Intro to Node.js

Marco Fabbri
marco.fabbri28@unibo.it

Dipartimento di Informatica: Scienza e Ingegneria (DISI)
ALMA MATER STUDIORUM—Università di Bologna

DS 2013/2014
Distributed Systems – 10th of April 2014
OUTLINE

1. What is Node?
2. Getting Started
3. Event-Loop and Callbacks
4. Event-Emitter Pattern
5. Streams
6. Modules
DISCLAIMER

All the material presented in these slides has been built by the author upon a collection of documents kindly made available by the Node.js community (mainly nodejs.org and nodetuts.com).

Then, credits for all the stuff (text & images) goes to the Node.js community.

Credits for all the mistakes goes to the author.
OUTLINE

1. What is Node?
2. Getting Started
3. Event-Loop and Callbacks
4. Event-Emitter Pattern
5. Streams
6. Modules
What is Node?

Node is a platform built on **Chrome's JavaScript runtime (V8)** for *easily building fast, scalable network applications*.

Node uses an **event-driven, non-blocking I/O** model that makes it lightweight and efficient, perfect for *data-intensive real-time applications* that run across distributed devices.
What is Node? (2)

```javascript
var http = require('http');
http.createServer(function (req, res) {
    res.writeHead(200, {'Content-Type': 'text/plain'});
    res.end('Hello World
');
}).listen(1337, "127.0.0.1");

console.log('Server running at http://127.0.0.1:1337/');
```
What is Node? (3)

In the previous “hello world” web server example many client connections can be handled concurrently.

Node tells the operating system (through epoll, kqueue, /dev/poll, or select) that it should be notified when a new connection is made, and then it goes to sleep. If someone new connects, then it executes the callback. Each connection is only a small heap allocation.

[http://nodejs.org/about/]
What is Node? (4)

...Node brings browser-like event-oriented Javascript to the server.

*Node simply enters the event loop after executing the input script. Node exits the event loop when there are no more callbacks to perform.* This behavior is like browser javascript—the event loop is hidden from the user.

[http://nodejs.org/about/]
What is Node? (5)

Recap…

• Event-driven, non-blocking I/O model (=> lightweight)

• Event-loop & JavaScript's callbacks (async by default)

• Single-threaded

• Building blocks for scalable network programs (e.g. HTTP streaming parser, sockets API…)

• Runs as command line application

• Open-source, started by Ryan Dahl @ Joyent in 2009

• Much C code “under the hood” (fast)
What could you build?

- Chat server
- Fast file upload client
- Ad server
- Streaming, real-time data applications
What are they building?

Take a look at http://nodejs.org/industry/

and https://github.com/joyent/node/wiki/Projects,-Applications,-and-Companies-Using-Node

...Dow Jones, eBay, PayPal [just to name a few]
OUTLINE

1. What is Node?

2. Getting Started

3. Event-Loop and Callbacks

4. Event-Emitter Pattern

5. Streams

6. Modules
Download from nodejs.org

Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.

Current Version: v0.10.26
REPL

mfabbri@whitewinter ~> node
> console.log('hello world');
Hello world
undefined
>

Script

```javascript
mfabbri@whitewinter ~> cat hellonode.js
console.log('hello world');

mfabbri@whitewinter ~> node hellonode.js
hello world
mfabbri@whitewinter ~> 
```
OUTLINE

1. What is Node?
2. Getting Started
3. Event-Loop and Callbacks
4. Event-Emitter Pattern
5. Streams
6. Modules
“JavaScript has certain characteristics that make it very different than other dynamic languages, namely that it has no concept of threads. Its model of concurrency is completely based around events.”

–Ryan Dahl
A Simple Example…

# helloNode

setTimeout(function() {
    console.log('world!');
}, 2000);

console.log('hello ');
Continued

# helloNode

setInterval(function() {
    console.log('world!');
}, 2000);

console.log('hello ');
The Event-Loop (1)

Asynchronous I/O platforms work by using an event loop. The loop can be simplified to 2 phases:
The Event-Loop (2)

But we can evolve this into three phases:

• Waiting for events on resources

• When an event happens on one resource, discover event handler

• Invoke event handler

• Repeat

The resources here can be files and network sockets, the operating system low-level resource involved in I/O.
The event loop is then sitting idle, listening for events on this list of resources.

Once an event happens, the event loop reacts to it and tries to find an event listener for this event and resource. If it finds one, it executes it.

Once that event handler returns, the event loop restarts the cycle, waiting for the next event.
Another Example...

```javascript
# hellonode-webserver.js

var http = require('http');

var server = http.createServer(function(req, res) {
    res.writeHead(200, { 'content-type': 'text/plain' });
    res.end("Hello World!");
});

server.listen(8080, 'localhost');
```
Continued

# hellonode-webserver.js

```javascript
var http = require('http');

var server = http.createServer(function(req, res) {
    res.writeHead(200, { 'content-type': 'text/plain' });
    res.write("hello\n");
    setTimeout(function() {
        res.end("world\n");
    }, 2000);
});

server.listen(8080, 'localhost');
```

24
...Continued

ab -n 200 -c 200 "http://127.0.0.1:8080/"

Benchmarking 127.0.0.1 (be patient)....

[CUT]

Concurrency Level: 200
Time taken for tests: 2.041 seconds
Complete requests: 200
Failed requests: 0
Write errors: 0

[CUT]
Blocking VS Non-Blocking

Node based around a NON-BLOCKING even-oriented paradigm (asynchronous operations).

• Blocking code
  Read file from Filesystem, set equal to “contents”
  Print contents
  Do something else

• Non-blocking code
  Read file from Filesystem
    whenever you’re complete, print the contents
  Do Something else
Asynchronous by Default

In Event-driven code, when you have to do I/O, *the call stack does not wait until this operation is complete.*

Instead, you define a function that gets invoked once the operation ends. A **callback**.

* passing functions around is possible due to the “*functional nature*” of JavaScript
The Callback Pattern (1)

```javascript
fs.readFile('/etc/passwd', function (err, results) {
  console.log('File contents:', results);
});
```

is the same as

```javascript
var callback = function (err, results) {
  console.log('File contents:', results);
};
fs.readFile('/etc/passwd', callback);
```
The Callback Pattern (2)

Node has a de-facto standard:

The function call that initiates I/O accepts a callback function as last argument and an error as the first argument of the callback if an error occurs.

i.e.:

If an error occurred, the error will be passed in as the first argument, and you should not expect any additional argument to be filled in. You should handle the error — probably informing the user that an error occurred.
The Callback Pattern (3)

E.g.:

```javascript
fs.readFile('/etc/passwd', function (err, results) {
  if (err) return handleError(err);
  console.log('File contents:', results);
});
```
OUTLINE

1. What is Node?

2. Getting Started

3. Event-Loop and Callbacks

4. Event-Emitter Pattern

5. Streams

6. Modules
The Event Emitter Pattern

The Callback pattern is useful when you are issuing granular I/O simple operations either fail or succeed.

But if you have an object that is more complex, like a TCP connection, you may want to perform some actions when the connection is opened, when it is closed, when some data arrives and when there is an error.
Another Example...

```javascript
# echo-server.js
var net = require('net');

var server = net.createServer(function(socket) {
    socket.on('data', function(data) {
        socket.write(data);
    });
});

server.listen(4242, 'localhost');
```
# turns-into-chat-server.js

```javascript
var net = require('net');

var server = net.createServer(function(socket) {
  sockets.push(socket);

  socket.on('data', function(data) {
    sockets
      .filter(function (s) { return s != socket; })
      .forEach(function (s) {
        s.write(socket.remotePort + ': ');
        s.write(data);
      });
  });

  socket.once('end', function() { unsubscribe(socket); });

  socket.once('error', function() { unsubscribe(socket); });
});

server.listen(4242, 'localhost');
```
Events in Node

Many objects in Node emit events: a net.Server emits an event each time a peer connects to it, a fs.readStream emits an event when the file is opened. All objects which emit events are instances of `events.EventEmitter`. You can access this module by doing: `require("events");`;

[…]

Functions can then be attached to objects, to be executed when an event is emitted. These functions are called listeners. Inside a listener function, `this` refers to the EventEmitter that the listener was attached to.

[from http://nodejs.org/api/events.html]
EventEmitter Example

```javascript
var EventEmitter = require('events').EventEmitter;

var logger = new EventEmitter();

logger.on('error', function(message){
    console.log('ERR: ' + message);
});

logger.emit('error', 'Spilled Milk');
logger.emit('error', 'Eggs Cracked');
```
Back to the HTTP Server Example...

```javascript
http.createServer(function(request, response){ ... });
```

is the same as

```javascript
var server = http.createServer();
server.on('request', function(request, response){ ... });
```
Event Type

The event type is represented by a string, usually lower-cased and containing only letters.

Each event type passes arguments to the listeners, and these arguments are specific to the event type — some documentation should be provided about which arguments the listeners for this event type should expect.
Listener

The event listener is just a simple JavaScript function, either declared inline or declared in advance.

Also, you can have more than one event listener for the same event type on the same event emitter, it's just a matter of registering it using the .on() method.

If some events will happen only once, at most once or if you're only interested in the first instance of that event, you should use .once instead of .on like we did on the socket close and end events.

After it is registered, you can remove an event listener from the event emitter by calling .removeListener(<event type>, <event listener>), for this you have to use named functions (declared in advance)
The Special Case of Error

The event type is just a meaningless identifier that is used by the event emitter to dispatch events with the exception of the error event type.

If an event of type error occurs and the event emitter has listeners for it, everything goes as planned.

But if there are no event listeners for when an error event occurs, an uncaught exception is thrown into the event loop.

You can still catch that error if you register for the global uncaughtException event type emitted by the global process object. (Really, this is not advised, you should always listen to the error event)
OUTLINE

1. What is Node?
2. Getting Started
3. Event-Loop and Callbacks
4. Event-Emitter Pattern
5. Streams
6. Modules
What Are Streams?

Wonderful abstractions, very useful for “data plumbing”.

Streams can be *readable, writeable, or both*.

A **read stream** is a source of data: it behaves like a data faucet, *emitting data events* (remember the TCP chat server?) An example of a read stream is a file read stream.

A **write stream** is a stream that you write data to. An example of a write stream is a file write stream.
Read a File (with Some Flow Control)

```javascript
var fs = require('fs');

var readStream = fs.createReadStream('/var/log/system.log');

readStream.on('data', function(data) {
    console.log('got some data:', data);
    readStream.pause();
    setTimeout(function() {
        readStream.resume();
    }, 1000);
});

readStream.on('end', function() {
    console.log('ended');
});
```
Write a File (with Some Flow Control)

```javascript
var fs = require('fs');

var writeStream = fs.createWriteStream('/tmp/write.txt');

var interval = setInterval(function() {
    writeStream.write((new Date()).toString() + '
');
}, 100);

setTimeout(function() {
    clearInterval(interval);
    writeStream.end();
}, 5000);

writeStream.on('drain', function() {
    console.log('drained.');
});
```
Pipe

```javascript
var fs = require('fs');
var file = fs.createReadStream("readme.md");
require filesystem module
var newFile = fs.createWriteStream("readme_copy.md");
file.pipe(newFile);
```

NOTE: Pipe also deals with back-pressure: when the write is not flushed, the read stream is paused until the write stream emits the **drain** event.
Pipe

```javascript
var fs = require('fs');
var file = fs.createReadStream("readme.md");
require filesystem module
var newFile = fs.createWriteStream("readme_copy.md");
file.pipe(newFile);
```

NOTE: Pipe also deals with back-pressure: when the write is not flushed, the read stream is paused until the write stream emits the drain event.
OUTLINE

1. What is Node?
2. Getting Started
3. Event-Loop and Callbacks
4. Event-Emitter Pattern
5. Streams
6. Modules
Modules

Modules are the way in Node that you can package code for reusability.

Node provides several core modules that we have used previously.

You can also use the module system for creating and using your own local modules, and you can also install and use third-party modules.
require()

The require command is the only mechanism provided of getting a reference to a module.

```javascript
var http = require('http');
```

First, it's a synchronous call. Require can do some I/O — like accessing the file system. This is designed like this so that you don't have to use nested callbacks to load all your dependencies.

!!! This means that require is designed to be used before Node starts the event loop and should not be used inside callbacks, with the risk of blocking the event loop.
Creating a Local Module

You can use the Node module system to create your own module. In Node, a module is one file.

#hello.js
var hello = function() {
    console.log("hello!");
}
exports = hello;

hello.js exports a simple function

#goodbye.js
exports.g goodby e = function() {
    console.log("bye!");
}

goodbye.js exports an object with a simple function
Requiring a Local Module

A local module is defined by the path to that module. In this case we're using a relative path, but the require function also accepts an absolute path. The path is always relative to the directory of the current module we are in.

One thing that we can omit is the file termination.

```javascript
#app.js
var hello = require('./custom_hello');
var gb = require('./custom_goodbye');
hello();
gb.goodbye();
```
Where Does require Looks for Modules?

i.e. where are node_modules store?

```javascript
var some_module = require('some_module');

<$my_app>/node_modules/
<$home_dir>/node_modules/
<$node_dir>/node_modules/
```
It’s the Node package manager ([nodejs.org](nodejs.org))

- Comes with Node
- Module Repository
- Dependency Management
- Easily publish modules
- "Local Only"

“Core” is small. “Userland” is large.
Installing a NPM Module

- Locally [i.e. in $my_app]
  
  \> npm install request

- Install modules with executables globally
  
  \> npm install coffee-script -g

- ...and then locally (to “require” them)
  
  \> npm install coffee-script

```javascript
var coffee = require('coffee-script');
```
Defining Your Dependencies

- in $my_app/package.json

```json
{
    "name": "My App",
    "version": "1",
    "dependencies": {
        "connect": "1.8.7"
    }
}
```

> npm install

- Install into the app node_modules directory
- modules dependencies are managed recursively (=> no conflicts)
- version number adopts **semantic versioning** Major.Minor.Patch
  [http://semver.org](http://semver.org)