“”

bom dia obrigado por ter vindo, essa e uma talk sobre padrões async elegantes e programação funcional.

técnicas que você pode usar agora para escalar sua app.

Javascript e multithreading e isso possa soar um pouco louco para você…

Porque de certa forma, é verdade: o loop de eventos do JavaScript significa que o seu programa faz uma coisa de cada vez. Essa decisão de design intencional nos protege de uma classe inteira de problemas de multithreading, mas também deu origem ao equívoco de que o

oncorrência é sobre lidar com várias

sas ao mesmo tempo e paralelismo é

e fazer várias coisas ao mesmo tempo.”

Rob Pike

Mas, na verdade, o design do JavaScript é adequado para resolver uma infinidade de problemas de simultaneidade sem cair às “armadilhas” de outras linguagens multithread.

unction whatever(func, callback){}

MANDIC + RIVENDEL. ESPECIALISTAS EM CLOUDS.

ASYNC PATTERNS NO JAVASCRIPT

mandic CLOUD SOLUTIONS + rivendel

Especialistas em Clouds.

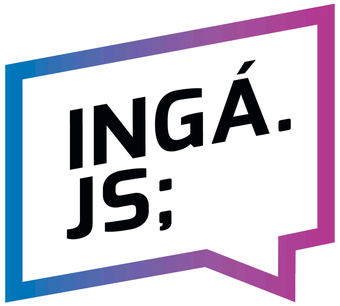
Leonardo Santos

aws vmware openstack. Azure Google Cloud Platform

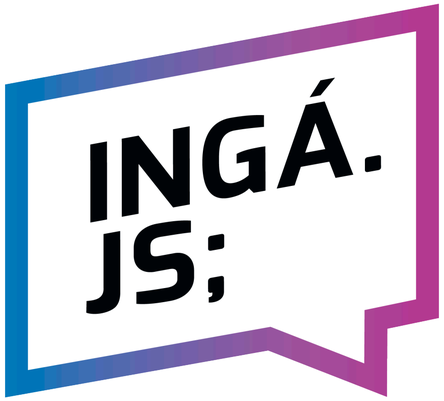
maringá


Colômbia
Bogotá
Cali
Pereira
Boa Vista
Suriname
RORAIMA
AMAPÁ
Macapá
Belém
São Luís
Quito
Equador
Guaiaquil
Manaus
AMAZONAS
PARÁ
MARANHÃO
Teresina
CEARÁ
Fortaleza
RIO GRANDE DO NORTE
PIAUÍ
PARAÍBA
João Pessoa
PERNAMBUCO
Recife
Brasil
ACRE
RONDÔNIA
TOCANTINS
ALAGOAS
Peru
Lima
MATO GROSSO
BAHIA
SERGIPE
Salvador
Cusco
GOIÁS
Brasília
Goiânia
Arequipa
La Paz
Bolívia
Cochabamba
Santa Cruz de La Sierra
MINAS GERAIS
Belo Horizonte
ESPÍRITO SANTO
Campo Grande
Ribeirão Preto
MATO GROSSO DO SUL
SÃO PAULO
RIO DE JANEIRO
São Paulo
Rio de Janeiro
Paraguai
Antofagasta
Salta
Assunção
PARANÁ
Curitiba
San Miguel de Tucumán
Chile
SANTA CATARINA
Florianópolis
RIO GRANDE DO SUL
Porto Alegre
Córdova
Santa Fé
Rosário
Uruguai
Santiago
Mendoza
Oceano Atlântico

DEVPARANÁ



MEETUP
DEVOPS
MARINGÁ



Abrir Call for paper

Verificar data do evento

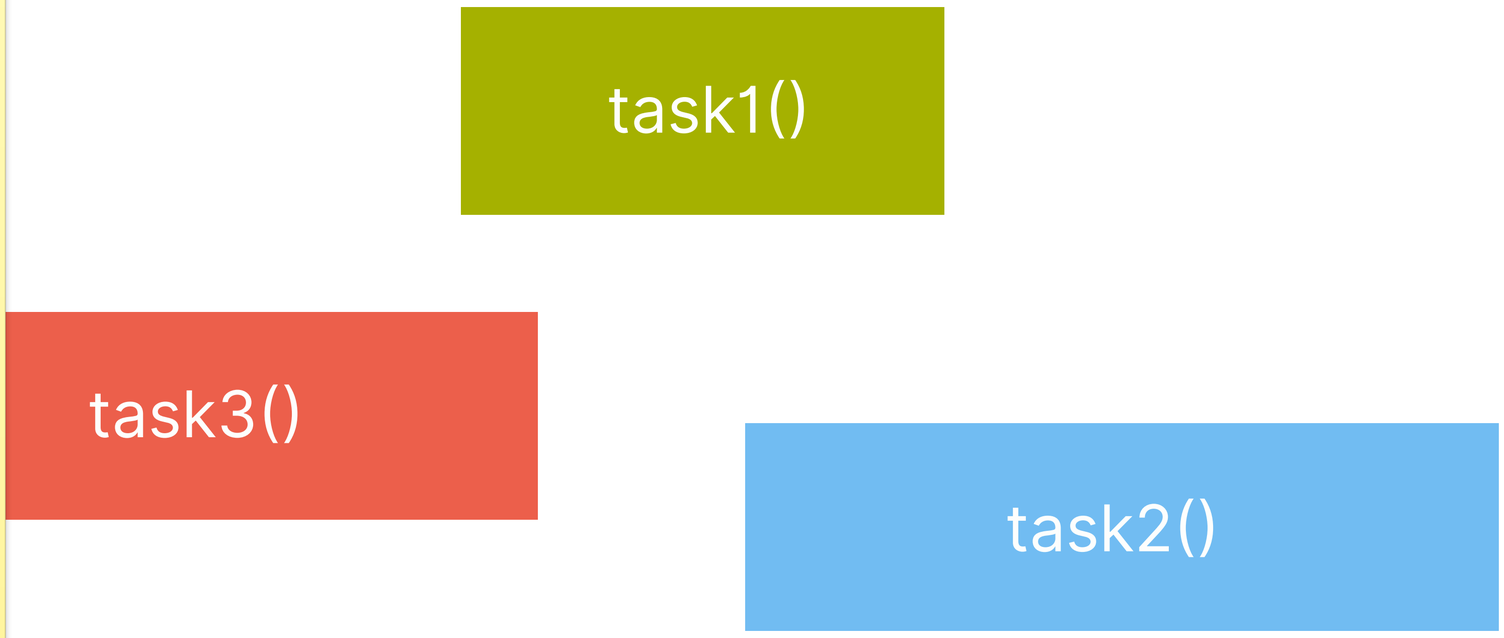
Verificar local

Cadastrar no meetup.com

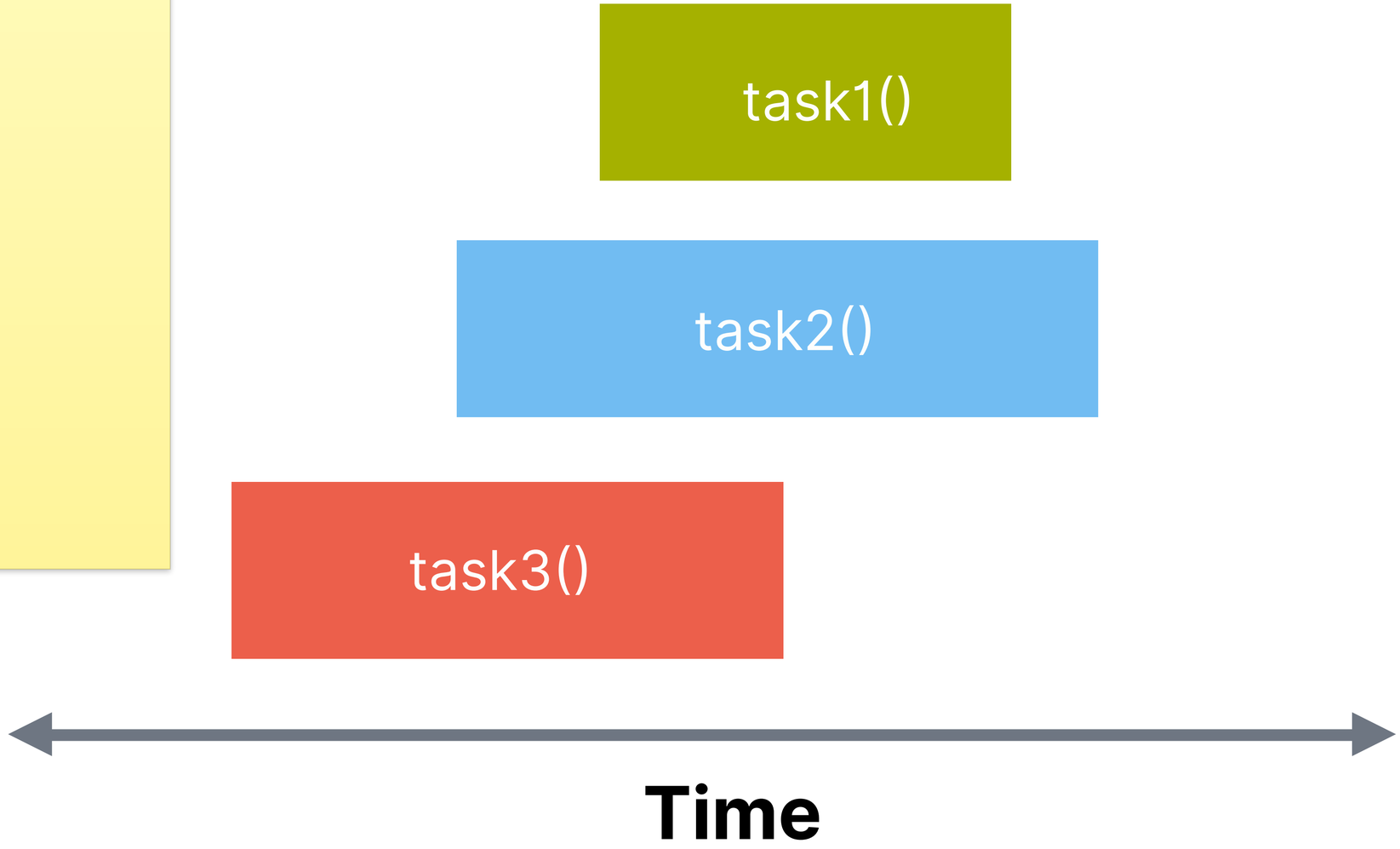
Mandar invites por e-mail

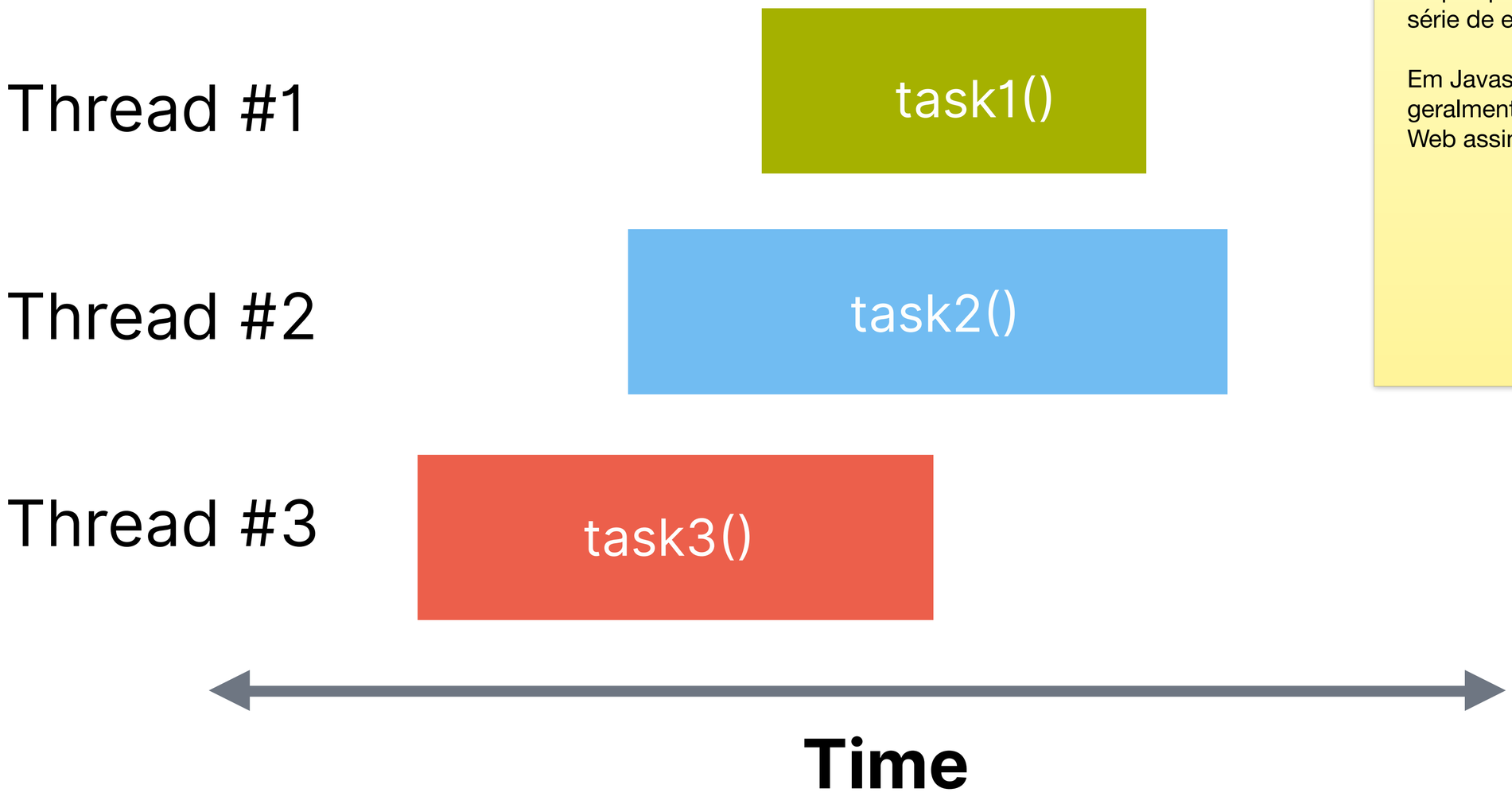
Comida

para programas simples, geralmente escrevemos código sequencial, uma etapa e executada de cada vez, e deve ser concluída antes que a próxima etapa seja iniciada.



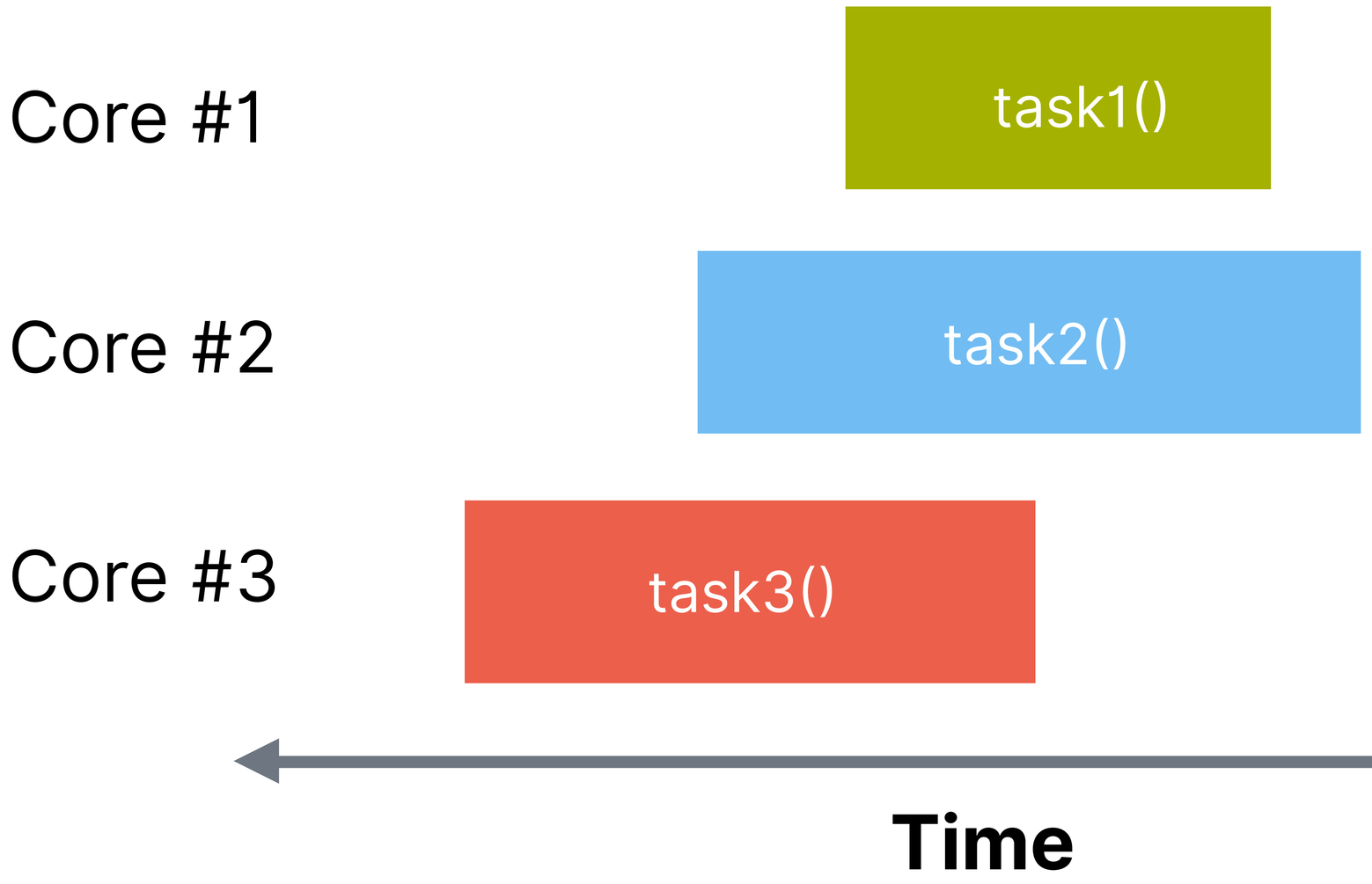
Até que uma requisição ajax, seja concluída, o javascript bloqueia qualquer linha abaixo da execução.





Ao contrario a concorrência ocorre quando a execuçào de uma série de etapas pode se sobrepor a outra série de etapas.

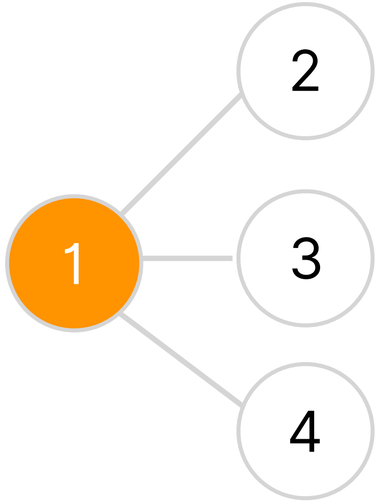
Em Javascript, a concorrência geralmente é realizada com APIs Web assincronas.



estratégias que você poderia usar um exemplo é
você pode mudar muito rapidamente entre
trabalhando nessas três tarefas assim em
particular e software que poderíamos usar
multi-threading para fazer isso uma única CPU
núcleo pode realmente fazer apenas uma coisa em um
tempo, mas mudando rapidamente poderíamos
tem essa execução sobreposta
comutação de contexto de linhas de tempo torna
Parece que estamos fazendo três coisas em
ao mesmo tempo, apesar de não sermos
há uma outra maneira que podemos fazer isso nós
poderia ter três máquinas separadas ou
Núcleos de CPU e dedicar um núcleo por tarefa
esta forma específica de concorrência é
muitas vezes chamado de

Ufa, uma desculpa a menos para rejeitar o JavaScript.

O JavaScript é ***altamente*** concorrente no Node e no navegador.




```
fetch( ‘person.json’)

setTimeout( refresh(),250)

db. transaction(['person'])
    .objectStore( ‘person')
    .get( 'lotr-1')
```

Magica ?

Event Loop & Web APIs

Source Code

```
fetch('person.json')
    .then(parse)

setTimeout( refresh(), 250)

db. transaction(['person'])
   .objectStore('person')
   .get('lotr-1')
   .then(render)
```

Call Stack

Web APIs

Fetch

IndexedDB

Timer

Callback Queue

Event Loop

```
stack.empty && !queue.empty
```

Source Code

```
setTimeout( refresh(), 250)

db. transaction(['person'])
    .objectStore('person')
    .get('lotr-1')
    .then(render)
```

Call Stack

```
fetch('person.json')
    .then(parse)
```

parse()

Web APIs

- Fetch
- IndexedDB
- Timer

Callback Queue

Event Loop

```
stack.empty && !queue.empty
```

Source Code

```
setTimeout( refresh(), 250)

db. transaction(['person'])
    .objectStore('person')
    .get('lotr-1')
    .then(render)
```

Call Stack

```
fetch('person.json')
    .then(parse)
```

Web APIs

Fetch

parse()

IndexedDB

Timer

Callback Queue

Event Loop

```
stack.empty && !queue.empty
```

Source Code

```
db. transaction(['person'])
    .objectStore(‘person')
    .get('lotr-1')
    .then(render)
```

Call Stack

```
setTimeout( refresh(), 250)
```

refresh()

Web APIs

Fetch

parse()

IndexedDB

Timer

Callback Queue

Event Loop

```
stack.empty && !queue.empty
```

Source Code

```
db. transaction(['person'])
    .objectStore(‘person')
    .get('lotr-1')
    .then(render)
```

Call Stack

```
setTimeout( refresh(), 250)
```

Web APIs

Fetch

parse()

IndexedDB

Timer

refresh()

Callback Queue

Event Loop

```
stack.empty && !queue.empty
```

Source Code

Call Stack

```
db. transaction(['person'])
    .objectStore(‘person')
    .get('lotr-1')
    .then(render)
```

render()

Web APIs

Fetch *parse()*

IndexedDB

Timer *refresh()*

Callback Queue

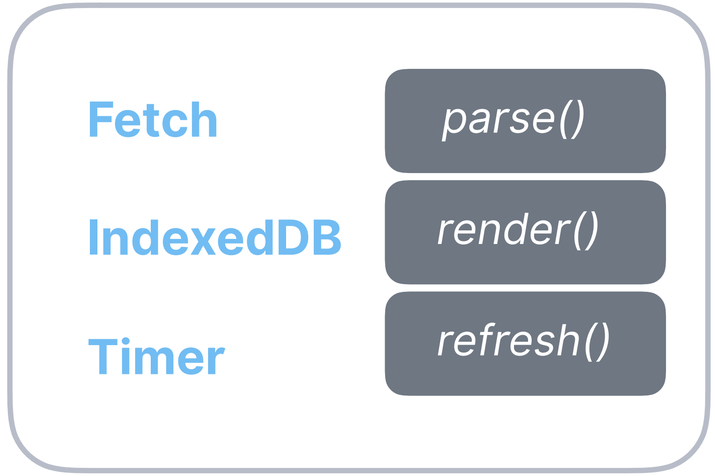
Event Loop

```
stack.empty  &&  !queue.empty
```

Source Code

Call Stack

Web APIs



Callback Queue

Event Loop

stack.empty && !queue.empty

Source Code

Call Stack

Web APIs

Fetch

IndexedDB

Timer

parse()

refresh()

Callback Queue

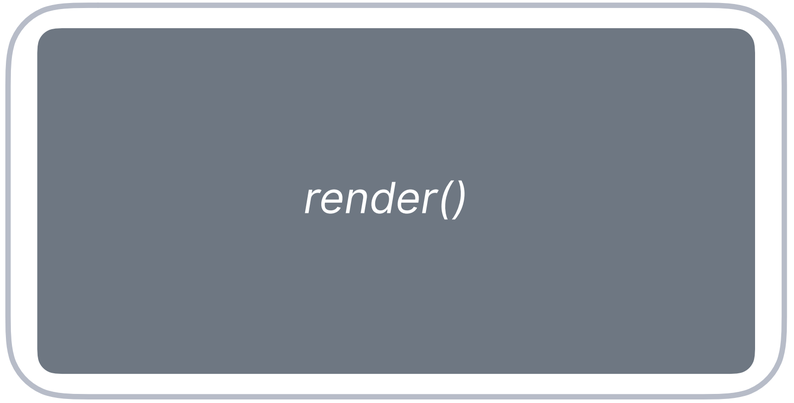
render()

Event Loop

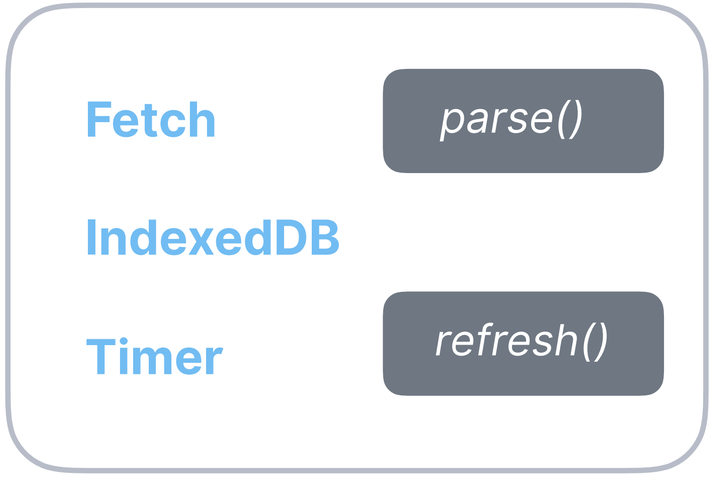
stack.empty && !queue.empty

Source Code

Call Stack



Web APIs



Callback Queue

Event Loop

stack.empty && !queue.empty

Source Code

Call Stack

render()

Web APIs

Fetch

IndexedDB

Timer

refresh()

Callback Queue

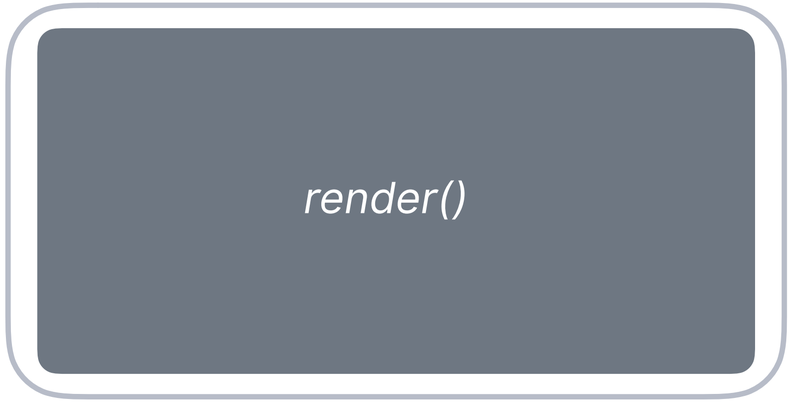
parse()

Event Loop

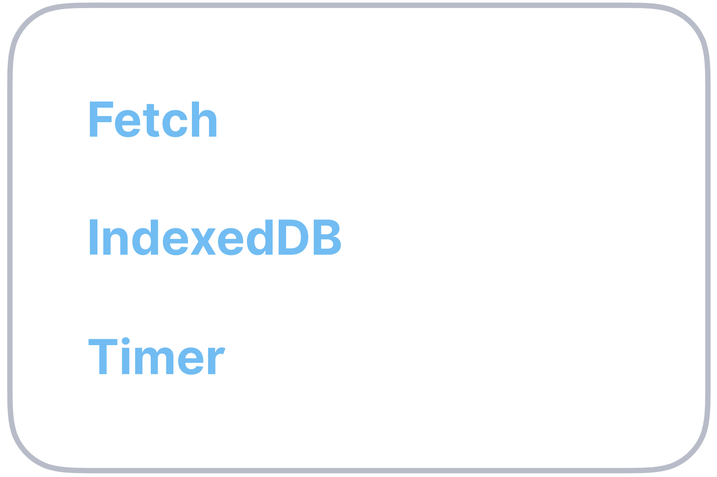
stack.empty && !queue.empty

Source Code

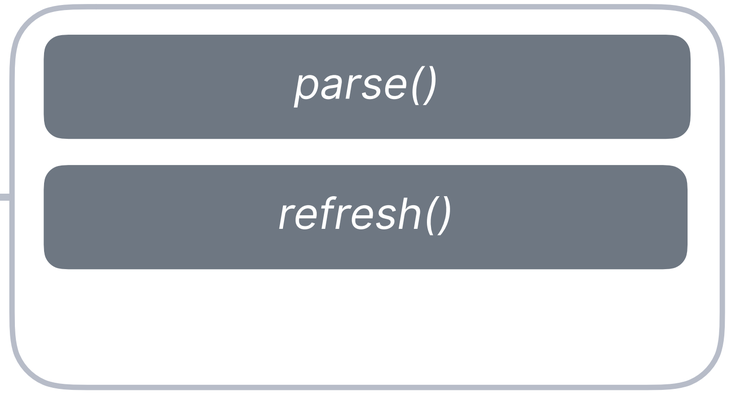
Call Stack



Web APIs



Callback Queue



Event Loop

stack.empty && !queue.empty

Source Code

Call Stack

Web APIs

Fetch

IndexedDB

Timer

Callback Queue

parse()

refresh()

Event Loop

stack.empty && !queue.empty

Source Code

Call Stack

parse()

Web APIs

Fetch

IndexedDB

Timer

Callback Queue

refresh()

Event Loop

stack.empty && !queue.empty

Source Code

Call Stack

Web APIs

Fetch

IndexedDB

Timer

Callback Queue

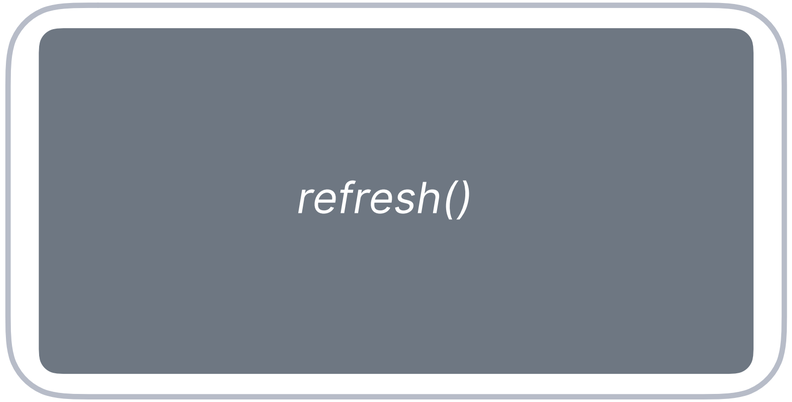
refresh()

Event Loop

stack.empty && !queue.empty

Source Code

Call Stack



Web APIs

Fetch

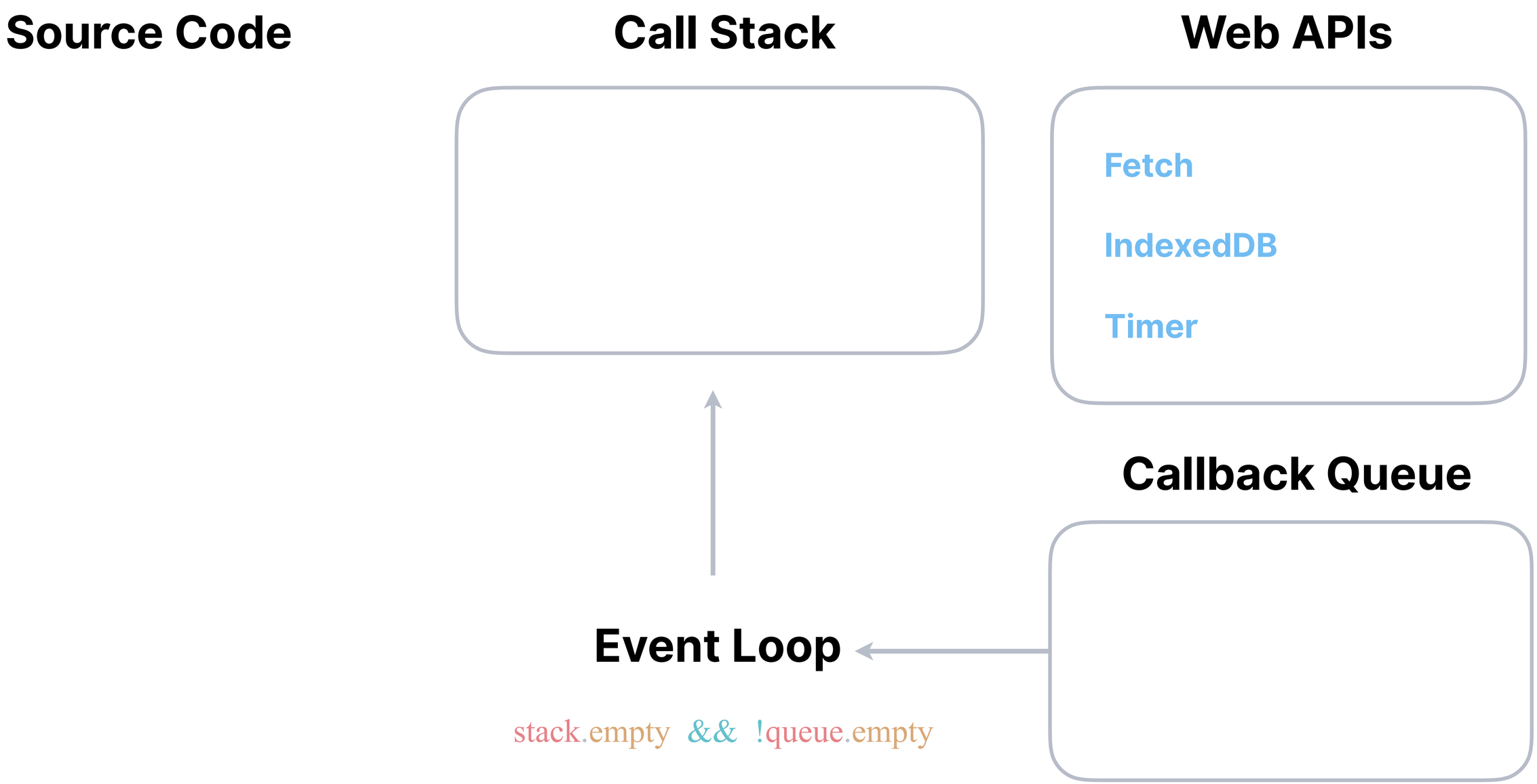
IndexedDB

Timer

Callback Queue

Event Loop

stack.empty && !queue.empty


Source Code
Call Stack
Web APIs
Fetch
IndexedDB
Timer
Callback Queue
Event Loop
stack.empty && !queue.empty

bit.ly/event-loop-help

Hoje, vamos cobrir:

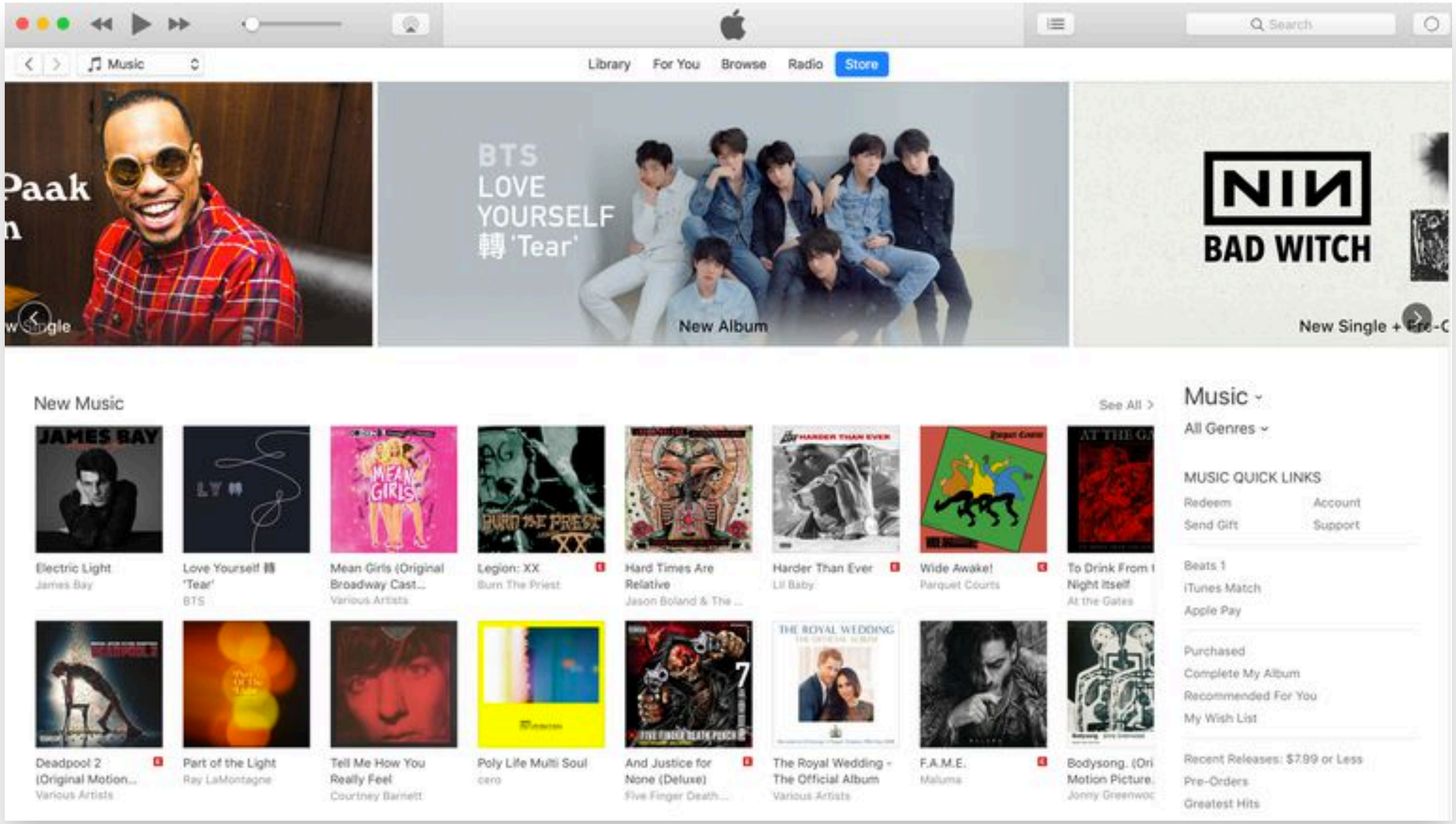
1. Declare dependências simultâneas com IIFEs assíncronos

2. Gerenciar simultaneidade com programação funcional

3. Crie seus próprios encadeamentos com os Clusters de Trabalhadores da Web

1. Declare concurrent dependencies with

Async IIFEs


Music
Library
For You
Browse
Radio
Store
Search
BTS
LOVE
YOURSELF
轉 'Tear'
New Album
NIN
BAD WITCH
New Single
New Music
See All
Electric Light
James Bay
Love Yourself 轉 'Tear'
BTS
Mean Girls (Original Broadway Cast...
Various Artists
Legion: XX
Burn The Priest
Hard Times Are Relative
Jason Boland & The ...
Harder Than Ever
Lil Baby
Wide Awake!
Parquet Courts
To Drink From t Night Itself
At the Gates
Deadpool 2 (Original Motion...
Various Artists
Part of the Light
Ray LaMontagne
Tell Me How You Really Feel
Courtney Barnett
Poly Life Multi Soul
cero
And Justice for None (Deluxe)
Five Finger Death...
The Royal Wedding - The Official Album
Various Artists
F.A.M.E.
Maluma
Bodysong. (Ori Motion Picture.
Jonny Greenwoc
Music
All Genres
MUSIC QUICK LINKS
Redeem
Account
Send Gift
Support
Beats 1
iTunes Match
Apple Pay
Purchased
Complete My Album
Recommended For You
My Wish List
Recent Releases: $7.99 or Less
Pre-Orders
Greatest Hits

1. Read File

2. Parse ID3 Metadata

3. Calculate Duration

4. Import Album

5. Import Song

1. Read File

2. Parse ID3 Metadata

3. Calculate Duration

4. Import Album

5. Import Song

```
// Read the file
let buffer = await read(file)

// Parse out metadata
let meta = await parser(file)
let songMeta = mapSongMeta(meta)
let albumMeta = mapAlbumMeta(meta)

// Compute
let duration = await getDuration(buffer)

let albumID = await importAlbum(albumMeta)

// Import the song
let songID = await importSong({
  ...songMeta,
  albumID,
  duration
})

return songID
```

```
let [ buffer, meta ] = await Promise.all([
  read(file),
  parse(file)
]);

let songMeta = mapSongMeta(meta);
let albumMeta = mapAlbumMeta(meta);

let [
  duration,
  albumID
] = await Promise.all([
  getDuration(songMeta),
  importAlbum(albumMeta)
]);
```

Async IIFEs

- *Async Immediately Invoked Function Expression (IIFE)*

```
(async() => {

  /* do things */

})();
```

Async IIFEs

- *Async Immediately Invoked Function Expression (IIFE)*

```
let task = (async () => {
  let thing = await otherTask;
  let result = await doThings(thing)
  return result;
})();
```

```
// Read the file
let readTask = read(file);

// Parse out the ID3 metadata
let metaTask = (async () => {
  let meta = await parser(file);
  let songMeta = mapSongMeta(meta);
  let albumMeta = mapAlbumMeta(meta);
  return { meta, songMeta, albumMeta };
})();

// Import the album
let albumImportTask = (async () => {
  let { albumMeta } = await metaTask;
  let albumId = await importAlbum(albumMeta);
  return albumId;
})();
```

```
})();

// Import the album
let albumImportTask = (async () => {
  let { albumMeta } = await metaTask;
  let albumId = await importAlbum(albumMeta);
  return albumId;
})();

// Compute the duration
let durationTask = (async () => {
  let buffer = await readTask;
  let duration = await getDuration(buffer);
  return duration;
})();

// Import the song
let songImportTask = (async () => {
```

```
// Import the song
let songImportTask = (async () => {
  let albumId = await albumImportTask;
  let { meta, songMeta } = await metaTask;
  let duration = await durationTask;

  let songId = await importSong({
    ...songMeta, albumId, file, duration, meta
  });

  return songId;
})();

let songId = await songImportTask;

return songId;
```

```
// Read the file
let readTask = read(file);

// Parse out the ID3 metadata
let metaTask = (async () => {
  let meta = await parser(file);
  let songMeta = mapSongMeta(meta);
  let albumMeta = mapAlbumMeta(meta);
  return { meta, songMeta, albumMeta };
})();

// Import the album
let albumImportTask = (async () => {
  let { albumMeta } = await metaTask;
  let albumId = await importAlbum(albumMeta);
  return albumId;
})();

// Compute the duration
let durationTask = (async () => {
  let buffer = await readTask;
  let duration = await getDuration(buffer);
  return duration;
})();

// Import the song
let songImportTask = (async () => {
  let albumId = await albumImportTask;
  let { meta, songMeta } = await metaTask;
  let duration = await durationTask;

  let songId = await importSong({
    ...songMeta, albumId, file, duration, meta
  });

  return songId;
})();

let songId = await songImportTask;

return songId;
```

readTask

metaTask

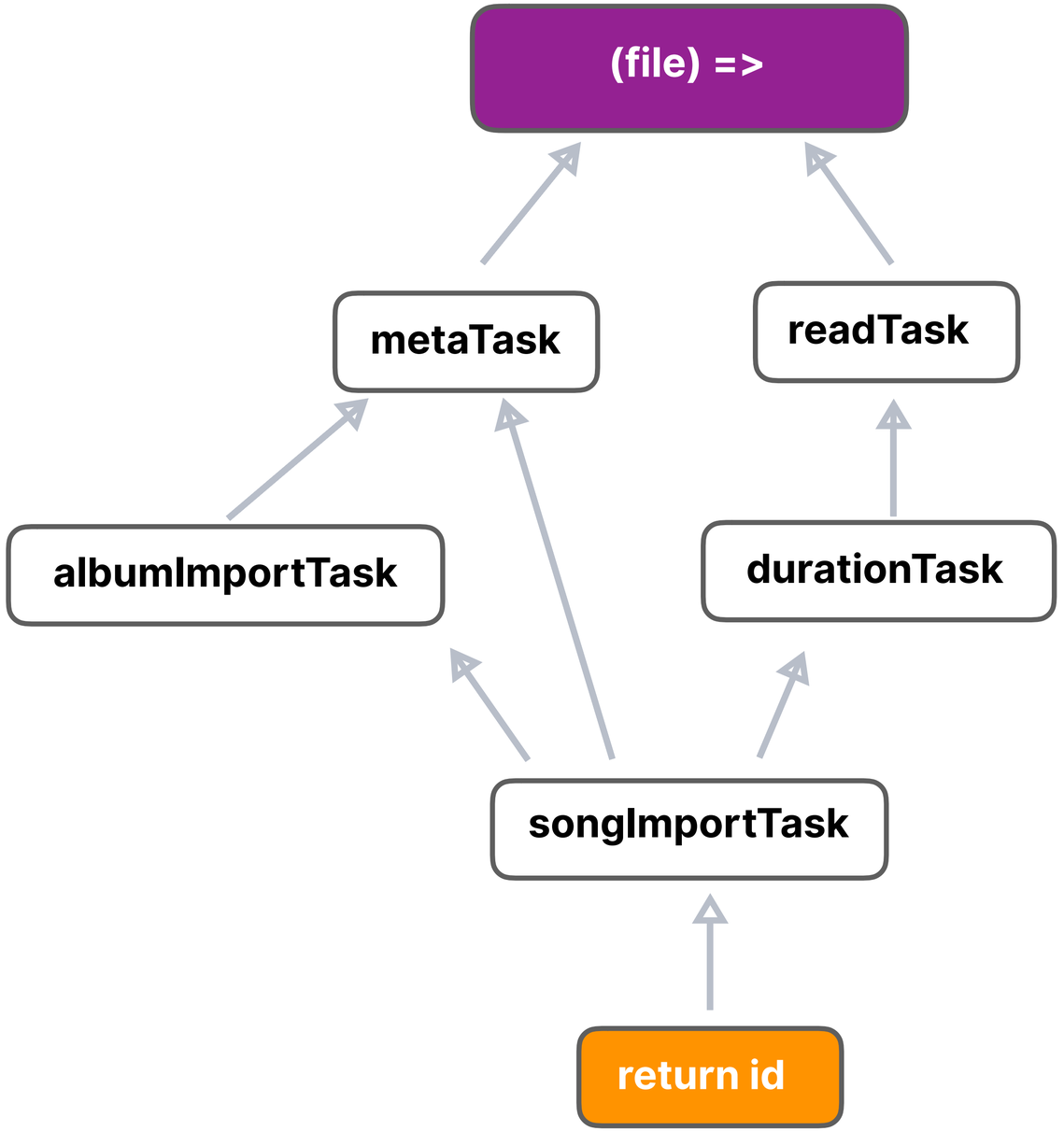
albumImportTask

durationTask

songImportTask

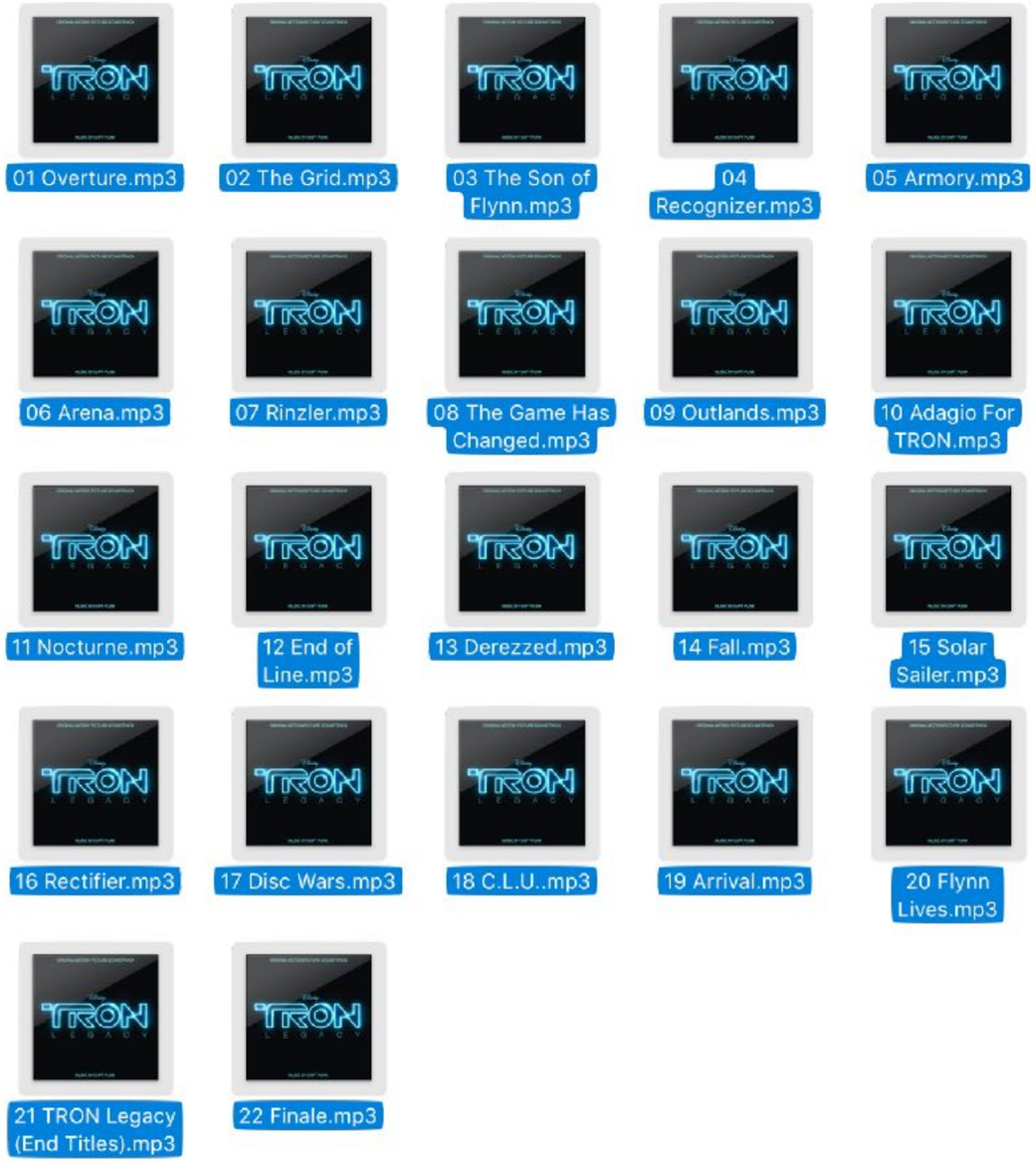
return id

```
// Read the file
let readTask = read(file);

// Parse out the ID3 metadata
let metaTask = (async () => {
  let meta = await parser(file);
  let songMeta = mapSongMeta(meta);
  let albumMeta = mapAlbumMeta(meta);
  return { meta, songMeta, albumMeta };
})();

// Import the album
let albumImportTask = (async () => {
  let { albumMeta } = await metaTask;
  let albumId = await importAlbum(albumMeta);
  return albumId;
})();

// Compute the duration
let durationTask = (async () => {
  let buffer = await readTask;
  let duration = await getDuration(buffer);
  return duration;
})();

// Import the song
let songImportTask = (async () => {
  let albumId = await albumImportTask;
  let { meta, songMeta } = await metaTask;
  let duration = await durationTask;

  let songId = await importSong({
    ...songMeta, albumId, file, duration, meta
  });

  return songId;
})();

let songId = await songImportTask;

return songId;
```



2. Como controlar a concorrência

Functional Programming


01 Overture.mp3
02 The Grid.mp3
03 The Son of Flynn.mp3
04 Recognizer.mp3
05 Armory.mp3
06 Arena.mp3
07 Rinzler.mp3
08 The Game Has Changed.mp3
09 Outlands.mp3
10 Adagio For TRON.mp3
11 Nocturne.mp3
12 End of Line.mp3
13 Derezzed.mp3
14 Fall.mp3
15 Solar Sailer.mp3
16 Rectifier.mp3
17 Disc Wars.mp3
18 C.L.U..mp3
19 Arrival.mp3
20 Flynn Lives.mp3
21 TRON Legacy (End Titles).mp3
22 Finale.mp3

Importando Livre Estou - Frozen.mp3 - 1%

Importando... Wesley Safão - 1% - 10%.mp3


Impor
98%.mp3

```
let semaphore = new Semaphore( 4);

await semaphore.acquire( );
/* do things */
semaphore. release( );
```

```
let Semaphore = max => {
  let tasks = []
  let counter = max

  let dispatch = () => {
    if (counter > 0 && tasks.length > 0) {
      counter--
      tasks.shift()()
    }
  }

  let release = () => {
    counter++
    dispatch()
  }

  let acquire = () =>
    new Promise(resolve => {
      tasks.push(resolve)
      setImmediate(dispatch)
    })

  return async fn => {
    await acquire()
    let result
    try {
      result = await fn()
    } catch (e) {
      throw e
    } finally {
      release()
    }
    return result
  }
}

export let limit = (max, fn) => {
  let semaphore = Semaphore(max)
  return (...args) => semaphore(() => fn(...args))
}

export default Semaphore
```

```
let Semaphore = max => {
  let tasks = []
  let counter = max

  let dispatch = () => {
    if (counter > 0 && tasks.length > 0) {
      counter--
      tasks.shift()()
    }
  }

  let release = () => {
    counter++
    dispatch()
  }

  let acquire = () =>
    new Promise(resolve => {
```

```
let release = () => {
  counter++
  dispatch()
}

let acquire = () =>
  new Promise(resolve => {
    tasks.push(resolve)
    setImmediate(dispatch)
  })

return async fn => {
  await acquire()
  let result
  try {
    result = await fn()
  } catch (e) {
    throw e
```

```
      setImmediate(dispatch)
    })

  return async fn => {
    await acquire()
    let result
    try {
      result = await fn()
    } catch (e) {
      throw e
    } finally {
      release()
    }
    return result
  }
}

export let limit = (max, fn) => {
  let semaphore = Semaphore(max)
```

```
      release()
    }
    return result
  }
}

export let limit = (max, fn) => {
  let semaphore = Semaphore(max)
  return (...args) => semaphore(() => fn(...args))
}

export default Semaphore
```

```
let semaphore = Semaphore(4)

let result = await semaphore(async () => {
  console.log('Acquired!')
  return await importMP3(file)
})
```

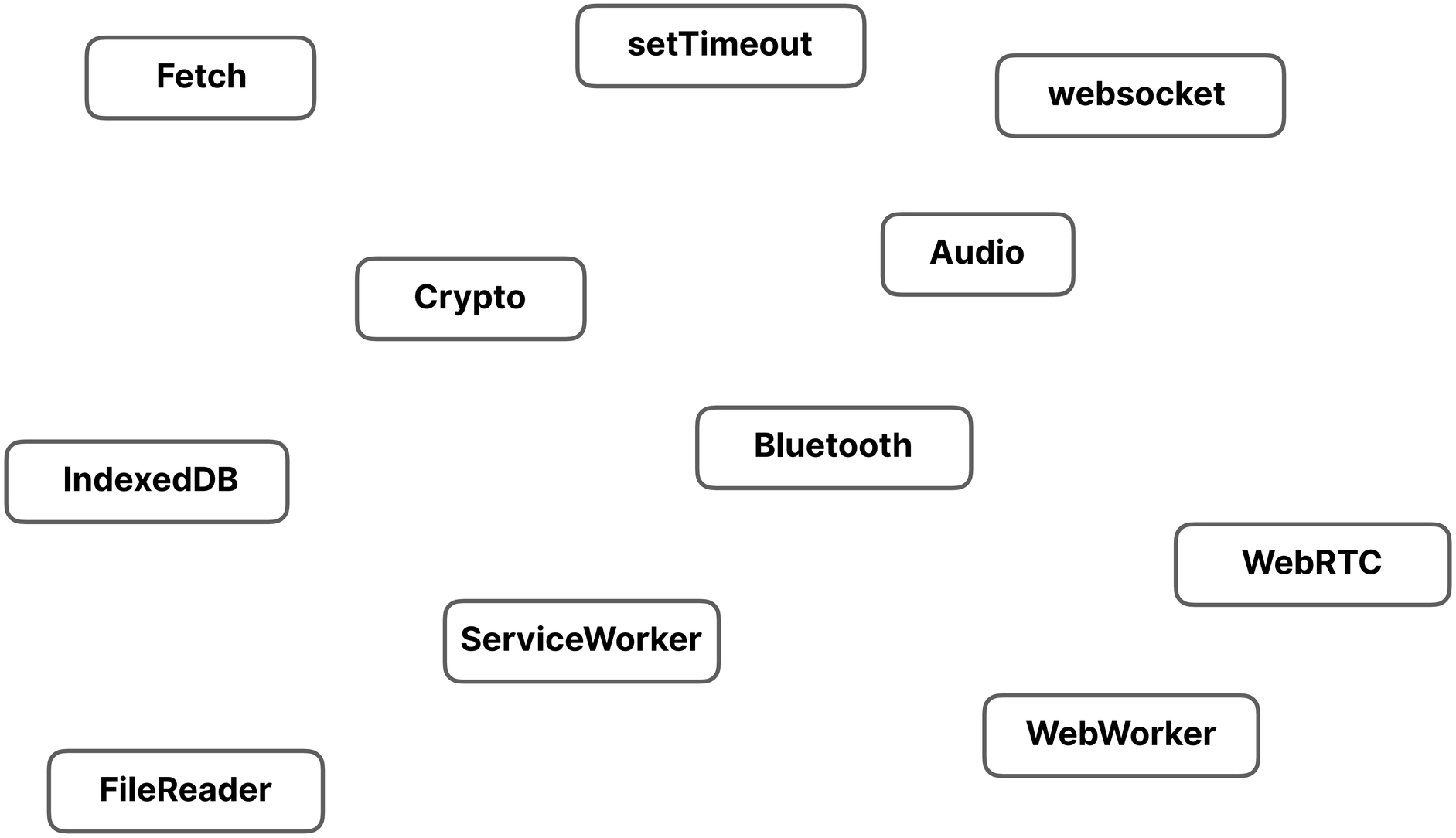
```
let importMP3 = async (data)  =>  /*  …  */

let limitedImportMP3 = limit(2, importMP3);
```

```
let importMP3 = async (data) => /* … */

 let limitedImportMP3 = limit(2 , importMP3);

limitedImportMP3(song1);
// starts immediately
limitedImportMP3(song2);
// starts immediately
limitedImportMP3(song3);
// waits for song1 or song2 to finish
```

```
let importMP3 = async (data) => /*  …  */
let limitedImportMP3 = limit(2,importMP3);

let limit  =  (max,  fn)  =>  {
    let semaphore  =  Semaphore(max);
    return (...args) =>
        semaphore(() => fn(...args));
 };
```

3. Create your own threads with

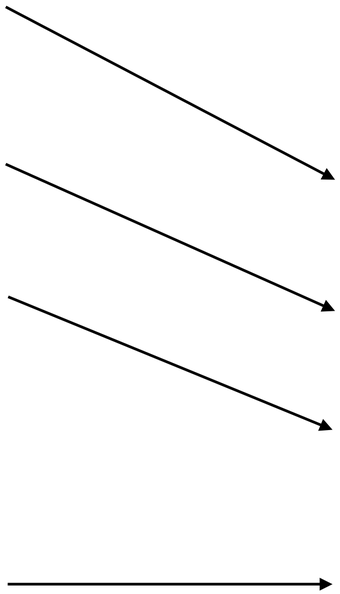
Web Worker Clusters

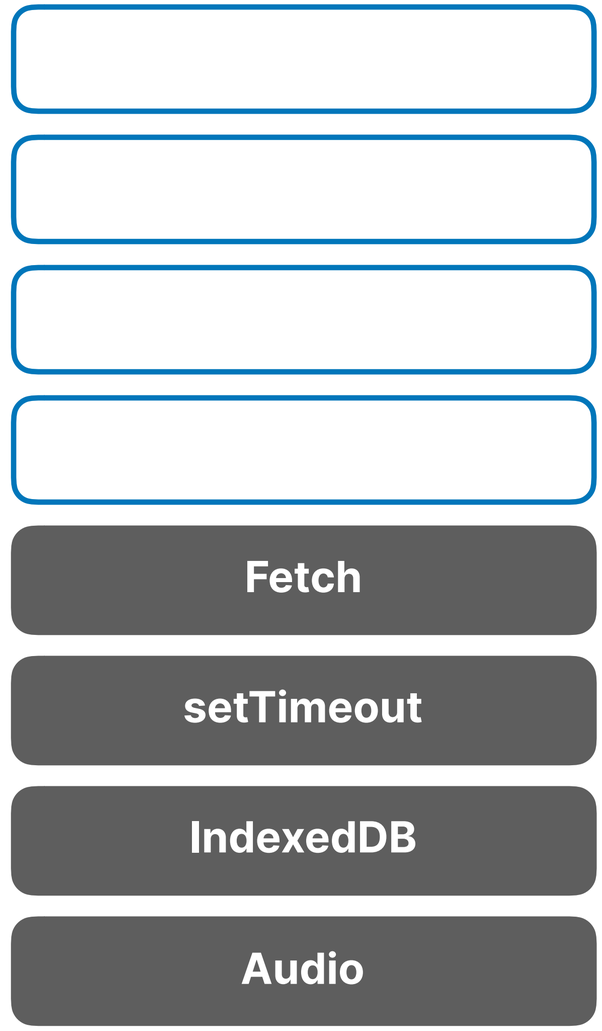

Fetch
setTimeout
websocket
Audio
Crypto
Bluetooth
IndexedDB
WebRTC
ServiceWorker
WebWorker
FileReader

Main Thread

Thread Pool

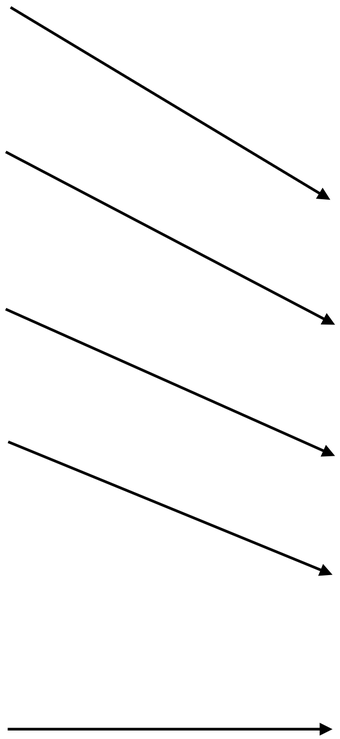
```
setTimeout( refresh(),  250)

db. transaction(['person'])
  .objectStore(‘person')
  .get('lotr-1')
  .then(render)

ctx.decodeAudioData(buffer)
```



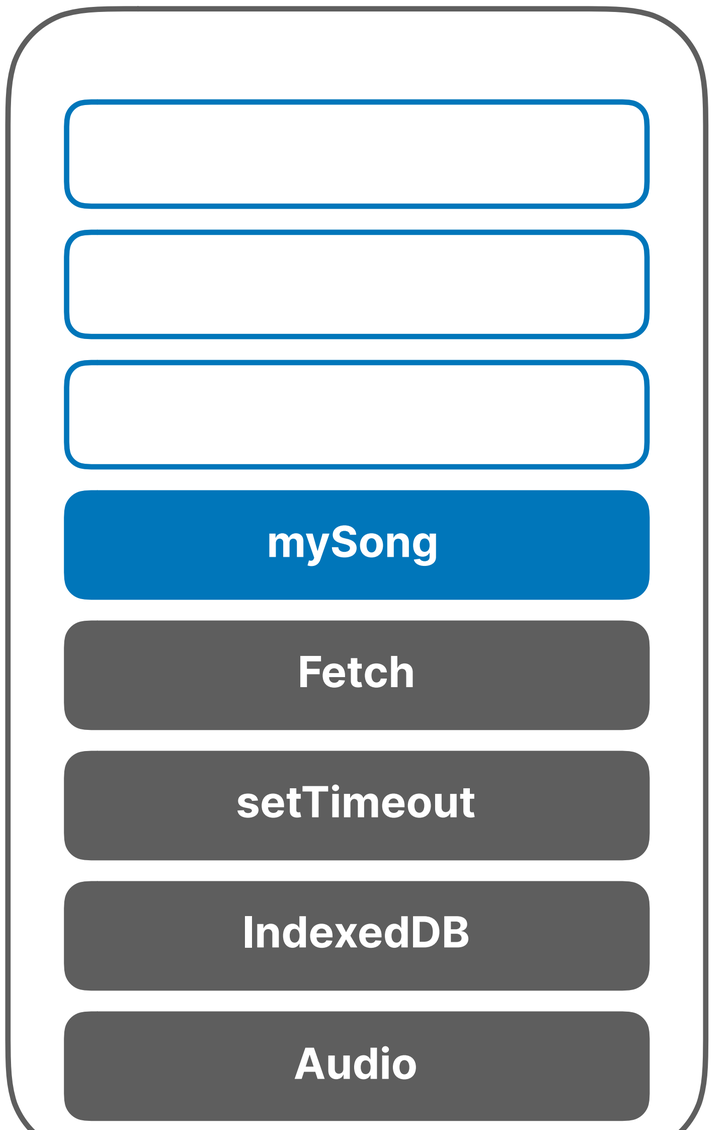


Main Thread

```
importMP3(file)

fetch('person.json')
        .then(parse)

setTimeout( refresh(),  250)

db. transaction(['person'])
  .objectStore('person')
  .get('lotr-1')
  .then(render)

ctx.decodeAudioData(buffer)
```



Thread Pool



```
let worker = new Worker('worker.js')

worker.postMessage({ all: ['the', 'data'] })

worker.onmessage(({ data }) => {
  console.log(data)
})
```

```
let cluster = Cluster('worker.js')

let result = await cluster({ all: ['the', 'data'] })
```

```
let importMP3 = Cluster('worker.js')

let song = await importMP3(songFile)
```

```
let maxWorkers = navigator.hardwareConcurrency || 4

let defaultHandler = async (worker, data) => {
  worker.postMessage(data)

  return await once('message')
}

let Cluster = (path, handler = defaultHeader, max = maxWorkers) => {
  let pool = []
  let semaphore = Semaphore(max)

  let useWorker = async fn => {
    let worker = pool.pop() || new Worker(path)
    let result

    try {
      result = await fn(worker)
    } catch (e) {
      throw e
```

```
let Cluster = (path, handler = defaultHeader, max = maxWorkers) => {
  let pool = []
  let semaphore = Semaphore(max)

  let useWorker = async fn => {
    let worker = pool.pop() || new Worker(path)
    let result

    try {
      result = await fn(worker)
    } catch (e) {
      throw e
    } finally {
      pool.push(worker)
    }

    return result
  }

  return async data =>
```

```
    } finally {
      pool.push(worker)
    }

    return result
  }

  return async data =>
    await semaphore(
        () => useWorker(
          worker => handler(worker, data)
        )
    )
}
```

```
import { done } from './rpc';

(async () => {
  while (true) {
    let { data } = await once(self, 'message')
    let song = await importMP3(data)
    done(song)
  }
})()
```

TL;DR:

1. JavaScript é altamente concorrente
2. Thread - não e só uma
3. Programação funcional
4. Crie suas próprias APIs assíncronas com
    1. Web Workers

Always bet on JavaScript

👉 **github.com/leonardoelias/awesome-meetup**

Leonardo Elias

leonardo.elias4@gmail.com
@leonardoelias_

Muito Obrigado