Node.js in 2020
Patterns and Antipatterns

github.com/HowProgrammingWorks

Timur Shemsedinov

Chief Technology Architect at Metarhia
Lecturer at Kiev Polytechnic Institute

github.com/tshemsedinov

vm Sandboxing

vm.Script, vm.runInContext and v8::Context

vm.createContext

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

const sandbox = {
  console: new Logger(),
  require: wrap(require),
  application: new Application(),
};

sandbox.global = sandbox;
vm.createContext(sandbox);
```

vm.Script

```
const vm = require('vm');
const fs = require('fs');

const code = await fs.readFile(fileName, 'utf8');
const src = `'use strict';\n context => ${code}`;

const script = new vm.Script(src);
script.runInContext(sandbox, { timeout: 5000 });
```

vm.Script

```
const options = {
  timeout: 5000,
  displayErrors: false, // default: true
  breakOnSigint: true,  // default: false, CTRL+C
};

script.runInContext(sandbox, options);
const object = { /* global */ };
script.runInNewContext(object, options);
script.runInThisContext(options);
```

v8 Serialization API

v8.serialize, v8.deserialize

v8.Serializer, v8.Deserializer

V8 serialization API

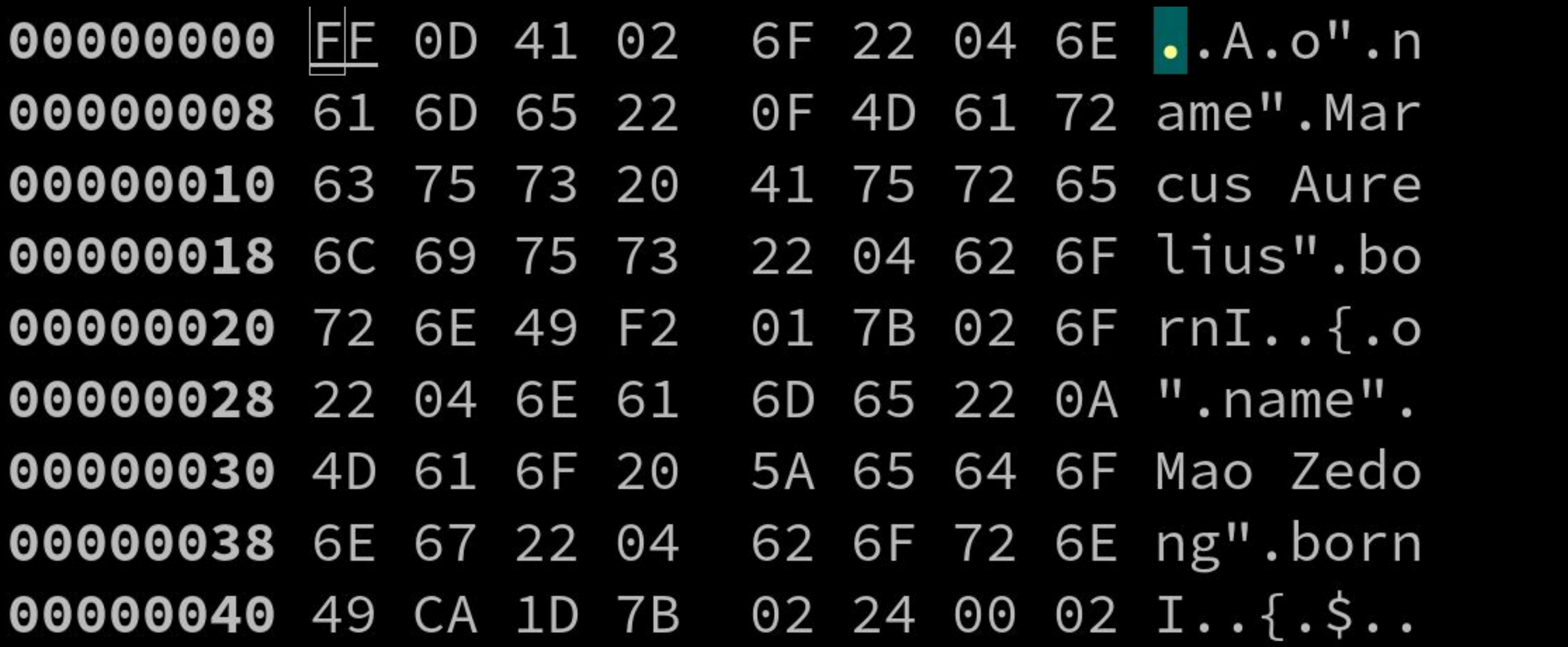
```
const v8 = require('v8');

const dataset = [
  { name: 'Marcus Aurelius', born: 121 },
  { name: 'Mao Zedong', born: 1893 },
];

const v8Data = v8.serialize(dataset);

const obj = v8.deserialize(v8Data);
```

V8 Serialization file format



Limit concurrency

to avoid resource starvation in high-intensive servers

How to limit concurrency?

- Counter variable
- Array, Linked list
- Asynchronous Queue
- Counting Semaphore
- Event Stream
- External balancer + monitoring
- It works somehow

Asynchronous Queue Usage

```
const q1 = metasync.queue(3).priority()
  .process((item, cb) => {});

const q2 = metasync.queue(1).wait(100).timeout(200)
  .process((item, cb) => {});

q1.pipe(q2);

q1.add({ id: 1 }, 0);
q1.add({ id: 3 }, 1);
```

Counting Semaphore

```
const semaphore = new CountingSemaphore(concurrency);

const handler = async (req, res) => {
  await semaphore.enter();
  ...
  semaphore.leave();
};
```

Reference implementations

Asynchronous Concurrent Queue

https://github.com/metarhia/metasync/blob/master/lib/queue.js

Counting Semaphore

https://github.com/HowProgrammingWorks/NodejsStarterKit/blob/master/lib/semaphore.js

Graceful shutdown

after fatal errors and for reload applications

When we need to exit?

- On fatal errors, unhandled exceptions
- Update infrastructure or platform
- Scheduled restart, release leaks, etc.
- Manual stop or restart, maintenance

How to shutdown graceful?

- Close server network ports
- Notify all clients with special events
- Wait for timeout
- Close all connections with socket.destroy()
- Save all data and release critical resources
- exit 0

fs.watch

fs.watchFile, fs.FSWatcher and live code reload

Watch

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

fs.watch(dirPath, (event, fileName) => {
  const filePath = path.join(dirPath, fileName);
  reloadFile(filePath);
});
```

Tricks

- Wait for 3-5 sec. timeout after fs.watch event
- Put all changes to collection to reload
- Ignore temporary and unneeded files

Antipatterns

real and imaginary
with solutions and without

What are the developers worried about?

- What is the best framework?
- How to organize folders structure in project?
- Which DBMS and ORM / ODM to use?
- How to scale my server?
- What is faster (e.g. callbacks or promises)?
- Where to put something to model or controller?
- How to separate project into microservices?

Answers

- Don’t think about framework, think about layers
- Data structures are more important than folders
- You can take any DBMS if it's PostgreSQL
- Do you really need to scale?
- Callbacks vs promises affect <1% of speed
- Forget about MVC, learn SOLOD, GRASP, GoF
- Design interfaces, not microservices

Think about...

- Proper error handling
- Modeling and software abstractions
- Coupling and Cohesion, Law of Demeter
- GRASP principle “Information Expert”
- Request queues and object pools
- How to minimize I/O and CPU usage with RAM
- Request context isolation

Layers

At least three:

- Data access layer
- Network layer
- Domain model (business-logic) layer

GRASP, SOLID, GoF

How can this be related to JavaScript?

GRASP

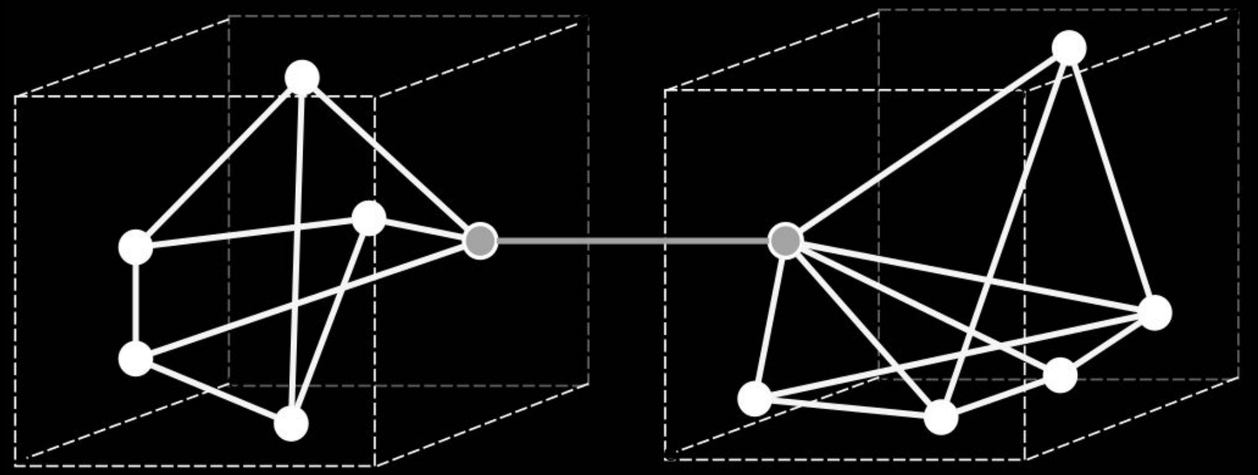
General responsibility
assignment software patterns

From book “Applying UML and Patterns:
An Introduction to Object-Oriented Analysis &
Design” // Craig Larman

GRASP: Coupling and cohesion

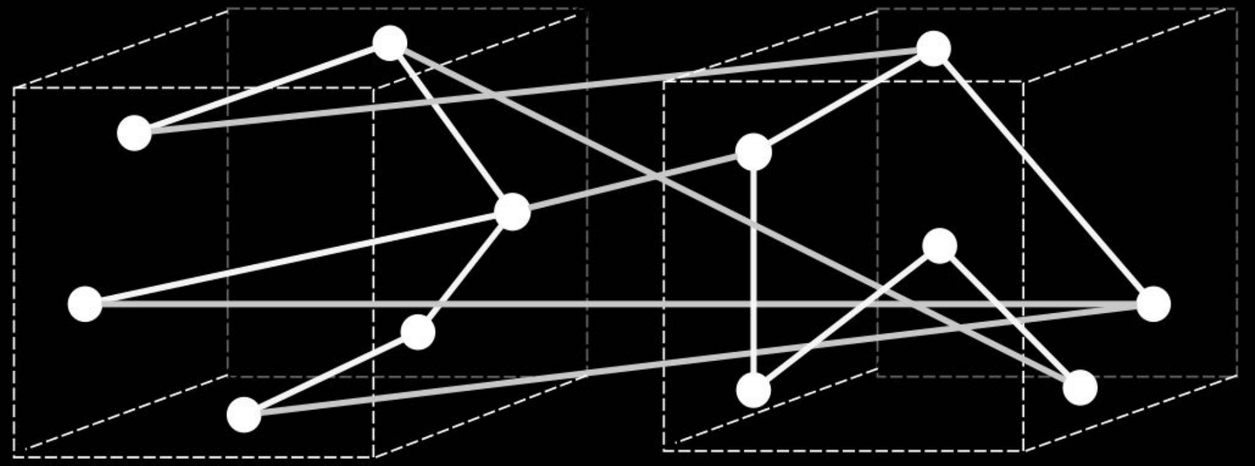
Cohesion

Inside module or program component



Coupling

Between modules or program components



GRASP:

General responsibility
assignment software patterns

Low Coupling
Information Expert
Controller
Pure Fabrication
Protected Variations

High Cohesion
Creator
Polymorphism
Indirection

SOLID: purpose

Michael Feathers
Robert Martin (aka Uncle Bob)

What SOLID give us:

- Facilitate modification
- Improved code ownership and TTM
- We can quickly understand each other

SOLID: 5 principles

- The Single Responsibility Principle
- The Open Closed Principle
- The Liskov Substitution Principle
- The Interface Segregation Principle
- The Dependency Inversion Principle
  (do not confuse with dependency injection
  and inversion of control)

GoF Patterns

Gang of Four (GoF): Эрих Гамм, Ричард Хелм, Ральф Джонсон, Джон Влиссидес

Design Patterns — Elements of Reusable Object-Oriented Software (23 шаблона)

GoF patterns classification

Creational: Abstract factory, Builder, Singleton, Factory method, Object pool, ...
Structural: Adapter, Bridge, Composite, Facade, Decorator (wrapper), Proxy...
Behavioral: Chain of Responsibility, Command, Observer, Iterator, Strategy...
Communicational: CQS, CQRS, Event sourcing...

Node.js

Patterns and Antipatterns

Node.js antipatterns classification

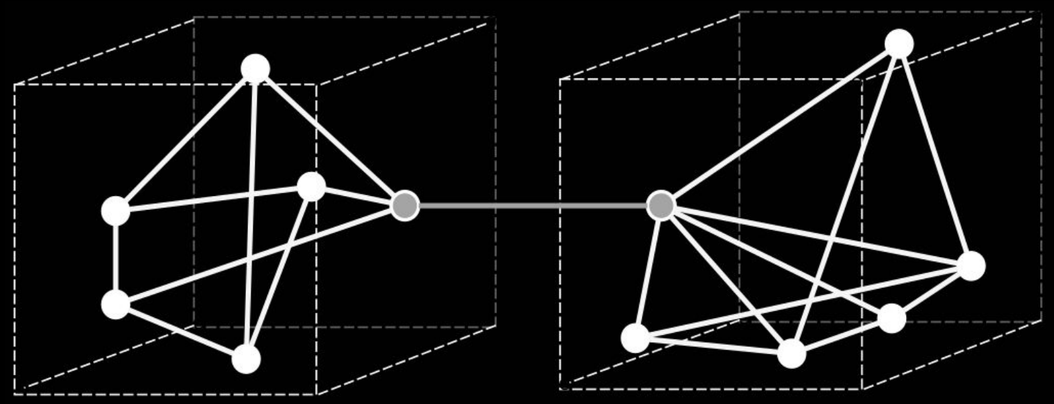
- Structure and arch.
- Initialization
- Dependency issues
- Application state
- Middlewares
- Context isolation
- Security issues
- Asynchronity issues
- Blocking operations
- Memory leaks
- Databases and ORM
- Error handling

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.

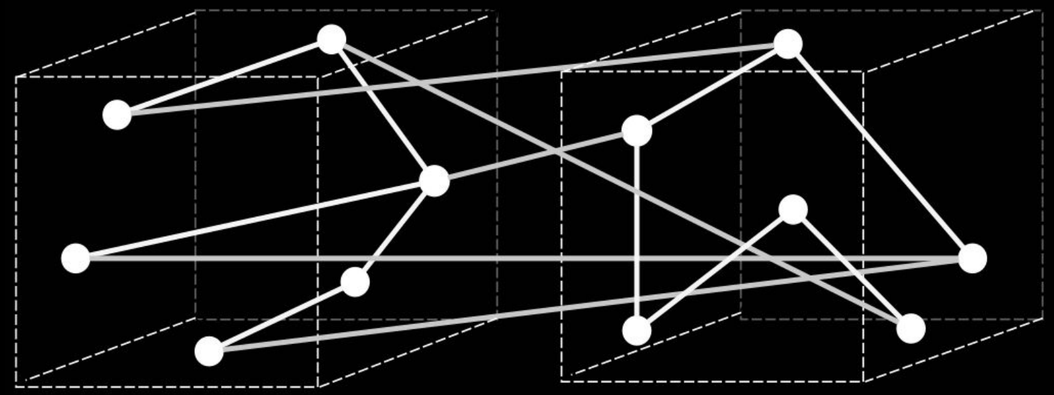
Middlewares

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



Middlewares changes:

- Socket state
- Db connection state
- Server state



Don’t create global state

```
let groupName;

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

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

Don’t mixin to req, res, ctx

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

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

Special place for the state: res.locals

```
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 mixin 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 from 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) => {
  });
});
```

Connection leaks is easy

```
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
  });
});
```

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: 1000000, 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(1000000).fill(1);
  let i = 0;
  for await (const number of numbers) i++;
  clearInterval(timer);
  console.dir({ i, ticks });
})();

// { i: 1000000, 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

Links

github.com/HowProgrammingWorks/
AbstractionLayers

github.com/HowProgrammingWorks/
MemoryLeaks

Error ignoring

with callbacks and Promises, EventEmitter, unhandled exceptions

Callback-last, error-first

```
const fn = (arg1, arg2, callback) => {
  if (...) callback(null, result);
  else callback(new Error('message'));
};

fn(arg1, arg2, (err, result) => {
  if (err) {
    console.log(err.message);
    return;
  }
  console.log(result);
});
```

Error ignoring

```
fn1(arg1, arg2, (err, res1) => {
  fn2(res1, arg3, (err, res2) => {
    fn3(res2, arg4, arg5, (err, res3) => {
      doSomething(arg5, res3);
    });
  });
});
```

Error ignoring

```
const cb3 = (err, res3) => {
  doSomething(arg6, res3);
};
const cb2 = (err, res2) => {
  fn3(res2, arg4, arg5, cb3);
};
const cb1 = (err, res1) => {
  fn2(res1, arg3, cb2);
};
fn1(arg1, arg2, cb1);
```

Promise.catch

```
doSomething(arg1, arg2)
  .then(
    data => { console.log({ data }); },
    err => { console.log({ err }); }
  );

doSomething(arg1, arg2)
  .then(data => { console.log({ data }); })
  .catch(err => { console.log({ err }); });
```

Promise.catch

```
doSomething(arg1, arg2)
  .then(
    data => { console.log({ data }); },
    err => { console.log({ err }); }
  );

doSomething(arg1, arg2)
  .then(data => { console.log({ data }); })
  .catch(err => { console.log({ err }); });
```

.then(fulfilled, rejected).catch(rejected)

```
doSomething(arg1, arg2)
  .then(
    data => { console.log({ data }); },
    err => { console.log({ err }); }
  )
  .catch(err => { console.log({ err }); });

// Помним про .finally()
```

Error ignoring

```
(async () => {

  const file1 = await readFile('file1.ext');
  const file2 = await readFile('file2.ext');
  console.dir({ file1, file2 });

})();
```

Error ignoring

```
(async () => {

  try {
    const file1 = await readFile('file1.ext');
    const file2 = await readFile('file2.ext');
  } catch (err) {
    console.error(err);
  }

})();
```

Catch unhandled errors

```
process.on('uncaughtException', err => {
  console.log('on uncaughtException: ' + err.message);
  process.exit(1);
});

process.on('warning', warning => {
  console.log({ warning });
});

// UnhandledPromiseRejectionWarning: Error: msg...
```

Catch unhandled errors

```
process.on('rejectionHandled', promise => {
  console.log({ rejectionHandled: { promise } });
});

process.on('multipleResolves', (type, p, reason) => {
  console.log({
    multipleResolves: { type, promise: p, reason }
  });
});
```

Node.js Starter Kit

25 kb core,
minimum dependencies:
pg (1.2 mb) and ws (0.24 mb)

Starter Kit Purpose

- Demonstrate modern node.js features (v14.x)
- Optimize for readability and understanding
- Minimum code size and dependencies
- Give structure and architecture examples
- Show patterns and code cohesion
- Not for production use

Starter Kit Feature List

- Serve API with auto-routing, HTTP(S), WS(S)
- Server code live reload with file system watch
- Graceful shutdown and application reload
- Code isolation, sandboxing and security
- Implemented dependency injection
- Layered architecture: core, domain, API, client

Starter Kit Feature List

- Multi-threading for CPU utilization
- Serve multiple ports in threads
- Serve static files with memory cache
- Request queue with timeout and size
- Execution timeout and error handling

Starter Kit Feature List

- Application configuration
- Simple logger and to terminal and file
- Database access layer (Postgresql)
- Persistent sessions (stored in DB)
- Unit-tests and API tests example

github.com
/HowProgrammingWorks
/NodejsStarterKit

Contacts

github.com/tshemsedinov
youtube.com/TimurShemsedinov
github.com/HowProgrammingWorks
patreon.com/tshemsedinov
t.me/HowProgrammingWorks
t.me/NodeUA

timur.shemsedinov@gmail.com