Node.js

L. Vilímek J.Šmolík

Ackee s.r.o

2016

What is Node.js I

Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world.

- Built on top of Chrome's Javascript runtime.
- Asynchronous and Event driven. All APIs for Node are asynchronous = non-blocking. Node js server never waits for an API to return data, but rather moves to next API call after calling the first, using notification mechanism of Events. More on that later in Event loop.

What is Node.js II

- Single threaded and hihgly scalable. Node uses a single threaded model with event looping. The event mechanism helps to respond in non blocking way as opposed to traditional servers, which create limited threads to handle requests. A single threaded program in Node can provide service to a much larger number of requests compared to same program that runs on traditional servers like Apache HTTP Server.
- No buffering. Node.js application never buff any data. These applications simply output the data in chunks.
- MIT License.

What is Node.js III

Is used in in number of known projects: eBay, General Electric, GoDaddy, Microsoft, PayPal, Uber, Wikipins, Yahoo!, and Yammer, ...

Where to use

- I/O bound Applications
- Data Streaming Applications
- Data Intensive Real time Applications (DIRT)
- JSON APIs based Applications
- Single Page Applications

Where not to use

It is not advisable to use Node.js for CPU intensive applications.

Callbacks reminder

callback = a function that is called at the completion of given task.

Synchronous

```
1 var data = readFileSync(file);
2 //handle data
```

Asynchronous

```
1 readFileAsync(file, function(data) {
2     //handle data
3 }
```

Event loop I

Part of Javascript runtime is an event queue, containing a list of messages and their associated callback functions.

Event loop II

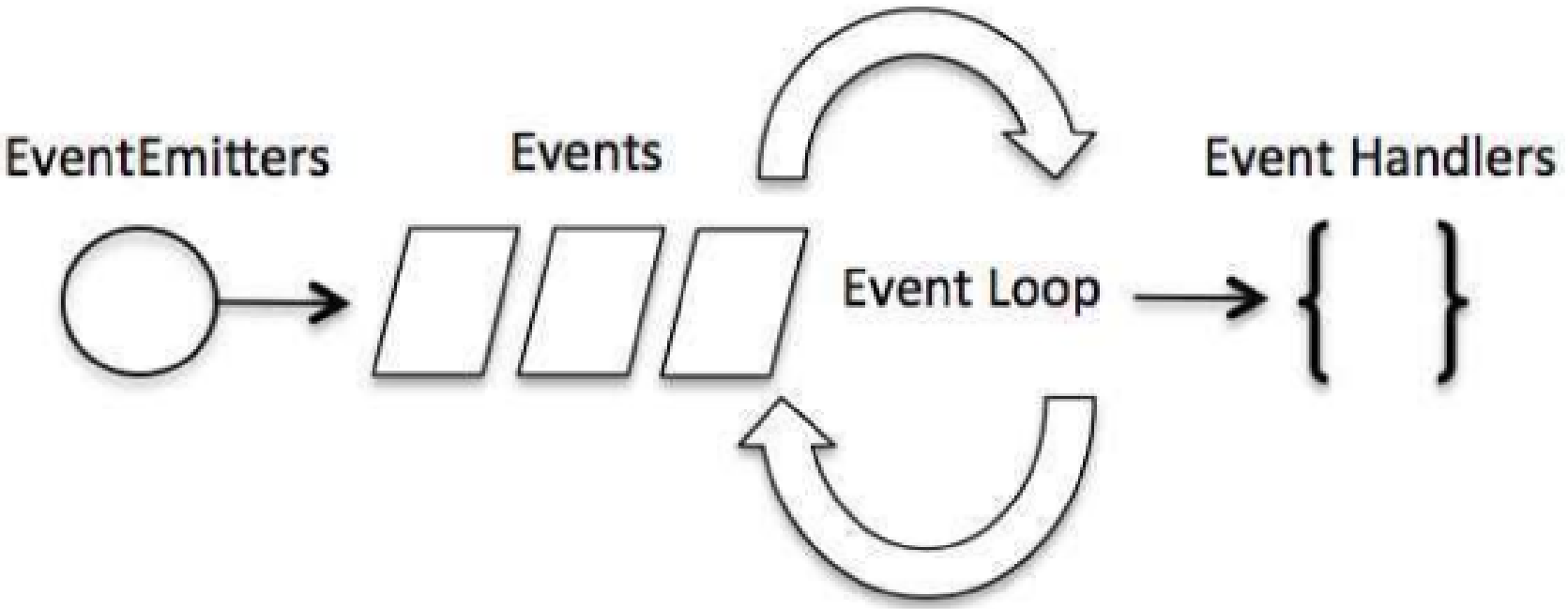

Figure: Event loop

Event loop III

In a loop, the queue is polled for the next message (each poll referred to as a "tick") and when a message is encountered, the callback for that message is executed.

High scalability $\rightarrow$ can process high number of requests without waiting for any function to return result.

Callbacks I

- make CPU intensive tasks in chunks if possible, otherwise entire server is blocked,
- don't forget to call callback when task is finished,
- don't forget to handle errors before passing results to another callback,
- good practice is to create callbacks that take first argument as error from processed task and if neccessary, callback as last.

Task

t1-node-argv Try node.js environment!

Packages I

Node follows the ComonJS module system, using built in require function to include modules, that exist in separate files. `require` reads a javscript file, executes the file and returns the `exports` object.

Files and modules are in one-to-one correspondence

Node comes with NPM package manager

Installing a package

```
1 npm install <Package>
```

Installs Package in one of either

Packages II

- `./node_modules` for local packages
- `path/to/user/data/npm/node_modules` for global packages (if -g option specified)

```
1 npm install
```

Installs dependent packages specified in `package.json`.

- metadata relevant to project

Package.json I

A file that contains a single JSON object wrapping set of package options. Notable are

- name. Name of your application/package.
- version. Application version.
- dependencies. Object defining dependencies of your application. Key: package name, Value: supported version. Example: `"bcryptjs": "2.3.0"`
- devDependencies. Dependencies that are installed when not in production mode.
- scripts. Object defining scripts that can be run with `npm run <Script>`. Key: script name identifier, Value: command to run. Example: `"start": "node server.js"`

Package.json II

```
1  {
2   "name": "Hello world",
3   "version": "1.0.0" ,
4   "scripts": {
5      "start": "node server.js"
6    },
7    "dependencies": {
8      "accept-language": "^2.0.16",
9      "async": "1.3.0",
10     "bcryptjs": "^2.3.0"
11   }
12 }
```

Package.json III

NPM scripts provides interface to the application. For example start is supposed to start the application, `stop` to stop it, `restart`, `postinstall`, `preinstall`.

Module usage and native modules I

Some modules are provided by default through Node API.

- fs. File I/O, wrapper around standard POSIX functions.
- crypto. Cryptographic functionality, wrappers for OpenSSL's hash, HMAC, cipher, decipher..
- http. Interface designed to support features of HTTP protocol.
- and more...

Module usage and native modules II

```
1  var crypto = require('crypto');
2
3  var hash = crypto.createHmac('sha256', 'NotASecret')
4                .update('I love Node.')
5                .digest('hex');
6
7  console.log(hash);
```

Task

t2-http-simple Try to build simple HTTP server in node!

Express

Express is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications.

A simplest, single file express app:

```
1  var express = require('express');
2  var app = express();
3
4  app.get('/', function (req, res) {
5    res.send('Hello World!');
6  });
```

Routing

Express route definition: `app.METHOD(PATH, HANDLER)`

- METHOD is an HTTP request method, i.e. `get`, `put`, `post`, ..., or `all` for any method.
- PATH is a path on the server
- HANDLER function that is executed when the route is matched. By convention, fist argument passed represents HTTP Request, usually named req, second HTTP Response, usually named res.

```
1 app.delete('/user', function (req, res) {
2   res.send('Got a DELETE request at /user');
3 });
```

Request

Representation of current HTTP request.

Notable properites

- app. Reference to Express app.
- baseUrl. URL path on which a router was mounted.
- body. Key-value pairs of data submitted in request body. Undefined by default, but populated when you use body-parsing middleware (body-parser, multer)
- query. Parsed query string.
- params. Route wildcards.

Route params

= wildcards in URL.

```
'/user/:name'
```

Matches for example URL `'/user/unicorn'`. `unicorn` is then passed as param `name` to Request object and can be accessed in route handler as

```
req.params.name
```

Query strings

Query strings are not part of the route path. They can be obtained through `req` object

```
req.query
```

Returns an automatically parsed object, where keys are the names of query string variables and values their values.

```
'/route?a=1&b=one&c[]=x&c[]=y'
```

```
req.query = {
    a: "1",
    b: "one",
    c: ["x", "y"]
}
```

Task

t3-express Try to build HTTP server with Express!

Middlewares I

So far, Express lookls like an enhanced router. But it's much more than that in functionality.

An express application is basically a series of middleware function calls.

Middleware functions are functions that have access to request object (req), response object (req) and next middleware function in current request-response cycle. Next-middleware-function is commonly named next.
Middleware function can

- execute any code,

Middlewares II

- make changes to request and response objects,
- end the request-response lifecycle, or
- call the next middleware.

A middleware *must end the request-response* cycle (=ends the response stream), or *must pass control* to next middleware. Otherwise the request will be left hanging.

Middleware pipeline

```
app.get(route, middleware1, middleware2, ...,
↪ handler)
```

Basic middleware types

- Application-level

Middlewares III

- Router-level
- Error-handling

Application-level middleware

Bound to app using `app.use`, or `app.METHOD`

```
1 app.use('/user/:id', function (req, res, next) {
2   console.log('Request Type:', req.method);
3   next();
4 });
```

Router-level middleware

Router level middle ware works in the same way as application-level middleware, except it is bound to an instance of `express.Router()`.

```
1 var router = express.Router();
2 app.use('/', router); //mounting the router
3
4 router.use(fn) //fn is a middleware called every
  ↪  request to the router.
```

Router-level middleware may call `next('route')` to pass control to the next matching route.

Error-handling middleware

Special type of middleware that always takes *four arguments* (otherwise its treated as ordinary middleware).

Signature of error handling middleware is `(error, req, res, next)`.

```
1 app.use(function(err, req, res, next) {
2   console.error(err.stack);
3   res.status(500).send('Something broke!');
4 });
```

Third-party middleware

Middleware that adds functionality to Express apps. Install as a node module, then load into your app at the application or router level.

Example Cookie parser: Populates req.cookies with data parsed from Cookie header.

```
1 var express = require('express');
2 var app = express();
3 var cookieParser = require('cookie-parser');
4
5 // load the cookie-parsing middleware
6 app.use(cookieParser());
```

Then the `req.cookies` contains object keyed by the cookie names.

Task

t4-express-node-template Get to know node-express-template. Try to run it, then check `config/routes`

Asynchronous paradigm I

In one threaded execution environment, like Node.js, you have to be careful of what you do synchronously, otherwise your application stops responding to other requests.

Code is executed in smaller blocks of execution - which is a function block. A function is a smallest piece of code that will run as a whole.

Consider:

Asynchronous paradigm II

```
1  function f() {
2    console.log("foo");
3    setTimeout(function(){console.log("bar");}, 0);
4    console.log("baz");
5    h();
6  }
7
8  function h() { console.log("blix");
9
10 f();
```

Callbacks

When working with asynchronous calls (HTTP requests, database queries, file reads..), you always have to wait for the result. Async functions dont return value directly, but rather call you back with callback function you provide.

```
1 //wrong!! data will be undefined or some garbage
2 var data = fetchDataAsync(...)
3 //correct
4 fetchDataAsync(function(data) {
5     //data
6 })
```

Callback hell I

A problem you will encounter with this callback-only approach is that if you're chaining multiple callbacks, you have to do it like this:

```
1 doAsync1(function () {
2   doAsync2(function () {
3     doAsync3(function () {
4       doAsync4(function () {
5     })
6   })
7 })
```

- hard to read

Callback hell II

- hard to change
- ..only calls async code in given sequence, no error handling..
- $\rightarrow$ callback-hell, pyramid of doom

Callback hell can be rewritten using newer and more comfortable approach - Promises.
Promises are natively supported in ES6 and shiped to Node, but there are two major libraries that are widely used and can be used in browser safely:

- Bluebird,
- Q.

Solving Callback hell I

Let the async functions return promises, you can then rewrite this code to something less nested:

```
1 doAsync1()
2 .then(doAsync2)
3 .then(doAsync3)
4 .catch(handleError);
```

Another example

Solving Callback hell II

```
1  doAsync1()
2  .then(doAsync2)
3  .then(doAsync3)
4  .then((dataFromPrevPromise => {
5      throw new Error('error');
6      return 'ok';
7  })
8  .then(message => {
9      console.log(message);
10 })
11 .catch(err => {
12     if (err.message === 'error') {
13         return 'ok' //its our error, thats ok
```

Solving Callback hell III

```
14      }
15      throw err;
16  });
```

A value returned in `then` handler is passed as result of a promise. If an error occurrs, all next `thens` are skipped to catch, which can handle the error and change the state of chained promise to be fulfilled with returning any value.

Next tick I

Remind setTimeout example: Javascript ran all code in which is setTimeout was called, and on the next tick, callback function of setTimeout was executed.

- callback function is put on event queue
- $\rightarrow$ other other code in queue can be processed in meantime

Next tick II

```
1  function compute() {
2      // performs complicated calculations continuously
   ↪  for a while
3      // ...
4      process.nextTick(compute);
5  }
6  http.createServer(function(req, res) {
7       res.writeHead(200, {'Content-Type':
   ↪  'text/plain'});
8       res.end('Hello World');
9  }).listen(5000, '127.0.0.1');
10
11 compute();
```

Next tick III

This pattern allows the server to do CPU intensive task, while the server still responds to requests between the calculation callbacks. Without the nextTick, compute will just recursively call it self without interleaving it with other functions.

Modules I

Node.js has a simple module loading system. It allows you to separate functionality and to have a lot reusability of your code.
Example:
circle.js

```
1  const PI = Math.PI;
2  exports.area = (r) => PI * r * r;
3  //exports.area = function(r) { return PI * r * r };
```

server.js

```
1  var circle = require('./circle.js');
2  console.log(circle.area(10));
```

Modules II

The core functionality of modules is as following:

- Modules are cached after the first time they are loaded.
- Node.js has several modules compiled into the binary (i.e. 'fs')
- Node.js allows cyclic module dependencies
- Private variables are allowed

Modules III

You can export set of methods/objects, specified in exports.* or you can rewrite export itself.

```
1  const PI = Math.PI;
2  module.exports = (r) => PI * r * r;
```

```
1  var circle = require('./circle.js');
2  console.log(circle(10));
```

Task

continue on t4-express-node-template Writing own modules.

Introduction

Mongoose is a MongoDB object modeling tool designed to work in an asynchronous environment.

> *Mongoose provides a straight-forward, schema-based solution to model your application data. It includes built-in type casting, validation, query building, business logic hooks and more, out of the box.*

Getting started

```
npm install mongoose
```

```
var mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/test');
```

Schema, model I

Everything in Mongoose starts with a Schema. Each schema maps to a MongoDB collection and defines the shape of documents within that collection.

```
1  var mongoose = require('mongoose');
2  var Schema = mongoose.Schema;
3
4  var blogSchema = new Schema({
5    title:  String,
6    author: String,
7    body:   String,
8    comments: [{ body: String, date: Date }],
9    date: { type: Date, default: Date.now },
```

Schema, model II

```
10      hidden: Boolean,
11      meta: {
12        votes: Number,
13        favs:  Number
14      }
15    });
16    mongoose.model('Blog', blogSchema);
```

- String
- Number
- Date
- Buffer

Schema, model III

- Boolean
- Mixed
- ObjectId
- Array, typed Array

Methods, statics

```
1 blogSchema.methods.findSimilarType = function() {..}
2 blogSchema.statics.search = function() {..}
```

- statics available directly on model,
- methods on model isntance

Serialization

```
1  blogSchema.options.toJSON = {
2      transform: function(document, ret, options) {
3          ret.id = ret._id;
4          delete ret._id;
5          delete ret._event;
6          delete ret.__v;
7          return ret;
8      }
9  };
```

Hooks I

You can define different time of hooks on schema

- `init`
- `validate`
- `save`
- `remove`
- `update`

Hooks II

```
1  blogSchema.pre('save',function (next) {
2      if(this.isNew) {
3          //...do something if is new
4          next();
5      } else {
6          //...do something else
7          next();
8      }
9  });
```

Persisting object

```
1  var mongoose = require('mongoose');
2  var Blog = mongoose;
3
4  var blog = new Blog({title: 'New blog post'});
5  blog.save(function(err) {
6     if (err) {return handleError(err);} //validation
   ↪  failed, connection closed, ...
7     console.log('Blog saved successfully');
8  });
```

Querying

Common mongodb queries can be used on models `Model#find`, `Model#findOne`, ..

```
1 Blog.find({title: 'New blog post'}, function(err,
  ↪  res) {...});
```

`Model#aggregate`

```
1 Blog.aggregate([{$match: match}, {$project:
  ↪  {comments: 1, _id: 0}, {$unwind:
  ↪  '$comments'}]).exec(function(err, res){...});
```

References and subdocuments I

```
1  var userSchema =  = new Schema({
2      username: {type: String, unique: true}
3  });
4  mongoose.model('Comment', commentSchema);
5  ----
6  var commentSchema = new Schema({
7      text: String,
8      author: {type: mongoose.Schema.Types.ObjectId,
   ↪   ref: 'User', required: true}, //== reference by
   ↪   ObjectId for User model
9  });
10 mongoose.model('Comment', commentSchema);
```

References and subdocuments II

```
11  ----
12  var blogSchema = new Schema({
13    title:  String,
14    ...
15    comments: [commentSchema] //== subdocuments as
     ↪  array of Comment models
16  });
17  mongoose.model('Blog', blogSchema);
```

Object population

MongoDB has(had) no join, but we still want references to related documents in other collections.

Population is an automatic process of replacing specified paths in the document with document(s) from other collection(s).

```
1 Comment.findOne(query).populate('author').exec(callback);
```

Plugins

Schemas are pluggable and you can extend their functionality after you defined one with specified plugins.

Third party exmaple

npm module: `mongoose-times`. Adds `created` and `lastUpdated` date properties to the schema.

```
1 var times = require('mongoose-times');
2 var blogSchema = new Schema({
3   title:  String,
4   ...
5 });
6 blogSchema.plugin(times)
```

Custom plugin I

```
1  module.exports = function lastModifiedPlugin (schema,
   ↪  options) {
2    schema.add({ lastMod: Date })
3
4    schema.pre('save', function (next) {
5      this.lastMod = new Date
6      next()
7    })
8
9    if (options && options.index) {
10     schema.path('lastMod').index(options.index)
```

Custom plugin II

```
11      }
12  }
```

..and in the schema:

```
1  blogSchema.plugin(lastMod, { index: true });
```

Bookshelf

Popular frameworks/libraries

- Sequelize - promise-based ORM for Node, PostgreSQL, MySQL, MariaDB, SQLite and MSSQL
- Knex - SQL Query builder and Bookshelf, Javasript ORM built on top of it. Postgres, MySQL, MariaDB, SQLite3, and Oracle

We use knex & bookshelf for it's simplicity, speed and flexibility such as mixing raw SQL and object queries.

Knex I

Knex.js is a "batteries included" SQL query builder

Simple query builder and schema builder.

Example queries:

```
knex.select('title', 'author', 'year').from('books')
```

Outputs results for:

```
select `title`, `author`, `year` from `books`
```

```
knex('users').where({
  first_name: 'Test',
  last_name:  'User'
}).select('id')
```

Knex II

Outputs results for:

```
select `id` from `users` where `first_name` = 'Test'
  and `last_name` = 'User'
```

Any time, you can use raw sql to help you:

```
knex.select('year',
  knex.raw('SUM(profit)')).from('sales').groupBy('year')
```

Outputs results for:

```
select `year`, SUM(profit) from `sales` group by year
```

Creating schema - migrations I

Migrations allow for you to define sets of schema changes. Migration is a set of instructions with up/down interface, to make the changes as well as rolling them back when needed.

Migrations consist of set of files, each defines a module with up and down methods. Up/down method must return a knex promise.

```
1  exports.up = function(knex, Promise) {
2      return knex.schema.createTable('user',
   ↪   function(table){
3          table.increments('id').primary();
4          table.string('name).defaultTo('Anonymous');
5          table.enu('state', ['ANONYMOUS',
   ↪   'REGISTERED']);
```

Creating schema - migrations II

```
6           table.integer('category_id')
7               .unsigned()
8               .references('id')
9               .inTable('category');
10  };
11
12  exports.down = function(knex, Promise) {
13      return knex.schema.dropTable('user');
14  };
```

Bookshelf I

What it is? ORM for Node, built on top of Knex.
What it's not? It does not handle changes in schema, only eases relation quering and loading. Database schema has to be created in advance by knex.
Bookshelf entities are simplest as they can be.

- One model per table.
- You do now define properties and types. Entity properties are column names and their values.
- You only define relations, serialization and few configurations (as table name, id attribute, plugins used);

Bookshelf II

Model examples (One-to-many):

```
1  var Book = bookshelf.Model.extend({
2    tableName: 'books',
3    pages: function() {
4      return this.hasMany(Page);
5    }
6  });
7
8  var Page = bookshelf.Model.extend({
9    tableName: 'pages',
10   book: function() {
11     return this.belongsTo(Book);
```

Bookshelf III

```
12      }
13  });
```

Bookshelf IV

These models assume database schema has been created. If not, bookshelf will throw an error when trying to read unexisting tables, or missing columns. With knex, creation can look like following:

```
1  exports.up = function(knex, Promise) {
2    return knex.schema.createTable('books',
   ↪   function(table) {
3      table.increments('id').primary();
4      table.string('name');
5    }).createTable('pages', function(table) {
6      table.increments('id').primary();
7      table.string('content');
8      table.integer('book_id').references('books.id')
```

Bookshelf V

```
9       });
10  };
```

Entity manipulation I

Persisting entity is simple as:

```
1 new Book({name: 'Lexicon'}).save() //returns a
  ↪ promise with saved entity
```

- The object passed to bookshelf model always has to correspond with defined table
- Bookshelf doesn't care about what's in the object you are trying to save → creates insert query accordingly
- missing required properties or too much properties undefined on schema will trigger an SQL error

Fetching an entity can be simple as

```
1 Book.where({id: 123}).fetch() //returns a promise
  ↪ with model or null
```

Entity manipulation II

or more complicated, where you can use knex queries.

```
1 Book.query(qb => {
2     qb.select('book.id');
3     qb.leftJoin('book_authors', 'book.id',
  ↪   'book_authors.book_id');
4     qb.where('book_authors.author_id', 43);
5 });
```

Notice interlink with knex.

Entity manipulation III

Assuming we've got query above saved in `query` variable, we can now fetch all Book as models we defined earlier.

```
query.orderBy('id').fetchAll({withRelated:
↪  ['pages']});
```

You can use dot notation to fetch with related to related objects: i.e using

```
withRelated: ['pages.paragraphs']
```

Entity manipulation IV

You can also query for related objects if you dont want all of them or anything else you can do with a query builder. i.e. ordering:

```
1 new Book({'ISBN-13': '9780440180296'}).fetch({
2   withRelated: [
3     'genre', 'editions',
4     { chapters: function(query) {
  ↪   query.orderBy('chapter_number'); }}
5   ]
6 })
```

Java comparision I

Bookshelf stands between MyBatis and Hibernate frameworks.

- ORM is not directly connected in and out of bookshelf models.
- Querybuilder is similar to the hibernate one.
- Database structure is manipulated through migrations, not by models.