ASYNC PROGRAMMING

AGENDA

1. NodeJS event-driven non-blocking IO model
2. Evolution of async programming in JavaScript
3. Testing async code

NODEJS EVENT-DRIVEN NON-BLOCKING IO MODEL

JavaScript runtime is **SINGLE THREADED**

Long-running operations **MUST NOT BLOCK**

- Network request
- Database query
- Filesystem operations

CALLBACK

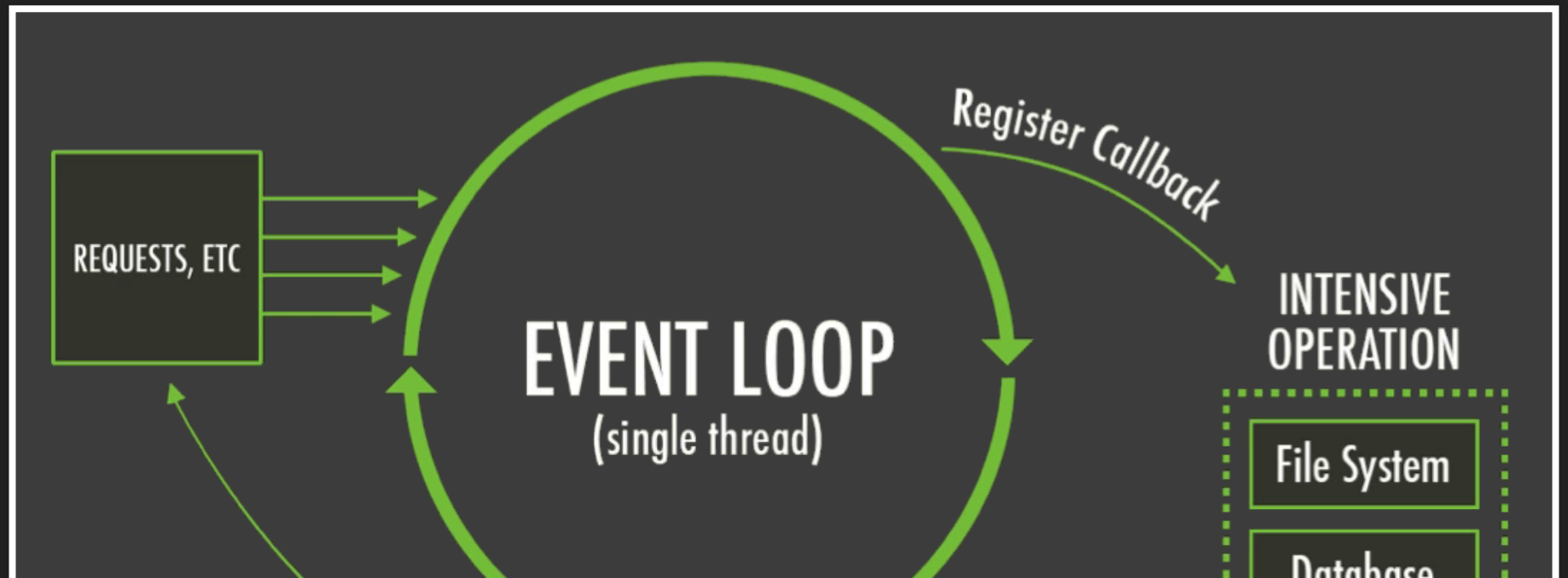
Cannot wait for result

NOTIFICATION OF COMPLETION EVENT needed

Callback

```
const fs = require('fs');
fs.readFile('myFile.txt', function(err, contents) {
  // continue working with file contents
});
```

EVENT LOOP


REQUESTS, ETC
EVENT LOOP
(single thread)
Register Callback
INTENSIVE
OPERATION
File System

OTHER PLATFORMS

New thread for every new request

Threads wait/block for IO

GOOD FOR

I/O intensive tasks

Waiter metaphore

BAD FOR

CPU intensive tasks

TESTING ASYNC

CONTENTS

ASYNC OVERVIEW

WHY ASYNC?

JS CONCURRENCY MODEL

- single threaded
- event based

Single threaded -> non-blocking I/O model

Every long-running operation is asynchronous

- network request
- filesystem operation
- IndexedDB

NodeJS is great for I/O heavy operations

I/O has to be non-blocking, otherwise browser and UI would freeze

NodeJS can respond to multiple requests resource efficiently with one thread

Caller is notified of completion

But... how?

CALLBACKS

A function that is called upon completion.

```
fs.writeFile(path, data, function(error) {
  // ...
});
```

GUARANTEES?

- Will it be called?
- Only once?

CONTROL FLOW PATTERNS?

- Loops
- Parallel
- Conditions

CALLBACK PYRAMID/HELL

```
fs.readFile('file1.txt', function(error, content) {
  let otherPath = content.split(';')[2];
  fs.readFile(otherPath, function(error, userIds) {
    let firstId = userIds.split(';')[0];
    db.fetch(firstId, function(error, userData) {
      // ...
    });
  });
});
```

Error handling is left out in the example

CALLBACK PYRAMID/HELL

- Hard to refactor
- Error handling cannot be separated from bussiness logic

ERROR HANDLING?

NodeJS convention

```
callback(error, result);

fs.readFile('file.txt', function(error, result) {
  if (error) {
    // do some error handling
    // maybe call a callback given to us
    return; // VERY IMPORTANT
  }
  // success case
});
```

RETURN!!! otherwise callback would be called multiple times

LIB: ASYNC

ASYNC

Collection and control flow patterns

- map, filter, reduce, ...
- series, waterfall, parallel, ...

ASYNC

```
async.filter(
  ['file1.txt', 'file2.dat', 'file3.mp4'],
  fs.exists,
  function(results) {
    // ...
  }
);
```

Still, just a helper for some use cases, it still has all the problems of callbacks.

PROMISES

PROMISES

Represents the outcome of an async operation

```
let promise = new Promise(function(resolve, reject) {
  // ... some async operation ...
  // notify the outer world that it has finished
  resolve(value); // success
  reject(reason); // error case
});

promise.then(function(value) { console.log(value); });
promise.catch(function(reason) { console.log(reason); });
```

PROMISES

States

- Pending
- Fulfilled = success
- Rejected = error

Adding handlers later to a finished promise --> handler is called

CHAINING

```
readFile('file1.txt')
// return value in handler
.then(function(content) { return content.split(';')[2]; })
// return a promise
.then(function(otherPath) { return readFile(otherPart); })
.then(function(content) { userIds.split(';')[0]; })
.then(function(firstId) { return db.fetch(firstId); })
.then(function(userData) { doSomeWork(); })
// every error propagates here
.catch(function(error) { handleError(error); });
```

Handlers

- return value
- return promise
- throw an error

Benefits

- composable scheme, composition always wins
- return value handled intelligently
- error handling can be separated from logic
- we have throw and catch
- errors propagate through the chain

MULTIPLE PROMISES

```
Promise.all([p0, p1, p2]).then(function(values) {
  // values[0] -> p0
  // values[1] -> p1
});

Promise.race([p0, p1, p2]).then(function(value) {
  // value -> first promise to resolve
});
```

LIBRARIES

- Bluebird
- Q
- bunch of others

Utility methods, progress, cancellation, etc.

DRAWBACKS

Not intuitive, learning curve

- pitfalls
- anti-patterns

MUST READ: we have a problem with promises

E.g. nesting .then handlers --> callback hell

The JS community learned and standardized promises the hard way.

GENERATORS + PROMISES

GENERATORS

Suspend execution of a function and continue later from the point of suspension.

GENERATORS

```
let genFunc = function* (x) {
  console.log(x);

  let y = yield 2*x;

  console.log(y);
  return 2*y;
};

let gen = genFunc(5); // NO OUTPUT, return: generator
gen.next(); // out: 5, return: { value: 10, done: false }
gen.next(13); // out: 13, return: { value: 26, done: true }
```

Two-way communication with yield and next()

GENERATORS

```
let genFunc = function* () {
  try { yield; } catch(error) { console.log(error); }
  yield 5;
};

let gen = genFunc();
console.log(gen.next());
// {value: undefined, done: false }
console.log(gen.throw(new Error('from outside')));
// [Error: from outside]
// { value: 5, done: false }
console.log(gen.next());
// { value: undefined, done: true }
```

Errors can be triggered from outside

GOING ASYNC

So far, so sync

What if we yield promises?

```
let genFunc = function* () {
  let content = yield readFile('file1.txt');
  let id = content.split(';')[2];
  let userData = yield db.fetch(id);
  return userData.name;
};
let gen = genFunc();
gen.next().value
  .then(content => gen.next(content).value,
        error => gen.throw(error).value)
  .then(userData => gen.next(userData).value)
  .then(name => console.log(name));
```

RUNNER

Libraries, helpers, samples for the general generator + promise runner

- co
- Q.spawn

Write your own!

Using then and gen.next can be generalized

There are libraries for that job, but for fun and education try to write your own. It's not hard, but if you succeed you understand the topic.

```
co(function* () {
  let content = yield readFile('file1.txt');
  let id = content.split(';')[2];

  try {
    let userData = yield db.fetch(id);
  } catch (error) {
    console.log(error);
  }

  return userData.name;
}).then(name => console.log(name));
```

ASYNC / AWAIT

ASYNC / AWAIT

Language feature for co + generator + yield

~~ES7 / ES2016~~ stage 3

Did not make it into ES2016, but close to finalized standard, browsers soon will support it.

Can be transpiled with Babel.

ASYNC / AWAIT

```
async function myFunc() {
  let content = await readFile('file1.txt');
  let id = content.split(';')[2];

  try {
    let userData = await db.fetch(id);
  } catch (error) {
    console.log(error);
  }

  return userData.name;
};

myFunc().then(name => console.log(name));
```

ASYNC / AWAIT

Async functions **always return a promise**

Understanding promises is essential for co+generator and async/await

returning promise --> integrates nicely into asyn stack

ASYNC TESTS

CONTEXT

- Mocha
- Chai

MOCHA

Test framework and test runner

- describe, it
- before, after, beforeEach, afterEach hooks
- describe.only, it.skip

CHAI

Assertion library

- assert, **expect**, should styles
- assertion throws error or simply returns
- plugins

Viktor's plugin: chai public equal

DONE

DONE

Callback given to test function to signal end of async test

```
it('should fetch the user name', function(done) {
  getUsername(id, function(error, result) {
    if (error) {
      done(error);
      return;
    }
    expect(result).to.eql('Joe');
    done();
  });
});
```

PROBLEMS WITH DONE

- forget to call done after expect
- callback hell
- careful error propagation needed
- timeout instead of error if tested API uses promises

Testing multiple steps can become callback hell

PROMISE

PROMISE

Assert in *then* or *catch* and return the promise

```
it('should fetch the user name', function() {
  return getUsername(id).then(function(result) {
    expect(result).to.eql('Joe');
  });
});
```

- errors: rejected promise
- shorter test

CHAI AS PROMISED

Chai plugin for assertions with promises

```
let chai = require('chai');
let chaiAsPromised = require('chai-as-promised');
chai.use(chaiAsPromised);

// ...

it('should fetch the user name', function() {
  return expect(getUsername(id)).to.eventually.eql('Joe');
  return expect(getUsername(id)).to.become('Joe');
  // error testing
  return expect(getUsername(id)).to.be.rejectedWith('some error');
}):
```

PROMISES NEVER DONE

Never use promises with done

```
it('should fetch the user name', function(done) {
  getUsername(id).then(function(result) {
    expect(result).to.eql('Joe');
    done();
  }, function(error) { done(error); });
  // OR WITH CHAI AS PROMISED
  expect(getUsername(id)).to.eventually.eql('Joe').and.notify(done);
}):
```

Why not: not needed, mocha accepts promises as return values

GENERATORS

GENERATOR TEST FUNCTIONS

co-mocha package

```
require('co-mocha');
it('should fetch the user name', function* () {
  let userName = yield getUsername(id);
  expect(userName).to.eql('Joe');
});
```

ASYNC / AWAIT

ASYNC TEST FUNCTIONS

```
it('should fetch the user name', async function() {
  let userName = await getUsername(id);
  expect(userName).to.eql('Joe');
}):
```

```
npm install babel-register babel-polyfill

mocha --compilers js:babel-register \
      --require babel-polyfill testfile.js
```

ADDITIONAL INFO

PROMISIFY

From callback API to promise API...

- Wrap manually
- Use library

MANUAL WRAPPING

```
let fs = require('fs');
function readFile(path) {
  return new Promise(function(resolve, reject) {
    fs.readFile(path, function(error, content) {
      if (error) {
        reject(error);
        return;
      }
      resolve(content.toString('utf8'));
    });
  });
}
```

USING A LIBRARY

```
let Promise = require('bluebird');
let readFile = Promise.promisify(fs.readFile.bind(fs));

Promise.promisifyAll(fs); // --> fs.readFileAsync(path);

function promiseReadFile(path) {
  return Promise.fromCallback(function(callback) {
    fs.readFile(path, callback);
  });
}

somePromise.asCallback(callback); // back to callback api
somePromise.nodeify(callback); // back to callback api
```

TESTING ANGULAR1 $HTTP

- with *angular-mocks* $http behaves *synchronously*
- returning its promise **DOES NOT WORK**
- Conscious decision of angular team

WRONG

```
it('should fetch the data from the backend', function() {
  // ...mock backend setup...
  let service = new MyService($http);
  let promise = service.load().then(() => {
    expect(service.items).to.eql(['a', 'b', 'c']);
  });
  $httpBackend.flush();
  return promise;
});
```

RIGHT

```
it('should fetch the data from the backend', function() {
  // ...mock backend setup...
  let service = new MyService($http);
  service.load();
  $httpBackend.flush();
  expect(service.items).to.eql(['a', 'b', 'c']);
});
```

OTHER RESOURCES

- Promise/A+ spec + implementations list
- HTML5 rocks on promises
- Promises website by Forbes Lindesay
- Comparison of promise libraries
- Promise visualization tool
- ES6 generators
- Understanding JavaScript async and await
- Testing with ES6, mocha, and babel
- Babel JS