ExpressJS, Front-End and Storage

Asynchronous Server Technologies

César Berezowksi

Big Data Consultant @ Adaltas

cesar@adaltas.com

Recap

- Developer tools: terminal, editor, github, stack overflow, travis-ci…
- Best practices on a node project :
    - scripts : don’t repeat long and complicated commands
    - examples : tell people how to use your code
    - npm : external libraries
    - modules : split your code intelligently
    - unit testing : check that your code does what it is supposed to do
    - transpilers : write cleaner code faster

Recap

- Project on github linked to travis CI

```
myproject/
 |-- .gitignore
 |-- .travis.yml
 |-- package.json
 |-- readme.md
 |-- bin/
 |-- src/ -> coffee code
 |-- lib/ -> compiled js from coffee
 +-- test/
```

Final project

- Based on code from class
- Simple dashboard app :
  - User login
  - A user can insert metrics
  - A user can retrieve his metrics displayed nicely in a graph
  - A user can only access his own metrics

Questions ?

Today

- Nodemon (tool)
- ExpressJS (framework)
- Postman (tool)
- LevelDB (database)

Nodemon

What is it ?

- A simple utility
- Watches your development files
- Restarts the server on saving

How to use it ?

- `npm i --save nodemon`
- `nodemon src/app.coffee`

ExpressJS

What is it ?

- Minimalist framework for NodeJS apps
- Provides features for web app development
- Create robust APIs
- Functions to expose a front end

What’s an API ?

- Application Programming Interface
- In web : REST
  - expose a set of HTTP routes
  - use HTTP verbs (GET/POST/PUT/DELETE)
  - client connects to communicate
  - usually communicating in JSON

How to use an API ?

- Combination of two sides :
  - Back-end : rest api
  - Front-end : web pages w/ JS, mobile app, …
- Express brings both for the web !

Create a basic server

- Manually : use node-http
- With express :

```
express = require 'express'
app = express()

app.set 'port', 1337

app.listen app.get('port'), () ->
    console.log "server listening on #{app.get 'port'}"
```

API's Routing

- Manually : parse the url and apply corresponding logic
- With Express :

```
app.get '/', (req, res) ->
  # GET

app.post '/', (req, res) ->
  # POST

app.put '/', (req, res) ->
  # PUT

app.delete '/', (req, res) ->
  # DELETE
```

API’s routing

- You can add parameters in the routes :

```
app.get '/hello/:name', (req, res) ->
  res.send req.name
```

Prepare a front end

- Create a `view/` directory
- Create a `layout.jade` file in it :

```
doctype html
html
  head
    title My Web Page
    block head
  body
    block content
```

Prepare a front end

- Create a `view/` directory
- Create an `index.jade` file in it :

```
extends layout

block head
  # Here will go our css/js links

block content
  p Hello world !
```

Prepare a front end

- Tell express to use our Jade views

```
app.set 'views', "#{__dirname}/../views"
app.set 'view engine', 'jade'
```

- Render our index on /

```
app.get '/', (req, res) ->
  res.render 'index', {}
```

Make it sexy !

- Expose static content (JS, CSS, Images, …)
- Download bootstrap getbootstrap.com/getting-started/#download
- Download JQuery code.jquery.com/jquery-2.1.4.min.js
- Add the css in `public/css` and the js in `public/js`

Make it sexy !

- In our `app.coffee`

```
app.use '/', express.static "#{__dirname}/../public"
```

- In our `index.jade`

```
block head
  script(type="text/javascript" src="js/jquery-2.1.4.min.js" charset="utf-8")
  script(type="text/javascript" src="js/bootstrap.min.js" charset="utf-8")
  link(rel='stylesheet', href='/css/bootstrap.min.css')
```

- Notice how the font changed ?

Let’s bring some AJAX

- Technologies used to dynamically update static pages
- Use JS embedded in HTML
- Get data from a server
- Update page without reloading

Let’s bring some AJAX

- Prepare the data on the back-end
- Let’s create a new module called `metrics` :

```
module.exports =
  ###
  `get()`
  -------
  returns some hard-coded metrics
  ###

  get: () ->
    return = [
      timestamp:(new Date '2013-11-04 14:00 UTC').getTime(), value:12
    ,
      timestamp:(new Date '2013-11-04 14:30 UTC').getTime(), value:15
    ]
```

Let’s bring some AJAX

- Expose the metrics on the back-end

```
app.get '/metrics.json', (req, res) ->
  res.status(200).json metrics.get()
```

And retrieve them on the front-end !

- In our `index.jade`

```
block content
  div.container
    div.col-md-6.col-md-offset-3
      p hello world !
      button(type="button" class="btn btn-success"
id="show-metrics") Bring the metrics
      #metrics
```

And retrieve them on the front-end !

- In our `index.jade`

```
script
  :coffee-script
    $('#show-metrics').click (e) ->
      e.preventDefault()
      $.getJSON "/metrics.json", {}, (data) ->
        content = ""
        for d in data
          content += "<p>timestamp: #{d.timestamp}, value: #{d.value}</p>"
        $('#metrics').append content
```

Postman

What is it ?

- Dashboard to test your API
- Simulate HTTP request
- Specify custom body & headers
- getpostman.com

How about storing ?

Databases

- RDBMS -> MySQL, PostGreSQL, Hive
- NoSQL
  - Column families: HBase, Cassandra
  - Document Store: MongoDB, ElasticSearch
  - Key Value: LevelDB
  - Graph DBs: Titan, Neo4J

LevelDB

- In-memory key-value store embedded in Node
- OpenSource
- NoSQL DB, Key Value store
- Originally written by Google
- leveldb.org

Why LevelDB for our project ?

- It’s blazing fast
- In memory & backed by the file system
- Keys are ordered : suitable for metrics
- Data compression with Snappy
- Embedded in the app, nothing else to setup / manage

Some limitations

- Not an SQL database
- Only a single process at a time

Let’s setup

- `npm install --save levelup leveldown level-ws`
- Create a `db/` directory at root

Use the db

- To open the db :

```
levelup = require 'levelup'
levelws = require 'level-ws'
db = levelws levelup "path/to/db_file"
```

- To write :

```
db.put key, value, (err) ->
  if err then …
```

- To read:

```
db.get key, (err, value) ->
  if err then …
```

The metrics

- Key : `metrics:#{id}:#{timestamp}`
- Value : an integer

Read/write metrics

- One by one ? Too heavy !
- Use streaming :

```
    stream = db.createReadStream(…)
    stream = db.createWriteStream()
```

Let’s post some metrics

- In our metrics.coffee, add a `save` function

```
save: (id, metrics, callback) ->
  ws = db.createWriteStream()
  ws.on 'error', callback
  ws.on 'close', callback
  for metric in metrics
    {timestamp, value} = metric
    ws.write key: "metric:#{id}:#{timestamp}", value: value
  ws.end()
```

Let’s post some metrics

- Install body-parser to parse the request’s body

```
npm i --save body-parser
```

- Configure Express to use it

```
app.use require('body-parser')()
```

Let’s post some metrics

- Using Postman :
    - Set up a POST request on /metrics
    - Set the header `Content-Type:application/json`
    - Add an array of metrics as RAW body :

```
[
    { "timestamp":"1384686660000", "value":"10" }
]
```

Or use a script ?

- Create script `bin/populatedb` with execution rights

```
#!/usr/bin/env coffee

metric = require '../src/metrics'

met = [
  timestamp:(new Date '2013-11-04 14:00 UTC').getTime(), value:12
,
  timestamp:(new Date '2013-11-04 14:10 UTC').getTime(), value:13
]

metric.save 0, met, (err) ->
  if err then throw err
  console.log 'Metrics saved'
```

From now on

Your work

- Front :
  - Work on the front’s layout with CSS
  - Display the metrics in a graph with d3.js
- Back :
  - Add `get` and `remove` to the metrics module
  - Use Postman to test the API
  - Enhance the populatedb script to add multiple metric batches