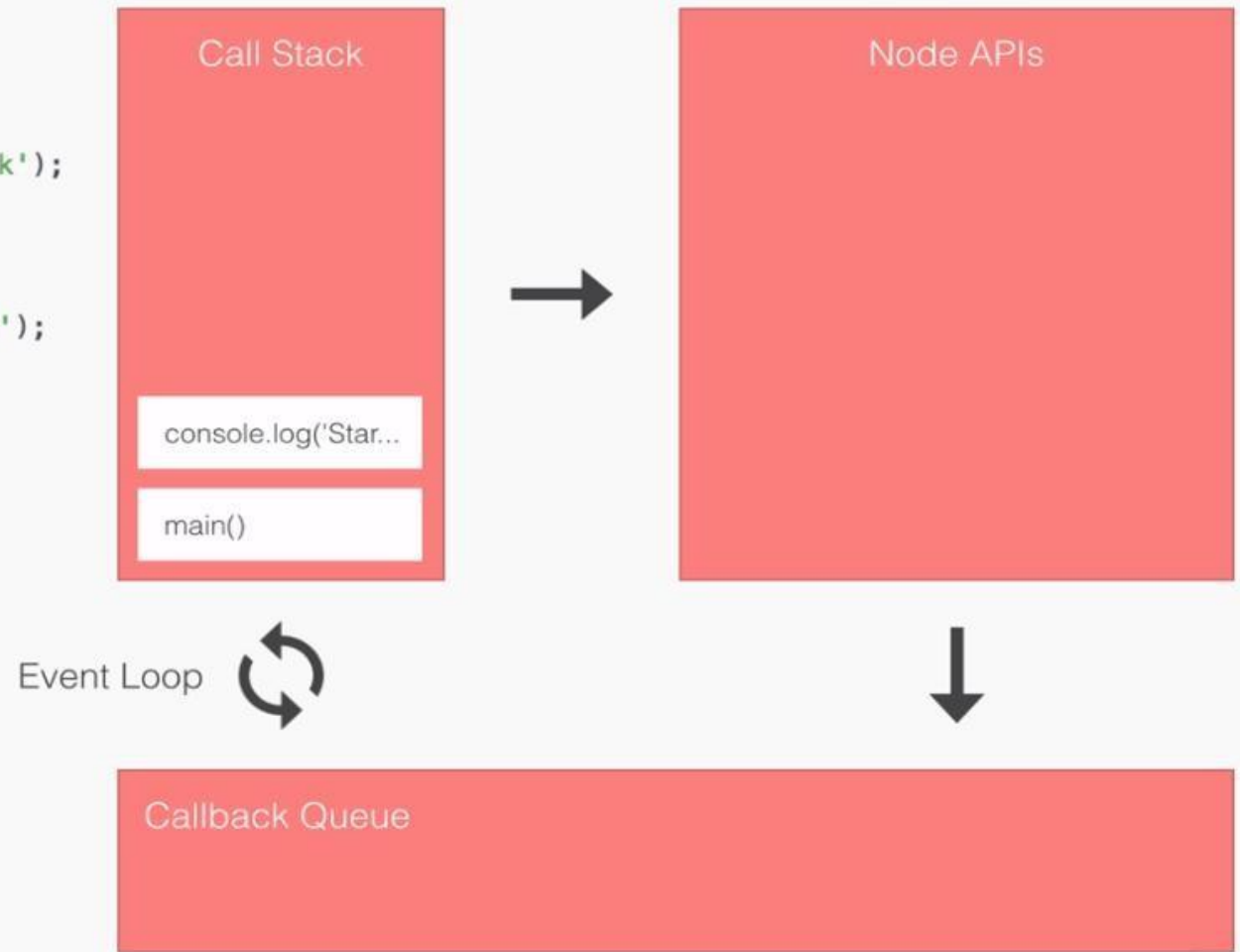
```
1 console.log('Starting app');
2
3 setTimeout(() => {
4   console.log('Inside of callback');
5 }, 2000);
6
7 setTimeout(() => {
8   console.log('Second setTimeout');
9 }, 0);
10
11 console.log('Finishing up');
12
```



► 4. Call Stack et Event loop

Exemple 3

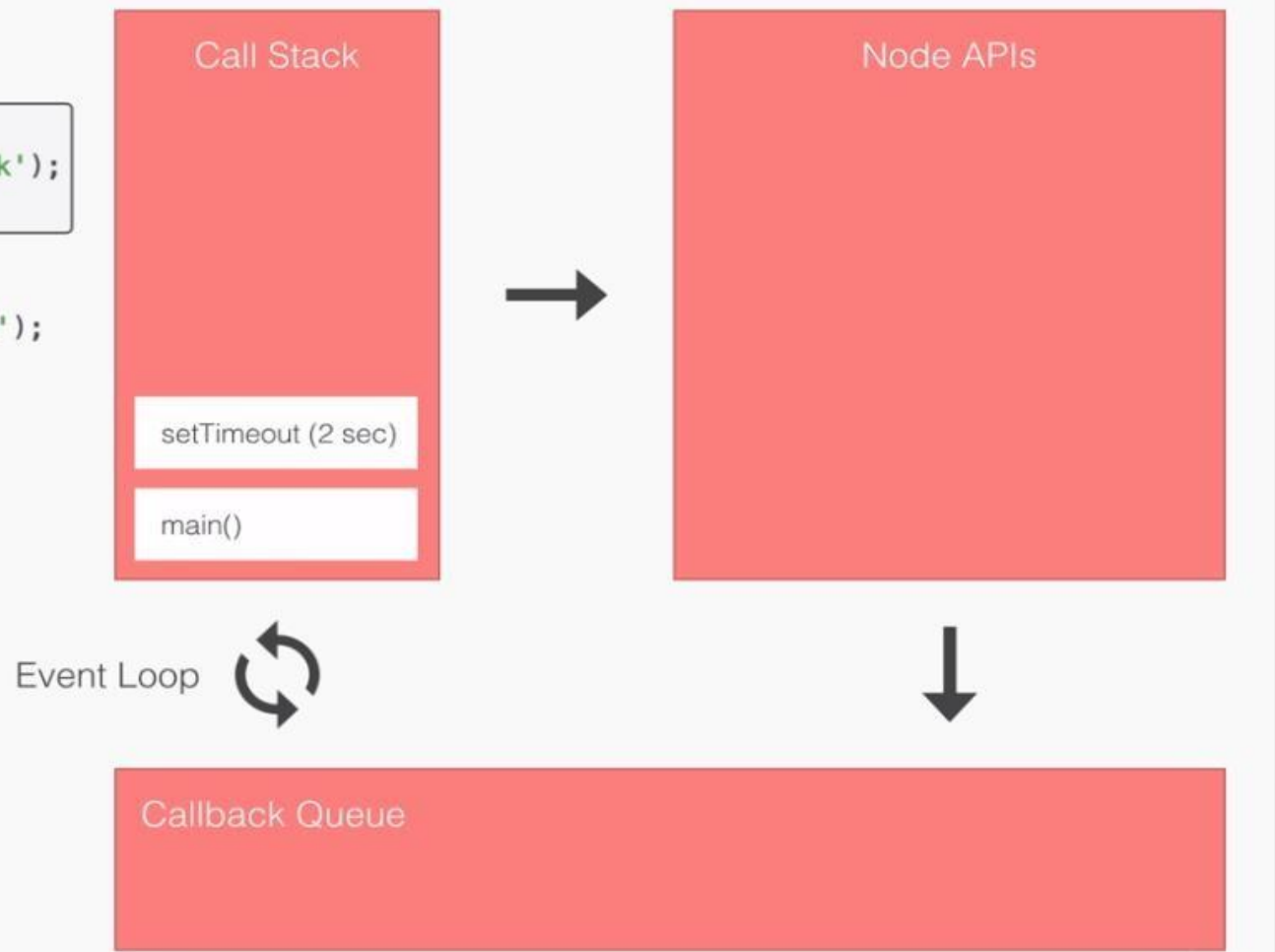
```
1 console.log('Starting app');
2
3 setTimeout(() => {
4   console.log('Inside of callback');
5 }, 2000);
6
7 setTimeout(() => {
8   console.log('Second setTimeout');
9 }, 0);
10
11 console.log('Finishing up');
12
```



► 4. Call Stack et Event loop

Exemple 3

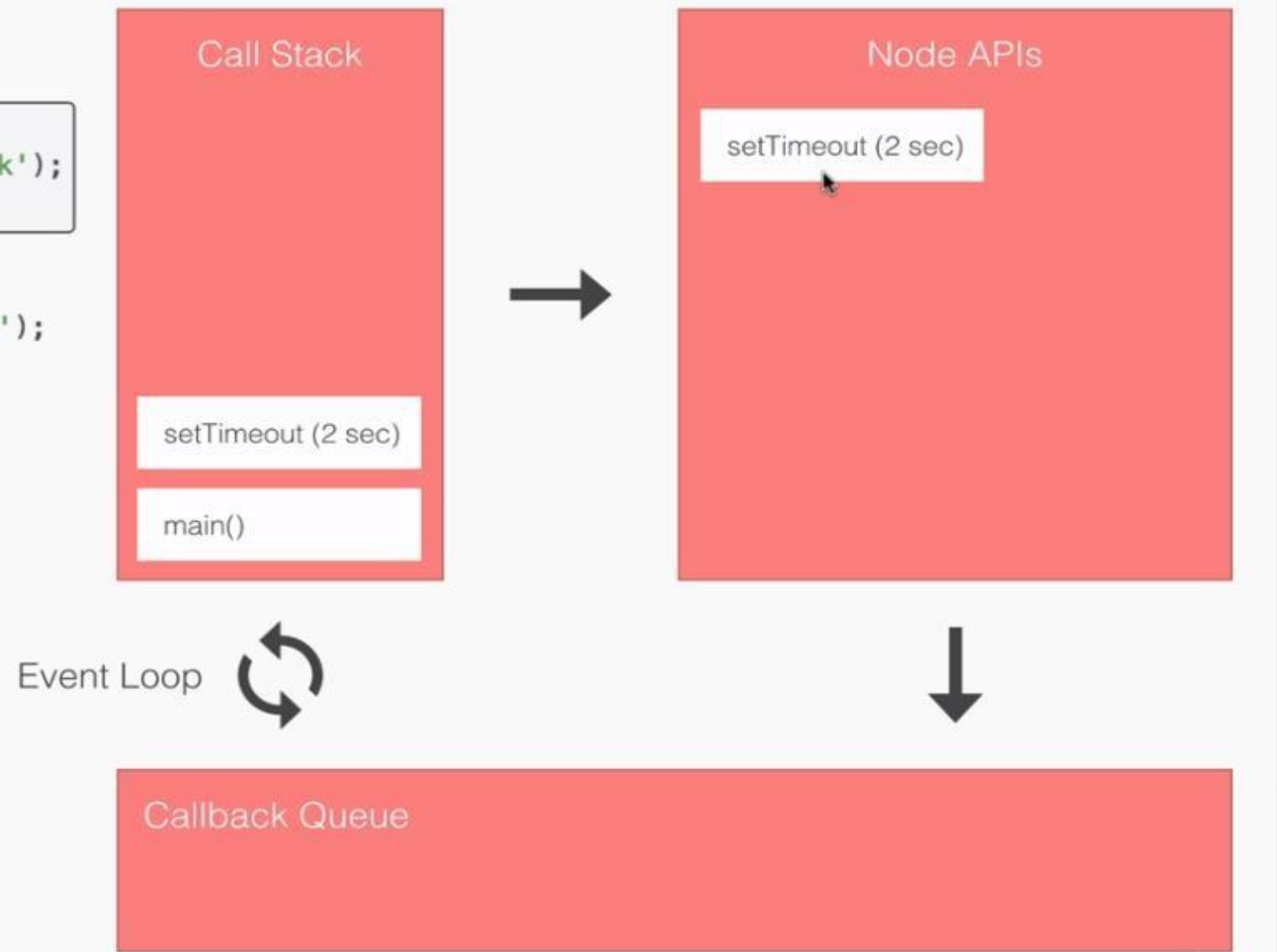
```
1 console.log('Starting app');
2
3 setTimeout(() => {
4   console.log('Inside of callback');
5 }, 2000);
6
7 setTimeout(() => {
8   console.log('Second setTimeout');
9 }, 0);
10
11 console.log('Finishing up');
12
```



► 4. Call Stack et Event loop

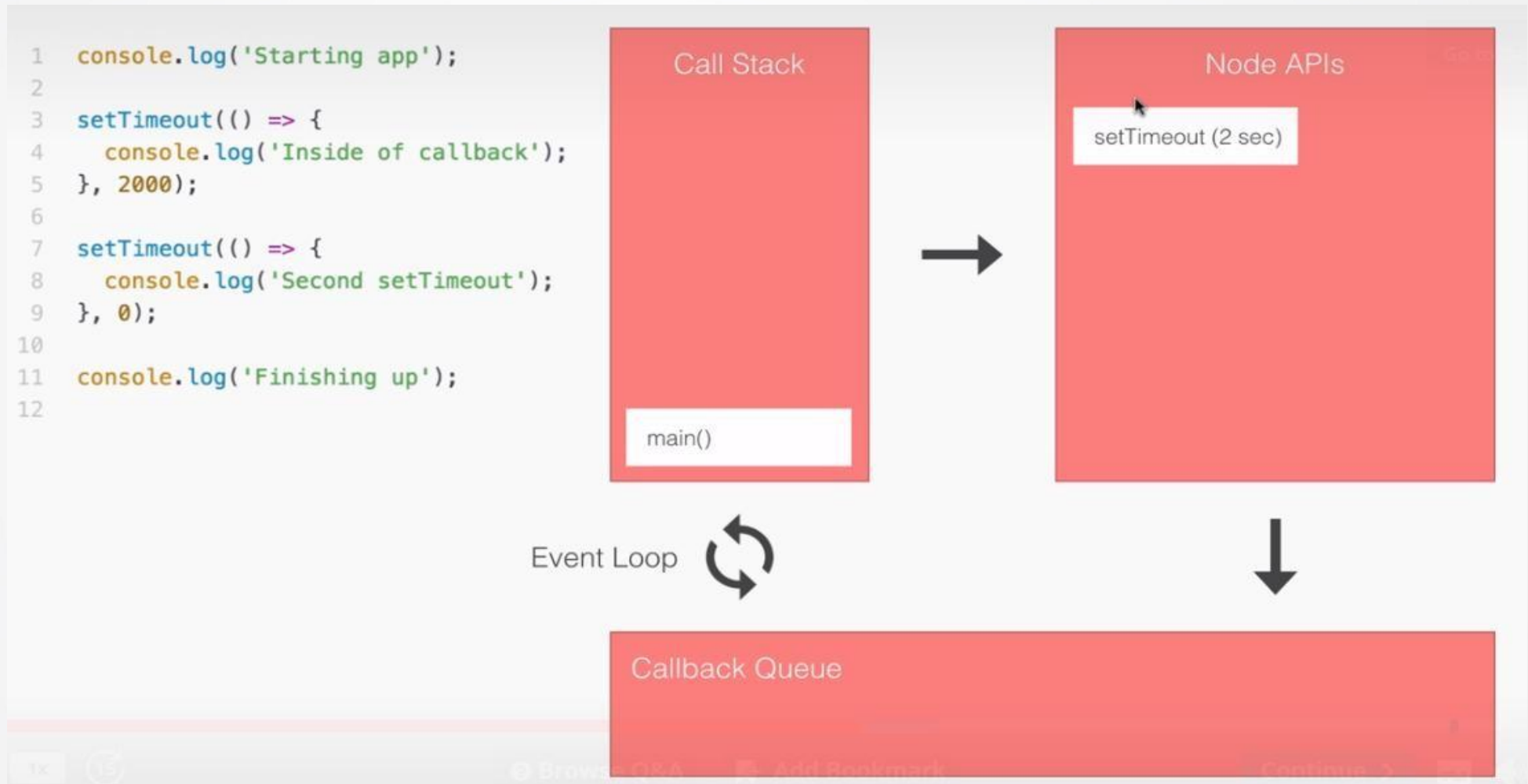
Exemple 3

```
1 console.log('Starting app');
2
3 setTimeout(() => {
4   console.log('Inside of callback');
5 }, 2000);
6
7 setTimeout(() => {
8   console.log('Second setTimeout');
9 }, 0);
10
11 console.log('Finishing up');
12
```



▶ 4. Call Stack et Event loop

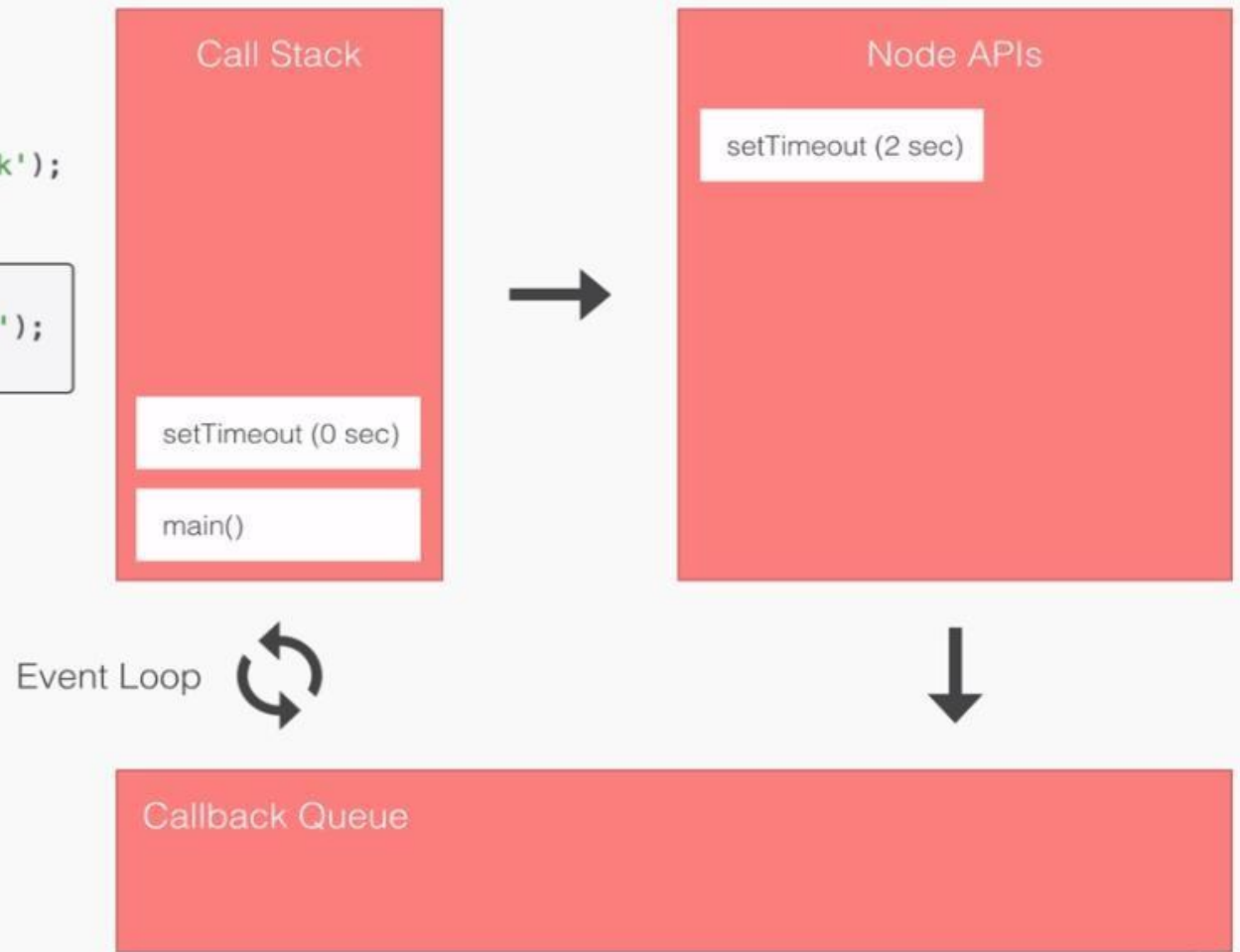
Exemple 3



▶ 4. Call Stack et Event loop

Exemple 3

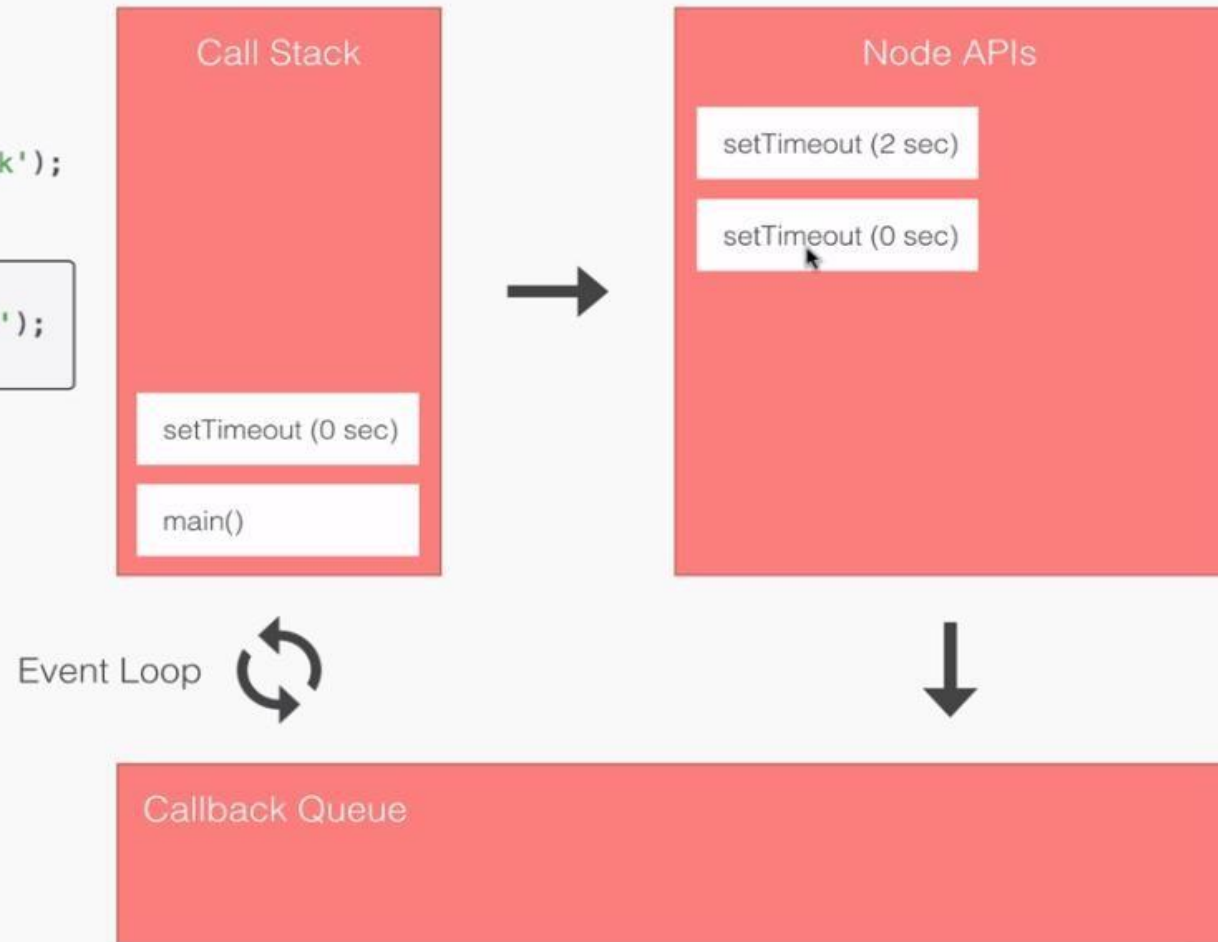
```
1 console.log('Starting app');
2
3 setTimeout(() => {
4   console.log('Inside of callback');
5 }, 2000);
6
7 setTimeout(() => {
8   console.log('Second setTimeout');
9 }, 0);
10
11 console.log('Finishing up');
12
```



4. Call Stack et Event loop

Exemple 3

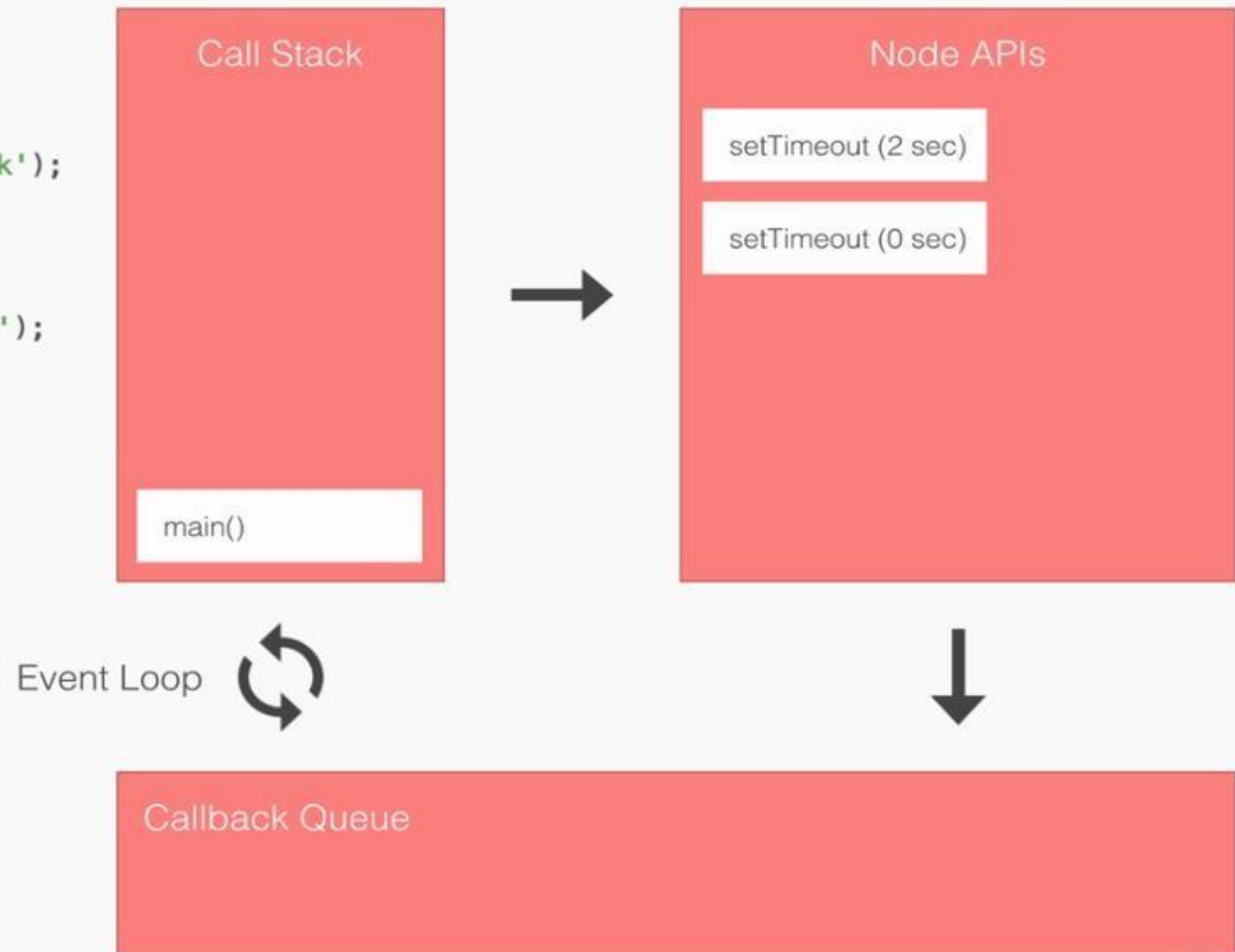
```
1 console.log('Starting app');
2
3 setTimeout(() => {
4   console.log('Inside of callback');
5 }, 2000);
6
7 setTimeout(() => {
8   console.log('Second setTimeout');
9 }, 0);
10
11 console.log('Finishing up');
12
```



▶ 4. Call Stack et Event loop

Exemple 3

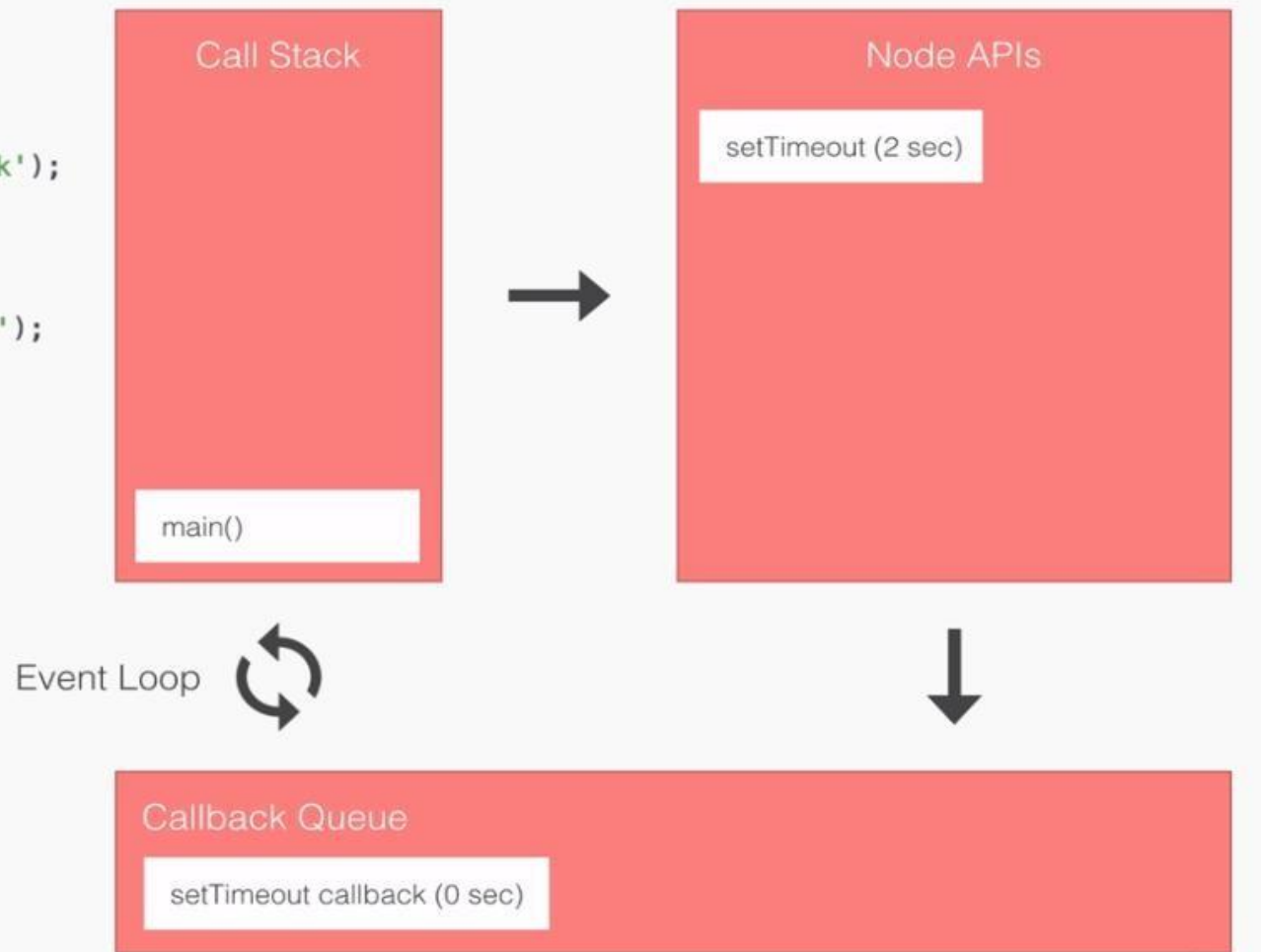
```
1 console.log('Starting app');
2
3 setTimeout(() => {
4   console.log('Inside of callback');
5 }, 2000);
6
7 setTimeout(() => {
8   console.log('Second setTimeout');
9 }, 0);
10
11 console.log('Finishing up');
12
```



▶ 4. Call Stack et Event loop

Exemple 3

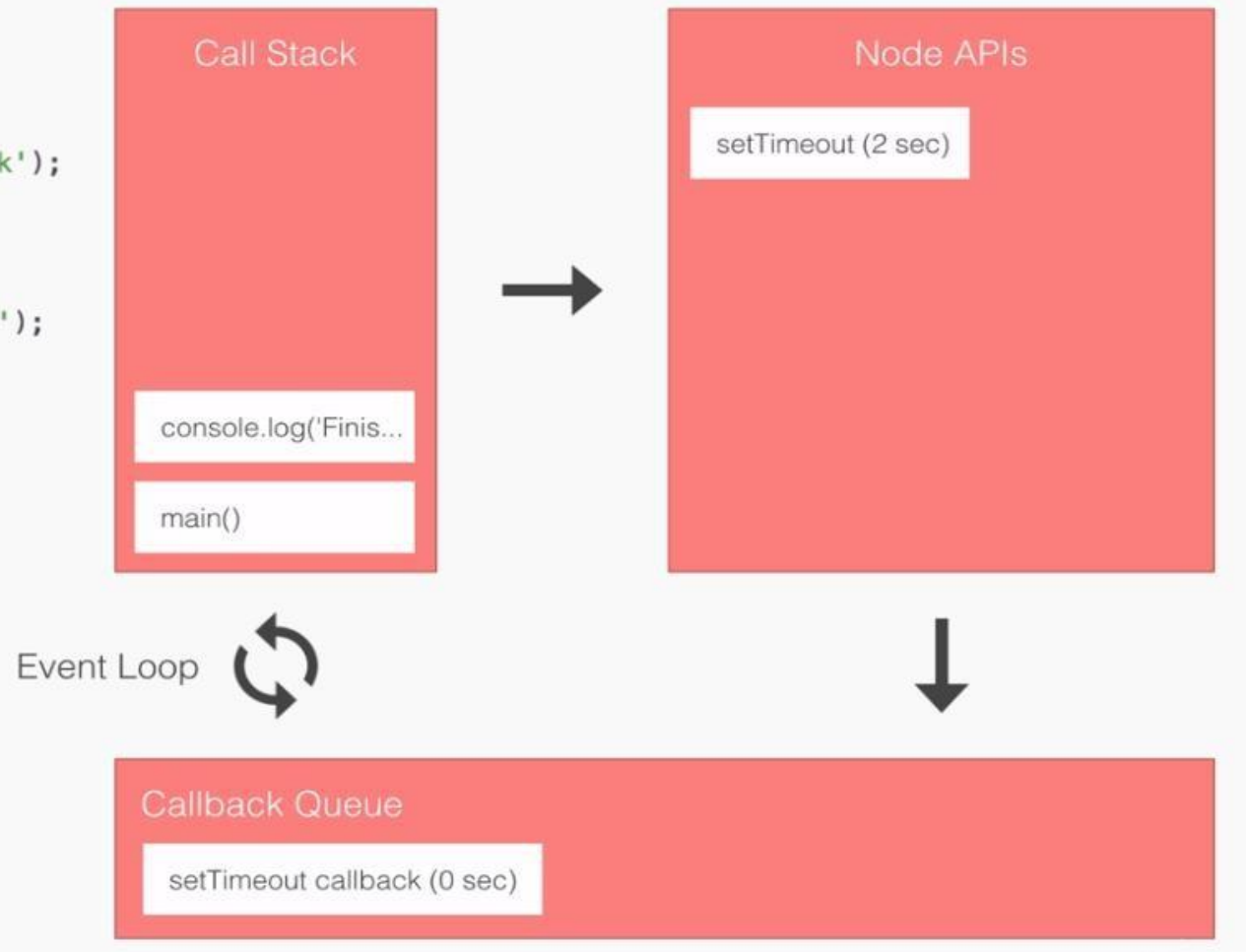
```
1 console.log('Starting app');
2
3 setTimeout(() => {
4   console.log('Inside of callback');
5 }, 2000);
6
7 setTimeout(() => {
8   console.log('Second setTimeout');
9 }, 0);
10
11 console.log('Finishing up');
12
```



► 4. Call Stack et Event loop

Exemple 3

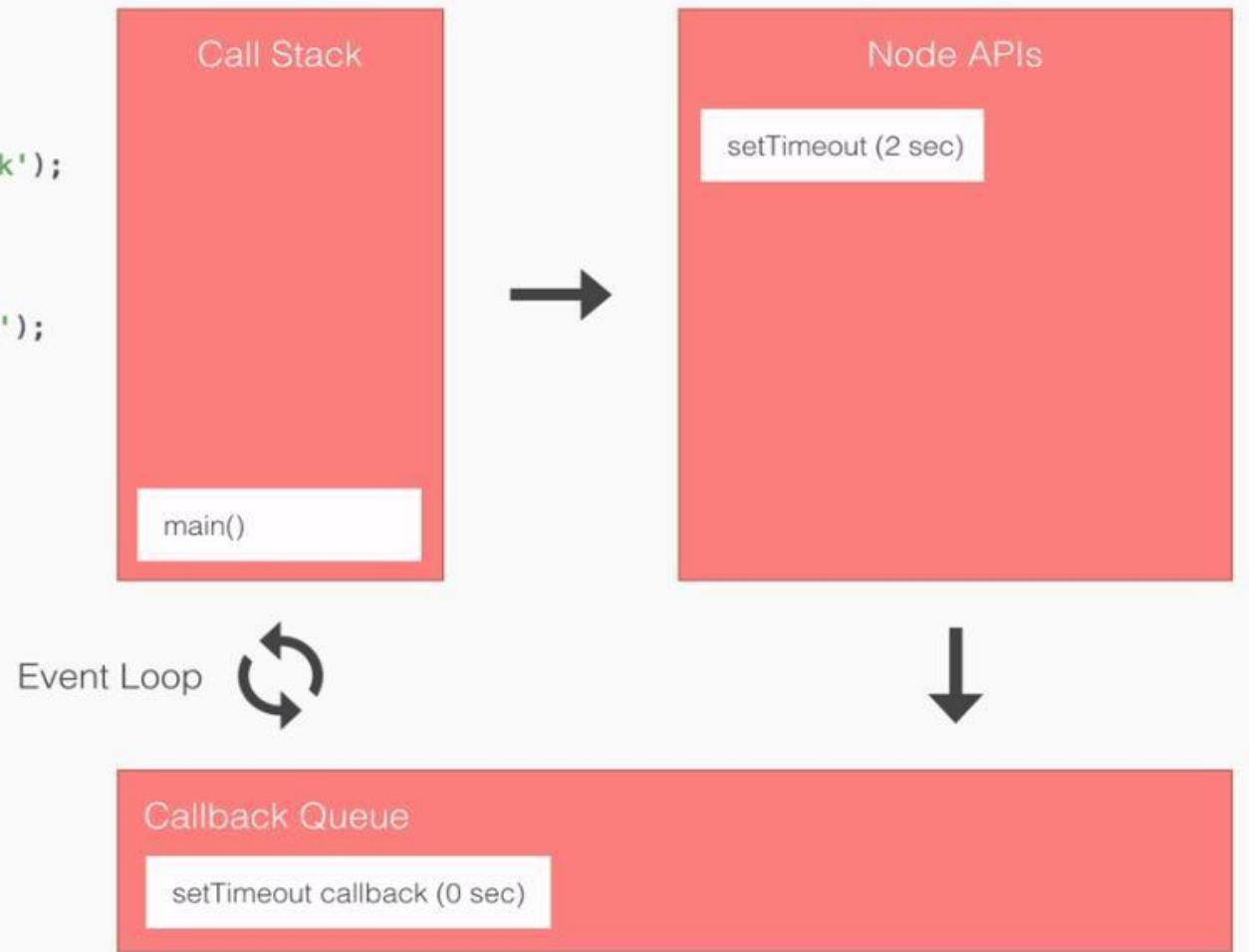
```
1 console.log('Starting app');
2
3 setTimeout(() => {
4   console.log('Inside of callback');
5 }, 2000);
6
7 setTimeout(() => {
8   console.log('Second setTimeout');
9 }, 0);
10
11 console.log('Finishing up');
12
```



▶ 4. Call Stack et Event loop

Exemple 3

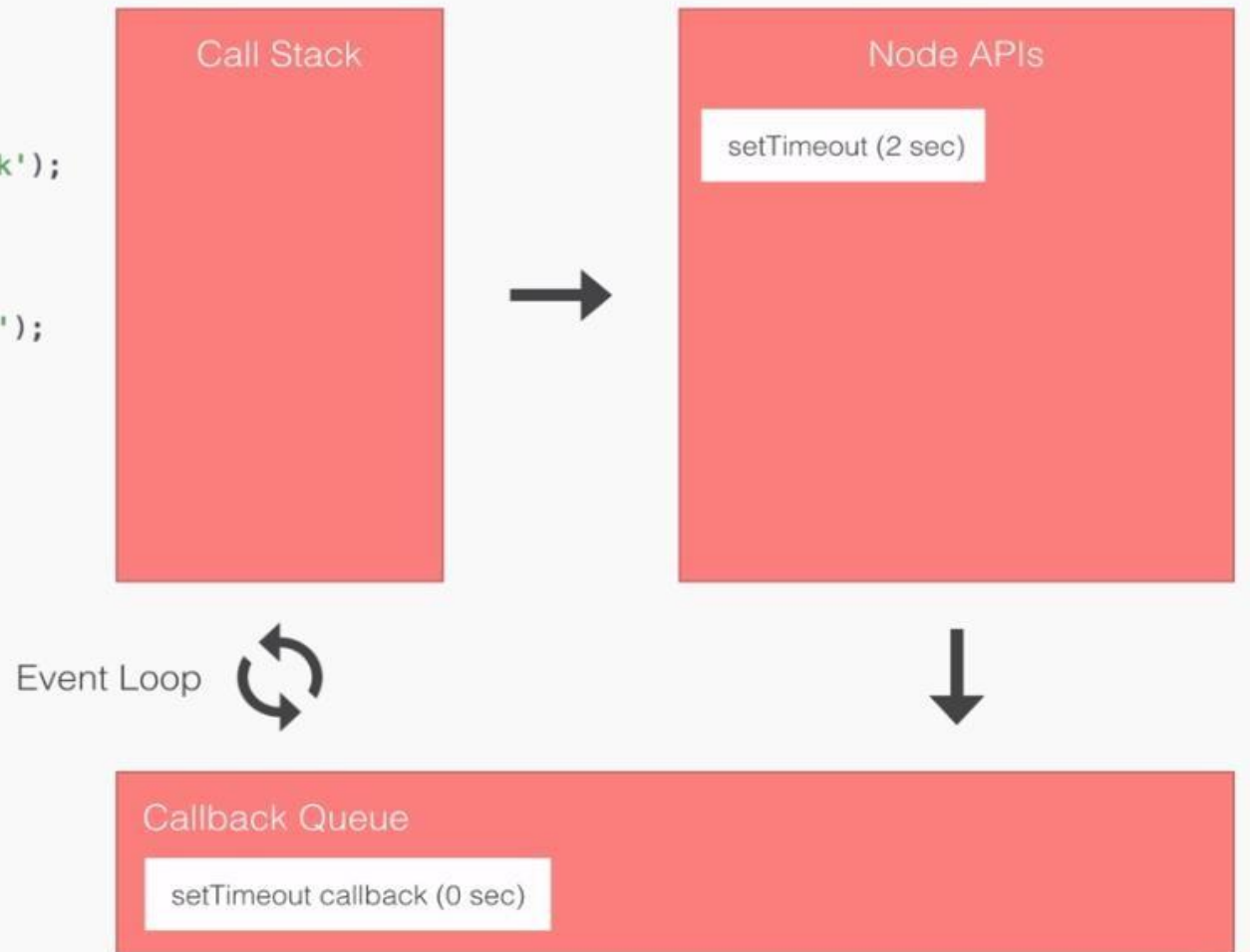
```
1 console.log('Starting app');
2
3 setTimeout(() => {
4   console.log('Inside of callback');
5 }, 2000);
6
7 setTimeout(() => {
8   console.log('Second setTimeout');
9 }, 0);
10
11 console.log('Finishing up');
12
```



▶ 4. Call Stack et Event loop

Exemple 3

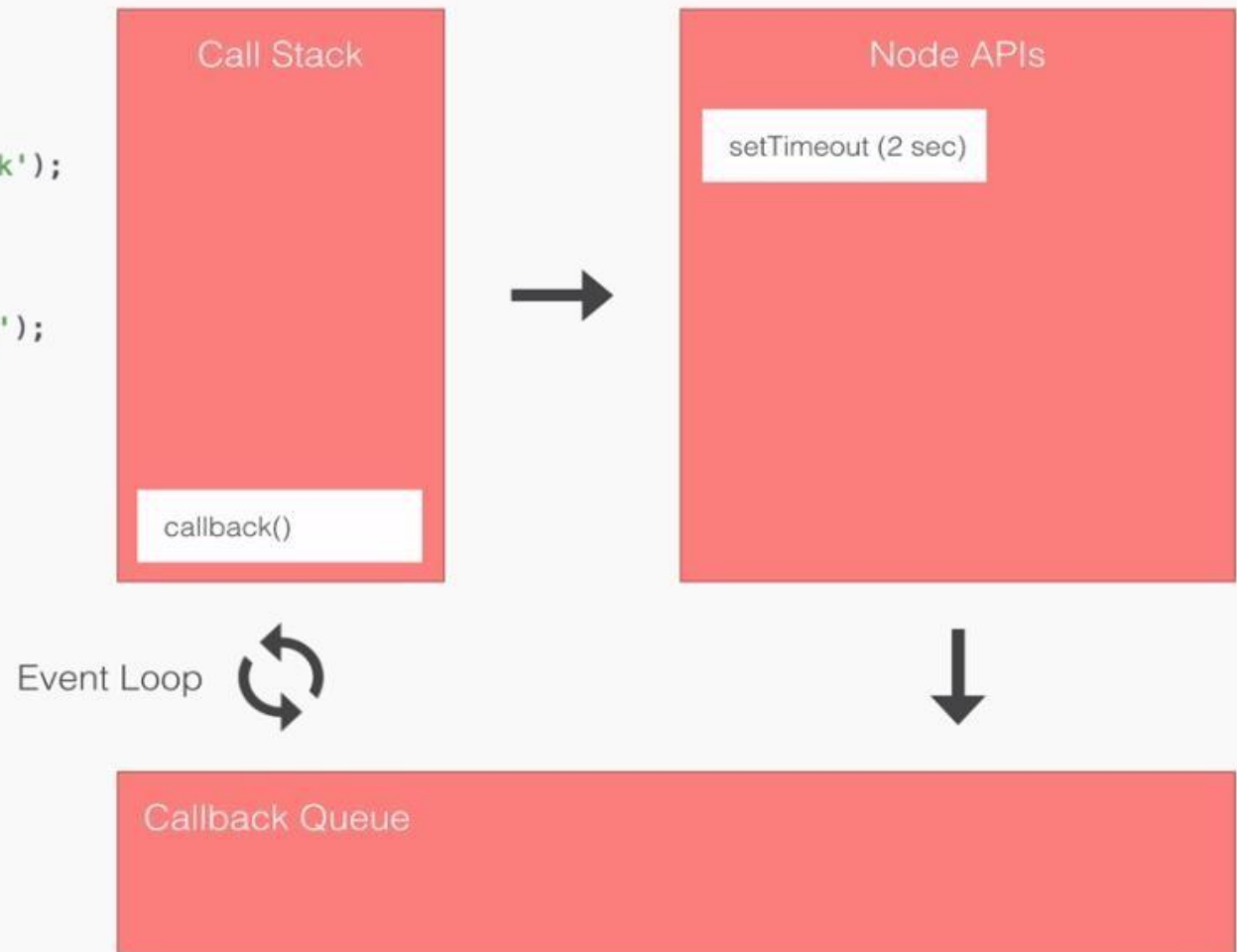
```
1 console.log('Starting app');
2
3 setTimeout(() => {
4   console.log('Inside of callback');
5 }, 2000);
6
7 setTimeout(() => {
8   console.log('Second setTimeout');
9 }, 0);
10
11 console.log('Finishing up');
12
```



▶ 4. Call Stack et Event loop

Exemple 3

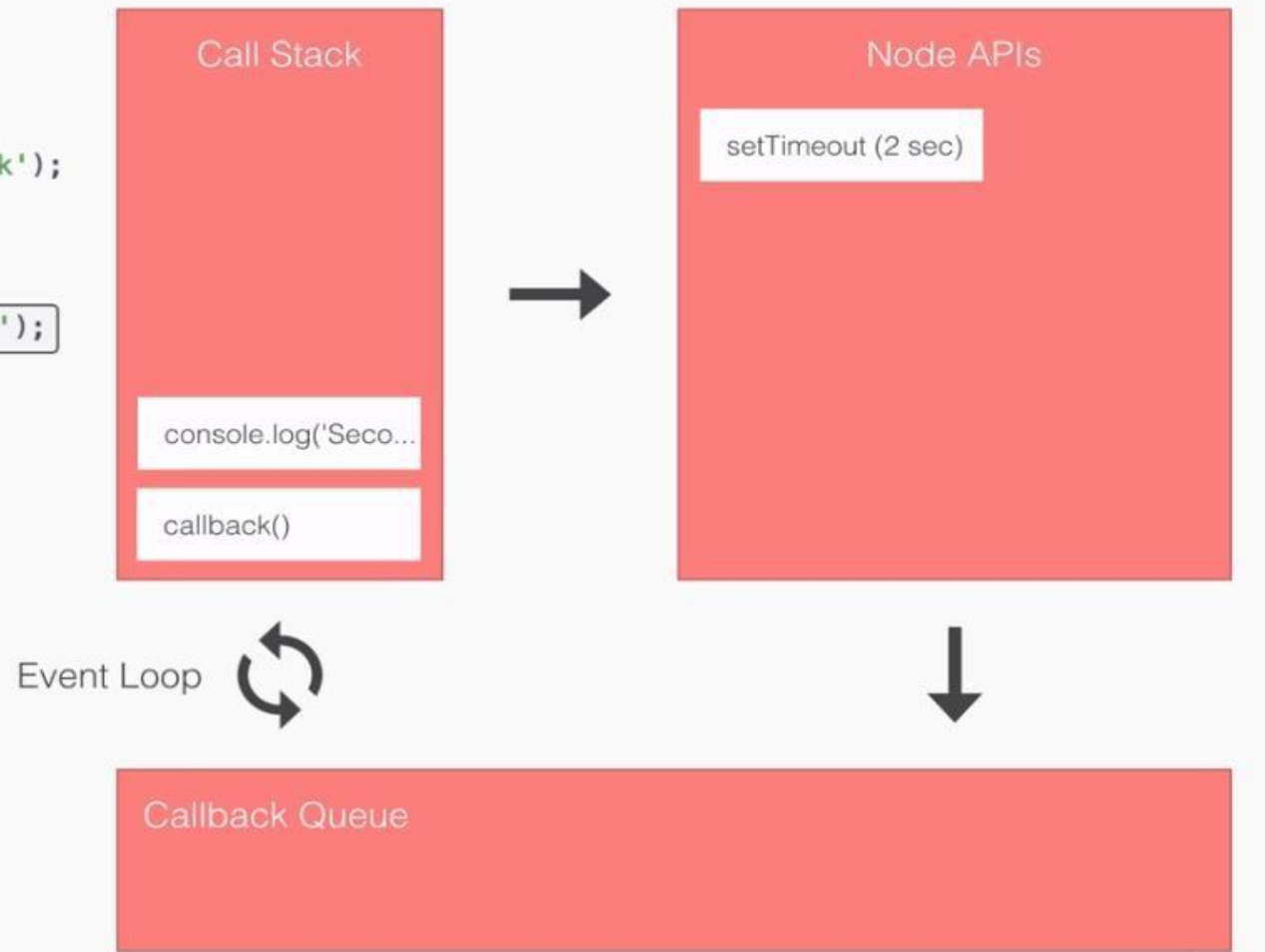
```
1 console.log('Starting app');
2
3 setTimeout(() => {
4   console.log('Inside of callback');
5 }, 2000);
6
7 setTimeout(() => {
8   console.log('Second setTimeout');
9 }, 0);
10
11 console.log('Finishing up');
12
```



▶ 4. Call Stack et Event loop

Exemple 3

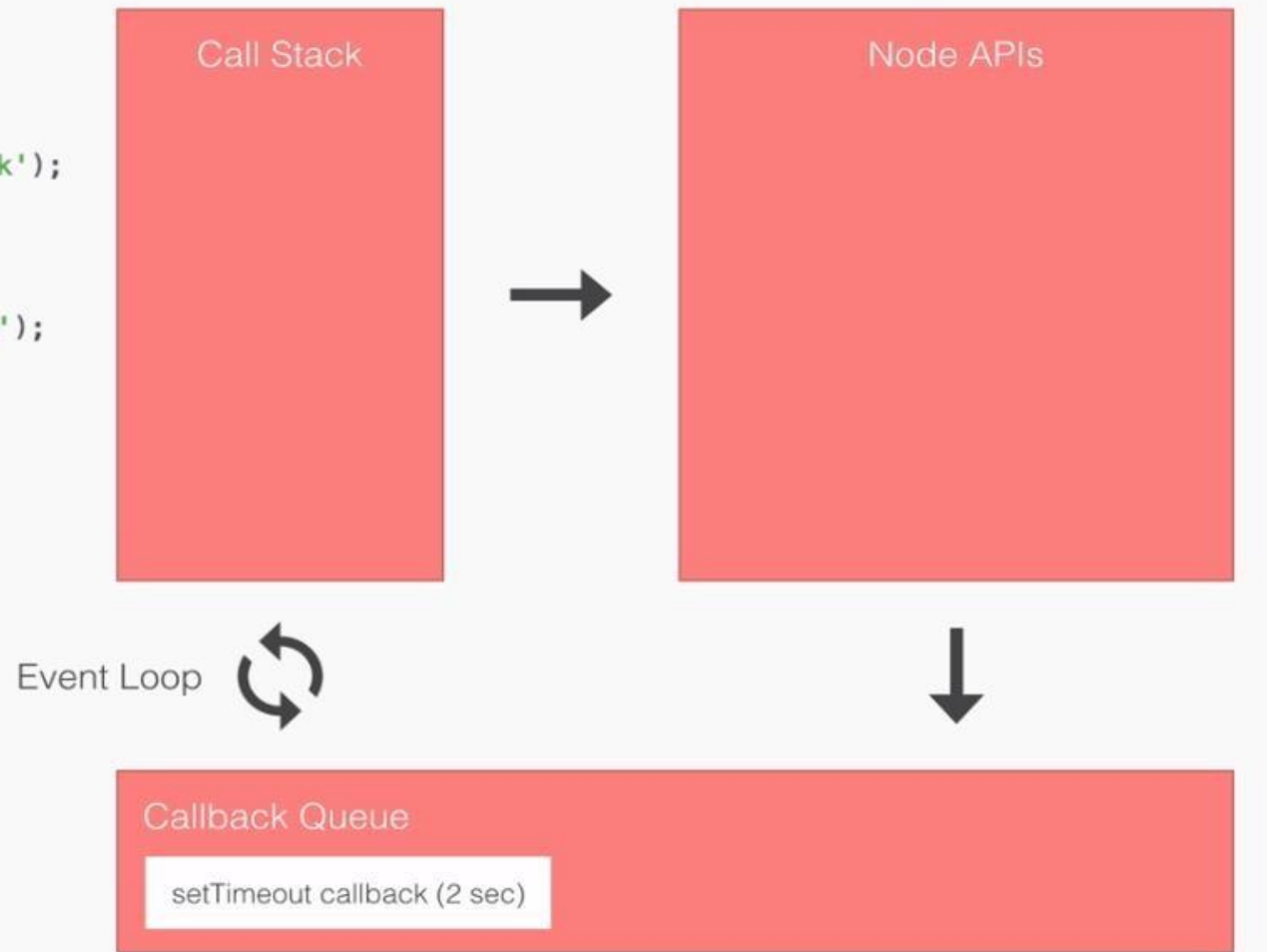
```
1 console.log('Starting app');
2
3 setTimeout(() => {
4   console.log('Inside of callback');
5 }, 2000);
6
7 setTimeout(() => {
8   console.log('Second setTimeout');
9 }, 0);
10
11 console.log('Finishing up');
12
```



▶ 4. Call Stack et Event loop

Exemple 3

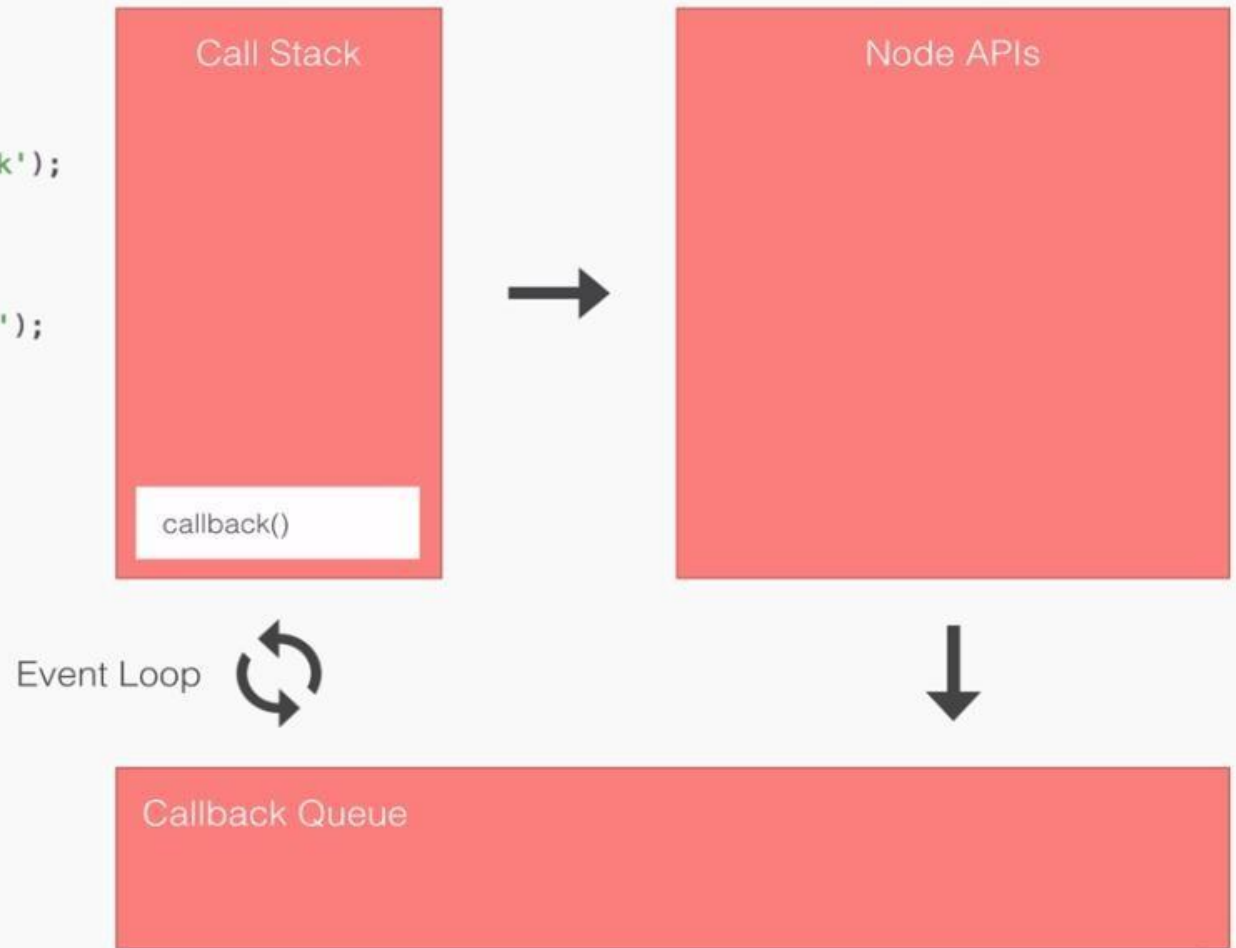
```
1 console.log('Starting app');
2
3 setTimeout(() => {
4   console.log('Inside of callback');
5 }, 2000);
6
7 setTimeout(() => {
8   console.log('Second setTimeout');
9 }, 0);
10
11 console.log('Finishing up');
12
```



▶ 4. Call Stack et Event loop

Exemple 3

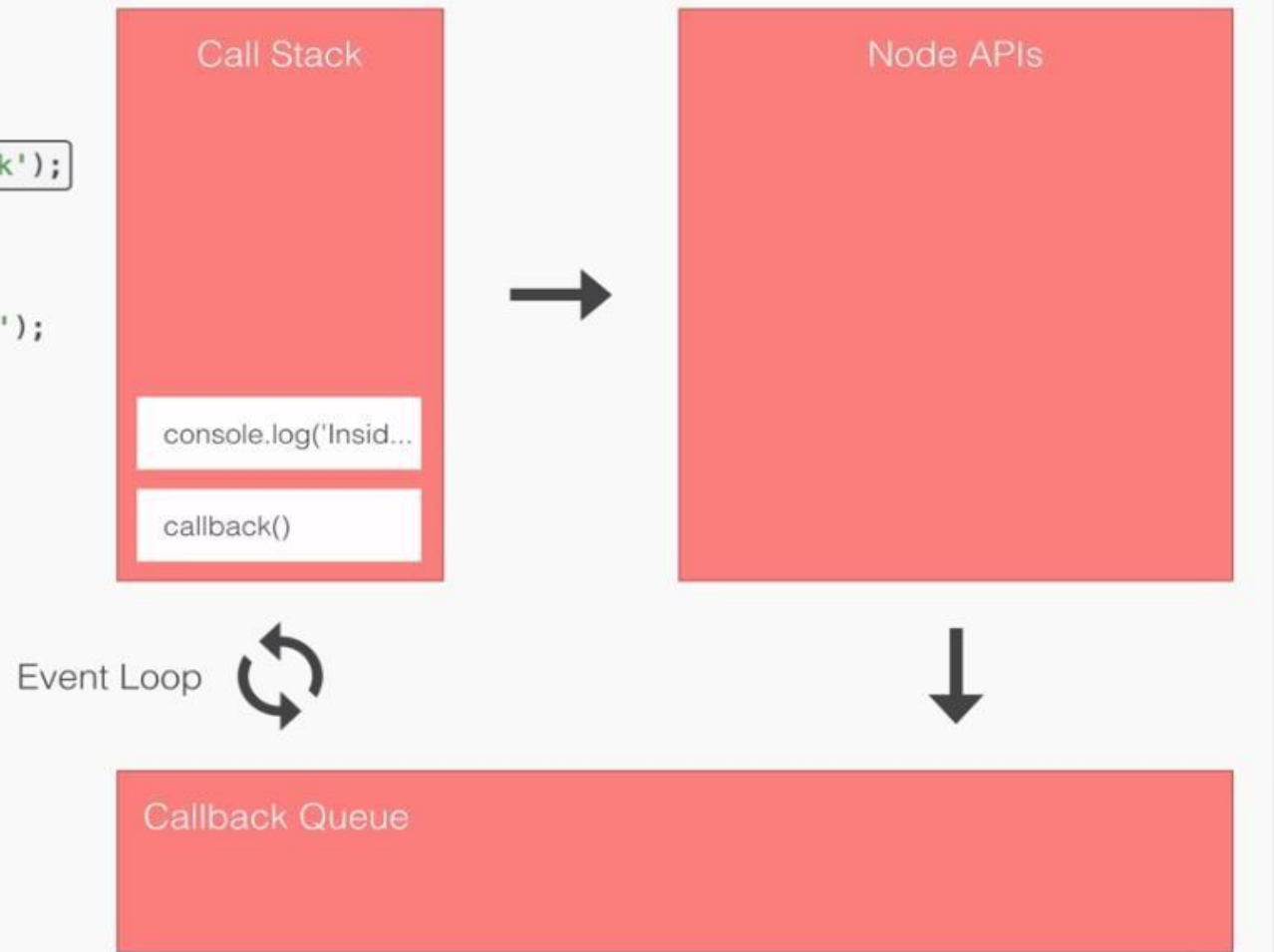
```
1 console.log('Starting app');
2
3 setTimeout(() => {
4   console.log('Inside of callback');
5 }, 2000);
6
7 setTimeout(() => {
8   console.log('Second setTimeout');
9 }, 0);
10
11 console.log('Finishing up');
12
```



► 4. Call Stack et Event loop

Exemple 3

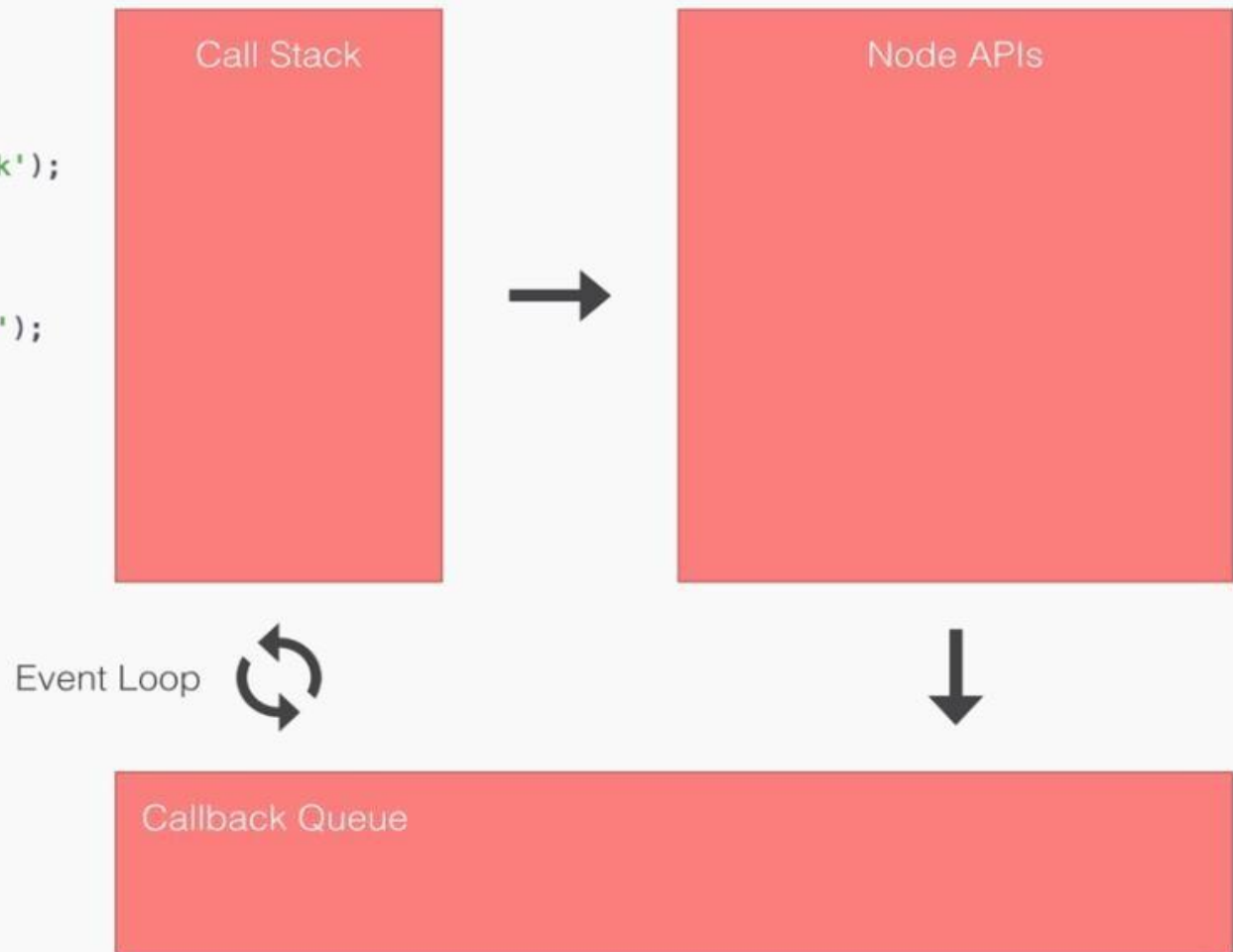
```
1 console.log('Starting app');
2
3 setTimeout(() => {
4   console.log('Inside of callback');
5 }, 2000);
6
7 setTimeout(() => {
8   console.log('Second setTimeout');
9 }, 0);
10
11 console.log('Finishing up');
12
```



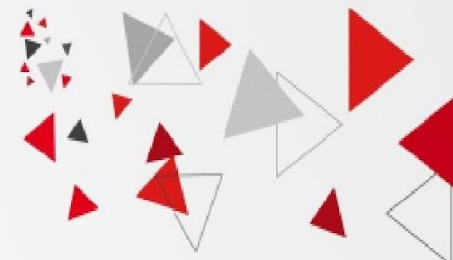
► 4. Call Stack et Event loop

Exemple 3

```
1 console.log('Starting app');
2
3 setTimeout(() => {
4   console.log('Inside of callback');
5 }, 2000);
6
7 setTimeout(() => {
8   console.log('Second setTimeout');
9 }, 0);
10
11 console.log('Finishing up');
12
```



► Exemples

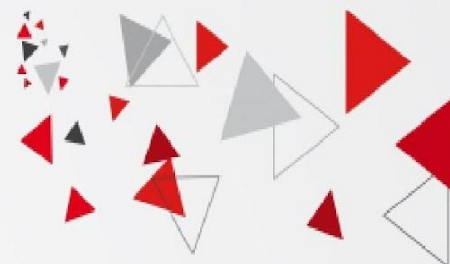


- La boucle d'événements(**Event Loop**) commence à exécuter des appels asynchrones une fois que la pile d'appels(**Call Stack**) est vide. Le mélange de code sync et async a un ordre d'évaluation distinct, le code synchrone est toujours exécuté avant l'async.

Donner l'ordre d'exécution de résultat (sans taper ou exécuter le code):

- ```
console.log("A")
setImmediate(_ => console.log("B"))
setImmediate(_ => console.log("C"))
console.log("D")
```
- ```
setImmediate(_ => setTimeout(_ => console.log("A")), 0)
setImmediate(_ => console.log("B"), 0)
setImmediate(_ => setTimeout(_ => console.log("C")), 10)
setTimeout(_ => console.log("D"), 10)
```

► Examples



3.

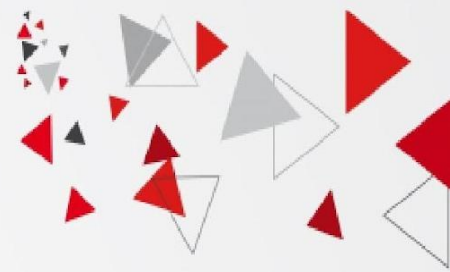
```
let x = "A"
setTimeout(_ => { console.log(x); x = "B" }, 3)
setTimeout(_ => { console.log(x); x = "C" }, 2)
setTimeout(_ => { console.log(x); x = "D" }, 1)
setTimeout(_ => { console.log(x) }, 4)
```

4.

```
let t1 = setTimeout(_ => { console.log("A"); setTimeout(_ => console.log("B")) }, 100)
let t2 = setTimeout(_ => { console.log("C"); setTimeout(_ => console.log("D")) }, 200)
clearTimeout(t1)
setTimeout(_ => clearTimeout(t2), 250)
```




Interprétation



1

1	A
2	D
3	B
4	C

2

1	B
2	A
3	C
4	D

3

1	A
2	D
3	C
4	B

4

1	C
2	D



► Examples

- Question: Quelle est la sortie du code ci-dessous et pourquoi?

```
setTimeout(function() {  
    console.log("A");  
}, 1000);  
  
setTimeout(function() {  
    console.log("B");  
}, 0);  
  
getDataFromDatabase(function(err, data) {  
    console.log("C");  
    setTimeout(function() {  
        console.log("D");  
    }, 1000);  
});  
  
console.log("E");
```



Interprétation

- Le compilateur ne s'arrêtera pas sur les méthodes setTimeout et getDataFromDatabase. Ainsi, la première ligne qu'il enregistrera est **E**.
- Les fonctions de rappel/callbacks (premier argument de `setTimeout`) s'exécutent après le setTimeout défini de manière asynchrone.

- ☐ **E** n'a pas de `setTimeout`
- ☐ **B** a un délai d'expiration défini de 0 millisecondes
- ☐ **A** a un délai d'expiration défini de 1000 millisecondes
- ☐ **D** doit demander une base de données, après il doit attendre 1000 millisecondes pour qu'il vienne après **A**.
- ☐ **C** est inconnu, lorsque les données de la base de données sont demandées. Cela peut être avant ou après **A**.



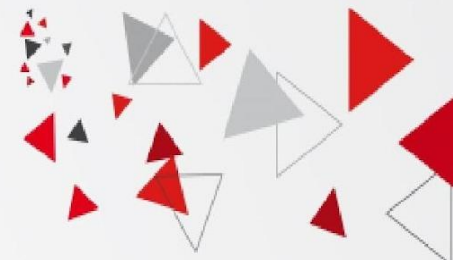
► Exemples

- Le code suivant enregistre les éléments du tableau de manière réursive. Mais, il échoue sur les grands tableaux. Utilisez le mode asynchrone pour résoudre ce problème.

```
function logArray(xs) {  
  if (!xs.length) {  
    return  
  }  
  let [x, ...xsRest] = xs  
  console.log(x)  
  logArray(xsRest) //échoue autour du 8000-9000 appel  
}  
  
function range(n) {  
  return [...Array(n).keys()]  
}  
  
logArray(range(10000))
```



Interprétation



```
function logArray(xs) {  
  if (!xs.length) {  
    return  
  }  
  let [x, ...xsRest] = xs  
  console.log(x)  
  setImmediate(_ => {  
    logArray(xsRest)  
  })  
}
```

► 5. Création de serveurs en NodeJS

Objectif: Créer un serveur Web

- Au niveau des composants matériels: un serveur web est un ordinateur qui stocke les fichiers qui composent un site web (par exemple les documents HTML, les images, les feuilles de style CSS, les fichiers JavaScript) .
- Cet ordinateur est connecté à Internet et est généralement accessible via un nom de domaine tel que google.com
- Au niveau des composants logiciels: un serveur web contient différents fragments qui contrôlent la façon dont les utilisateurs peuvent accéder aux fichiers hébergés. On trouvera a minima un serveur HTTP. Un serveur HTTP est un logiciel qui comprend les URL et le protocole HTTP(GET ,POST).



► Références

1. <https://nodejs.org/en/docs/guides/blocking-vs-non-blocking>
2. https://nodejs.org/api/events.html#events_events
3. https://medium.com/@shubhamkamboj_30683/nodejs-single-threaded-non-blocking-async-have-callback-queue-and-event-loop-7f0051ca322a
4. <https://www.freecodecamp.org/news/understanding-node-js-event-driven-architecture-223292fcbc2d/>