SZAKDOLGOZAT

Tamás Csaba
2013

LightWeight Logger

Workout Logger System

Készítette: Tamás Csaba

2013

Konzulens: Rátósi Tivadar

Nyilatkozat

Alulírott **Tamás Csaba** kijelentem, hogy a szakdolgozat saját munkám eredménye. A dolgozat elkészítéséhez felhasznált valamennyi forrást az irodalomjegyzékben feltüntettem a hivatkozás szabályainak megfelelően. A dolgozat elkészítéséhez meg nem engedett segédanyagokat nem használtam fel.

Szakdolgozatomat más intézményben nem adtam be.

Kelt: 2013.04.14

Tamás Csaba

Tartalomjegyzék

1. Bevezetés

Pehelysúlyú, egyszerűen használható, mobilbarát workout logger. Segítségével könnyedén nyomon tudod követni, az eredményeidet. Segít abban, hogy folyamatosan fejlődni tudjon a sportteljesítményed.

A projekt ötlet onnan jött, hogy már több mint 5 éve komolyan sportolok és valahogy szerettem volna nyomon követni a fejlődésemet, illetve egy idő után azt éreztem, hogy megrekedtem egy szinten, először papír alapon írogattam az eredményeket, de rájöttem, hogy hosszú távon ez nem célravezető, mivel rengeteg adat halmozódik fel és néha el is hagytam pár cetlít, ekkor jött a az ötlet, egy workout logger site-ról, ami mobilbarát, mivel így edzés után rögtön fel tudom vinni az eredményeket. Mára már nagyon sok embernek van okostelefonja mobilnet kapcsolattal, így könnyen kapcsolódni tud az internetre szinte bárhonnan.

Jelenleg fronted fejlesztőként dolgozom a Virgo Systems Kft-nél, így jött az ötlet, hogy szerver oldali JavaScript a **NodeJS** segítségégével készítsem el az oldalt. A NodeJS mellé akkor már érdemes egy noSQL alapú szintén aszinkron működést preferáló adatbázis kezelő rendszert használni, ami persze a **MongoDB** lett. Ez a páros a PHP+MySQL kombináció nagyon ütős alternatívája.

Ezekről a rendszerekről egy rövid ismertetőt, illetve telepítési leírást a következő 2 fejezetben adók.

2. NodeJS

A NodeJS alapjait Ryan Dahl fejlesztette le 2009-ben. Ryan egy viszonylag kis dolgot szeretett volna megoldani: valós időben kijelezni, hogy egy HTML feltöltés hol tart éppen, ebből lett a Node. Maga a rendszer C/C++-ban íródott, és egy esemény alapú I/O rendszert takar a Google V8 JavaScript motorja felett.

A Node.js-t úgy írták meg, hogy (szinte) minden esemény aszinkron legyen, ezért az program sosem blokkolódik, azaz nem kell várni, hogy egy művelet befejeződjön, vele párhuzamosan futtathatunk további műveleteket.

Példa egy egyszerű aszinkron függvényre::

```
function isLinkOrDir (path, cb) {
      fs.lstat(path, function (er, s) {
            if (er) return cb(er);
            return cb(null, s.isDirectory() || s.isSymbolicLink())
      });
}
```

Ezeket az aszinkron kéréseket nagyon jól lehet egymás után fűzni (chaning) pl: egy későbbi adatbázis lekérés:

```
Workout.find({ user: userId }).sort({createdAt: -1 }).populate('user')
  .exec(function(err, workouts) {
      if (err) return res.render('500');
      console.log(workout);
    });
})
```

2.1. Telepítés

Szinte akármilyen operációs rendszerre telepíthetjük a Node-ot, Linuxtól kezdve a Mac OS X-en át a Solarisra. Fordíthatjuk ezen kívül Windows alatt Cygwin-en, FreeBSD-re és OpenBSD-re. A rendszer telepítése elég egyszerű, letöltjük az aktuális stabil változatot, konfiguráljuk, és fordítjuk (make). A fordításhoz szükségünk van egy C++ (gcc) fordítóra, és a Python 2.4 vagy újabb változatára.

Ubuntu alatt csomagkezelő segítségével is könnyen telepíthetjük:

```
sudo apt-get install python-software-properties python g++ make
sudo add-apt-repository ppa:chris-lea/node.js
sudo apt-get update
sudo apt-get install nodejs
```

Forrásból telepítjük, Windows vagy más rendszerre:

```
mkdir node-latest
cd node-install
curl http://nodejs.org/dist/node-latest.tar.gz | tar xz
```

```
--strip-components=1
./configure
make install
```

2.2. NodeJS a gyakorlatban

Az egyik lényeges különbség például a PHP-hoz képest azon kívül, hogy JavaScript-ben írjuk az alkalmazásunkat, hogy nincsen szükségünk HTTP szerverre (mint amilyen az Apache httpd), mivel mi magunk írjuk a HTTP szerverünket (server.js).

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

http.createServer(function (request, response) {

      response.writeHead(200, {'Content-Type': 'text/plain'});

      response.end('Leight Weight\n');

}).listen(3000);

console.log('Szerver a http://127.0.0.1:3000/ címen fut');
```

Az első sorban inicializáljuk a node http modulját, itt ejtenék róla szót, hogy a node CommonJS alapú JavasScript-ben vagy C++-ban írt moduláris rendszert biztosít számunkra. Mindezt egy npm nevezetű package menedzser rendszeresen keresztül, erről részleteket később írók.

A http modulnak a createServer metódusával készítjük el a szerverünket, aminek a visszatérési értéke a html request (kérés) illetve a response (válasz). A kérésbe tartozik minden olyan művelet vagy adattovábbítás ami a kiszolgáló felé zajlik pl: GET, POST stb. kérés vagy a böngésző adatait is innen tudjuk elérni. A response a szerver válasza ebben az esetben egy 200-as html fejléccel visszaküldött text/plain típusú fájl, amiben a „Leight Weight” és egy sortörés szerepel.

A következő paranncsl tudjuk mindezt futtatni:

```
node /home/user/node/server.js
```

2.3. NPM

A Node hivatalos csomagkezelője az npm, amellyel kényelmesen listázhatjuk, telepíthetjük, frissíthetjük és eltávolíthatjuk az elérhető modulokat, valamint mi magunk is publikálhatjuk az általunk fejlesztett modulokat az npm rendszerébe. Rendszerkövetelményei azonosak a Node-éval, telepítése egyszerű:

```
curl http://npmjs.org/install.sh | sh
```

A következő paranccsal installálhatjuk a modulokat:

```
npm install async -g
```

A -g kapcsolóval globális telepítjük az adott kiterjesztést így bárhol elérhető lesz a számunkra. Viszont ilyenkor nem árt sudo-val futtatni, mivel az usr/lib/node_moduls mappába fog kerülni.

Fontosabb core modulok :

- **sys**: Loggolást, hibakeresést, futtatást segítő eszközök
- **fs:** fájl rendszerhez biztosít hozzáférést
- **http:** HTTP kliens és szerver alkalmazásokhoz
- **net:** hálózati feladatok ellátásához (socketek, streamek)
- **crypto:** titkosítás, kódolás (md5, sha1, rsa, sha256 stb.)
- **path:** fájlrendszer elérési utakhoz
- **url:** URL kezeléshez
- **querystring:** Az URL query részének kezeléséhez

3. MongoDB

Napjainkban egyre elterjedtebbek a noSQL rendszerek egyes értelmezések szerint Not only SQL, azaz nem csak SQL, más értelmezés szerint egyszerűen csak nem SQL adatbázis szoftverek gyűjtőneve. A NoSQL adatbázisok elsősorban nem táblázatokban tárolják az adatokat és általában nem használnak SQL nyelvet lekérdezésre.

A noSQL rendszer előnyei, hogy szinte mindenki objektum-orientált környezetben, nyelvekkel dolgozik, így az adatokat is ilyen szemlélettel lenne jó kezelni. Erre alapvetően nagyjából jó is a klasszikus relációs modell, nagyon leegyszerűsítve egy táblát egy osztálynak lehet megfeleltetni első megközelítésben. Amit viszont a merev relációs adatbázisok nem tudnak szépen kezelni, az a struktúraváltozás, amit a mélyebb objektum-orientált koncepciók maguk után vonnak (pl.: öröklődés). Relációs adatbázisoknál előre kell definiálni a struktúrát, így minden változás ALTER-ekkel jár, amik egyrészt kódból nem mindig kezelhetőek, másrészt lassúak sok adat esetén és fölösleges overhead-et jelentenek. A sebesség és kezelhetőség miatt ezért sok fejlesztő “hackel”, TEXT mezőkben egyénileg kódoltan tesz le adatokat (serialize, json, marshal), amiknek a változásai már kódszinten jóval könnyebben kezelhetőek, viszont egyrészt így értelmét veszti az SQL, másrészt kívül is esnek a standard SQL koncepciókon, lekérdezéseken, operátorokon (pl. JOIN egy json-encodeolt adathalmaz tetszőleges elemére). Ezt felismerve jöttek létre a noSQL megoldások, kezdve az SQL fogalmaival teljesen szakító eszközökkel (pl. Redis), vagy az arany középutat választó MongoDB-vel, ami az adatokat egy JSON pontosabban BSON objektumban tárolja és egyszerű, objektumként viselkedik.

A mongo kicsit hibrid megoldás, mert bár nem kötött sémával dolgozik, de igyekszik nem túlzottan elrugaszkodni a relációs modelltől sem. Ismeri az adatbázis és a tábla fogalmát, a táblákat viszont gyűjteménynek (collection) hívja. A szokott módon egy adatbázis több gyűjteményt is tartalmazhat. A gyűjteményeket és adatbázisokat a nevük azonosítja.

A legtöbb NoSQL adatbázis szerver, (mint a MongoDB is) erősen optimalizált írás és olvasás műveletekre, míg ezen túl nem sok műveletet támogatnak (find, update, delete stb.).

3.1. Telepítés

Ubuntu:

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv 7F0CEB10
```

```
sudo echo "deb http://downloads-distro.mongodb.org/repo/ubuntu-upstart dist
10gen" | tee -a /etc/apt/sources.list.d/10gen.list
```

```
sudo apt-get -y update
```

```
sudo apt-get install mongodb-10gen
```

Ezzel a hivatalos gyártó a 10gen által készített csomagot érdemes használni, az Ubuntu alapból elérhető saját csomagban, de ez egy nagyon régi 2011-ben készült 1.1.6-os verziót tartalmazza.

Windows:

http://docs.mongodb.org/manual/tutorial/install-mongodb-on-windows/

3.2. Egy kis Mongo

Telepítés után nyissunk meg egy terminált/konzolt és gépeljük be a `mongo` parancsot. Alapból a test adatbázishoz fog kapcsolódni, de a `db` parancsot kiadva, megbizonyosodtunk róla. Adatbázist váltani a `use mydb` paranccsal tudunk. Maga a db mindig az aktuális adatbázist jelenti amihez kapcsolódtunk, így elhelyezni adatokat a következőképpen tudunk:

```
db.users.insert({
    first: 'arnold',
    last: 'schwarzenegger',
    dob: '03/06/1925',
    gender: 'm',
    hair_colour: 'barna',
    occupation: 'bodybuilder',
    nationality: 'amerikai'
});
```

Itt a db maga az adatbázis a users a collection (tábla) az elhelyezett adat pedig a document.

Kilistázni és keresni a következő paranccsal tudunk:

```
db.nettuts.find()
```

Miután kiadtuk ezt a parancsot, megkapjuk az elmentett dokumentumot amit a mongo automatikusan kiegészít egy ObjectId-val, amit indexnek is nevezhetünk, mivel egyedileg azonosítja a document-et. Következőképpen néz ki a mi esetünkben.

```
> db.users.find()
{ "_id" : ObjectId("516ab4924da1b1ea75e06499"), "first" : "arnold", "last" : "schwarzenegger", "dob" : "03/06/1925", "gender" : "m", "hair_colour" : "barna", "occupation" : "bodybuilder", "nationality" : "amerikai" }
```

Gyakorlatilag az insert és a find metódussal el lehet végezni az adatbázis műveletek 90%-át, de ezen kívül még számtalan más függvény is elérhető a MongoDB-ben, akár még térképészeti alakzatok alapján végzett keresést is támogat.

3.3. Mongo után jöjjön a Mongoose

A Mongoose egy MongoDB-re épülő ORM (Object Data Manager) rendszer. Mely segítségével egyszerűen tudunk létrehozni adatbázis shema-kat (modelleket) és tudjuk rendszerezni validálni stb. Telepítését az NPM rendszeren keresztül történik.

```
npm install mongoose
```

Egy egyszerű példa a működésére:

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

var Cat = mongoose.model('Cat', { name: String });
```

```
var kitty = new Cat({ name: 'Cirmos' });
kitty.save(function (err) {
    if (err) // ...
    console.log('meow');
});
```

Ebben a példában kapcsolódunk az adatbázishoz majd létrehozunk egy új modellt Cat néven, amiben egy 'Cat' nevű kulcshoz egy String értéket rendelünk. Majd egy kitty nevű változóban példányosítjuk az objektumot és megadjuk névnek 'Cirmos'-t ezután ezt a példányt mentjük, majd a mentés callbackje kiírja a consolra hogy 'meow'.

Később még részletezem a modell leírásnál, a Mongoose további funkcióit bevezetésnek egyenlőre ennyit.

4. Project felépítése

Klasszikus **MVC** szemlélet szerint épül fel. Az MVC szoftvertervezésben használatos szerkezeti minta. Összetett, sok adatot a felhasználó elé táró számítógépes alkalmazásokban gyakori fejlesztői kívánalom az adathoz (modell) és a felhasználói felülethez (nézet) tartozó dolgok szétválasztása, hogy a felhasználói felület ne befolyásolja az adatkezelést, és az adatok átszervezhetők legyenek a felhasználói felület változtatása nélkül. A modell-nézet-vezérlő ezt úgy éri el, hogy elkülöníti az adatok elérését és az üzleti logikát az adatok megjelenítésétől és a felhasználói interakciótól egy közbülső összetevő, a vezérlő bevezetésével.

Az alkalmazás struktúrája a következő:

```
-app/
  |__controllers/
  |__models/
  |__views/
-config/
  |__routes.js (url-ek kezelése)
  |__config.js (beállítások)
  |__passport.js (autentika config)
  |__express.js (express.js config)
  |__middlewares/ (köztes beépülők)
-public/
  |__css/
  |__js/
  |__img/
  |__......
-server.js
-package.json
```

Mint már említettem a node esetén mi írjuk a szervert, ez a `server.js` ami talán legjobban egy index.php-hez hasonlít azzal a különbséggel, hogy a NodeJS a szerver indításakor fogja felparsolja és hasonlóan, mint a Java egy virtuális gép segítségével futtatja a kódot. Tehát az alkalmazás a szerver indításakor lefut és az egyes kéréseknél már csak az aktuális események kiszolgálása történik meg.

A külső kérések számára csak a public mappa tartalma lesz elérhető így semmiképpen nem láthatnak bele a fájljaink tartalmába.

A `package.json` az npm által használt modulokat tartalmazza, nézzünk egy kicsit bele:

```
{
    "name": "LeighWeigt",
    "description": "LeightWeight Workout Logger",
    "version": "0.0.1",
    "private": true,
    "author": "Tamás Csaba <tamcsaba@gmail.com>",
    "engines": {
        "node": "0.10.2",
        "npm": "1.2.15"
    },
    "scripts": {
      "start": "./node_modules/.bin/nodemon server.js"
    },
     "dependencies": {
      "express": "latest",
      "jade": "latest",
      "express-validator": "latest",
      "mongoose": "latest",
      "socket.io": "latest",
      "connect-mongo": "latest",
      "connect-flash": "latest",
      "passport": "latest",
      "passport-local": "latest",
      "passport-facebook": "latest",
      "passport-twitter": "latest",
      "underscore": "latest",
      "gzippo": "latest",
      "async": "latest",
      "nodemon": "latest",
      "view-helpers": "latest",
  }
}
```

A `package.json` fájlban adhatjuk meg alkalmazásunk nevét alap adait és függőségeit is. Továbbá különböző script-ket rendelhetünk hozzá. Jelen esetben az `npm start` parancsra, lefut a 'nodemon' nevezetű kiterjesztés és elindítja a server.js-t. A nodemon arra szolgál, hogy amennyiben bármilyen változás történik, a projektünkön belül, ezt azonnal érzékeli és automatikusan újraindítja a szervert.

A másik fontos része a fájlnak a `dependencies` ezeket a kiterjesztéseket használjuk majd a projektünkben (node_modules) mappába kerülnek telepítésre. A csomag neve után

megadhatjuk, melyik verziót kívánjuk telepíteni pl: '1.4.5' vagy melyiknél legyen nagyobb '>1.4' vagy akár '1.4.x' formában is. Jelenleg én mindenből a legfrissebb verziót használom.

4.1. Project beindítása

Miután feltelepítettük a NodeJS-t és a MongoDB-ét csak a projekt mappájába kell navigálnunk majd kiadni a következő 2 parancsot.

```
npm install
```

```
npm start
```

4.2. Project alapjai

Igyekeztem minden szempontból a legmodernebb megoldásokkal és rendszerekkel dolgozni. A következőkben tárgyalom, hogy mely technológiák ezek.

4.2.1. Frontend

Eddig jobbára a backendről volt szó, nézzük mi a helyzet a frontendel. A LeightWeight a Twtter Bootstrap nevezetű frameworkre épül, ez van egy egyedi stílussal megspékelve. Talán a Bootstrapről annyit, hogy jelenleg az egyik legmodernebb és lekiforrottabb keretrendszer, nagyon sok komponenst kapunk kézhez, amiket nagyon egyszerűen fel lehet használni. Maga a rendszer teljes körűen támogatja a responsive site-ok elkészítését.

Igyekeztem mindenhol diszkrét JavaScript szabályai szerint eljárni, hogy JS nélkül is használható legyen az oldal. A diszkrét Javascript azt mondja, hogy a HTML kódunkban ne használjunk Javascriptet, válasszuk le, s tegyük külön fájlba teljesen scriptjeinket, s építsük fel úgy az oldalt, hogy azok nélkül is teljes funkcionalitással bírjon - maximum nem olyan kényelmesen. Ismerősnek tűnhet az ötlet: a mai modern CSS technikák pontosan ezt mondják a stíluslapok esetén is: válasszuk szét a megjelenést és a tartalmat. A Javascript esetén a tartalom, s a használhatósági javítások szétválasztásáról van szó. A HTML azt mondja meg, mi ez a szöveg, a CSS azt, hogy hogyan nézzen ki, a Javascript pedig azt, hogy hogyan viselkedjen (az oldal). A Javascript egy nagyon jó eszköz a weblapok használhatóságának növelésére.

Az oldal megalkotásakor kicsit elrugaszkodtam a HTML nyelvtől, Jade template engine használatával hoztam létre a view-t. A Jade egy kicsit elrugaszkodik a hagyományos HTML-től igyekszik minél kisebbre összehúzni a layout méretét, azzal, hogy nem kell záró tageket használnunk és igyekszik minél több dolgot rövíditeni.

JADE	HTML
`h1#bar.foo foobar`	`<h1 id=”bar” class=”foo”>foobar</h1>`

`#content`
`  .block`
`  input#bar.foo1.foo2`
	`<div id="content">`
	`  <div class="block">`
	`  <input id="bar" class="foo1 foo2">`
	`  </div>`
	`</div>`

input(type="text", placeholder="your name")	<input type="text" placeholder="your name">

!!! 5
html
  head
  title my jade template
  body
  h1 Hello World
	<!DOCTYPE html>
	<html>
	  <head>
	  <title>my jade template</title>
	  </head>
	  <body>
	   Hello World
	  </body>
	</html>

Látszik, hogy a Jade segítéségel mennyivel könnyeben és gyorsabban lehet kódólni, mivel nem kell figyelnünk a lezáró tag-re csak a helyes syntax-ra.

4.2.2. Backend

A server.js fájlban található, a rendszer magja itt behúzunk minden fontos modult és hozzá kapcsolódó config fájlokat.

```
var env = process.env.NODE_ENV || 'development',
       config = require('./config/config')[env],
```

A node rendszerek különlegessége, hogy többféle környezetet (enviroment) biztosít számunkra, melyek, más és más beállításokkal rendelkezhetnek és lehetőségünk van globális tulajdonságokat megadni, ami mindegyikre jellemző. Tehát a gyakorlatban ez úgy néz ki, ha nem adjuk, meg a futtatási környezetet, akkor development módban fog futni az alkalmazásunk, dev módban, a hibákról folyamatos értesítést kapunk a konzolon mind pedig a böngésző kimenetben is (ha egyáltalán elindul a szerver). Illetve másik fontos dolog, hogy más-más adatbázissal dolgozhatunk a különböző környezetekben, de bármilyen más tulajdonságokat egyedileg meg lehet határozni.

Miután beolvastuk a beállításokat beállítjuk azokat az express számára. Az express az alkalmazásunk alaprétege, biztosítja számunkra a routingot, a template, süti, session kezelést is, ezáltal lényegesen egyszerűbbé teszi a munkánkat, hiszen az express függvényein keresztül könnyedén el tudjuk érni őket!

Az express konfiguráció után, inicializáljuk az adatbázis modelljeinket. Ezt a node egyik beépített fájlbeolvasó függvényével lehet megoldani, fontos, hogy ebben az esetben ne aszinkron módon tegyük, meg mert akkor még előfordulhat, hogy úgy hivatkozunk a fájlok tartalmára, hogy a rendszer még nem olvasta be azokat.

```
var models_path = __dirname + '/app/models';
fs.readdirSync(models_path).forEach(function (file) {
  require(models_path+'/'+file);
});
```

Végül pedig a routing beállítása és az alkalmazás indítása következik.

5. Felhasználókezelés

Az oldal elkészítésének első lépése a felhasználókezelés megoldása volt, itt fontos szempont volt az, hogy a legelterjedtebb autentika rendszereket is integráljam az oldalba, itt elsősorban arra gondolók, hogy facebook, twitter stb account segítségével is be tudjon lépni a látogató.

5.1. Modell

Ehhez tartózó adatmodell a következőképpen néz ki (`app/models/user.js`):

```
var UserSchema = new Schema({
	name: { type: String, required: true },
	email: { type: String, required: true, unique: true },
	username: { type: String, required: true , unique: true },
	provider: { type: String, required: true },
	hashed_password: { type: String, required: true },
	salt: { type: String, required: true },
	facebook: {},
	twitter: {}
});
```

A séma elég lineáris tároljuk a nevét, felhasználó nevét, email címét jelszavát stb. mindegyiket String értékként.

Itt térnék ki a Mongoose validálási lehetőségeire, mert a felhasználó adatok mentésekor ez meghatározó. Már a séma definiálásakor beállíthatjuk hogy az adott mező szükséges és egyed-e. Ebben az esetben minden mező szükséges, és a felhasználónévnek és e-mail címnek egyedinek kell lennie.

Viszont ennél sokkal több lehetőséget add a Mongosse egyedi validálási metódusokat írhatunk a különböző kulcs érték párokhoz.

```
UserSchema.path('email').validate(function (email) {
	return /^([a-z0-9_\.-]+)@([\da-z\.-]+)\.([a-z\.]{2,6})
$/i.test(email);
}, 'Nem érvényes az e-mail címed!');
```

Ebben az esetben az e-mail címhez, írhatunk egy kis RegEx-et, amiben megnézzük hogy a sting kezdete egy olyan karakterlánc-e ami a-z vagy 0-9-ig tartalmaz karaktereket és aláhúzást pontot vagy kötőjelet, majd következik a @ ezután szám illetve szintén a-zig jöhet bármi plusz pont és kötőjel is, majd pedig pont után 2-6 karakternyi kisbetűk és pontot tartalmazó végződés jön, ami illeszkedik a (.hu .com .info stb-re).

Amit talán még megemlítenék az a Virtuals, ezek olyan attribútumok amik más attribútumomból állnak össze, pl. ilyent használunk a jelszó elővarázsoláshoz, mivel magát a jelszót nem tároljuk az adatbázisban csak a hash-elt verzióját és magát a sót. Ami egy random karaktersor amivel úgymond megfűszerezzük, a jelszót, hogy semmiképpen ne lehessen visszafejteni, a szivárványtábla alapján. A szivárványtábla nem más, mint egy hatalmas

adatbázis, ami különféle karakterkombinációkból hash eljárással készült kódokat tartalmaz - ezeket hasonlítja össze, ha megvan az egyező hash-pár, megvan a jelszó is. Ezt a rést védjük azzal,hogy megsózzuk a jelszót ezáltal egy random karaktersorral lesz megtoldva, egy x algoritmus szerint, amivel már nem tud mit kezdeni a kódvisszafejtő.

```
UserSchema.virtual('password').set(function(password) {
    this._password = password
    this.salt = this.makeSalt()
    this.hashed_password = this.encryptPassword(password)
  }).get(function() { return this._password });
```

Így néz ki egy felhasználó (statikus adatai) az adatbázis users collectionjában:

```
[
  'provider' => 'local',
  'name' => 'Tamás Csaba',
  'email' => 'tamcsaba@gmail.com',
  'username' => 'tamascsaba',
  'hashed_password' => 'e1cdfd1dc7bfb9191a885ad494b0268fb1367334',
  'salt' => '1328329057909',
  '_id' =>
  MongoId::__set_state(array(
     '$id' => '515e9fb50b17e6ee21000010',
  )),
  '__v' => 0,
]
```

5.2. Routing és View

Mielőtt a controllere térnénk nézzük bele egy kicsit a routingba hiszen innen indul ki minden folyamat. A (`config/router.js`) fájlban vannak inicializálva a controllerek, illetve felépítve a routing tábla.

```
var users = require('../app/controllers/users');
app.get('/user/login', users.login);
app.get('/user/register', users.register);
app.get('/user/logout', users.logout);
app.post('/user/create', users.create);
```

Az app maga az alkalmazásunk amit még a projekt elején a `server.js`-ben definiáltunk. 4 -féle http request érkezhet a server felé:

- get
- post

- delete
- put (update)

Ezekre a kérésekre tudnunk „ráülni” és szabályózni hogy egy adott url-re érkező bizonyos kérésre milyen esemény fusson le a controllerben. Egymás után füzhetünk akár több eseményt is, illetve lehetőség van változónevek definiálására is az URL-ben az alábbi módón.

```
app.get('/user/:userId', auth.requiresLogin, users.show);
```

A profil oldal routingja, ahol a megadott userid alapján először megnézi az autentikáért felelős middleware metódus hogy be vagy-e lépve majd megmutatja a profil oldalt. Ami az alapján generálódik ki hogy a saját profilod-e (akkor szerkeszteni is tudod).

A view felépítése a layoutból indul ki, ahol van egy alapértelmezett template-ünk, az esetek 99%-ban ezt terjeszti/extendálja ki a többi template. Mikor Ajaxos kéréseket használunk, akkor pedig layout nélkül jelenítjük meg a tartalmat, mint a bejelentkezésnél és a regisztrációnál is.

5.3. Controller

A felhasználókezelés controlerje az (`app/controlers/users.js`) fájlban található a routingból hivatkozunk ide.

Kezdjük a beléptetéssel. Itt említeném meg, hogy miután lefutott a routung metódusnak 2 visszatérési értéke van a

request (req) illetve a **response** (res), mindkettő egy-egy objektum.

A request a **kérés** ami a szerver fele intézünk, ki tudjuk belőle nyeri az URL-t amit lekértünk:

```
// ?name=tobi
req.param('name')
// => "tobi"
```

vagy a sütiket:

```
// Cookie: name=tj
req.cookies.name
// => "tj"
```

A request tartalmaz minden olyan információt ami a user felől érkezik.

A response ennek az „ellentéte a válasz ami a szervertől jön pl:

```
res.redirect('/admin'); //átirányítjuk a user-t admin oldalra

res.send(404, 'Az oldal nem található!'); //küldünk 404-es HTML státusszal
egy hibaoldalt
```

Ezek alapján egy egyszerű Ajax-os oldal kirendelése az alábbiak szerint történik.

```
exports.login = function (req, res) {
```

```
  if(!req.xhr) {
    res.render('users/login', {
      title: 'Bejelentkezés',
      message: req.flash('error')
    });
  } else {
    res.render('users/xhr/login', {
      message: req.flash('error')
    });
  }
};
```

Megnézzük, hogy a kérés Ajax-os-e, (`req.xhr`), ha igen akkor kirendereljük a templatet layout nélkül különben meg hozzárendereljük a layout-ot is. Ezen kívül leküldjük a szükséges változókat is (title, és errors).

A 2 legfontosabb része van a user controllernek felhasználó beléptetése és regisztrációja.

Kezdjük a regisztrációval. Itt kitérnék arra, hogy a node-ban nem csak adatbázis szinten tudjuk validálni az inputokat, hanem beépített validátor modullal rendelkezik a következő ellenőrzési eljárásokat, alkalmazóm a felhasználó adataira.

```
  req.assert('name', 'Nem adtad meg a neved!').notEmpty();
  req.assert('email', 'Érvényes e-mail címet kell megadnod!').isEmail();
  req.assert('password', 'Jelszadnak legalább 6 karakteresnek kell lennie
(max 20)!').len(6,20);
  req.assert('username', 'Felhasználóneved minimum 4 karakter és csak angol
ábécé betűi lehetnek!').is(/[a-zA-Z]+$/).min(4);
  req.assert('digits', 'Nem megfelelő ellenőrzőkódot írtál
be!').is(req.session.captcha);
```

Ebből az utolsó sor a captcha, ami olyan automatikus teszt, ami képes megkülönböztetni az emberi felhasználót a számítógéptől. A kifejezés a "**C**ompletely **A**utomated **P**ublic **T**uring test to tell **C**omputers and **H**umans **A**part" (teljesen automatizált nyilvános Turing-teszt a számítógép és az ember megkülönböztetésére) rövidítése. A teszt során a számítógép generál egy feladványt, amit csak egy ember tud helyesen megválaszolni, de a válasz helyességét a gép is könnyedén el tudja dönteni. Ebben az esetbrn egy egyszerű algorítmus ami canvas segítségével kirajzol egy random számsort és húz bele 2db Bézier görbét, ami a számítógépes grafikában gyakran használt parametrikus görbe.

A vektorgrafikában a Bézier görbéket szabadon alakítható sima görbék modellezésére használják. A képszerkesztő programok, mint például az Inkscape, Adobe Photoshop vagy a GIMP a görbe vonalak rajzolásához egymáshoz kapcsolt Bézier görbék sorozatát használják. Ezeket a görbéket nem korlátozza a raszterképek felbontása és interaktívan alakíthatóak.

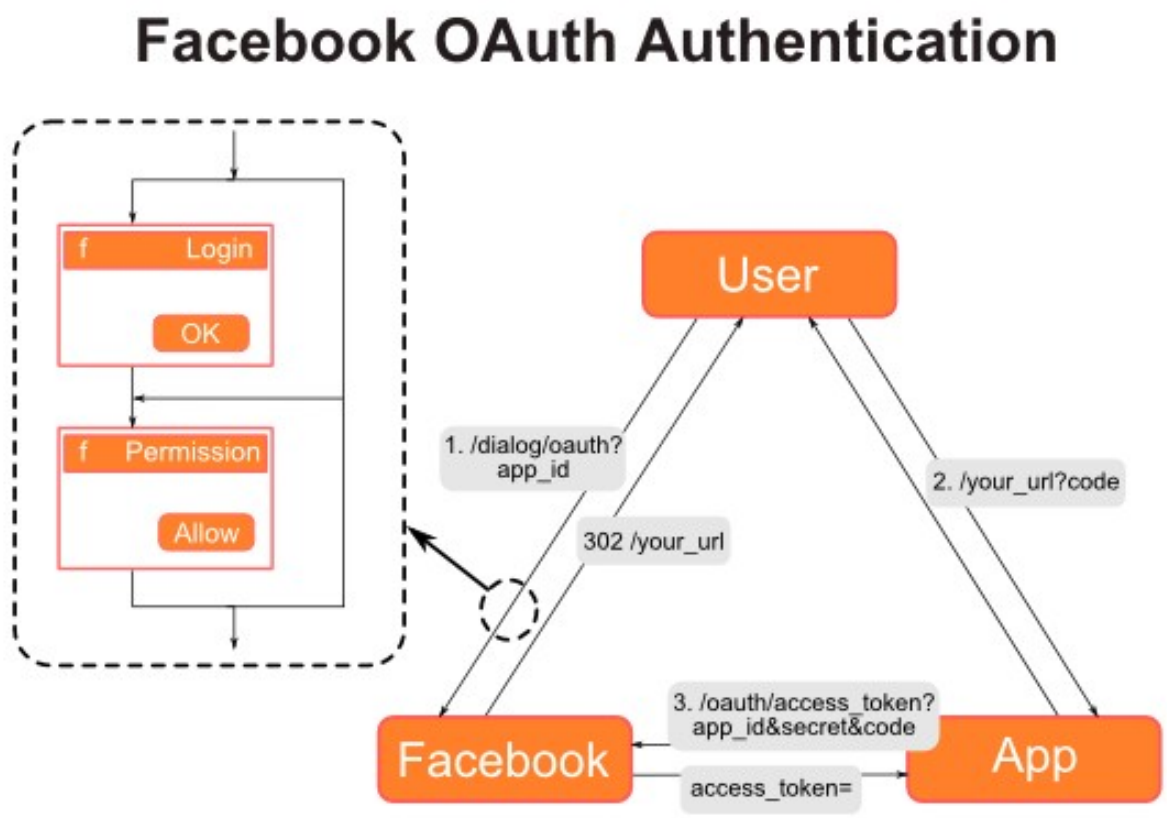
Amennyiben a captcha és a többi adatot helyesen adta meg a user akkor elmenti a rendszer azokat a `user.save()` metódussal, aminek a callback-jében lekezeljük az esetleges hibákat és beléptetjük a felhasználót (persze csak ha nem volt error.

Másik fontos része a user controllernek a beléptetés, ez 3 féleképpen történhet meg local, facebook vagy twitter provider által. Kezdjük a local-lal itt a felhasználó adatait egyszerűen összevetjük az adatbázisban lévőkkel, és amennyiben egyezést találunk, akkor beléptetjük a usert-t. Beállítjuk a sessionjét és lerakjuk a sütijét, ezt a folyamatot a user serializálásának nevezzük.

A Facebook és Twitter beléptetés is hasonlóan működik, gyakorlatilag vehetjük egy kalap alá a kettőt, mivel mindegyik OAuth-ra épül. Az OAuth egy nyílt szabványú engedélyezési protokoll, amely lehetővé teszi, hogy külső felek a felhasználó jelszavának ismerete nélkül hozzáférjenek a felhasználói adatokhoz. Az alkalmazások nem közvetlenül a felhasználók jelszavát használják, hanem az OAuth mechanizmus mint egyfajta „szolgálati kulcs” (token) segítségével férnek hozzá az adatainkhoz, és hajtanak végre műveleteket.

Ezek az alkalmazásokon (app-okon) keresztül kommunikálunk, ami már hozzáfér adatainkhoz, de gyakorlatilag bármihez, de csak akkor ha ezt engedélyezzük azt, illetve mindig figyeljünk arra, hogy ne adjunk ki és ne kérjünk be felesleges adatokat, mert ez sok felhasználóban visszatetszést kelthet. Az app alapú szekvenciális előnye, hogy könnyebb kiszűrni az esetleges adathalász tevékenységeket is.

Az alábbi ábra jól szemléleti az a folyamatot, hogy működik az azonosítás.



Térjünk vissza a gyakorlati oldalára miután megadtuk a facebook-nak, twitternek az appunk beállításait már le tudjuk kérni felhasználók adatait. Ezután, megnézzük, hogy szerepel-e az

adatbázisban, a user amennyiben nem létrehozzuk a felhasználót (migráljuk az accountot a saját rendszerünkbe).

Később már local azonosítással is be tud lépni a user.

Facebook esetén így néz ki:

```
passport.use(new FacebookStrategy({
        clientID: config.facebook.clientID,
        clientSecret: config.facebook.clientSecret,
        callbackURL: config.facebook.callbackURL
    },
    function(accessToken, refreshToken, profile, done) {
      User.findOne({ 'facebook.id': profile.id }, function (err, user) {
        if (err) { return done(err); }
        if (!user) {
          user = new User({
              name: profile.displayName,
              email: profile.emails[0].value,
              username: profile.username,
              provider: 'facebook',
              facebook: profile._json
          });
          user.save(function (err) {
            if (err) console.log(err);
            return done(err, user);
          });
        }
        else {
          return done(err, user);
        }
      });
    }
  ));
```

A felhasználó kezelés lényegében ennyi, ezen kívül még pár kisebb dolog van, mint a Profil oldal és annak a kezelése, hogy bizonyos olalakhoz, ne férjen hozzá jogosulatlan személy ehhez middleware-ket használtam, mielőtt lefutna az oldal kirenderelése megnézi, hogy van-e hozzáférése a látogatónak hozzáférése, ha nincs akkor a login oldalra dobja.

6. Edzés rögzítés

Az oldal másik fontos része, sikeres regisztráció után a dashboard oldalra navigáljuk a felhasználót, itt lehetősége nyílik, megtekinteni mások edzéseit és loggolni saját eredményeit.

6.1. Modell

Adatbázis szinten ez jelenti a másik nagy egységet collection-t (MongoDB esetén a colletion-ok gyakorlatilag a relációs adatbázis tábláinak felelnek meg).

Egy edzés sémája a következőképpen épül fel (`app/models/workout.js`):

```
var WorkoutSchema = new Schema({
  title: {type : String, default : '', trim : true},
  time: {type : Number, default : 60, trim : true},
  body: {type : String, default : '', trim : true},
  user: {type : Schema.ObjectId, ref : 'User'},
  comments: [{
    body: { type : String, default : '' },
    user: { type : Schema.ObjectId, ref : 'User' },
    createdAt: { type : Date, default : Date.now }
  }],
  tags: {type: [], get: getTags, set: setTags},
  createdAt  : {type : Date, default : Date.now}
});
```

Kicsit bonyolultabb, mint a felhasználók, de amit fontos kiemelni, az ennyi `user: {type : Schema.ObjectId, ref : 'User'},` mivel ezzel az egy utasítással, **`ref: 'User'`** fűzzük össze a felhasználót az edzéseivel, gyakorlatilag megfelel egy join-nak relációs adatbázis szinten, viszont annál sokkal gyorsabb, főleg mivel a MongoDB ObjectId alapján tud a leggyorsabban keresni.

6.2. View

Megjelenésben a dashboard a site központi része, ahol megtekintheted mások edzéseit illetve, rögzítheted a saját eredményeid, ami publikálásra kerül az edzésfalra. Itt emelném ki a Jade template Engie adta remek lehetőségeket, mint a layout extendálás és más templatek include-ja. Következőképpen néz ki:

```
extends ../layouts/default // kiterjesztjük a sablonunkat az alap
layout-tal
```

```
include modals/new-workout //behúzzuk a bootstrap-es modal ablakot
```

6.3. Controller

A controller része viszonylag egyszerű, de több részre tagolódik külön vettem, magát

az edzésre vonatkozó (workouts), a kommentező (comments), és a tag kezelő (tags) controllert, mert így sokkal átláthatóbb.

A workouts-ban található az edzés elkészítésének folyamata, amely 2 lényeges részre bontható, az edzés készítés és megjelenítés. Készítés során formból felolvassa az adatokat a rendszer majd elmenti az adatbázisba, megjelenítéskor pedig időrendbe bontva (legújabbak a legelején) kilistázza azokat.

Példa a saját edzéseim listázására:

```
var userId = req.user._id;
Workout
    .find({ user: userId })
    .sort({createdAt: -1 })
    .populate('user')
    .exec(function(err, workouts) {
      if (err) return res.render('500');
      res.render('workouts/my', {
        title: 'Saját edzéseim',
        workouts: workouts
      });
    });
```

Itt fontos megjegyezni a Mongoose `populate` metódusát, mely összefűzi a 2 collection-t (workouts, users) ezáltal elérhetővé válnak az edzésen belül a hozzá kapcsolódó felhasználói adatok.

A kommentezés során az edzésekhez füzünk egy kisebb egységet, mint ahogy a modellben is látható. Lekérjük a felhasználót és a request-ben szereplő edzést, majd hozzáfűzzük az általa megadott szöveget (persze, csak ha minden adat passzol).

```
exports.create = function (req, res) {
  var workout = req.workout;
  var user = req.user;

  if (!req.body.body) return res.redirect('/workouts/'+ workout.id);
  workout.comments.push({
    body: req.body.body,
    user: user._id
  });
  workout.save(function (err) {
    if (err) return res.render('500');
    res.redirect('/workouts/'+ workout.id);
  });
}
```

A tag-ek Stringből vesszővel elválasztott egységenként képzett tömbelemek. Így könnyű őket szeparálni és lekérni azokat az edzéseket, amelyekben szerepelnek.

```
Workout.list(options, function(err, workouts) {
  if (err) return res.render('500');
  Workout.count({ tags: req.param('tag') }).exec(function (err, count) {
    res.render('workouts/index', {
      title: 'Tag: ' + tag,
      workouts: workouts,
      page: page,
      count: count,
      pages: count / perPage
    });
  });
});
```

A jelenlegi felépítés (MVC) előnye, hogy nagyon könnyen lehet hozzáfejleszteni további controllereket és kibővíteni a jelenlegieket.

7. Továbbfejlesztési lehetőségek

Nagyon sok irányba lehetne továbbfejleszteni az alkalmazást, kiterjeszteni a dashboard-on található statisztikát további adatokkal, illetve edzésteljesítményt is lehetne vizsgálni, ehhez szét kéne szeparálni a workouts szekciót, mivel fel kell venni, hogy milyen gyakorlatot, milyen teljesítménnyel tudott teljesíteni a felhasználó (pl.: megtett táv, megemelt súly, elvégzett ismétlésszám). A jelenlegi rendszer erőssége, hogy gyakorlatilag bármilyen sportteljesítmény követésére használható.

Amennyiben specializálódna a rendszer egy bizonyos sportra (crossfit, erőemelés, bodybuilding, úszás, futás....) akkor az ehhez tartozó gyakorlatok bemutatására szolgáló, edzésgyakorlatok szekcióval is bővíteni lehetne az oldalt.

Illetve edzésed mellett, lehetne az étkezést is loggolni, mivel ennek is komoly hatása van a sportteljesítményre.

Az alkalmazásba be van építve a Socket.IO ami segítésével realtime lehet frissíteni a weboldal elemeit, kb, mint a facebook-nál, ha valaki ír egy kommentet vagy bejegyzést, az rögtön megjelenik a dashboard-on. Ennek a kidolgozása több időt vesz igénybe illetve nem lehet minden adatot folyamatosan socket-ken keresztül küldeni, mert feleslegesen terhelnénk vele a szervert, így jó átgondoltan kell használni, de a dashbord elemeire érdemes lenne alkalmazni.

Rendkívül sok irányba és sokféleképpen lehetne továbbfejleszteni a rendszert számtalan olyan technológia létezik mára mely segítségével „realtime” folyamatosan élő weboldalt lehet létrehozni, a LeightWeight Logger esetén ez lenne a cél.

8. Konklúzió

A LeightWeight Logger egy nagyon kompakt, sok irányba fejleszthető alkalmazás lett, tartalmaz minden olyan technológia elvárást, amit egy mai modern site-tól elvárunk, mind felépítés, mint megjelenés szempontjából.

Maga az oldal fejlesztése nem volt kis feladat, hiszen a hozzá használt eszközöket (NodeJS, MongoDb) eddig nem volt alkalmam használni, hiszen a hazai webfejlesztéssel foglalkozó cégek talán 99% PHP vagy Java alapokon fejleszt a hozzájuk választott adatbázis-kezelő rendszerrel, ami általában egy relációs adatbázis MySQL vagy Oracle.

Az oldal elkészítése során rengeteget tanultam, és egyre több lehetőséget látok a NodeJS-ben illetve a MongoDB-ben. Tisztában vagyok a rendszerek korlátaival, ugyanakkor nagyobb volumenű projekteket biztos, hogy nem node-dal fejlesztenék, hiszen megvannak a maga korlátai, de valószínűleg a közeljövőben, mint a NodeJS mint a MongoDB jobban el fognak terjedni. Sajnos magyarul még nagyon kevés dokumentáció érhető el hozzájuk, de angol nyelven már elég sok leírást lehet találni.

Örülök neki, hogy belevágtam az oldal elkészítésbe és megismertem ezeket technológiákat, remélem rajtam kívül sokan fognak érdeklődni iránt uk.

9. Irodalomjegyzék

http://blog.kodfejtok.hu/mongodb-elso-lepesek/

http://expressjs.com/api.html

http://mongoosejs.com/docs/guide.html

http://weblabor.hu/cikkek/nodejs-alapok

http://hu.wikipedia.org/wiki/MVC

http://hu.wikipedia.org/wiki/B%C3%A9zier_g%C3%B6rbe

http://hu.wikipedia.org/wiki/OAuth

http://hu.wikipedia.org/wiki/Captcha