async programming in JS

Part1

your desire to avoid the mess of callbacks **is not justification for blocking, synchronous code**

(@getify, Kyle Simpson)

Why we need this lesson ?

So why isn't everyone using event loops, callbacks, and non-blocking I/O?

For reasons both **cultural** and **infrastructural**.

your **desire to avoid the mess of callbacks** is not justification for blocking, synchronous code (@getify, Kyle Simpson)

That's just not really how our brains appear to be set up.

Developeri sú leniví naučiť sa a používať async patterny a radi skĺznu ku komfortnému a prirodzenejšiemu

- synchrónnemu a
- buffrovanému kódu

Node.js a JavaScript je o

- **asynch** a
- **chunk** processingu

Ak to nechcete, programujte v inom jazyku....
Už teraz je dosť zlých JS knižníc a kódov na svete....

Recap, node.js design

- To provide a purely **evented**, **non-blocking** infrastructure to script highly concurrent programs (web servers,...)
- The API should should be **fammiliar to client side JS** programmers and **old school UNIX hackers**
- No function should directly perform IO
- To **receive info** from disk, net or another process, **there must be callback**
- **Stream everything**, never force buffering of data
- Have build-in support for most important protocols TCP, DNS, HTTP

Recap - callbacks

- When your JS program makes **async operation**, you set up the "**response" code** in a function (commonly called a "**callback**"), and the JS engine **tells the hosting environment**, "Hey, I'm going to suspend execution for now, but whenever you finish with that request, and you have some data, please ***call* this function *back***.“ (@getify)
  - Cooeration of JS engine and hosting environment
  - Chunks of program – main and functions
- Event loop

Recap – “všetok“ váš kód bude vyzerať nejako takto

```
const fs = require("fs");

fs.readFile(from, (err, data) => {
    if (err) { /* */ ; return; }
    fs.writeFile(to, data, (err) => {
        if (err) { /* */ return; }


        console.log("done");

    });
});
```

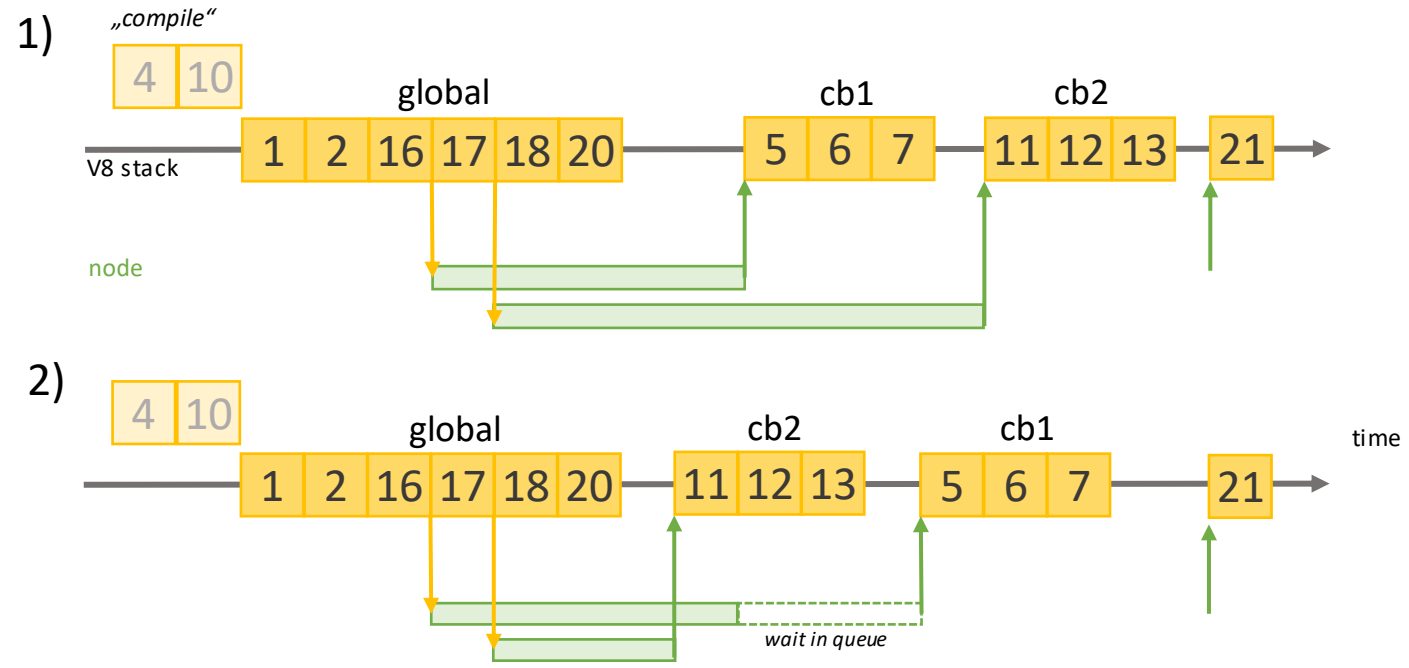
Async Callbacks

- **most typical API in node.js** core and userland modules
- function call **does not return data**
- instead function call accepts **another function as parameter (callback)**
- expected function **signature is commonly (err,data) or (err)**
- nesting of callbacks to achieve more steps in algorithm

Recap … a bežať nejako takto….

```
1  let a = 1;
2  let b = 2;
3
4  function cb1() {
5    a++;
6    b = b * a;
7    a = b + 3;
8  }
9
10 function cb2() {
11   b--;
12   a = 8 + b;
13   b = a * 2;
14 }
15
16 const https = require("https");
17 https.get("https://nodejs.org/en/", cb1);
18 https.get("https://nodejs.org/en/", cb2);
19
20 process.on("beforeExit", () => {
21   console.log(a, b);
22 });
```

/samples/concurrency.js



- **async**: some code happends ***now*** [1,2,...] and some ***later*** [5,5,6], [11,12,13],[21]
- **concurrent**: callbacks happen in certain timeframe but not in predictable and *controllable* order
- **run-to-completion**: function is „*atomic*“ unit and must execute as whole, there is no chance for (11,5,12,... to happen)

```
$ node ./concurrency.js
11 22
$ node ./concurrency.js
11 22
$ node ./concurrency.js
183 180
$ node ./concurrency.js
11 22
```

async code styles

History of async

- JS ako jazyk nemal žiadnu notáciu pre asynchronicitu
- Callback bol fajn mechanizmus ako to zapísať syntakticky a čiastočne aj semanticky (všetko čo je odtabované je “LATER“)
- Callback ako taký ale nič nevraví o asynchronicite

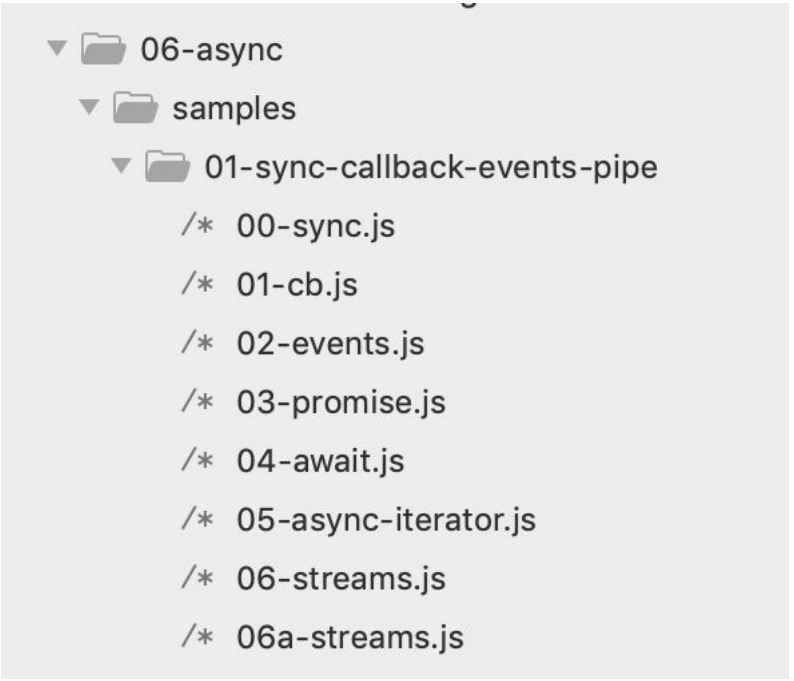
- promises (ako nezávislý štandard) potom ako súčasť ES
- generatory
- async await
- async iterators

Async styles

- sync

1. async **callback**
2. async **events**
3. async **promises**
4. async **async/await**
5. async **iterators/generators**
6. streaming, **pipe, pipeline**

- node.js core module **fs** is a good demo of all styles, it supports all of them, samples:



Async styles – sync

```
const fs = require("fs");

let data = fs.readFileSync(from);
fs.writeFileSync(to, data);

console.log("done");
```

- simple
- readable
- blocking
- buffering

Async styles – (single) callback

```
const fs = require("fs");

fs.readFile(from, (err, data) => {
    if (err) { /* */ ; throw err; }
    fs.writeFile(to, data, (err) => {
        if (err) { /* */ throw err; }
        console.log("done");
    });
});
```

- simple
- readable
- non-blocking
- buffering
  - callback sa vykoná raz zo všetkými dátami

Async styles – events (simplified*)

```
const fs = require("fs");
const stream = fs.createReadStream(from);

stream.on("data", (chunk) => {
    fs.appendFileSync(to, chunk); //*
});
stream.on("end", () => {
    console.log("done");
});
```

- simple
- readable
- non-blocking
- not buffering
  - unless you buffer yourself
  - callback sa vykoná vela krát z čiastočnými dátami

Async styles – events – fully evented version ?

https://stackoverflow.com/questions/11293857/fastest-way-to-copy-file-in-node-js

;-)))))))

- simple
- readable
- non-blocking
- not buffering
  - unless you buffer yourself
  - callback sa vykoná vela krát z čiastočnými dátami

Async styles – promises

```
const fs = require("fs").promises;

fs.readFile(from)
  .then((data) => {
    return fs.writeFile(to, data);
  })
  .then(() => {
    console.log("done");
  });
```

- simple
- readable
- non-blocking
- buffering
  - promise sa rezolvne raz zo všetkými dátami

Async styles – await

```
const fs = require("fs").promises;

let data = await fs.readFile(from);
await fs.writeFile(to, data)
console.log("done");
```

- simple
- readable
- non-blocking
- buffering
  - promise sa rezolvne raz zo všetkými dátami

Async styles – async iterators (simplified*)

```
const fs = require("fs");
const readable = fs.createReadStream(from);
const writable = fs.createWriteStream(to);

for await (const chunk of readable) {
  fs.appendFileSync(to, chunk); //*
}
console.log("done");
```

- simple
- readable
- non-blocking
- not-buffering

https://nodejs.org/api/stream.html#stream_consuming_readable_streams_with_async_iterators

Async styles – streams

```
const fs = require("fs");
const readable = fs.createReadStream(from);
const writable = fs.createWriteStream(to);

readable.pipe(writable)
  .on("finish", () => {
    console.log("done");
  });

console.log("main.end");
```

- simple
- readable
- non-blocking
- not buffering

- pipe – returns stream with events
  - looks like sync API by original definiction, but is evented again

Async styles – streams

```
const fs = require("fs");
const readable = fs.createReadStream(from);
const writable = fs.createWriteStream(to);

readable.pipe(writable)
  .on("finish", () => {
    console.log("done");
  });

console.log("main.end");
```

- simple
- readable
- non-blocking
- not buffering

- pipe – returns stream
  - looks like sync API (no callback), but is evented again
- Mixing pipe with events

Async styles – pipeline utility

```
const fs = require("fs");
const { pipeline } = require("stream");
const readable = fs.createReadStream(from);
const writable = fs.createWriteStream(to);

pipeline(readable, writable, (err) => {
  console.log("done", err);
});
console.log("main.end");
```

- simple
- readable
- non-blocking
- not buffering

- pipe – returns stream
  - looks like async API
- Single done callback

Async, concurrent, async function....

Nejaká základná terminológia

What is async and concurrency

Async

- async is about the **gap between *now* and *later***
- Implementation: Event loop
- Granularity: **function** is single event loop queue entry

Parallel

- things being able to execute at the same **instant of time**
- Implementation: processes, threads
- Granularity: Each statement is broken to **several low-level operations**

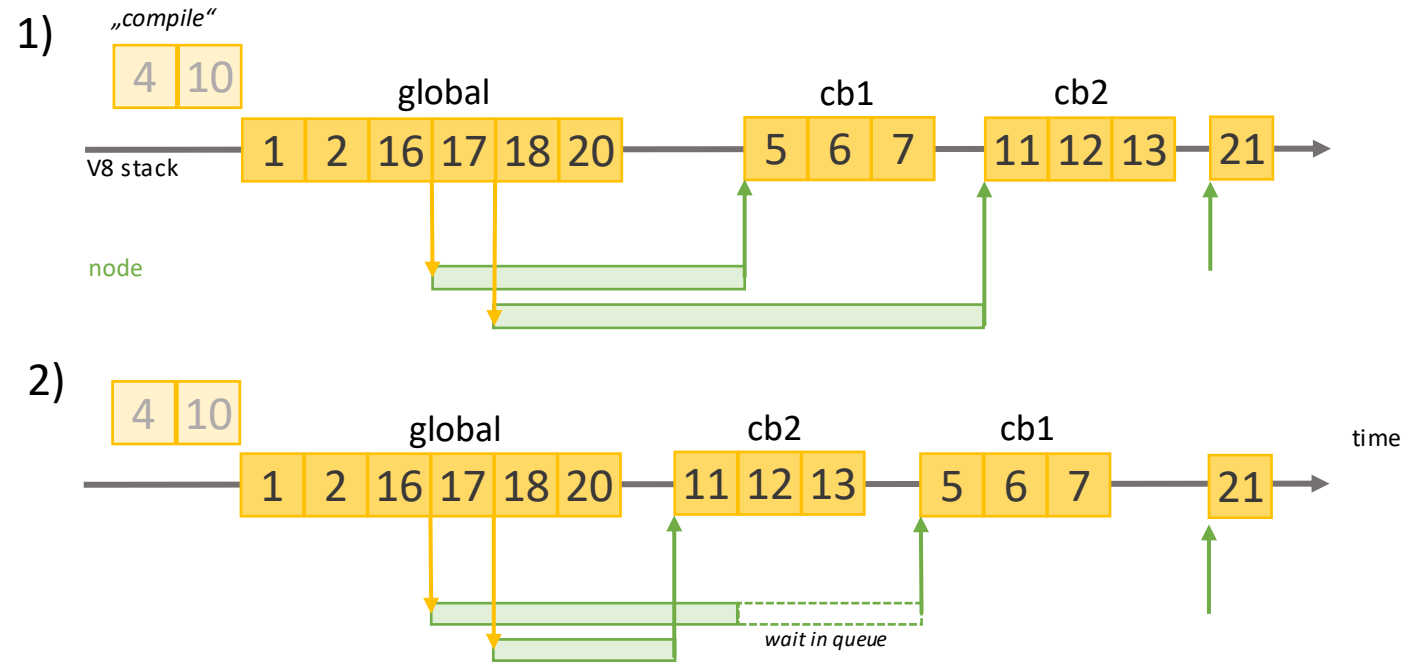
Concurrency

several **higher level** *„tasks“* occuring within **same time frame** (*simultaneously)*.
Not necesarilly in the same instant of time (paralell).

Async and Concurrency

```
1  let a = 1;
2  let b = 2;
3
4  function cb1() {
5    a++;
6    b = b * a;
7    a = b + 3;
8  }
9
10 function cb2() {
11   b--;
12   a = 8 + b;
13   b = a * 2;
14 }
15
16 const https = require("https");
17 https.get("https://nodejs.org/en/", cb1);
18 https.get("https://nodejs.org/en/", cb2);
19
20 process.on("beforeExit", () => {
21   console.log(a, b);
22 });
```

/samples/concurrency.js



- **async**: some code happends ***now*** [1,2,...] and some ***later*** [5,5,6], [11,12,13],[21]
- **concurrent**: callbacks happen **in certain timeframe** but not in predictable and *controllable* order
- **run-to-completion**: function is „*atomic*“ unit and must execute as whole, there is no chance for (11,5,12,... to happen)

```
$ node ./concurrency.js
11 22
$ node ./concurrency.js
11 22
$ node ./concurrency.js
183 180
$ node ./concurrency.js
11 22
```

Async function - definitions

- Pod **async funkciou** budeme zjednodušenie rozumieť jednu z konštrukcií, ktorá nám umožní zapísať, že niečo sa deje **NOW** a potom **LATER**. V zásade existujú 3 možnosti ako je async funkcia implementovaná:

a) funkcia, ktorá v signatúre očakáva callback

b) funkcia, ktorá vracia Promise

c) ES2017 AsyncFunction

```
function task1(params,callback) {
  //...
  callback(null, data);
  callback(err, data);
}

function task2(params) {
  //...
  return Promise.resolve(data);
  return Promise.reject(err);
}

async function task3(params) {
  //...
  return data; //impl. promise
  throw err;
}
```

Async function - usage

- Podľa toho potom rôzne zapisujeme „**later**“

a) funkcia, ktorá v signatúre očakáva callback

b) funkcia, ktorá vracia Promise

c) ES2017 AsyncFunction

Niektoré knižnice sú napísané štýlom A) iné B) iné podporujú A/B/C ..... Niektoré runtimes podporujú len A, niektoré A,B,C

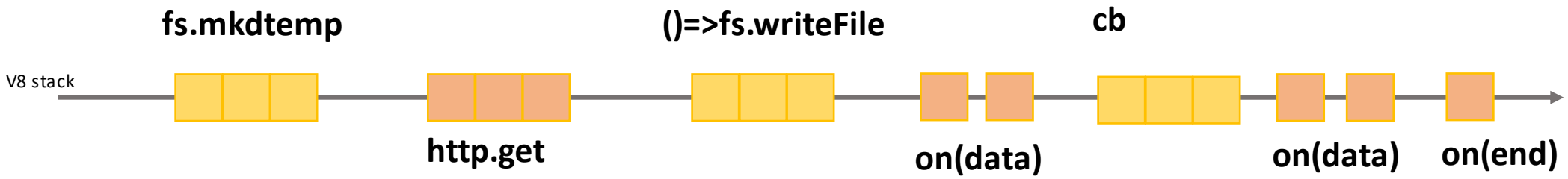
```
// how and where continuation
// is written

task1(args, (err, cb) => {
  if (err) {
    //...
  } else {
    //...
  }
});

task2(args)
  .then(data => {
    //...
  })
  .catch(err => {
    // ...
  });


try {
  let data = await task3(args);
  //...
} catch (err) {
  //...
}
```

1995-2020

2010
2015-2020

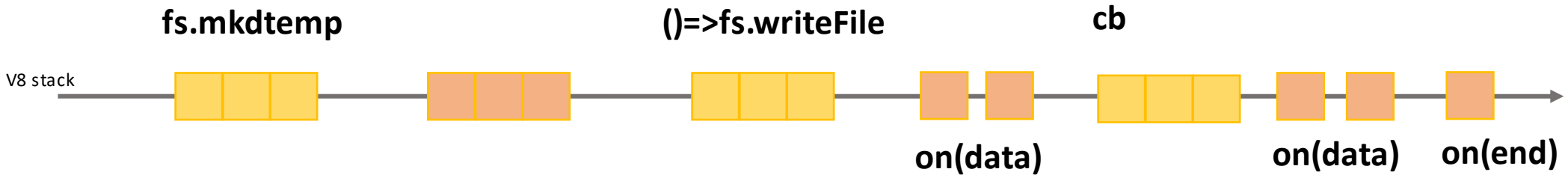
2017-2020

Async process



- async *procesom* alebo *taskom* budeme rozumieť logickú úlohu pozostávajúcu z viacerých časovo oddelene vykonávaných *chunkov* kódu
- V príklade zobrazené dva procesy:
  - Zápis do temp súboru
  - Čítanie dát z web stránky

Concurrency coordination



- ***„process“*** or „task“
  - in the majority of slides here, it means **higher abstraction of some logical process** to be performed (process server response, animate rotation of globe, print response results, process client request) of various granularity (viac funkcií volaných later)

- **Noninteracting**
  - As two or more "processes" are interleaving their steps/events concurrently within the same program, they don't necessarily need to interact with each other if the tasks are unrelated. **If they don't interact, nondeterminism is perfectly acceptable.**
- **Interaction**
  - More commonly, concurrent "processes" **will by necessity interact**, indirectly through **scope** and/or the DOM. When such interaction will occur, **you need to coordinate these interactions to prevent "race conditions“.**
- **Cooperative concurrency**
  - Another expression of concurrency coordination is called "cooperative concurrency.“ The goal is to take a long-running "process" and **break it up into steps** or batches so that other concurrent "processes" have a chance to interleave their operations into the event loop queue.

Async Patterns

because concurrency coordination is **needed** ….

Čo riešime za úlohy

- Všetci už vieme, že JS je:
  - asynchrónny,
  - single threaded (nikdy nebežia dve veci naraz v JS stacku)
  - atomický a run-to-completion na úrovni funkcie
- Potom čokoľvek programujeme nejaký ***top level async process*** treba posudzovať z hľadiska async patternov
  - buď jednotlivo
  - alebo ako ich kombináciu
- **ES/JS a node.js** majú limitovaný zabudovaný mechanizmus na ovládanie ***interaction*** medzi *úlohami,* na riešenie **cooperative concurency** (async.js a iné knižnice)

Prečo to riešime

- V JS je “všecko async“ a beží “concurrentne“
- V iných jazykoch si to musíte explicitne kódnuť inak ste blocking a serializovaný

- V JS to musíte riešiť od prvého dňa a naučiť sa bohužiaľ čítať všetky štýly a písať aspoň niektoré z nich

- Mali by ste JS concurrency využívať a nie bojovať proti nej

Čo riešime za úlohy – klasifikácia vzorov

- **series/parralel** - musia ísť úlohy za sebou (series) alebo môžu ísť paralelne (parallelne)
- **results passing/cummulating** - ak idú za sebou potrebuje tá ďalšia dáta z predošlej, alebo nám záleží len na poradí vykonania
- **[] of async-functions/[] of data + worker** – robím rôzne veci, alebo robím to isté nad inými dátami
- **[] / queue, doWhilst,...** - je zoznam úloh konečný alebo je dynamický a úlohy pribúdajú alebo odbúdajú
- **try, race** – potrebujem výsledok len prvej dokončenej úlohy

- limits –paralell limits, timeouts
- errors – kedy a ako sú signalizované chyby

Async.js

Terminológia vzorov na nasledovných slajdov a ukážky kódov používajú knižnicu async.js

- Async is a utility module which provides straight-forward, powerful functions for working with asynchronous JavaScript. Although originally designed for use with Node.js and installable via npm install async, it can also be used directly in the browser.

- https://caolan.github.io/async/v3/

- Existujú samozrejme iné knižnice
  - http://bluebirdjs.com/docs/getting-started.html
  - ...

Prečo async.js

- Async.js vznikla v čase **callbackov** a riešila peknou syntaxou to čo by inak malo šialené povnárané zápisy ,
- Dnes podporuje aj **promises** a **async** function, a dá sa tak použiť ako jednotná abstrakcia, nezávisle od APIčka knižníc z ktorými kódujete
- Veľa paternov sa dá napísať aj pomocou promises, alebo async iterátorov, ale nie všetky a nie na každom runtime (nie všetci bežia latest node a latest chrome)

Intro

- Majme nejake async funkcie (tasky)
- Povedzme si ako ich chceme vykonať a akú koordináciu potrebujeme
- Zvoľme vhodný pattern z knižnice

```
const task1 = (cb) => setTimeout(() => cb(null, 1));
const task2 = async () => 2;
let task3 = () => Promise.resolve(3);
task3 = asyncify(task3);

parallel([task1, task2, task3], done);
series([task1, task2, task3], done);
waterfall([task1, task2, task3], done);
tryEach([task1, task2, task3], done);
//...
each([1, 2, 3, 4], task4);
detect([1, 2, 3, 4], task5, done);

parallel([
  (cb) => detect([1, 2, 3, 4], task5, cb),
  task2,
  (cb) => series([task1, task2, task3], cb)
], done)
```

c) Function returning promise requires conversion with ***asyncify***

Sampel je len ilustratívny.....

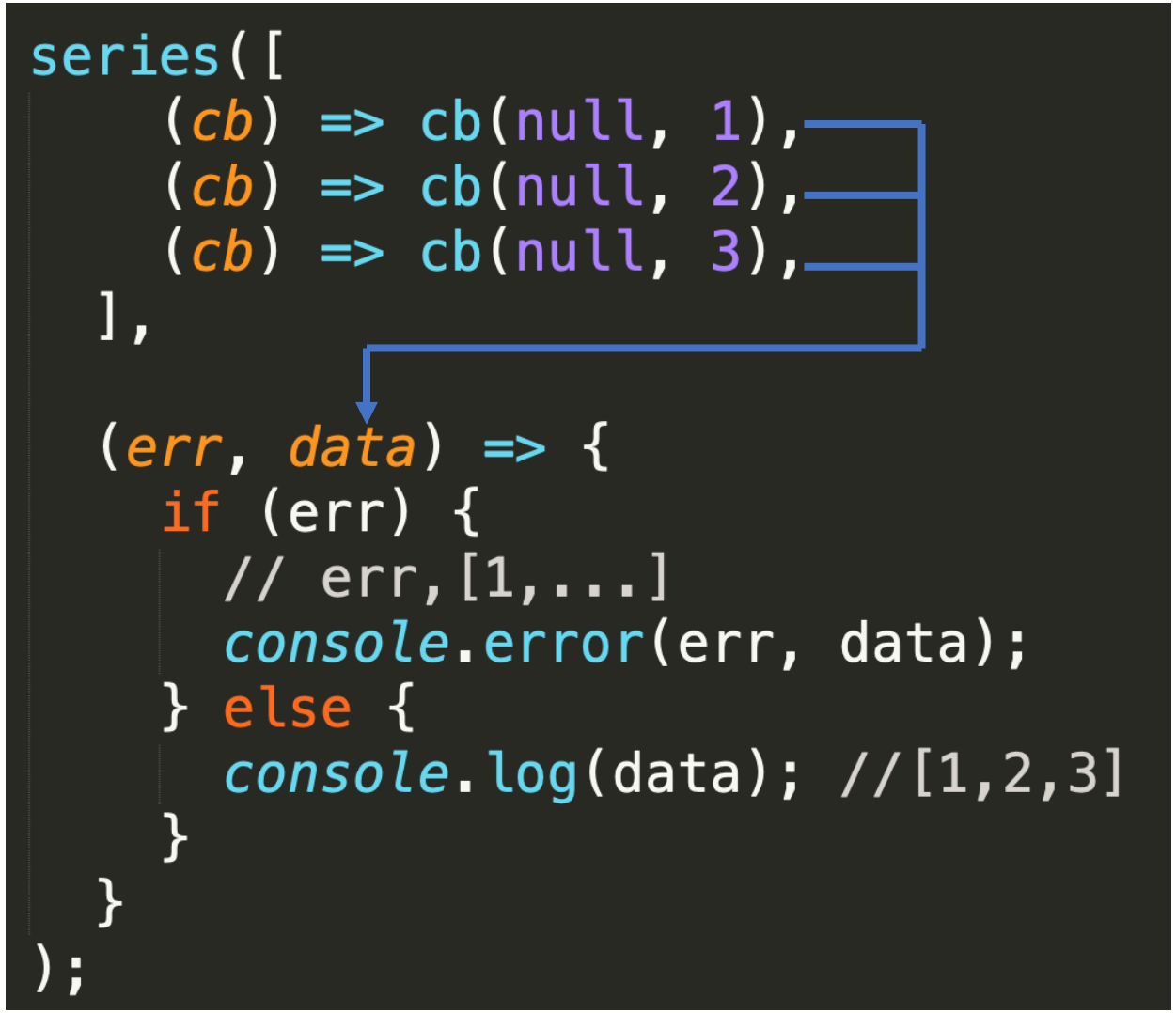
Series

- simplest sequential

Use when:

- tasks are **dependent on order** of execution
- **but do not need** data from previous task

- data ***returned*** from tasks can be **collected** in final callback



Series

- Výhodou async.js je aj flexibilita návratových hodnôt
- Ak chcete po dobehnutí môžete volať callback
- Ale si výsledok vyzdvihnete ako promise
- Alebo si na neho počkáte cez await

```
// flexible inputs
// any kind of async function
const task1 = (cb) => cb(null, 1);
const task2 = async () => 2;
let task3 = () => Promise.resolve(3);
task3 = asyncify(task3);

// flexible outputs
// a) result as callback
series([task1, task2, task3], print);

// b) result as promise
series({task1, task2, task3})
  .then(printD);

// c) await
(async () => {
  const data = await series([task1, task2]);
  printD(data);
})();
```

Series

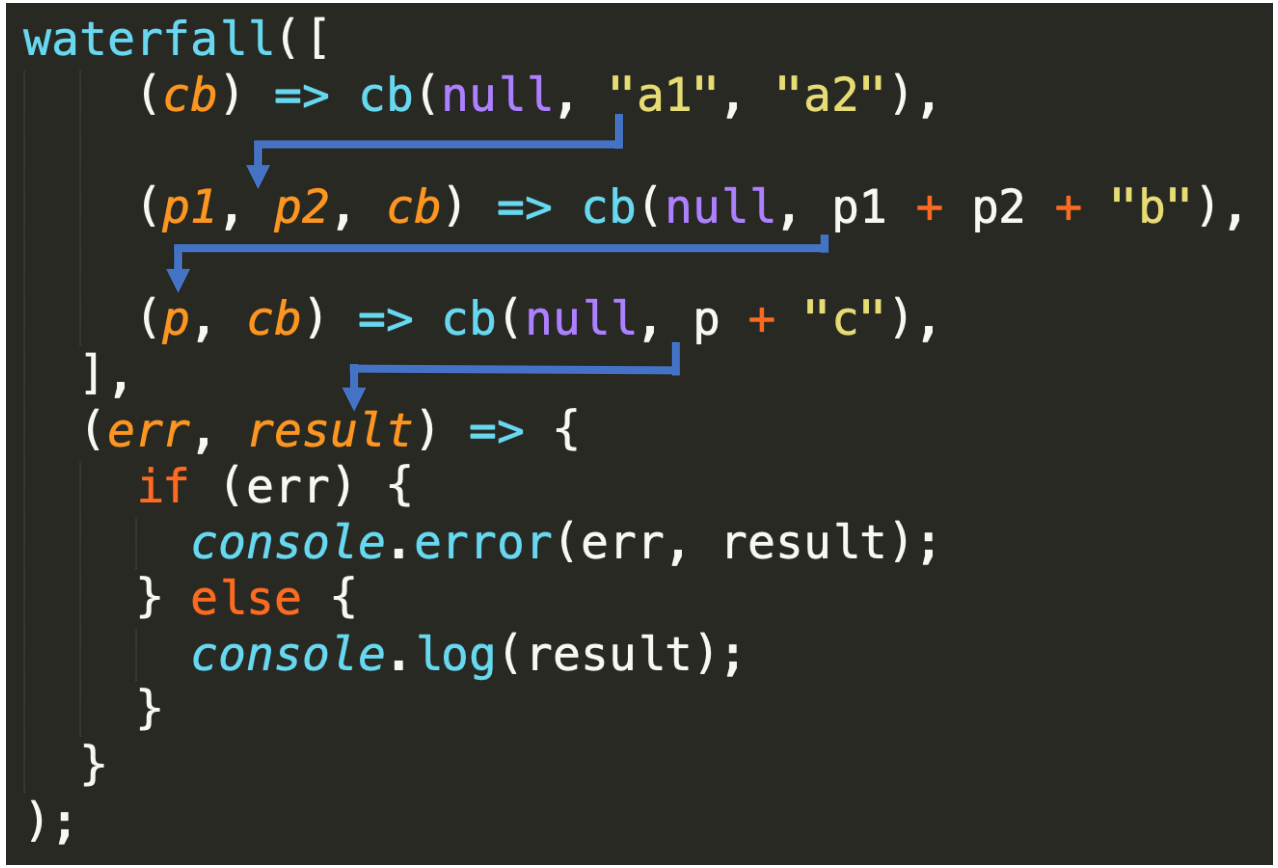
- TODO: Real life example ?

Waterfall

- sequential

Use when:

- tasks are using **data from previous task**
- next task gets data from *returned* from previous task
- final callback gets data of last task



Waterfall

- Znova platí, že kľudne vieme zmixovať async funkcie rôznych typov
- a ak majú vhodné signatúry vieme ich priamo zapísať do waterfall
- Ak nie “adaptneme“ ich cez arrows
- Ak su tam nejake sync kroky, zaradime ich ako funkcie do waterfall alebo dnuka do arrows

- Zase veľmi pekný, deklaratívny zápis, bez arrows a lúštenia kódu vieme čo to robí až potom čítame ako to robí.....

```
const task1 = (cb) =>
  /*...*/
  cb(null, "a1", "a2");
const task2 = async (p1, p2) =>
  /*...*/
  p1 + p2 + "b";
let task3 = (p) =>
  /*...*/
  Promise.resolve(p + "c");
task3 = asyncify(task3);

waterfall([
  task1,
  task2,
  task3
], print);
```

Waterfall

- TODO: budeme robit na cviku, writeTempFile ako sekvenciu 2 async funkcii z node.js
- toto je jedna z moznych variant ako k tomu pristupit

```
// 1. keep things normal, without any special async.js constructs
// like normal "functional style" programm
const tempDirName = path.join(os.tmpdir(), `${process.pid}-`);
const tempFileName = (folder) => path.join(folder, fileName);
const writeFile = (tempFileName) => {
  try {
    fs.writeFile(tempFileName, ...args, (e) => cb(e, tempFileName))
  } catch (err) { // sync errors from writeFile
    cb(err);
  }
};

// 2. then use the constracts, and adjust for async.js
waterfall([
  constant(tempDirName),  // a) variables to constant functions
  fs.mkdtemp,             // b) use 1:1, core API with cb
  asyncify(tempFileName), // c) convert sync to async
  writeFile               // d) create custom function if needed
], cb);
```

Waterfall

- V samploch mate trivi priklad na vykradanie JS dokumentacie
  - /samples/03-patterns/03a-waterfall-scrape-mdn.js

```
waterfall([
    (cb) => getPageHtml("Array", cb),
    parseMethods,
    // ... download method docs
  ],
  print
);
```

tryEach

- sequential

It runs each task in series but stops whenever **any of the functions were successful**

Use when:

- you have more alternatives to perform the task and first succesfull is enough
- If one of the tasks were successful, the callback will be passed the **result of the successful task**
- if all fail, cb will be called with error of last attempt

```
const a = (cb) => setTimeout(() => cb(new Error(), "a"), 3000);
const b = (cb) => setTimeout(() => cb(null, "b"), 2000);
const c = (cb) => setTimeout(() => cb(null, "c"), 8000);


tryEach([
    a,
    b,
    c
  ],
  (err, data) => {
    if (err) {
      console.error(err, data);
    } else {
      console.log(data);
    }
  }
);
```

doWhilst

- sequential – do...while for async functions,
- do, while and done are all ***functions, while*** is **sync** test

Use when:

- algorithms is best desribed by ***do...while*** but
- ***do*** contains some async function

```
20  const [, , URL] = process.argv;
21
22  const { doWhilst } = require("async");
23  const request = require("request")
24    .defaults({ json: true });
25
26
27  let pageIndex = 0;
28  const results = [];
29
30  doWhilst(
31    function _do(done) {
32      request(`${URL}&pageIndex=${pageIndex + 1}`,
33        (err, { statusCode }, result) => {
34          if (err || statusCode !== 200)
35            return done(err || new Error(statusCode));
36
37          results.push(...result.rules);
38          pageIndex++;
39          done(null, result);
40        }
41      );
42    },
43    function _while({ p, ps, total }) {
44      // has more records, index * pageSize
45      return p * ps < total;
46    },
47    function _done(err) {
48      if (err) throw err;
49      console.log(JSON.stringify(results, null, 2));
50    }
51  );
```

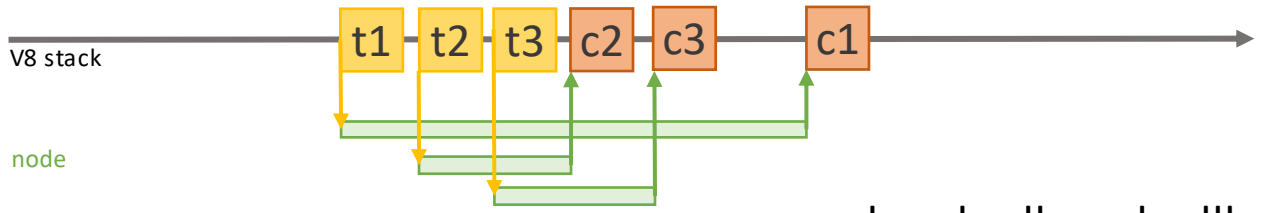
last

real life example: sonar API, ***paged search***
/samples/sonar-rules-cli.js

Running in series

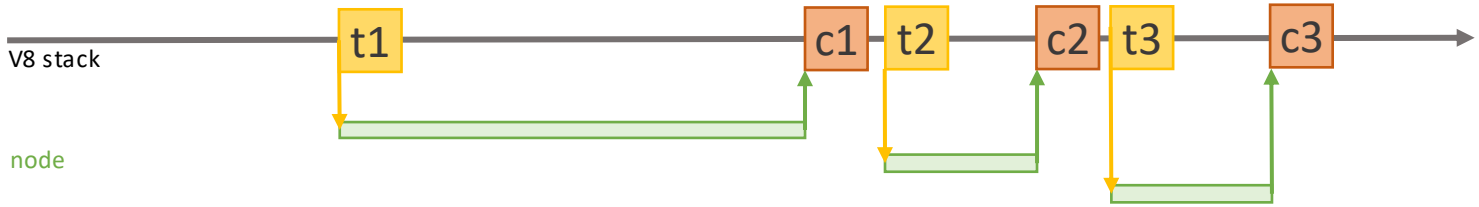
Normal JS code

un ordered callbacks



Series/waterfall/awaits

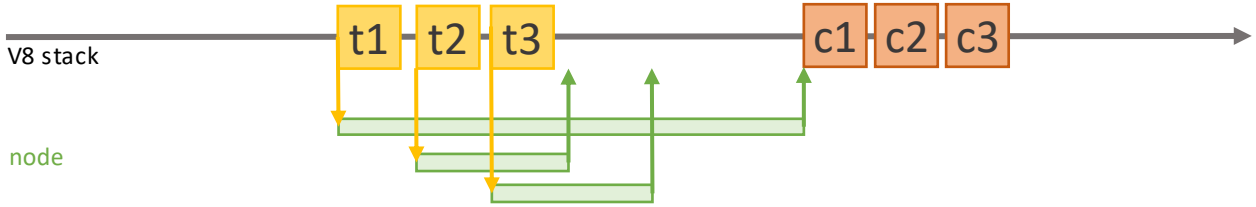
ordered calls and callbacks



???

unordered calls

ordered callbacks



parallel

- parallel

Use when:

- you have independent processes
- you need to wait for all of them to execute and then continue
- you want to collect results from all of them

```
parallel({
    task1: a,
    task2: b,
    task3: c
  },
  (err, data) => {
    // { task2: 'b', task1: 'a', task3: 'c' }
    if (err) {
      console.error(err, data);
    } else {
      console.log(data);
    }
  }
);
```

parallelLimit

- The same as parallel but **runs** a maximum of limit async operations ***at a time***.

- koľko približne bude trvať tento program ?
- ak limit je 2 ?
- a tasky *trvajú* 3,2 a 8 sekúnd

```
const a = (cb) => setTimeout(() => cb(null, "a"), 3000);
const b = (cb) => setTimeout(() => cb(null, "b"), 2000);
const c = (cb) => setTimeout(() => cb(null, "c"), 8000);

parallelLimit({
    task1: a,
    task2: b,
    task3: c
  },
  2,
  (err, data) => {
    if (err) {
      console.error(err, data);
    } else {
      console.log(data);
    }
  }
);
```

race

- parallel - Once any of the tasks complete or pass an error to its callback, the main callback is immediately called.

Use when:

- podobne ako sekvenčný tryEach teda ak máte alternatívne algoritmy a stačí vám ***výsledok najrýchlejšieho***

Ale pozor:

- ak ***najrýchlejší*** spadne aj callback sa zavolá z chybou
- v ok aj v err scenári - nik nezabezpečí odplánovanie tých spustených timerov
- koľko teda pobeží tento program
  - po callback ?
  - po úplný koniec node procesu ?

```
const a = (cb) => setTimeout(() => cb(null, "a"), 3000);
const b = (cb) => setTimeout(() => cb(null, "b"), 2000);
const c = (cb) => setTimeout(() => cb(null, "c"), 8000);

race([
    a,
    b,
    c
  ],
  (err, data) => {
    if (err) {
      console.error(err, data);
    } else {
      console.log(data);
    }
  }
);
```

any

- parallel

Problém z race:

- ak ***najrýchlejší*** spadne aj callback sa zavolá z chybou

Často ale potrebujeme iný scenár, teda prvý úspešný výsledok toho co zbehol bez chyby ako prvy

a uplny error, až keď všetky zlyhajú

See: https://esdiscuss.org/topic/promise-any

- TODO: máme v async.js paralelný pattern z takouto semantikou ? Pohladajte si v doksoch k async.js

auto, autoInject

- executes in *best order based on requirements*
- ***Deklaratívny*** zápis
  - čo chcete dosiahnuť a
  - čo na každý *task* potrebujete

```
async.auto({
    // this function will just be passed a callback
    readData: async.apply(fs.readFile, 'data.txt', 'utf-8'),
    showData: ['readData', function(results, cb) {
        // results.readData is the file's contents
        // ...
    }]
}, callback);

async.auto({
    get_data: function(callback) {
        console.log('in get_data');
        // async code to get some data
        callback(null, 'data', 'converted to array');
    },
    make_folder: function(callback) {
        console.log('in make_folder');
        // async code to create a directory to store a file in
        // this is run at the same time as getting the data
        callback(null, 'folder');
    },
    write_file: ['get_data', 'make_folder', function(results, callback) {
        console.log('in write_file', JSON.stringify(results));
        // once there is some data and the directory exists,
        // write the data to a file in the directory
        callback(null, 'filename');
    }],
    email_link: ['write_file', function(results, callback) {
        console.log('in email_link', JSON.stringify(results));
        // once the file is written let's email a link to it...
        // results.write_file contains the filename returned by write_file.
        callback(null, {'file':results.write_file, 'email':'user@example.com'});
    }]
}, function(err, results) {
    console.log('err = ', err);
    console.log('results = ', results);
});
```

Queue

- parallel

Use when:

- ak neviete dopredu zoznam taskov a postupne pribúdajú

Why:

- Paterny čo sme si doteraz ukázali pracovali spravidla zo známym počtom itemov

Creates a queue object with the specified concurrency. Tasks added to the queue are processed in parallel (up to the concurrency limit). If all workers are in progress, the task is queued until one becomes available. Once a worker completes a task, that task's callback is called.

```
// create a queue object with concurrency 2
var q = async.queue(function(task, callback) {
    console.log('hello ' + task.name);
    callback();
}, 2);

// assign a callback
q.drain(function() {
    console.log('all items have been processed');
});
// or await the end
await q.drain()

// assign an error callback
q.error(function(err, task) {
    console.error('task experienced an error');
});

// add some items to the queue
q.push({name: 'foo'}, function(err) {
    console.log('finished processing foo');
});
// callback is optional
q.push({name: 'bar'});

// add some items to the queue (batch-wise)
q.push([{name: 'baz'},{name: 'bay'},{name: 'bax'}], function(err) {
    console.log('finished processing item');
});

// add some items to the front of the queue
q.unshift({name: 'bar'}, function (err) {
    console.log('finished processing bar');
});
```

Cargo

Patterns - for tasks/ flows coordination

- coordinating various async tasks represented by **different async functions**
- or chunks (**cargo**), or datas (**queue**)
- for serial and parallel usage
- declarative dependencies and automatic ordering with **auto**

	pattern/API	parallel	tasks	pass data
series	**series**	no	Array | Iterable | Object	no
	waterfall	no	Array	yes
	seq	no	…functions	yes
	compose	no	…functions	yes
	applyEachSeries	no	Array | Iterable | Object	
	timesSeries	no	one function	
	tryEach	no	Array | Iterable | Object	no
parallel	**parallel**	yes	Array | Iterable | Object	no
	parallelLimit	yes	Array | Iterable | Object	no
	race	yes	Array	no
	queue	yes	**dynamic []** of any + worker	
	priorityQueue	yes	**dynamic []** of any (+ priority) + worker	
	applyEach	yes	Array | Iterable | Object	
	times	yes	one function	
	timesLimit	yes	one function	
others	**auto**	yes*	functions + requirements	yes
	autoInject	yes*	functions + requirements*	yes
	cargo	no	1 worker + payload (chunks)	

Patterns – for data collections

- when you want to run **same async algorithm** over collection of **data**
- mostly Array functions: map, filter, ... just with async function as callback (iteratee)

	Async API	parralel	description
each	**each**	yes	**Applies** the function iteratee to **each item** in coll, in parallel.
	eachLimit	yes*	
	eachSeries	no	
	eachOf	yes	Applies the function iteratee to each item in coll, in parallel. passes also index.
	eachOfLimit	yes*	
	eachOfSeries	no	
map	**map**	yes	Produces a new collection of values by mapping each value in coll through the iteratee function
	mapLimit	yes*	
	mapSeries	no	
	mapValues	yes	Produces a **new Object** by **mapping each value of obj** through the iteratee function
	mapValuesLimit	yes*	
	mapValuesSeries	no	
filter	**filter**	yes	Returns a **new array** of all the values in coll **which pass an async** truth **test**.
	filterLimit	yes*	
	filterSeries	no	
	reject	yes	The opposite of filter.
	rejectLimit	yes*	
	rejectSeries	no	
truth checks	**every**	yes	Returns **true if every** element in coll **satisfies an async test**.
	everyLimit	yes*	
	everySeries	no	
	some	yes	Returns true if **at least one element** in the coll **satisfies an async test**
	someLimit	yes*	
	someSeries	no	
	detect	yes	Returns the **first value** in coll **that passes an async truth test**
	detectLimit	yes*	
	detectSeries	no	
reduce	**reduce**	no	Reduces coll into a single value using an async iteratee to return each successive step
	reduceRight	no	
	transform	no	A relative of reduce. Takes an Object or Array, and iterates over each element in series
	groupBy	yes	Returns a **new object**, where each **value corresponds to an array of items**, from coll, **by returned key**
	groupByLimit	yes*	
	groupBySeries	no	
	concat	yes	**Applies** iteratee to each item in coll, **concatenating** the results. Returns the **concatenated list.**
	concatLimit	yes*	
	concatSeries	no	
sort	sortBy		**Sorts a list** by the results of running each coll value through an async iteratee.

Patterns – for single function

- basic async patterns and helpers to be applied on function.
- Mostly return wrapped function suitable for another async usage.

pattern/API	Description
retry	**Attempts to get a successful response** from task no more than times times before returning an error.
retryable	Converts function to **new retryable function**, without calling it
timeout	**Sets a time limit on an asynchronous function**. Returns wrapped function
memoize	**a memoized version of fn**
unmemoize	Undoes a memoized function, reverting it to the original, unmemoized form.
ensureAsync	Wrap an async function and **ensure it calls its callback** on a later tick of the event loop
asyncify	Take a **sync function and make it async, passing its return value to a callback**.
reflect,reflectAll	Wraps the async function in another function that always **completes with a result object, even when it errors**
apply	Creates a continuation function with some **arguments already applied**.
constant	Returns a function that when called, **calls-back with the values provided**.
setImmediate	
nextTick	
log	
dir	

Cooperative Concurrency

The World will stop

Cooperative Concurrency

What:

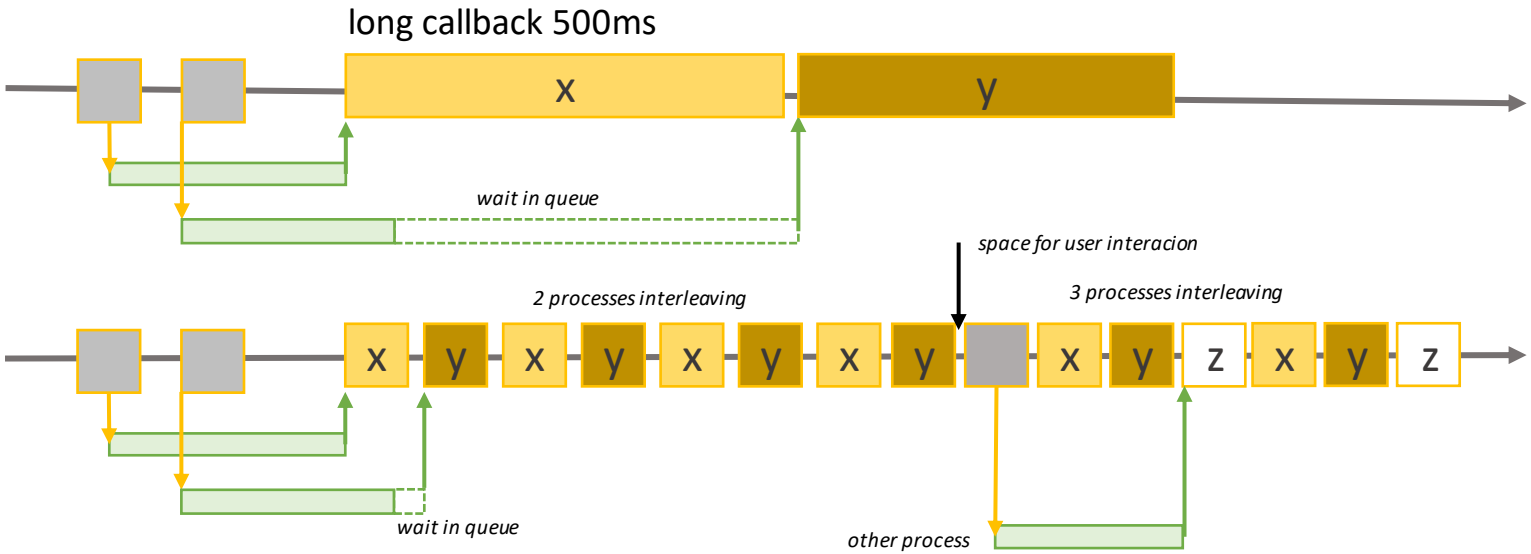
- The goal is to take a **long-running "process"**
- and **break it up into steps** or batches or chunks
- so that **other** concurrent "processes" have a chance to **interleave** their operations into the event loop queue

Why:

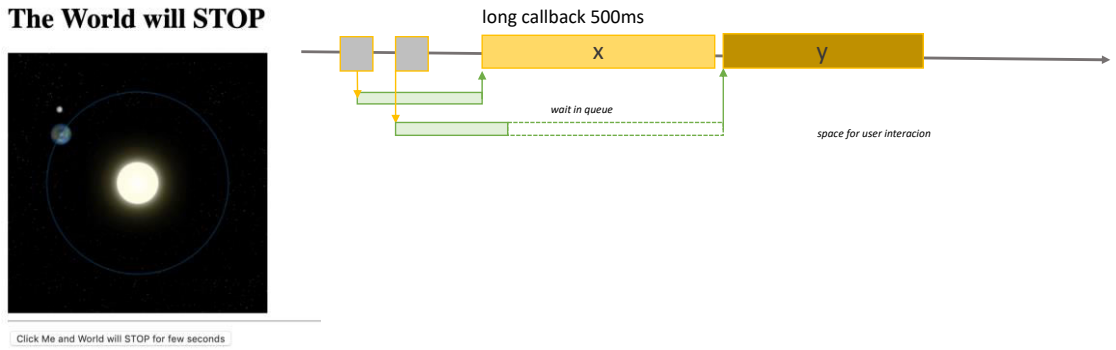
- to receive better performance of simultaneously running processes in single threaded environment
- This applies to **both Noninteracting** and **Interacting** „processes“

Cooperative Concurrency

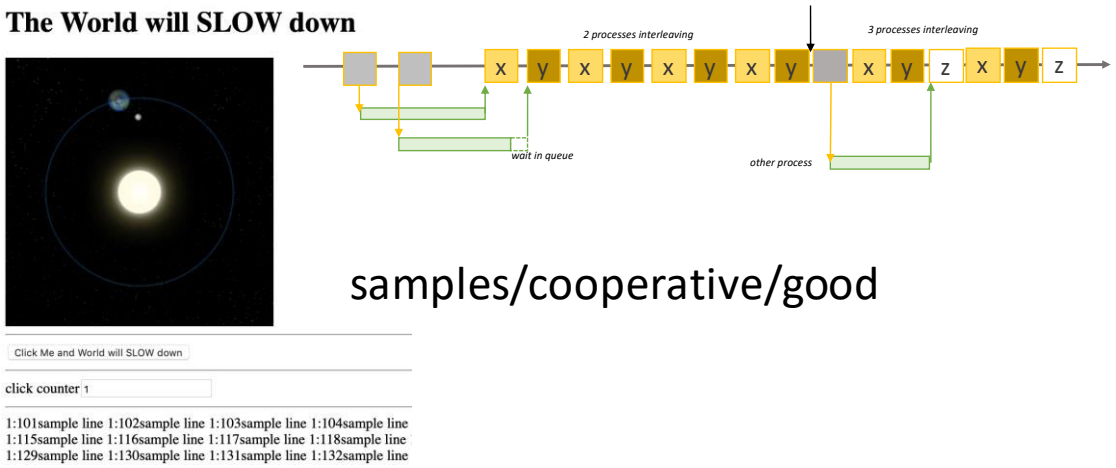
- if callback takes a lot of time,
  - eg. processes a lot of data items in „*loop*“
- nothing else can happen:
  - in browser – no rendering, no clicks, ...
  - on web server – no other requests can be served
- The idea is to split processing to „chunks“,
- ***give up*** stack after processing ***partial chunk***
- schedule next chunk processing
- this brings chance for user or other events to start other processes
- brings better ***perceived performance*** and ***responsivness***



Cooperative Concurrency **Demo**
The World will **stop** vs. The World will **slow down**



samples/cooperative/bad



samples/cooperative/good

```
  xhr.onload = function() {
    drawData(xhr.response.split("\n"));
  };
  xhr.send();
});

function drawData(data) {
  for (var i = 0; i < data.length; i++) {
    display.innerHTML += `${c}:${data[i]}`;
  }
}
```

```
  xhr.onload = function() {
    drawData(xhr.response.split("\n"));
  };
  xhr.send();

});
function drawData(data) {
  if (!data.length) return;
  let chunk = data.splice(0, 100);
  for (var i = 0; i < chunk.length; i++) {
    display.innerHTML += `${c}:${data[i]}`;
  }
  setTimeout(() => drawData(data), 0);
}
```

Cooperative Concurrency

What to do to be cooperative:

A. Split data to several chunks process with one function (previous sample)
   - for loop vs recursion
   - async recursion
   - async for ….
   - another async patterns

B. Split sequence of sync calls to several async calls in async sequence pattern
   - sequence of asyncified functions

Solutions

A. Make your own, rozdeluje long tasky na mensie async tasky, rodelujte data (vid predosly example, zalozene na setTimeout)
B. async.js, cargo, queue a ine

C. TODO: iné packages
    A. thaw.js https://github.com/robertleeplummerjr/thaw.js thaw.js helps keep the browser from freezing by splitting long computations into small parts which are processed more naturally when the browser isn't busy. This simulates asynchronous behaviour and keeps the browser from freezing. thaw.js thaws frozen browsers.
    B. TODO: pohladajte si dalsie

Callbacks

Callbacks

- When your JS program makes async operation, you set up the "response" code in a function (commonly called a "callback"), and the JS engine tells the hosting environment, "Hey, I'm going to suspend execution for now, but whenever you finish with that request, and you have some data, please *call* this function *back*.“ (@getify)

Callback signatures

We have already met several callbacks during revious lessons:

- cb(err, data)
- cb(err)
- cb(data)

- cb(err, data, data2)

TODO: examples from node and npm modules API

Callbacks (+)

- callbacks are by far the **most common way** that asynchronycity in JS programs is expressed and managed
- Indeed, the callback is the **most fundamental async pattern** in the language
- Countless JS programs, even very **sophisticated** and **complex** ones, have been written upon no other async foundation than the callback
- The callback function is the async work horse for JavaScript, and it does its job respectably

https://github.com/getify/You-Dont-Know-JS/blob/master/async%20%26%20performance/ch2.md

Order of code and execution

```
a(function() {
  b();

  c(function() {
    d();
  })

  e();
});
f();
```

```
a,f,b,c,e,d
```

Order of code and execution

```
a(function() {
  b();

  c(function() {
    d();
  })

  e();
});
f();
```

```
// order-async-ac.js          a,f,b,c,e,d
// order-sync-a.js            a,b,c,e,f,d
// order-sync-c.js            a,f,b,c,d,e

// order-sync-depends.js
//                            a,f,b,c,e,d
//                            a,b,c,d,e,f
//                            a,b,c,e,f,d
//                            a,f,b,c,d,e
```

prednasky/09-async/samples/callbacks/order-*.js

Number of callbacks

```
a(function() {
  b();

  c(function() {
    d();
  })

  e();
});
f();
```

```
// order:
// a,f,b,c,e,d
// a,f,b,c,e,b,c,e
// a,f,b,c,e,b,c,e,d,d
```

prednasky/09-async/samples/callbacks/order-*.js

Callbacks (-)

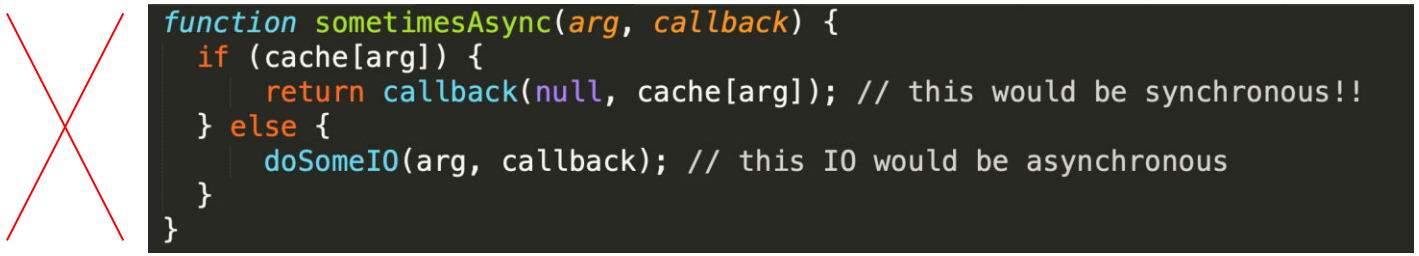
- It turns out that how we express asynchrony (with callbacks) in our code doesn't map very well at all to that **synchronous brain** planning behavior
- The reason it's difficult for us as developers to write async evented code, especially when all we have is the callback to do it, is that stream of consciousness thinking/planning is **unnatural for most of us.**
- We think in step-by-step terms, but the tools (callbacks) available to us in code are not expressed in a step-by-step fashion once we move from synchronous to asynchronous.
- And that is why it's so **hard to accurately author and reason about async JS code with callbacks**: because it's not how our brain planning works.

https://github.com/getify/You-Dont-Know-JS/blob/master/async%20%26%20performance/ch2.md

callback hell

- with callback **we *trust* the 3rd party** (the one who calls our function) to
  - call us **only once**
    - but it can call us multiple times,
    - or not at all
  - call us **in the async or sync** fashion, but not sync one time and async second time
  - solvable with some boilerplate code but clutter
- order of code is not sequential
- hardcoded continuations (names of next functions)
- error handling paths complicate the code
- error handling of sync parts of async. functions (writeFile and encoding)
- cluttered code *hard to reason* about
- ... indentation ... is just side effect not helping previous real problems

sometimes-async-function



```
function sometimesAsync(arg, callback) {
  if (cache[arg]) {
      return callback(null, cache[arg]); // this would be synchronous!!
  } else {
      doSomeIO(arg, callback); // this IO would be asynchronous
  }
}
```

prečo je takáto async funkcia problém (ozaj v skratke):

- nemôžte sa spoľahnúť na poradie (viď predošlý príklad)
  - b,c,e .... e vs.
  - b,c,d,e
- Maximum call stack size exceeded

> If you have an API which takes a callback,
> and *sometimes* that callback is called immediately,
> and *other times* that callback is called at some point in the future,
> then you will render any code using this API impossible to reason about, and cause the release of Zalgo.

http://blog.izs.me/post/59142742143/designing-apis-for-asynchrony
https://blog.ometer.com/2011/07/24/callbacks-synchronous-and-asynchronous/

```
a(function() {
  b();

  c(function() {
    d();
  })

  e();
});
f();
```

sometimes-async-function, never called, multiple call – solutions ?

Sometimes:

- As authors: Do not produce sometimes sync, async functions
- As consumers: When consuming sometimes async function wrap it with **async.ensureAsync(fn)**

Never:

- As authors: it is bug, do all callback paths coverage with your **tests**
- As consumers: wrap your functions with **async.timeout(fn)**

Multiple:

- As authors: it is bug, do all callback paths coverage with your **tests**
- As consumers: ???

solutions

async.**ensureAsync** and
async.**timeout**

```
function a(cb) {
  // sometime sync
  console.log("a");
  sometimes() ? setTimeout(cb, 0) : cb();
}
function c(cb) {
  console.log("c");
  // sometimes never
  if (sometimes()) setTimeout(cb, 0);
}
```

```
 3  a(function() {
 4    b();
 5    c(function(err) {
 6      if (err) throw err;
 7      d();
 8    })
 9
10    e();
11  });
12  f();
```

```
// order:
// a;b;c;e;f;d  - sync a
// a;b;c;e;f;   - never d
// a;f;b;c;e,d  - async a
// a;f;b;c;e    - never d
```

```
 3  ensureAsync(a)(function() {
 4    b();
 5    timeout(c, 100)(function(err) {
 6      if (err) throw err;
 7      d();
 8    })
 9
10    e();
11  });
12  f();
```

```
// order:
// a;f;b;c;e;d          - async a
// a;f;b;c;e; error     - never d
```

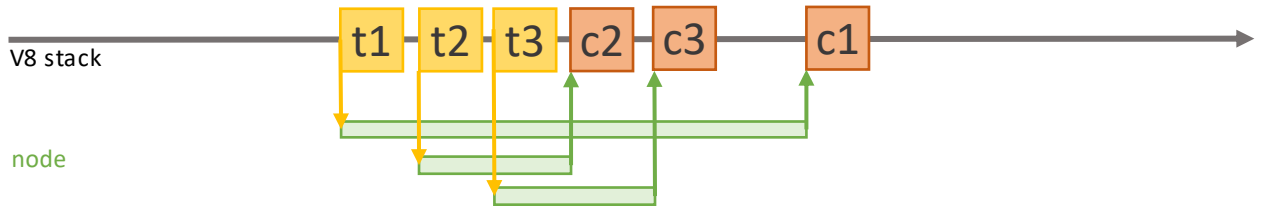

_aux

Links (not reviewed)

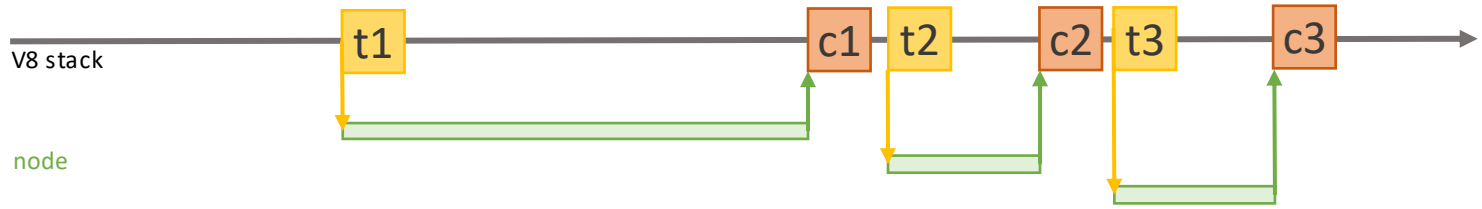
- https://medium.com/@ExplosionPills/javascript-synchronization-patterns-ec8c05ac05be
- https://itnext.io/understand-async-iterators-665259680044

Series/Waterfall – runs in series

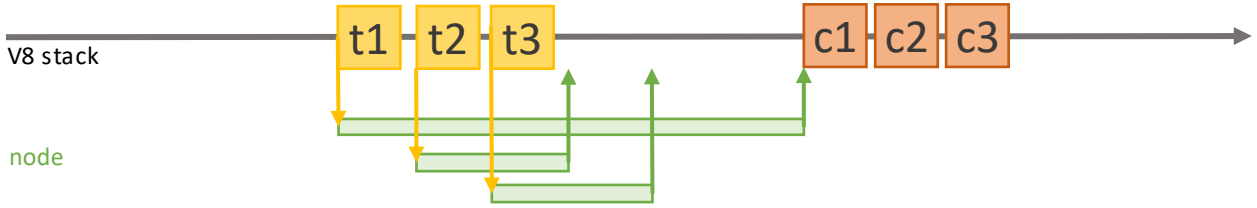
Normal JS code



Series/waterfall/awaits



???



- https://github.com/getify/revocable-queue/blob/master/README.md