From Node.js to Design Patterns

Luciano Mammino

@loige

👋 Hello, I am Luciano

Cloud & Full stack software engineer

Let's connect:

🌎 loige.co

🐦 @loige

🧳 lucianomammino

nodejsdesignpatterns.com

👇 Get the slides (and click around...)

loige.link/devcast

What is Node.js

Node.js is an **open-source**, **cross-platform**, JavaScript runtime environment that executes JavaScript code **outside a web browser**.

Node.js + JavaScript: when?

- Building for the web
  - Websites, APIs, Servers, Single Page Applications, Bots, etc…
- Command-line applications and tools
- Mobile applications (React Native, Ionic, NativeScript, etc.)
- Desktop apps (Electron)
- Embedded & IoT (Espruino, Johnny-Five)

Async I/O

In JavaScript and in Node.js, input/output operations (e.g. making an HTTP request) are **non-blocking**.

A blocking HTTP request (Java)

```java
OkHttpClient client = new OkHttpClient();
Request request = new Request.Builder()
  .url("https://google.com")
  .build();
Response response = client.newCall(request).execute();
System.out.println(response.body().string());
System.out.println("Request completed");
```

Code executed
“In order”

blocking...

blocking...

Output

```
<google home page HTML code>
Request completed
```

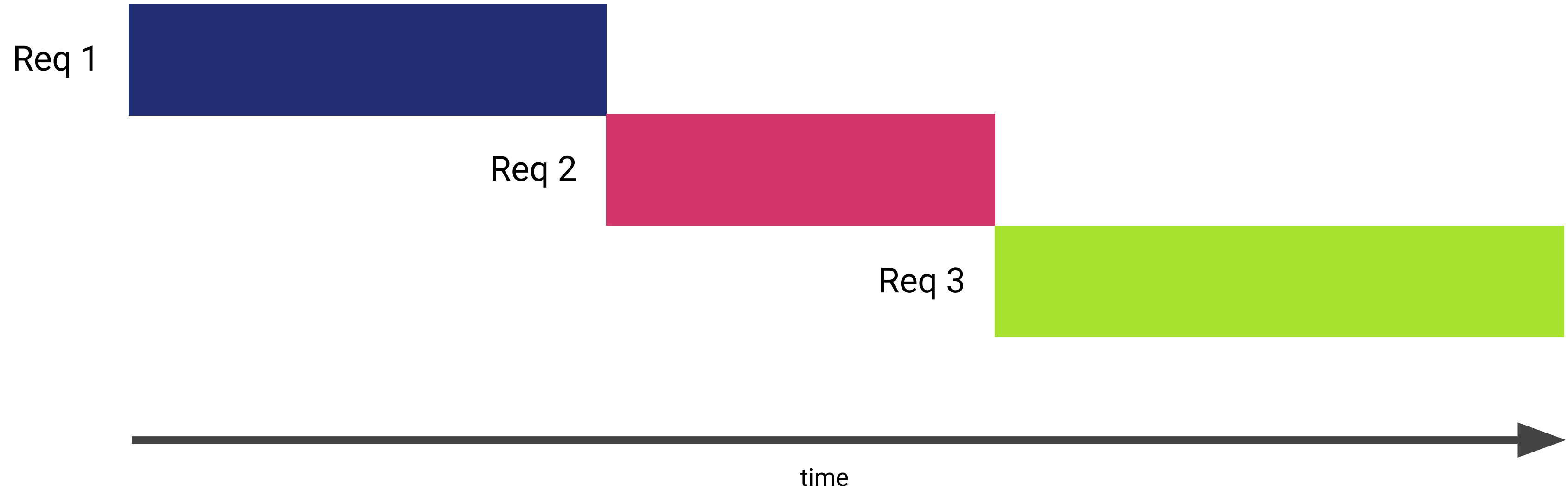
@loige

Is blocking I/O bad?

It depends…

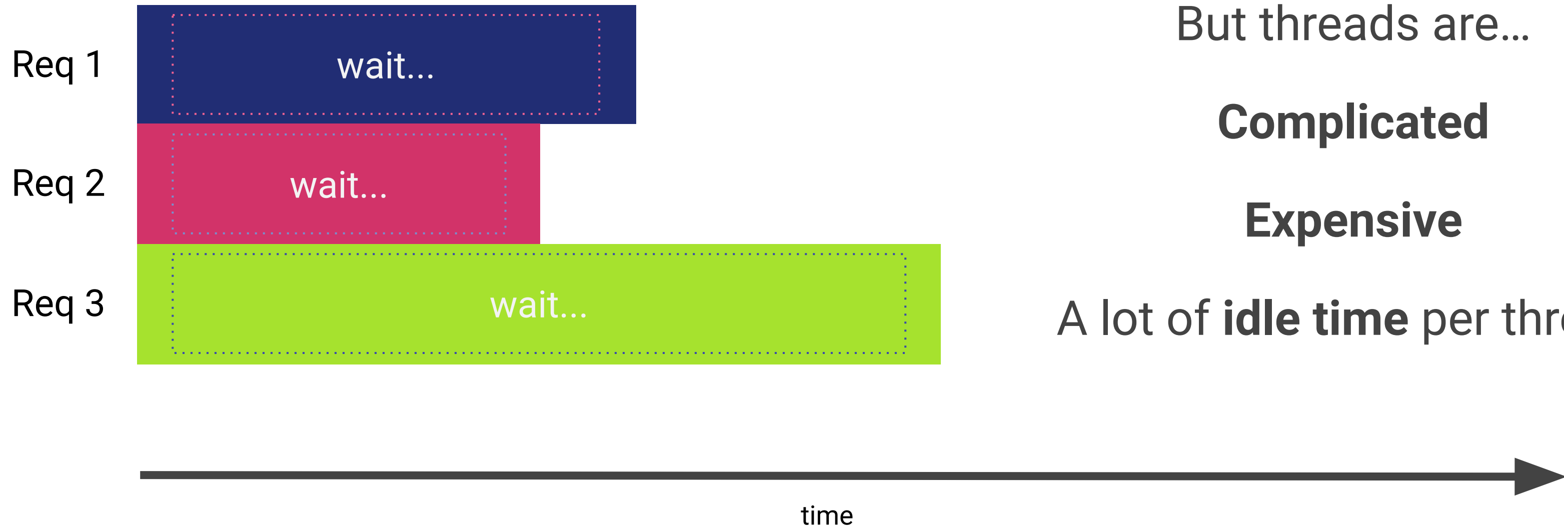
If your application is I/O heavy, then you might have a problem...

Let’s make 3 requests...



@loige

You can always use threads...



But threads are...

Complicated

Expensive

A lot of **idle time** per thread!

With Node.js async I/O

```
client.get('https://google.com',
  (err, resp) => {
    console.log(resp.body)
  }
)

console.log('Request completed (?)')
```

1, 3, 2

Request “in the background”

Callback function

Non-blocking: execution continues

⚠️ Code executed “**OUT** of order”

Output

```
Request completed (?)
<google home page HTML code>
```

Not really completed!

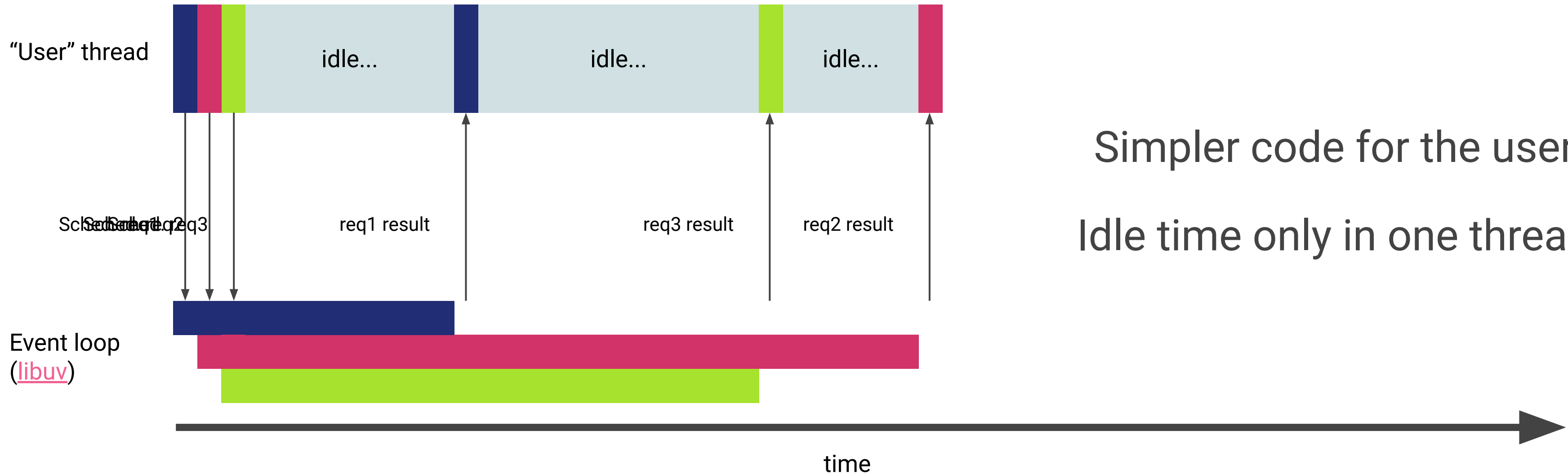
@loige

Mental model

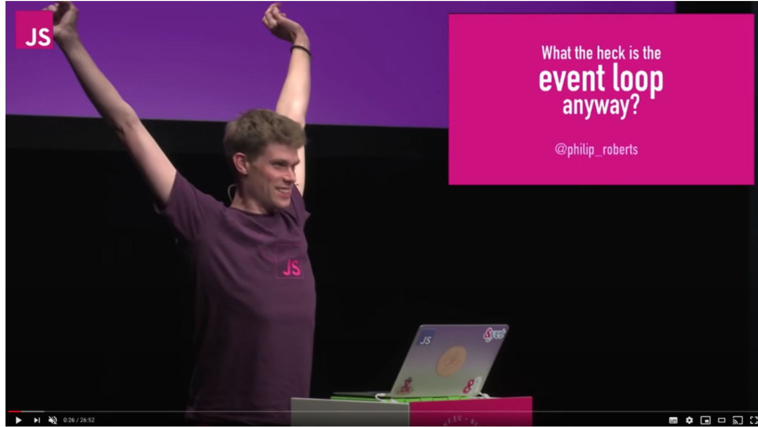
You “just” **schedule** async I/O and you will get **notified** when the operation is completed!

- Async I/O happens in the background asynchronously
- You don’t have to manage threads to get concurrency!

Let’s do 3 requests with Async I/O



I am oversimplifying a bit... 😛



Watch loige.link/event-loop-what-the-heck

if you want to go more in depth!

Many ways to handle async flows

Delete last reservation if confirmed

- Get a guest object from a guest id (async)
- Get the last reservation from the guest object
- Get the details of that reservation (async)
- Delete that reservation if “confirmed” (async)

Callbacks

```
function deleteLastReservationIfConfirmed (client, guestId, cb) {
  client.getGuest(guestId, (err, guest) => {
    if (err) { return cb(err) }
    const lastReservation = guest.reservations.pop()
    if (typeof lastReservation === 'undefined') {
      return cb(null, false)
    }
    client.getReservation(lastReservation, (err, reservation) => {
      if (err) { return cb(err) }
      if (reservation.status === 'confirmed') {
        client.deleteReservation(reservation.id, (err) => {
          if (err) { return cb(err) }
          return cb(null, true)
        })
      }
    })
  })
}
```

Promises

```
function deleteLastReservationIfConfirmed (client, guestId) {
  return client.getGuest(guestId)
    .then((guest) => {
      const lastReservation = guest.reservations.pop()
      if (typeof lastReservation !== 'undefined') {
        return client.getReservation(lastReservation)
      }
    })
    .then((reservation) => {
      if (!reservation || reservation.status !== 'confirmed') {
        return false
      }
      return client.deleteReservation(reservation.id)
    })
}
```

Async/Await

```
async function deleteLastReservationIfConfirmed (client, guestId) {
  const guest = await client.getGuest(guestId)
  const lastReservation = guest.reservations.pop()
  if (typeof lastReservation === 'undefined') {
    return false
  }
  const reservation = await client.getReservation(lastReservation)
  if (!reservation || reservation.status !== 'confirmed') {
    return false
  }
  return client.deleteReservation(reservation.id)
}
```

Advantages of Async/Await

- Easier to read and reason about (“sequential flow”)
- Easier to deal with conditional async operations
- Unified error handling

  (you can catch both synchronous and asynchronous errors)

⚠️ To fully understand async/await, you still need to understand callbacks and promises, don’t ignore them!

Other ways to handle async

- Events
- Streams
- Async iterators/generators

Some interesting async patterns

Sequential execution

```
const guestIds = ['Peach', 'Toad', 'Mario', 'Luigi']

for (const guestId of guestIds) {
  await deleteLastReservationIfConfirmed(client, guestId)
}
```

Sequential execution ⚠️ Common pitfall

Don’t use **Array.map** or **Array.forEach**!

```
const guestIds = ['Peach', 'Toad', 'Mario', 'Luigi']

guestIds.forEach(async (guestId) => {
  await deleteLastReservationIfConfirmed(client, guestId)
})
```

forEach will run all the functions without awaiting them, so all the delete invocations will happen concurrently!

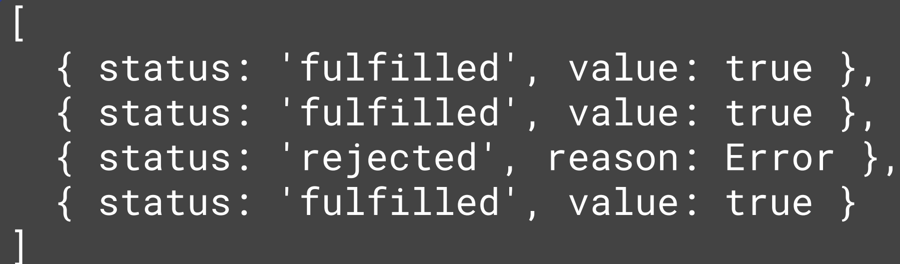
Concurrent execution

```
const guestIds = ['Peach', 'Toad', 'Mario', 'Luigi']

await Promise.all(
  guestIds.map(
    guestId => deleteLastReservationIfConfirmed(client, guestId)
  )
)
```

⚠️ **Promise.all** rejects as soon as one promise rejects. A failure will result in the failure of the entire operation.

Concurrent execution - Alternative

```
const guestIds = ['Peach', 'Toad', 'Mario', 'Luigi']

const results = await Promise.allSettled(
  guestIds.map(
    guestId => deleteLastReservationIfConfirmed(client, guestId)
  )
)
```



```
[
  { status: 'fulfilled', value: true },
  { status: 'fulfilled', value: true },
  { status: 'rejected', reason: Error },
  { status: 'fulfilled', value: true }
]
```

Concurrent execution - limited

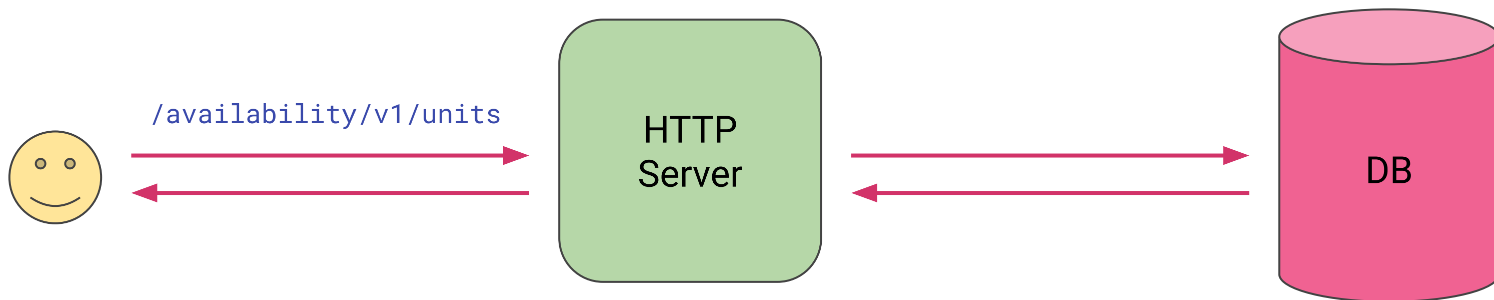
When you have a lot of tasks to run and what to keep limited concurrency

Uses the third-party async module (npm.im/async)

```
const mapLimit = require('async/mapLimit')

const guestIds = ['Peach', 'Toad', 'Mario', 'Luigi', '...']

const results = await mapLimit(
  guestIds,
  2, // max concurrency
  async (guestId) => deleteLastReservationIfConfirmed(client, guestId)
)
```

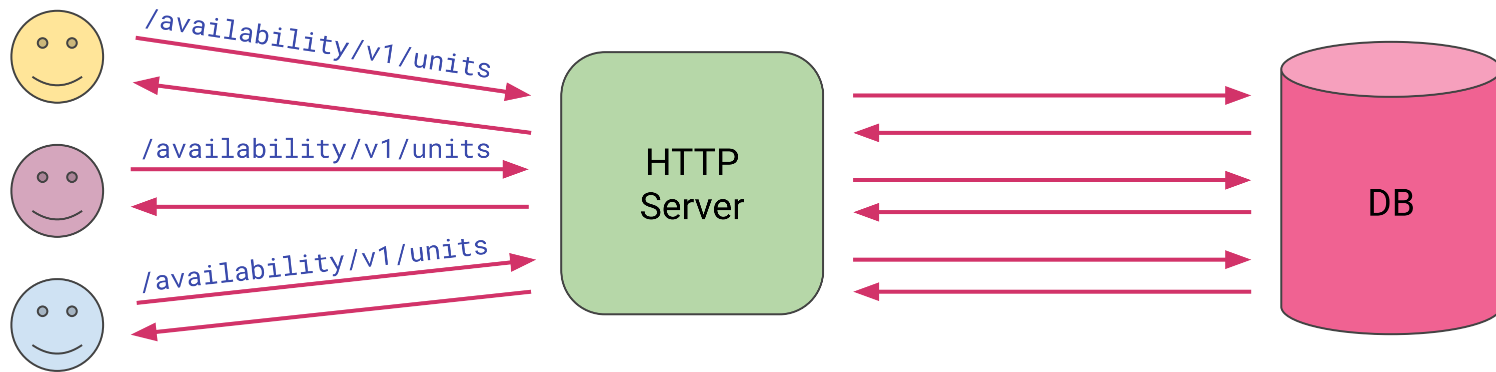
Request batching

Classic flow - one user



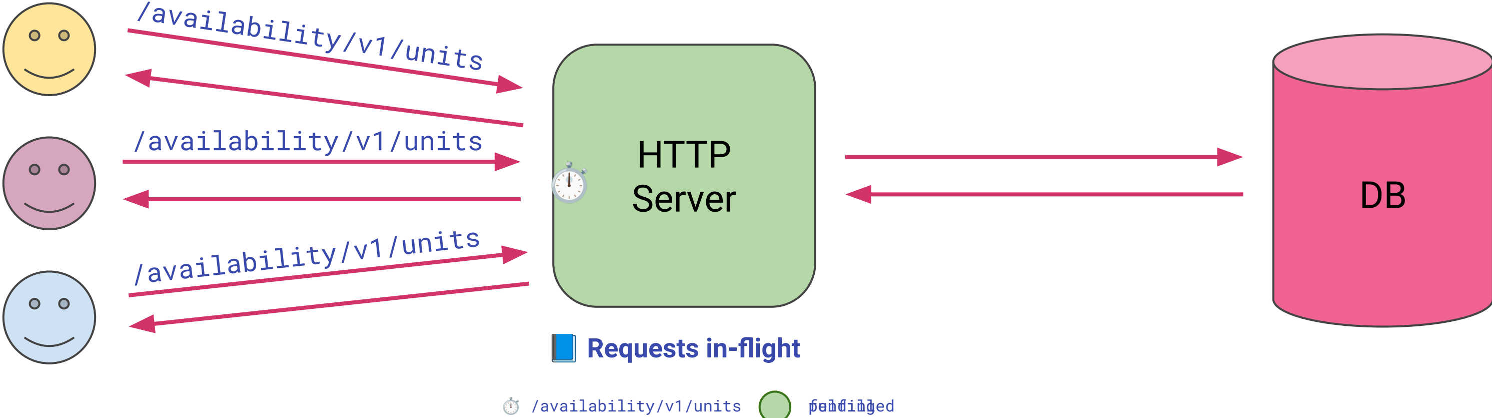
Request batching

Classic flow - multiple users (no batching)



Request batching

Classic flow - multiple users (with batching!)



@loige

The web server

```
const { createServer } = require('http')

const server = createServer(async (req, res) => {
  const url = new URL(req.url, 'http://localhost')
  if (url.pathname !== '/availability/v1/units') {
    res.writeHead(404, 'Not found')
    return res.end()
  }

  const units = await getAvailableUnitsFromDb()

  res.writeHead(200)
  res.end(JSON.stringify(units))
})

server.listen(8000)
```

@loige

Get units from DB

```
let pendingRequest = null

async function getAvailableUnitsFromDb () {
  if (pendingRequest) {
    console.log('batching')
    return pendingRequest
  }

  console.log('Getting data from db')
  pendingRequest = db.query('SELECT * FROM units WHERE "availability" > 0')
  pendingRequest.finally(() => {
    pendingRequest = null
  })

  return pendingRequest
}
```

@loige

Performance comparison

Without batching

```
❯ autocannon -c 20 localhost:8000/availability/v1/units
Running 10s test @ http://localhost:8000/availability/v1/units
20 connections
```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	1 ms	6 ms	12 ms	12 ms	6.01 ms	3.44 ms	70.16 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	2743	2743	3049	3223	3073.6	141.02	2743
Bytes/Sec	573 kB	573 kB	637 kB	674 kB	642 kB	29.4 kB	573 kB

With batching

+15% throughput

```
❯ autocannon -c 20 localhost:8000/availability/v1/units
Running 10s test @ http://localhost:8000/availability/v1/units
20 connections
```

Stat	2.5%	50%	97.5%	99%	Avg	Stdev	Max
Latency	1 ms	5 ms	10 ms	10 ms	4.98 ms	2.93 ms	65.52 ms

Stat	1%	2.5%	50%	97.5%	Avg	Stdev	Min
Req/Sec	3417	3417	3681	3731	3656.4	83.48	3417
Bytes/Sec	714 kB	714 kB	770 kB	780 kB	764 kB	17.5 kB	714 kB

In conclusion

- Node.js is a great runtime for I/O bound applications
- It's also great for full stack web development
- The async model allows you to express concurrent computation effectively
- You still need to master callbacks, promises and async / await!
- There are many patterns that can help you out to keep your code organised and more performant.

Want to learn more?

- **nodejsdesignpatterns.com** - possibly a great book :)
- **loige.link/javascript-mdn** - mozilla guides
- **eloquentjavascript.net** - free e-book
- **freecodecamp.org** - interactive code training
- **nodejs.org/api** - Node.js official docs

Thank you!

loige.link/devcast

@loige