Structure and architecture, API, security, asynchronity, routing, middlewares, data access, memory, cpu, state management, etc.

Node.js Antipatterns

Timur Shemsedinov

Node.js Antipatterns Classification

- Structure and arch.
- Initialization
- Dependency issues
- Application state
- Middlewares
- Context isolation
- Security issues
- Asynchronity issues
- Blocking operations
- Memory leaks
- Databases and ORM
- Error handling

No layers, everything mixed:

- Configuration and Dependency management
- Network protocols related code (http, tcp, tls…)
- Request parsing, Cookies, Sessions
- Logging, Routing, Business-logic
- I/O: fs, Database queries
- Generating responses and error generation
- Templating, etc.

github.com/HowProgrammingWorks/AbstractionLayers

Mixed Layers

```
router.get('/user/:id', (req, res, next) => {
  const id = parseInt(req.params.id);
  const query = 'SELECT * FROM users WHERE id = $1';
  pool.query(query, [id], (err, data) => {
    if (err) throw err;
    res.status(200).json(data.rows);
    next();
  });
});
```

What do we want?

```
async (arg1, arg2, arg3) => {
  const data1 = await getData(arg1);
  if (!data1) throw new Error('Message');
  const [data2, data3] = await Promise.all(
    [getData(arg2), getData(arg3)]
  );
  return await processData(data1, data2, data3);
}
```

github.com/HowProgrammingWorks/API

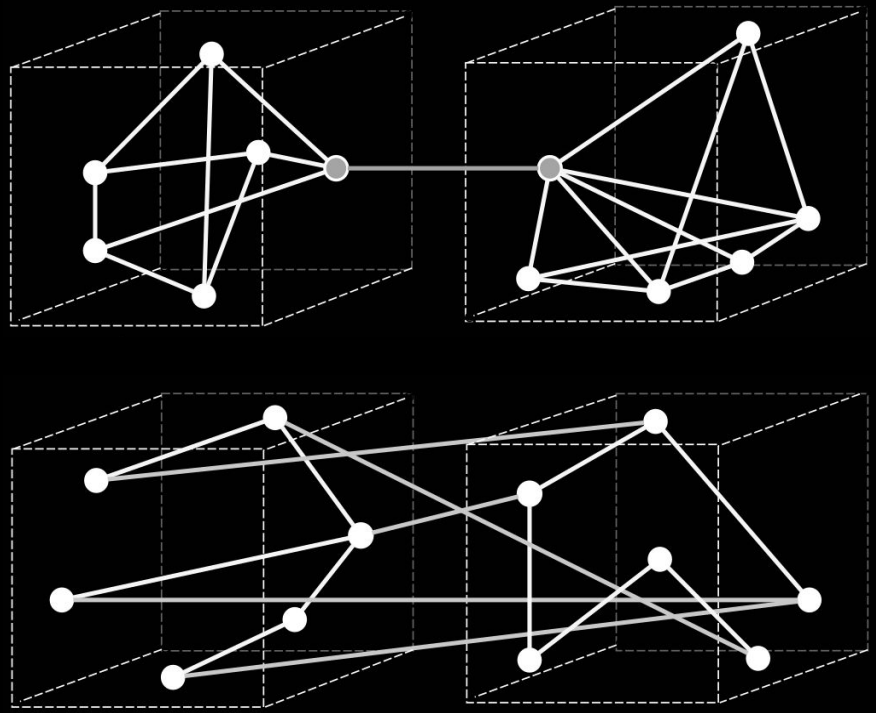
Transport agnostic, Framework agnostic

Middlewares

Middlewares is an extremely bad idea for low coupling and high cohesion

Middlewares changes:

- Socket state
- Db connection state
- Server state



Don’t use globals to pass state


```
let groupName;

app.use((req, res, next) => {
  groupName = 'idiots'; next();
});

app.get('/user', (req, res) => {
  if (res.groupName === 'idiots') {
    res.end('I know you!');
  }
});
```

Don’t mixins to req and res


```
app.use((req, res, next) => {
  res.groupName = 'idiots';
  next();
});

app.get('/user', (req, res) => {
  if (res.groupName === 'idiots') {
    res.end('I know you!');
  }
});
```

Don’t mixins locals to res


```
app.use((req, res, next) => {
  res.locals.groupName = 'idiots';
  next();
});

app.get('/user', (req, res) => {
  if (res.locals.groupName === 'idiots') {
    res.end('I know you!');
  }
});
```

Don’t mixins methods

```
app.get('/user/:id', (req, res, next) => {
  req.auth = (login, password) => { /* auth */ };
  next();
});

app.get('/user/:id', (req, res) => {
  if (req.auth(req.params.id, '111')) {
    res.end('I know you!');
  }
});
```

Don’t require in middleware / handler


```
app.get((req, res, next) => {
  req.db = new require('pg').Client();
  req.db.connect();
  next();
});

app.get('/user/:id', (req, res) => {
  req.db.query('SELECT * from USERS', (e, r) => {
  });
});
```

Don’t connect db in handlers


```
app.get((req, res, next) => {
  req.db = new Pool(config);
  next();
});

app.get('/user/:id', (req, res) => {
  req.db.query('SELECT * from USERS', (e, r) => {
  });
});
```

Don’t loose connection on error


```
const db = new Pool(config);

app.get('/user/:id', (req, res) => {
  req.db.query('SELECT * from USERS', (err, r) => {
    if (err) throw err;
    // Prepare data to reply client
  });
});
```

Why execution context isolation?

JS FEST

- Errors, crashes
- Memory leaks and other resources
- Application: data, database connections
- File system and root directory
- OS environment, PID, IPC
- OS Security: users, groups
- Networking: socket descriptors, ports, hosts

Execution isolation levels

- Hardware: servers, networks
- Virtual machine (hypervisor)
- Container (docker)
- Process (node)
- Thread (worker_threads)
- Sandbox (vm.createContext, vm.Script)
- Software context (object, closure)

Dependency issues

- Almost all we need is in
  - JavaScript and in Node.js API (just check)
- Adding dependencies from NPM tell yourself:
  - It’s my responsibility
    - authors give no warranties,
    - it’s just a good will if they help and support
  - and I will support fork

Security issues

- Malicious modules from NPM
- Path traversal
- Injections: SQL, NoSQL, Blind, JavaScript
- Sandbox escaping (vm)
- Buffer vulnerabilities
- Regular expressions

Path traversal


```
const serveFile = fileName => {
  const filePath = path.join(STATIC_PATH, fileName);
  return fs.createReadStream(filePath);
};

http.createServer((req, res) => {
  const url = decodeURI(req.url);
  serveFile(url).pipe(res);
}).listen(8000);

curl -v http://127.0.0.1:8000/%2e%2e/1-traversal.js
```

Path traversal fixed


```
const serveFile = fileName => {
  const filePath = path.join(STATIC_PATH, fileName);
  if (!filePath.startsWith(STATIC_PATH)) {
    throw new Error(`Access denied: ${name}`);
  }
  return fs.createReadStream(filePath);
};

http.createServer((req, res) => {
  const url = decodeURI(req.url);
  serveFile(url).pipe(res);
}).listen(8000);
```

Asynchronity issues

- Callback hell and promise hell
- Ignoring errors in callbacks
- Ignoring errors in promises
- Throwing errors and releasing resources
- Race conditions and deadlocks
- Queuing theory: need to limit concurrency

Callback hell

```
select(id, (err, data) => {
  check(data, (err, valid) => {
    convert(data, (err, res) => {
      cache(id, res, err => {
        send(data);
      });
    });
  });
});
```

Flat, decomposed, named

```
const saved = (err, data) =>
  err ? throw err : send(data1);
const converted = (err, res) =>
  err ? throw err : cache(res, saved);
const checked = (err, valid) =>
  err ? throw err : convert(data, converted);
const selected = (err, data) =>
  err ? throw err : check(data, checked);
select(id, selected);
```

Promises sequential execution


```
Promise.resolve(id)
  .then(select)
  .catch(...)
  .then(check)
  .catch(...)
  .then(convert)
  .catch(...)
  .then(cache)
  .catch(...)
  .then(send)
  .catch(...);
```

Promise hell

```
select(id).then(data => {
  check(data).then(valid => {
    if (valid) {
      Promise.all([
        convert(data),
        cache(data)
      ]).then(send);
    }
  });
});
```

Compose errback


```
compose
  (send, cache, convert, check, select)
  (id);

pipe
  (select, check, convert, cache, send)
  (id);
```

Compose AsyncFunction


```
await compose
  (send, cache, convert, check, select)
  (id);

await pipe
  (select, check, convert, cache, send)
  (id);
```

Universal composition: metasync

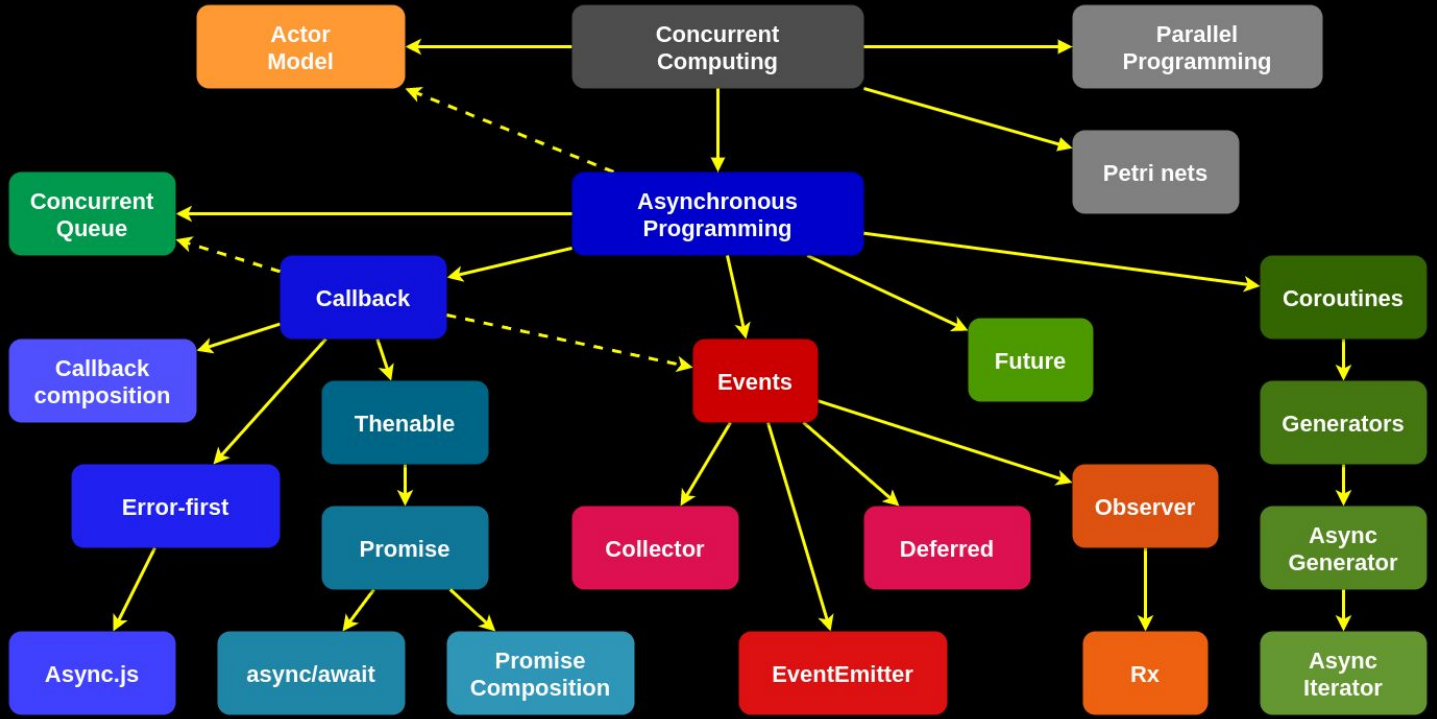

```
  metasync
    ([select, check, [[convert, cache]], send])
    (id);


// In arguments: Function | Errback | AsyncFunction
// Process: Parallel | Sequential
// Functionality: Cancelable | Timeout | Throttle
// Result: Thenable | Promise | Errback | EventEmitter
```

Limit execution concurrency


```
const queue = metasync.queue(3)
  .wait(2000)
  .timeout(5000)
  .throttle(100, 1000)
  .process((item, cb) => cb(err, result))
  .success(item => {})
  .failure(item => {})
  .done(() => {})
  .drain(() => {});
```

Asynchronous programming



27 lectures: https://habr.com/ru/post/452974/

Race condition demo


```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }

  async move(dx, dy) {
    this.x = await add(this.x, dx);
    this.y = await add(this.y, dy);
  }
}
```

Race condition demo


```
const random = (min, max) => Math
  .floor(Math.random() * (max - min + 1)) + min;

const add = (x, dx) => new Promise(resolve => {
  const timeout = random(20, 100);
  setTimeout(() => resolve(x + dx), timeout);
});
```

Race condition demo


```
const p1 = new Point(10, 10);
console.log(p1);

p1.move(5, 5);
p1.move(6, 6);
p1.move(7, 7);
p1.move(8, 8);

setTimeout(() => {
  console.log(p1);
}, 1000);
```

Race condition demo


```
Initial
Point { x: 10, y: 10 }

Expected
Point { x: 36, y: 36 }

Actual
Point { x: 18, y: 25 }
```

Do we need parallel primitives?

- Semaphore: Binary and Counting semaphore
- Condition variable
- Spinlock
- Mutex, Timed mutex, Shared mutex
- Recursive mutex
- Monitor
- Barrier

Do we need parallel primitives?

https://github.com
/HowProgrammingWorks
/Semaphore
/Mutex
/RaceCondition
/Deadlock
https://youtu.be/JNLrITevhRI

Simple Resource Locking

```
class Lock {
  constructor() {
    this.active = false;
    this.queue = [];
  }

  leave() {
    if (!this.active) return;
    this.active = false;
    const next = this
      .queue.pop();
    if (next) next();
  }
}
```

```
enter() {
  return new Promise(resolve => {
    const start = () => {
      this.active = true;
      resolve();
    };
    if (!this.active) {
      start();
      return;
    }
    this.queue.push(start);
  });
}
```

Race condition demo


```
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
    this.lock = new Lock();
  }
  async move(dx, dy) {
    await this.lock.enter();
    this.x = await add(this.x, dx);
    this.y = await add(this.y, dy);
    this.lock.leave();
  }
}
```

Web Locks API

```
locks.request('resource', opt, async lock => {
  if (lock) {
    // critical section for `resource`
    // will be released after return
  }
});
```

https://wicg.github.io/web-locks/

Web Locks: await

```
(async () => {

  await something();
  await locks.request('resource', async lock => {
    // critical section for `resource`
  });
  await somethingElse();

})();
```

Web Locks: Promise and Thenable


```
locks.request('resource', lock => new Promise(
  (resolve, reject) => {
    // you can store or pass
    // resolve and reject here
  }
));

locks.request('resource', lock => ({
  then((resolve, reject) => {
    // critical section for `resource`
    // you can call resolve and reject here
  })
}));
```

Web Locks: Abort


```
const controller = new AbortController();
setTimeout(() => controller.abort(), 200);

const { signal } = controller;

locks.request('resource', { signal }, async lock => {
  // lock is held
}).catch(err => {
  // err is AbortError
});
```

Web Locks for Node.js

github.com/nodejs/node/issues/22702
Open

github.com/nodejs/node/pull/22719
Closed

Don’t use blocking operations

- Sync calls like fs.readFileSync
- Console output like console.log
- Remember that require is synchronous
- Long loops (including for..of and for await)
- Serialization: JSON.parse, JSON.stringify
- Iteration: loops, Array.prototype.map, etc.
- CPU-intensive: zlib, crypto

Loop: for await of is blocking


```
(async () => {
  let ticks = 0;
  const timer = setInterval(() => ticks++, 10);
  const numbers = new Array(1000000).fill(1);
  let i = 0;
  for await (const number of numbers) i++;
  clearInterval(timer);
  console.dir({ i, ticks });
})();

// { i: 1000, ticks: 0 }
```

AsyncArray (short version)


```
class AsyncArray extends Array {
  [Symbol.asyncIterator]() {
    let i = 0;
    return {
      next: () => new Promise(resolve => {
        setTimeout(() => resolve({
          value: this[i], done: i++ === this.length
        }), 0);
      })
    };
  }
}  // github.com/HowProgrammingWorks/NonBlocking
```

Loop: for await of + AsyncArray


```
(async () => {
  let ticks = 0;
  const timer = setInterval(() => ticks++, 10);
  const numbers = new AsyncArray(10000000).fill(1);
  let i = 0;
  for await (const number of numbers) i++;
  clearInterval(timer);
  console.dir({ i, ticks });
})();

// { i: 10000, ticks: 1163 }
https://github.com/HowProgrammingWorks/NonBlocking
```

Memory leaks

- References
  - Global variables
  - Mixins to built-in Classes
  - Singletons, Caches
- Closures / Function contexts
  - Recursive closures
  - Require in the middle of code
  - Functions in loops

Memory leaks

- OS and Language Objects
  - Descriptors: files, sockets...
  - Timers: setTimeout, setInterval
- Events / Subscription / Promises
  - EventEmitter
  - Callbacks, Not resolved promises

github.com/HowProgrammingWorks/MemoryLeaks

ORM is an obvious antipattern

- ORM tool for DBMS performance degradation
- SQL is much more easier
- ORM hides DBMS functionality
- Additional security vulnerabilities
- Broken OOP principles

github.com/HowProgrammingWorks/Databases

Error handling antipatterns

JS FEST

- Ignoring callback errors
- Callback returns multiple times
- Unhandled stream errors
- Unhandled rejection
- Promise resolves multiple times

github.com/HowProgrammingWorks/PromiseError

Don’t ignore callback errors


```
const cbFunc = (arg1, arg2, arg3, callback) => {
  readData(arg1, arg2, (error1, data1) => {
    const arg4 = data1.field;
    checkData(arg3, arg4, (error2, data2) => {
      if (error2) callback(new Error('msg'));
      callback(null, { data1, data2 });
    });
  });
};
```

Don’t ignore stream errors


```
const sharp = require('sharp');

http.get(url, src => {
  const dest = fs.createWriteStream(fileName);
  const resize = sharp().resize(300, 300);
  const free = () => src.destroyed || src.destroy();
  resize.on('error', free);
  destination.on('error', free);
  src.pipe(resize).pipe(dest);
});
```

Use FP style, like Pump


```
const pump = require('pump');
const sharp = require('sharp');

http.get(url, src => {
  const dest = fs.createWriteStream(fileName);
  const resize = sharp().resize(300, 300);
  const fail = err => throw err;
  pump(src, resize, dest, fail);
});
```

Thanks! Questions?

https://github.com/tshemsedinov
https://www.youtube.com/TimurShemsedinov
timur.shemsedinov@gmail.com