Componentes principais do Node.js: 2-3 - Libuv

Como eu dito anteriormente, nem setTimeout , setInterval nem setImmediate existem em JavaScript. Essas são funções de timer e são funções assíncronas tratadas por uma conhecida biblioteca C chamada libuv.

Libuv é uma biblioteca C que permite ao Node.js criar:

- Async Functions
- Threads
- Timers
- Child Processes
- Event Loops

E muito mais

Ela foi originalmente feito para ajudar o Node.js, mas é extensível e também pode ser usado em outras linguagens. E o resultado que ele tem com seu loop de eventos é o que fortalece o núcleo do Node.js e o transforma em um dos maiores e mais usados runtimes do mundo.

Como a Libuv pode fazer isso? Permitindo que o operações assincronas sejam executadas, como por exemplo o setTimeout.

Timers são funções assincronas executadas em segundo plano, e retornam o contexto principal quando terminam. Cada vez que executamos setTimeout em JS, é a Libuv executando esse código em segundo plano e chamando o callback fornecido.
O loop de eventos seria então um loop infinito que fica buscando novos eventos e chamando as funções fornecidas de volta ao terminar a execução

Isso é o que todos conhecemos como parte de sigle thread do JavaScript. E, mesmo que você possa criar e manipular threads com Libuv, cada tarefa eventualmente enviará uma mensagem de volta ao loop de eventos para manter a consistência e a ordem, isso evitará que as tarefas entrem em conflito entre si e causem conflitos.

Em resumo, é assim que o Node.js pode trabalhar com multithreads usando o módulo Worker Threads.
Cada vez que uma tarefa for concluída, ela enviará uma mensagem de volta ao loop de eventos e o loop de eventos chamará seu callback e removerá a função da fila.

Componentes principais do Node.js: 3-3 - Camada C++

V8 é o mecanismo que interpreta JavaScript e pode chamar funções C++ personalizadas e libuv é a biblioteca que fornece o loop de eventos e outros recursos, como threading e execução de tarefas assíncronas no sistema operacional.

A última parte do sistema Node.js é o que chamo de camada C++.
A camada C++ é o mediador entre o código JavaScript que você escreve, o mecanismo V8 e o Libuv. Ele lida com respostas do V8 e Libuv e responde à camada JavaScript.

Imagine o seguinte pipeline:

1. Execute um programa C++ e envie um arquivo JS como argumento Executado pela camada C++, ou seja, seu programa.
2. Leia o conteúdo do arquivo Também executado pela camada C++.
3. Envie a string para o motor V8 e ele transforma o código em um objeto C++ V8 avaliando a string que você enviou.
4. Aguarde que eventos, temporizadores, processos e outras chamadas assíncronas concluam o processamento
   Este é o loop de eventos da Libuv rodando como um loop infinito.
5. Libuv conclui a tarefa e chama as funções C++ fornecidas. A camada C++ recebe a resposta.
6. A camada C++ chama a API V8 para responder à função JS. A camada C++ invoca a função de callback fornecida e finaliza a solicitação.

O que são Funções de Callback e sua Utilidade

As funções de callback em JavaScript são funções passadas como argumentos para outras funções. Elas são executadas após a conclusão de uma operação, permitindo um fluxo de trabalho assíncrono. Isso é crucial em operações que dependem de tempo, como solicitações de rede ou eventos de usuário.

Benefícios da Programação Assíncrona em JavaScript

Na programação assíncrona, operações não bloqueiam a execução do código. Isso é vital em JavaScript, especialmente para aplicações web, onde a resposta imediata a interações do usuário e a eficiência no carregamento são cruciais. A programação assíncrona, utilizando callbacks, promove uma experiência de usuário mais fluida e responsiva.

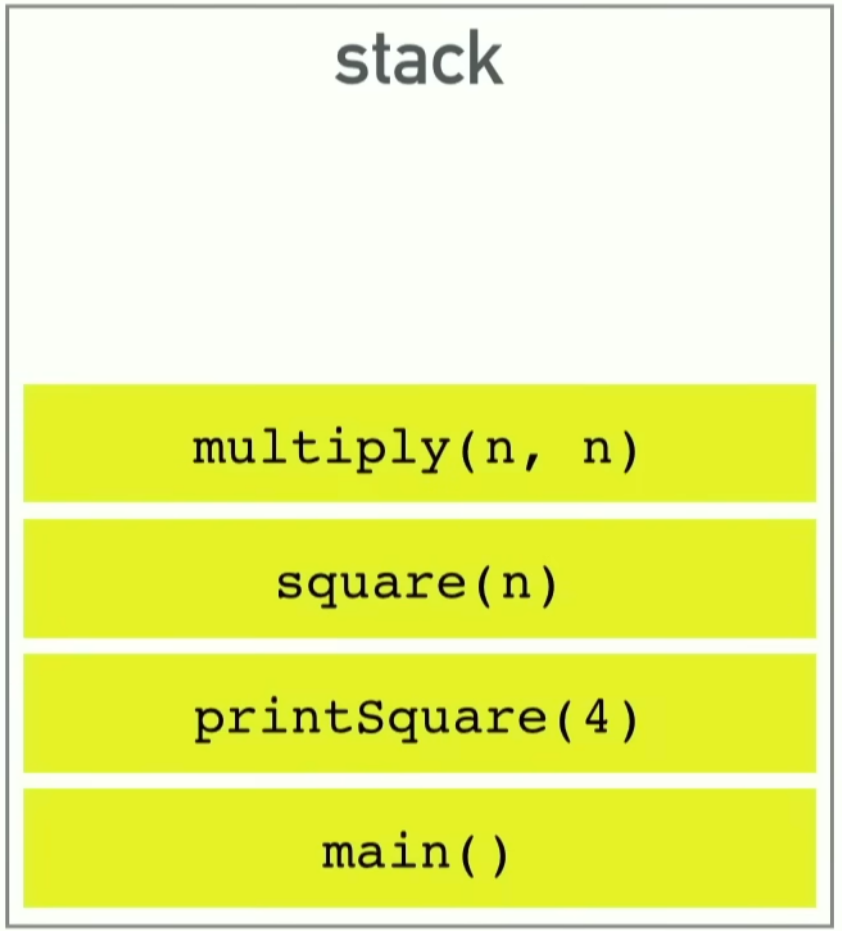
Por exemplo, em JavaScript, um callback pode ser usado para manipular a resposta de uma solicitação HTTP. Quando os dados são recebidos, a função de callback é chamada para processá-los. Este modelo é amplamente utilizado em APIs, interações com bancos de dados e manipulação de eventos de usuário.

Um pequeno exemplo de callbacks

```
function multiply(a,b) {
  return a * b;
}

function square(n) {
  return multiply (n, n)
}

function printSquare(n) {
  let squared = square(n);
  console.log(squared);
}

printSquare(4);
```

Call Stack

```
function multiply(a, b) {
    return a * b;
}

function square(n) {
    return multiply(n, n);
}

function printSquare(n) {
    var squared = square(n);
    console.log(squared);
}

printSquare(4);
```



Mais um exemplo de callback

```
function processarUsuario(id, callback) {
  // Simula uma operação de busca de dados
  const usuario = {
      id: id,
      nome: "Luiz"
  };
  callback(usuario);
}

processarUsuario(1, function(usuario) {
  console.log("Nome do usuário:", usuario.nome);
});
```

Fluxos Assíncronos

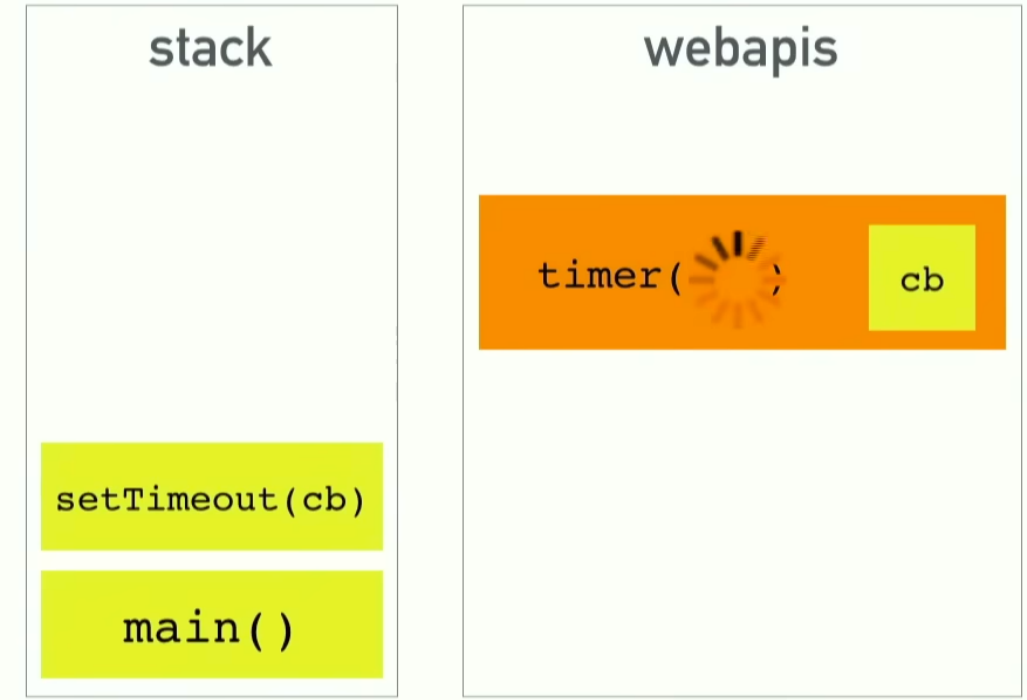
Vamos entender a ordem de execução do código e qual será sua saída subsequente:

```
console.log(1);
console.log(2);
console.log(3);
console.log(4);
```

Qual será a ordem agora?

```
console.log(1);
console.log(2);
setTimeout(() =>{
  console.log('callback function')
}, 5000);
console.log(3);
console.log(4);
```

```
console.log('Hi');

setTimeout(function cb() {
    console.log('there');
}, 5000);

console.log('JSConfEU');
```



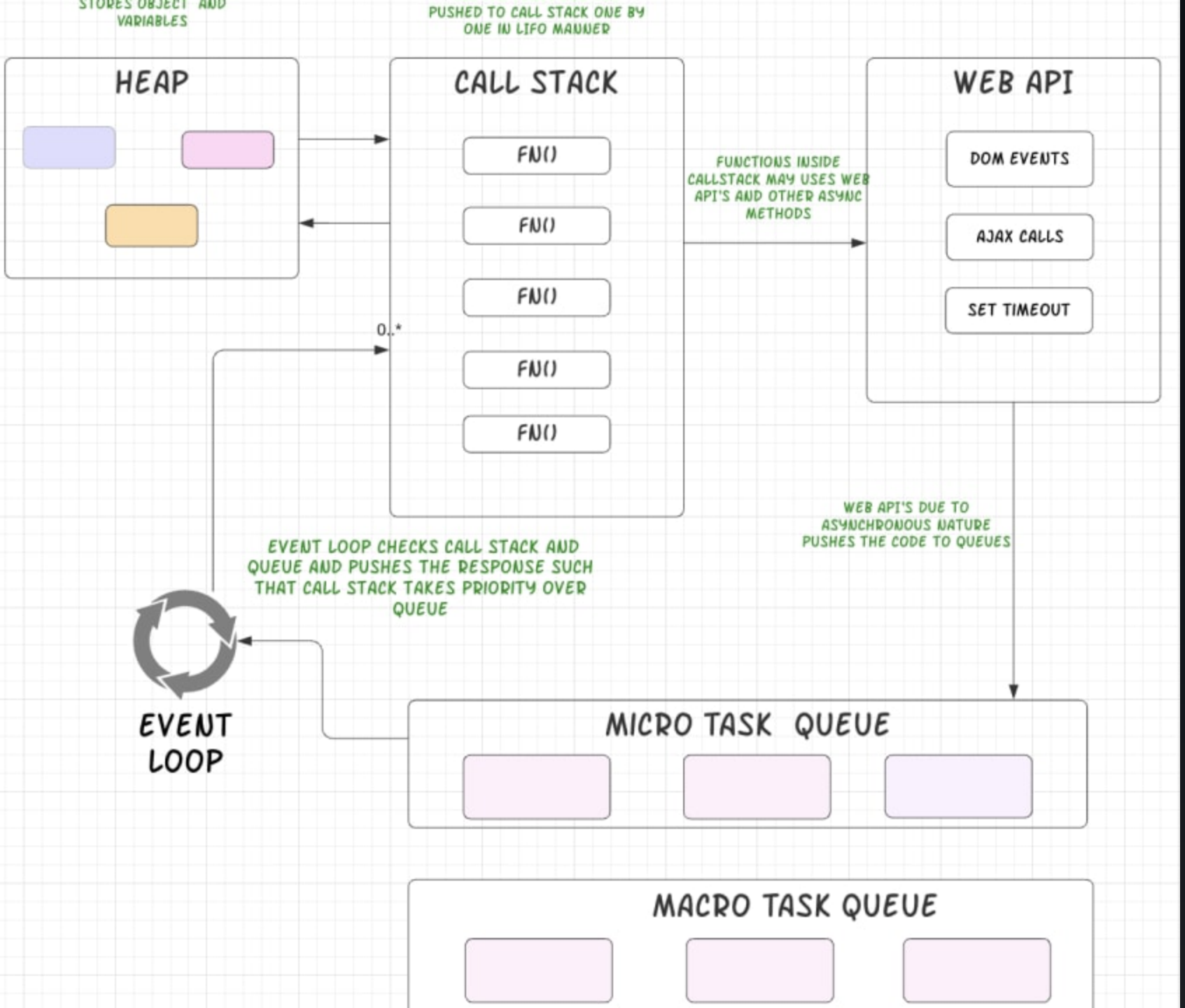
Console

```
Hi
```

event loop

task queue

;


STORES OBJECT AND VARIABLES
PUSHED TO CALL STACK ONE BY ONE IN LIFO MANNER
HEAP
CALL STACK
FN()
FN()
FN()
FN()
FN()
0..*
FUNCTIONS INSIDE CALLSTACK MAY USES WEB API'S AND OTHER ASYNC METHODS
WEB API
DOM EVENTS
AJAX CALLS
SET TIMEOUT
WEB API'S DUE TO ASYNCHRONOUS NATURE PUSHES THE CODE TO QUEUES
EVENT LOOP CHECKS CALL STACK AND QUEUE AND PUSHES THE RESPONSE SUCH THAT CALL STACK TAKES PRIORITY OVER QUEUE
EVENT LOOP
MICRO TASK QUEUE
MACRO TASK QUEUE

Entendendo cada sessão:

Heap - Armazena todas as referências de objetos e variáveis que definimos em nossa função.

Call Stack - Todas as funções que usamos em nosso código são empilhadas aqui no modo **LIFO**, de modo que a última função esteja no topo e a primeira função esteja na parte inferior.

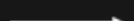
APIs da Web - Essas APIs são fornecidas pelo navegador, que fornece funcionalidade adicional sobre o mecanismo V8. As funções que usam essas APIs são enviadas para este contêiner que, após a conclusão da resposta da API da Web, é retirado deste contêiner.

Filas - As filas são usadas para calcular a resposta do código assíncrono de forma que não bloqueie a execução do mecanismo.

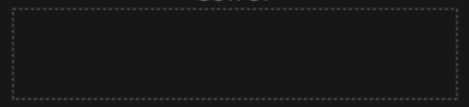
- **Fila de tarefas macro** - Esta fila executa funções assíncronas como eventos DOM, chamadas Ajax e setTimeout e tem prioridade mais baixa que a fila de tarefas.
- **Fila de micro tarefas** - Esta fila executa funções assíncronas que usam promessas e tem maior precedência sobre a fila de mensagens.

O loop de eventos verifica a pilha de chamadas; se a pilha estiver vazia, ele envia as funções das filas para a pilha de chamadas e a executa. As funções já presentes recebem prioridade mais alta e são executadas primeiro em comparação com funções na fila de mensagens.

1 || Functions get **pushed to** the call stack when they're **invoked** and **popped off** when they **return a value**



```
function greet() {
  return "Hello!"
}

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

greet()
respond()
```



Made with ❤ by **Lydia Hallie**

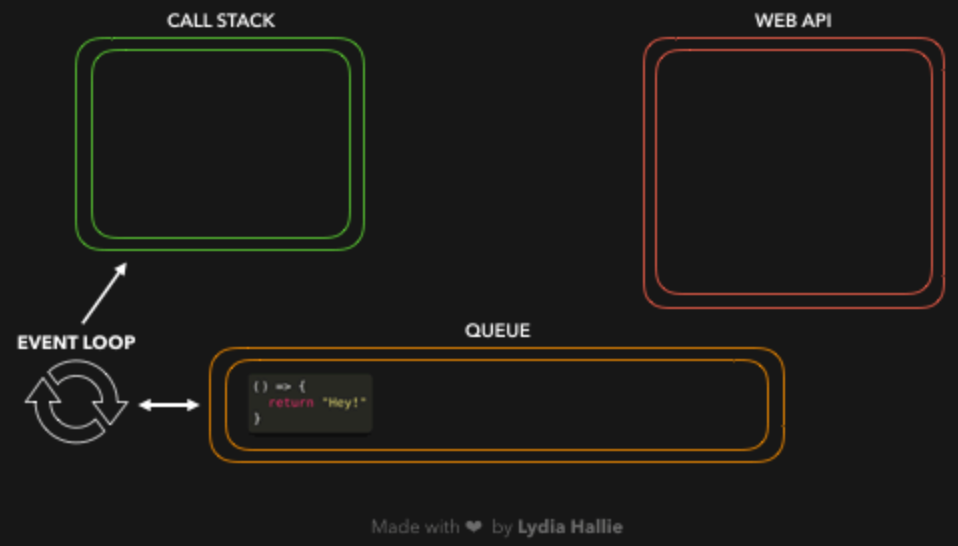
2 || **`setTimeout`** is provided to you by the *browser*, the **Web API** takes care of the callback we pass to it.

Made with ❤ by **Lydia Hallie**

3 || When the timer has finished (1000ms in this case), the callback gets passed to the **callback queue**

Made with ❤ by **Lydia Hallie**

4 || The **event loop** looks at the **callback queue** and the **call stack**.
If the call stack is empty, it pushes the first item in the queue onto the stack.

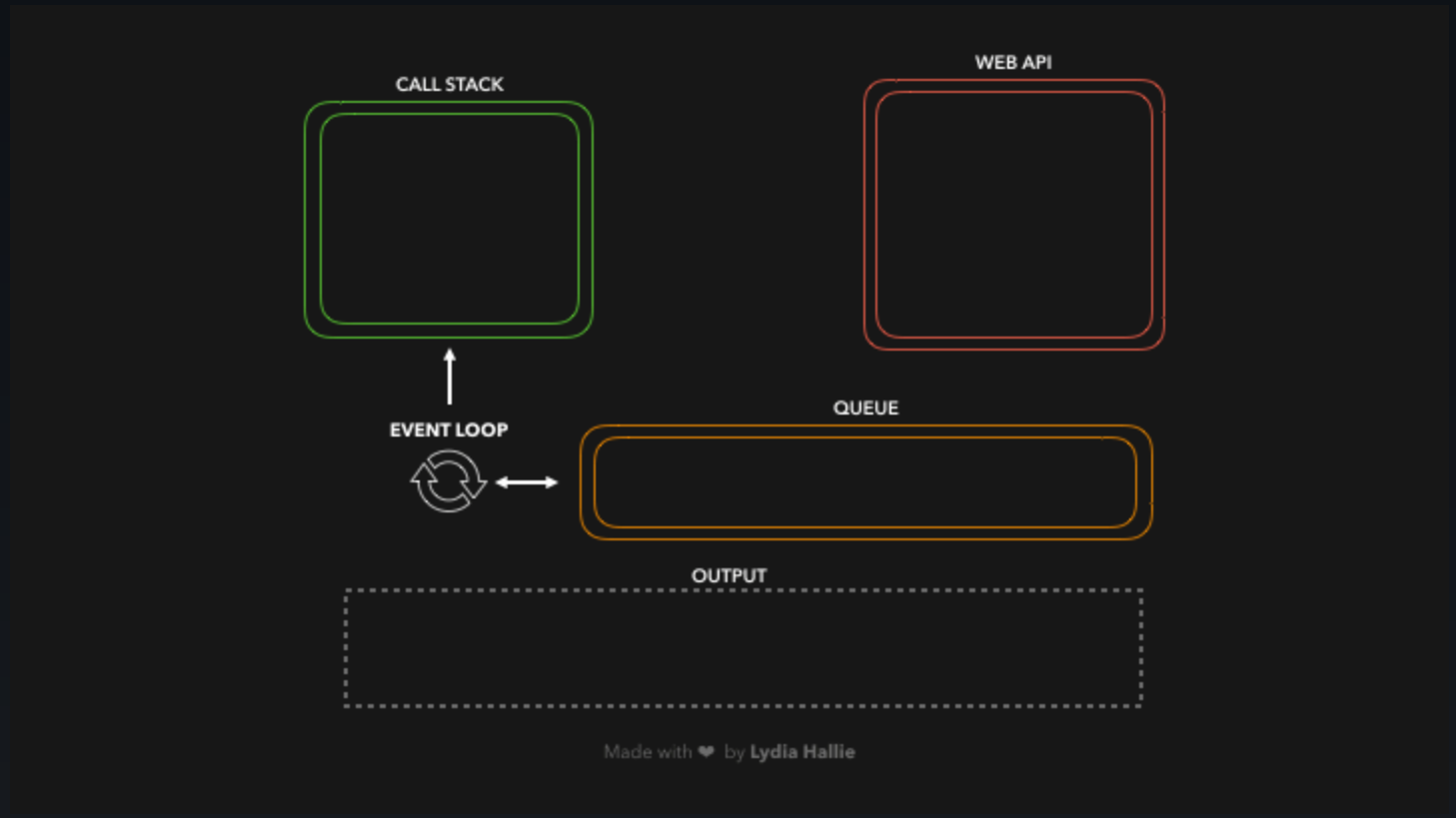


5 || The callback is added to the call stack and executed.
Once it returned a value, it gets popped off the call stack.

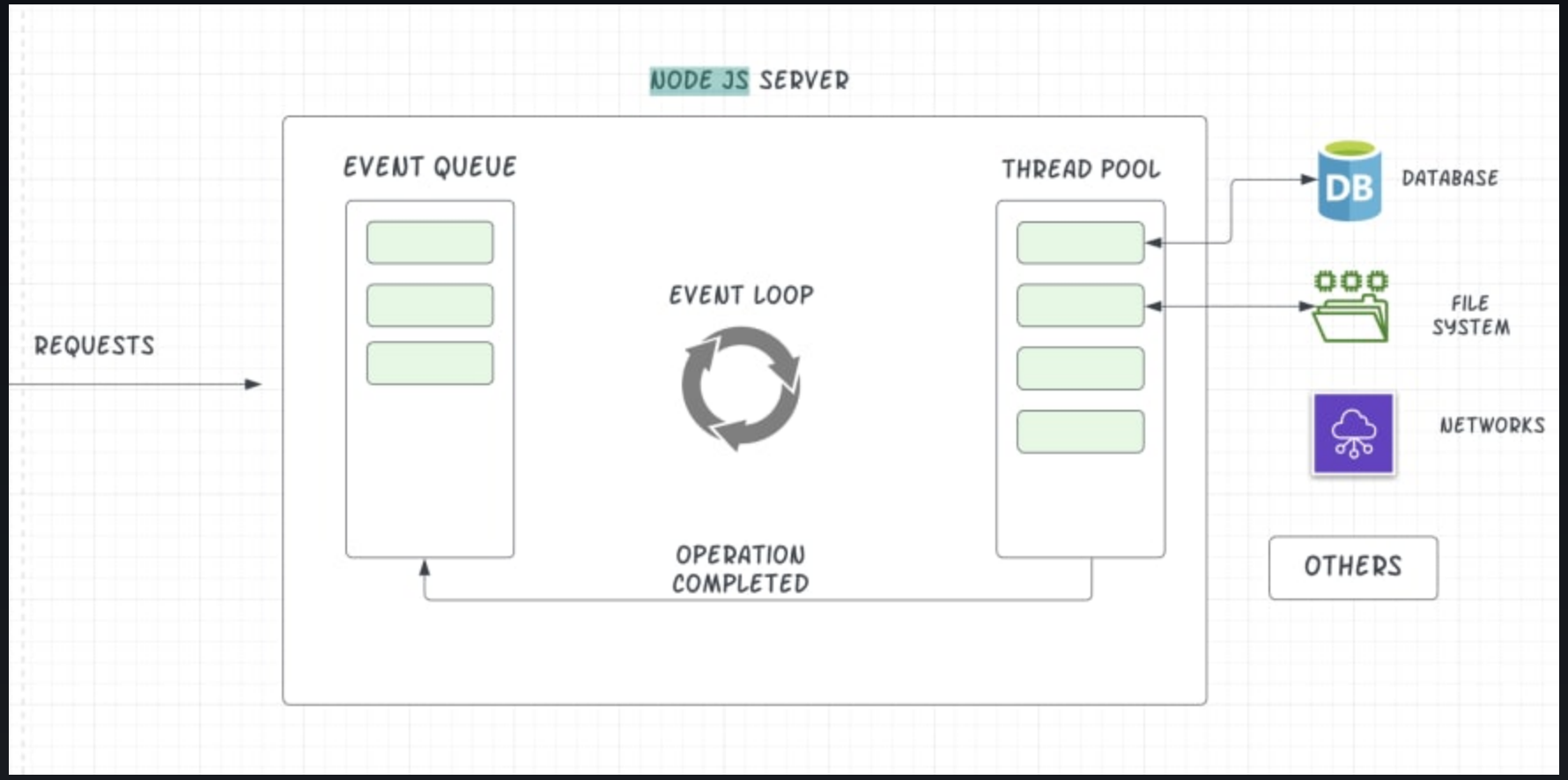
OUTPUT

```
function greet() {
  return "Hello!"
}

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

greet()
respond()
```

```
const foo = () => console.log("First");
const bar = () => setTimeout(() => console.log("Second"), 500);
const baz = () => console.log("Third");

bar();
foo();
baz();
```


CALL STACK
WEB API
EVENT LOOP
QUEUE
OUTPUT
Made with ❤ by Lydia Hallie

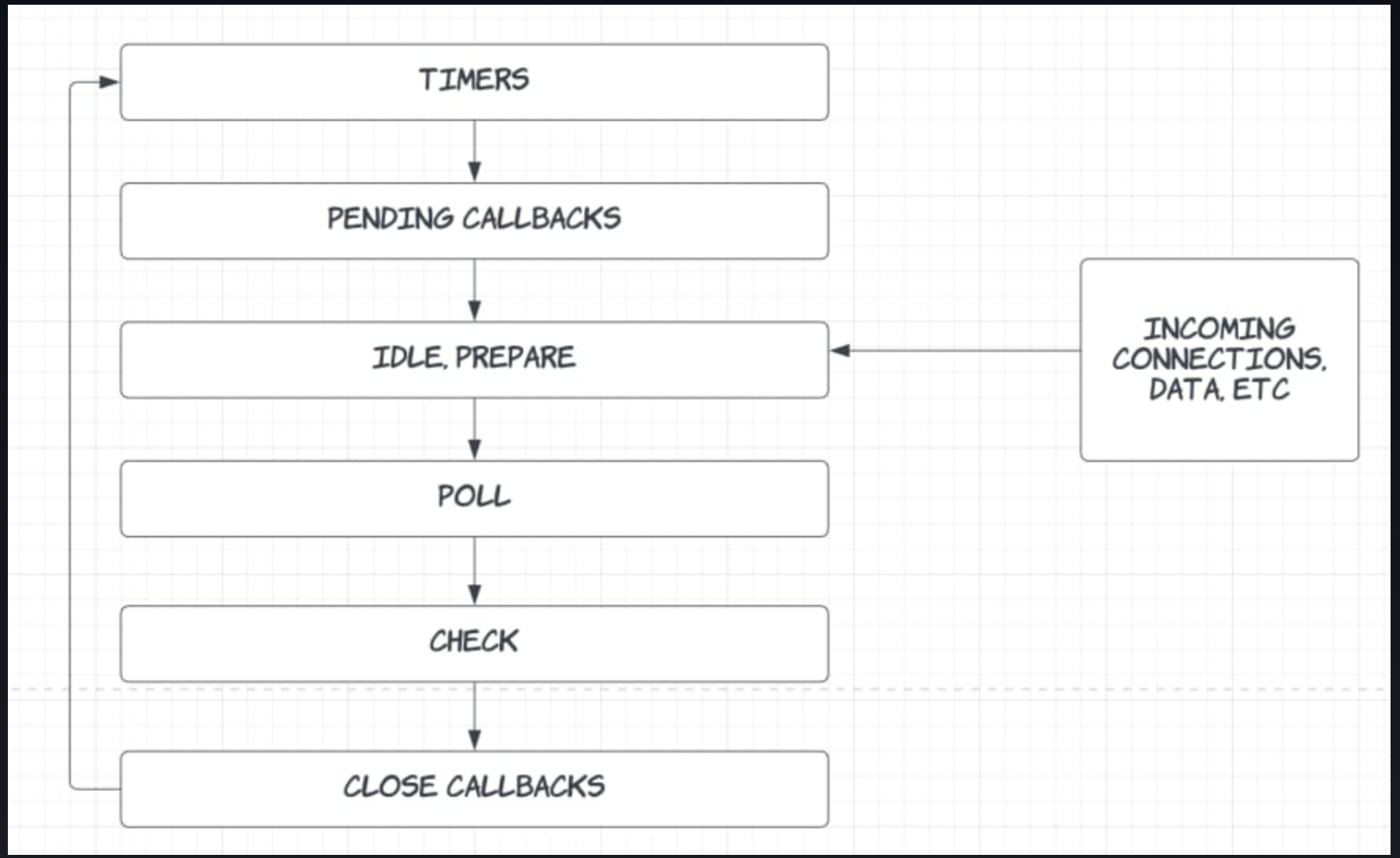
Node.js Event Loop



Event Queue - Após a conclusão do Thread Pool, uma função de callback é emitida e enviada para a fila de eventos. Quando a pilha de chamadas está vazia, o evento passa pela fila de eventos e envia o callback para a pilha de chamadas.

Threads Pool - O thread pool é composto por 4 threads que delegam operações que são muito pesadas para o loop de eventos. Operações de E/S, abertura e fechamento de conexões, setTimeouts são exemplos de tais operações.

O **loop de eventos** no Node Js possui diferentes fases que possuem fila **FIFO** de callbacksa para execução. Quando o loop de eventos entra em uma determinada fase, ele opera callbacks nessa fila de fase até que a fila se esgote e o número máximo de callbacks seja executado e então passa para a próxima fase.


TIMERS
PENDING CALLBACKS
IDLE, PREPARE
POLL
CHECK
CLOSE CALLBACKS
INCOMING CONNECTIONS, DATA, ETC

Vamos usar como base o seguinte código:

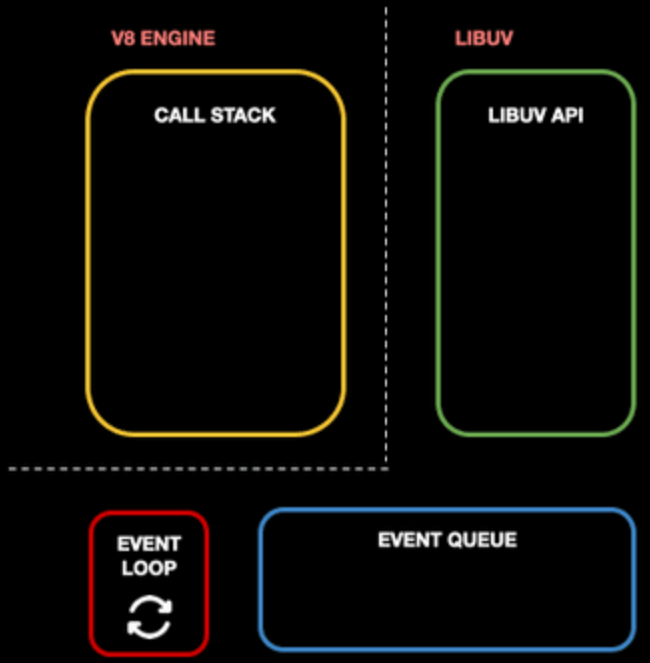
```
console.log("Iniciando o Node.js");
db.query("SELECT * FROM public.cars", function(err, res) {
  console.log("Query executed");
});
console.log("Antes do resultado da consulta");
```

O mecanismo JavaScript V8 gerencia uma call stack, uma peça essencial que rastreia qual parte do nosso programa está em execução. Sempre que invocamos uma função JavaScript, ela é enviada para a pilha de chamadas. Quando a função chega ao fim ou a uma instrução return, ela é retirada da pilha.

Em nosso exemplo, a linha de código `console.log('Starting Node.js')` é adicionada à pilha de chamadas e imprime `Iniciando o Node.js` no console. Ao fazer isso, ele chega ao final da função `log` e é removido da pilha de chamadas.

An invoked function is added to the call stack. Once it returns a value, it is popped off.

```
console.log("Starting Node.js");

db.query("SELECT * FROM public.cars", function (err, res)  {
  console.log("Query executed");
});

console.log("Before query result");
```



OUTPUT

Made with ❤️ by @FabriLallo and @AndrewHu368

A linha de código a seguir é uma consulta ao banco de dados. Essas tarefas são imediatamente interrompidas porque podem demorar muito. Eles são passados para o Libuv, que os trata de forma assíncrona em segundo plano. Ao mesmo tempo, o Node.js pode continuar executando outro código sem bloquear seu único thread.

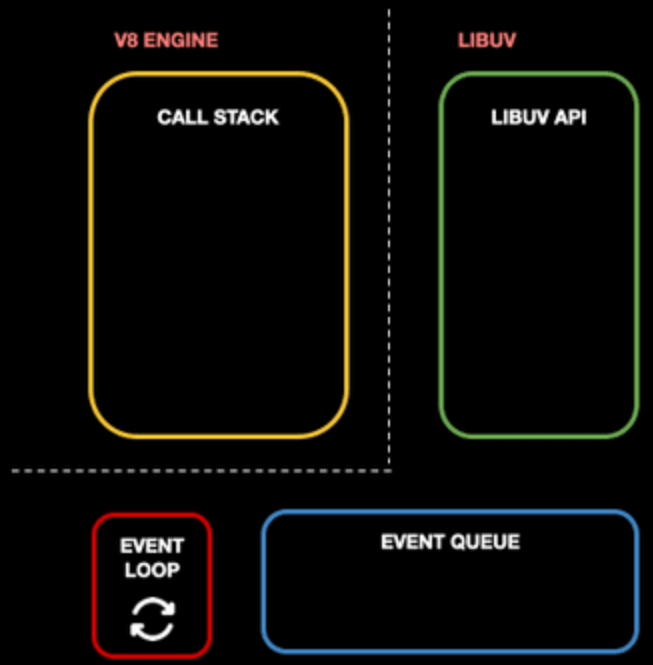
No futuro, o Node.js saberá o que fazer com a consulta porque associamos uma função de retorno de chamada a instruções para lidar com o resultado ou erro da tarefa.

Database queries or other I/O ops do not block Node.js single thread because Libuv API handles them.

```
console.log("Starting Node.js");

db.query("SELECT * FROM public.cars", function (err, res)  {
  console.log("Query executed");
});

console.log("Before query result");
```

OUTPUT

```
Starting Node.js
```

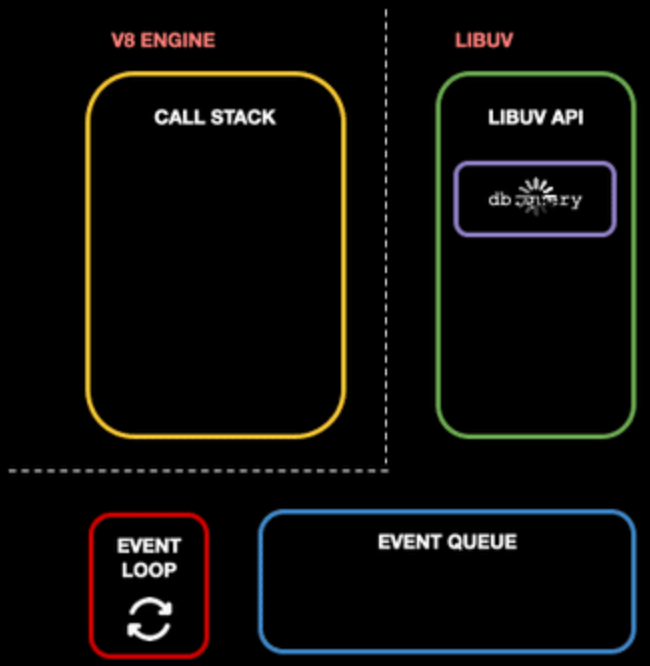


Made with ❤️ by @FabriLallo and @AndrewHu368

Embora o Libuv lide com a consulta em segundo plano, nosso JavaScript não é bloqueado e pode continuar com o console.log(”Antes do resultado da consulta”).

While Libuv asynchronously handles I/O operations, Node.js single thread keeps running code.

```
console.log("Starting Node.js");

db.query("SELECT * FROM public.cars", function (err, res)  {
  console.log("Query executed");
});

console.log("Before query result");
```



OUTPUT

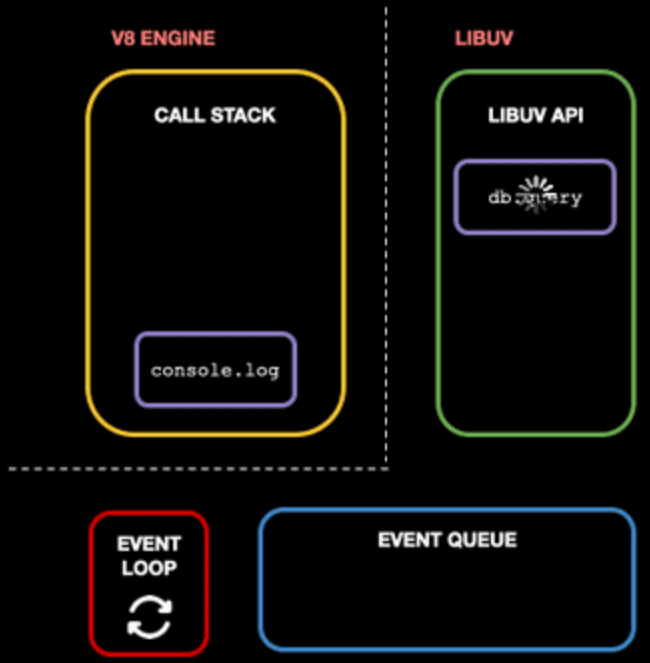
```
Starting Node.js
```

Made with ❤️ by @FabriLallo and @AndrewHu368

Quando a consulta é concluída, seu callback é enviado para a fila de eventos de E/S para ser executado em breve* . *O loop de eventos conecta a fila à pilha de chamadas. Verifica se este está vazio e move o primeiro item da fila para execução.

Callbacks of completed queries are moved to the event queue. If the call stack is empty, the event loop checks for callbacks and transfers the first.

```
console.log("Starting Node.js");

db.query("SELECT * FROM public.cars", function (err, res)  {
  console.log("Query executed");
});

console.log("Before query result");
```



OUTPUT

```
Starting Node.js

Before query result
```

Made with ❤️ by @FabriLallo and @AndrewHu368

Referencias

- Recreating Node.js from Scratch using V8, Libuv, and C++
- Difference between the Event Loop in Browser and Node Js?
- Mas que diabos é o loop de eventos? | Philip Roberts
- Node.js animated: Event Loop
- A Deep Dive Into the Node js Event Loop - Tyler Hawkins
- JavaScript Visualized: Event Loop
- Everything You Need to Know About Node.js Event Loop - Bert Belder, IBM