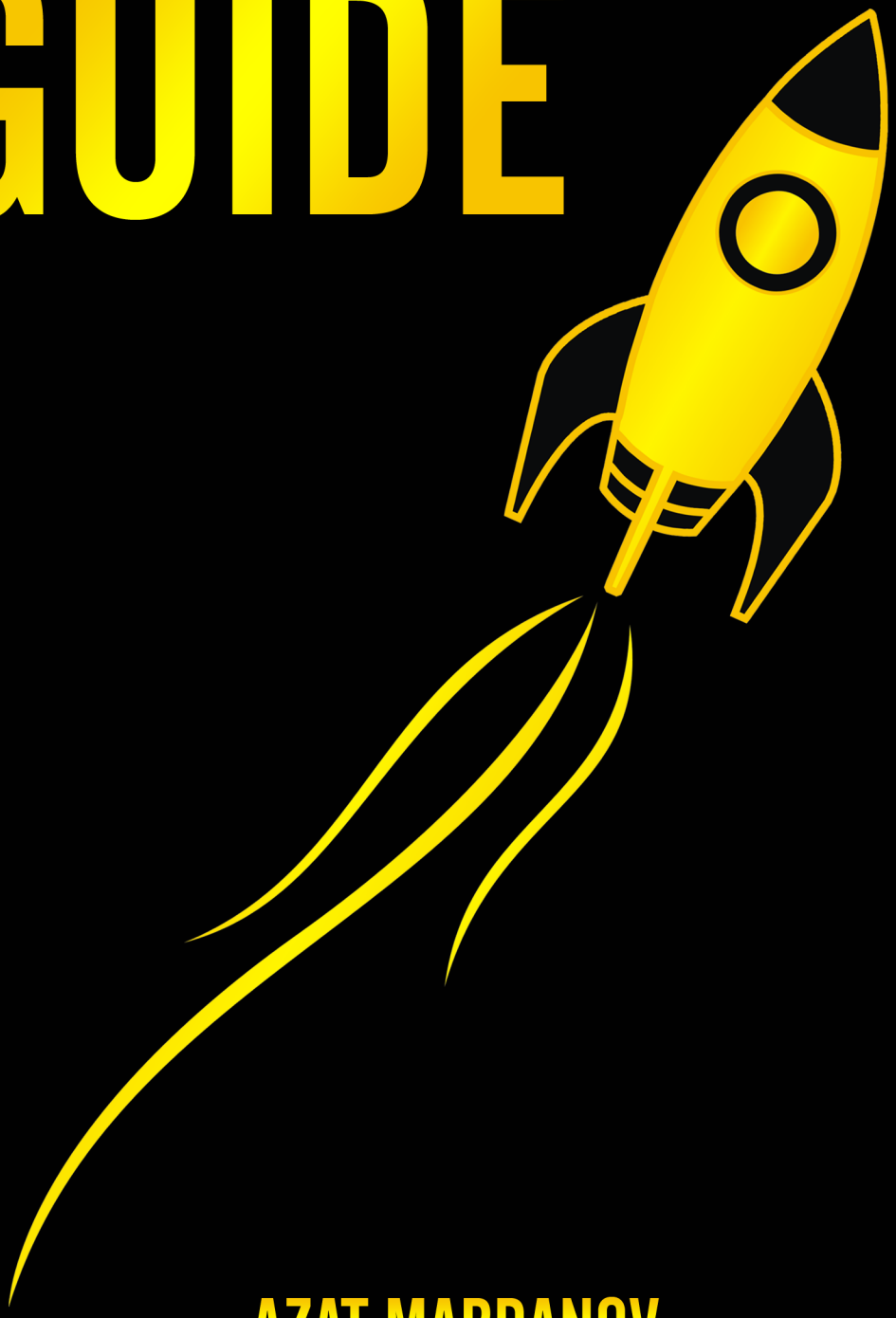
THE MOST POPULAR NODE.JS WEB FRAMEWORK MANUAL

EXPRESS.JS GUIDE



AZAT MARDANOV

Express.js Guide

The Most Popular Node.js Web Framework Manual

Azat Mardanov

This book is for sale at http://leanpub.com/express

This version was published on 2013-10-26

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Azat Mardanov by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I'm reading Express.js Guide — the most popular Node.js framework's manual by @azat_co #RPJS

The suggested hashtag for this book is #RPJS.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#RPJS

Also By Azat Mardanov

Rapid Prototyping with JS

Oh My JS

To Fyodor Mikhailovich Dostoyevsky and The Idiot

Contents

Foreword

Dear reader, you are holding a book which will open you to understanding and fluent usage of the Express.js framework - standard de facto in web application programming on Node.js. And I would especially recommend this book because it was written by a practicing engineer, one who has a comprehensive knowledge about the full stack of web application development and Express.js in particular.

Azat and I worked on the same Node.js/Express.js code base at Storify[1] — the social media curation tool that Washington Post, CNN, BBC, The White House and other news corps use — which was recently acquired by LiveFyre[2]. Right before the Express.js Guide release, he asked me to write the foreword, because it'll sound objective, sincere and unbiased coming from the creator of **another** Node.js framework — CompoundJS[3].

However, nobody is reading forewords. So, instead of a foreword, I'll share my story. Actually, I never thought it was worth sharing and there's definitely nothing exciting about it. But from the other point of view — thousands of young programmers living similar ordinary lives - it could be inspiring: it's a common story, but a kind of successful one.

My way to web development started when I was a student and joined a team as junior PHP programmer. I was working here for about 5 years and the main lesson I have learned was: education is nothing compared to real work experience. The next page of my professional life was work in outsourcing (PHP and Ruby on Rails). And then I found Node.js.

It was something that I always wanted: processes that do not have to wait for DB/IO operations which are keeping all the resources, but doing something useful instead. This is the simple reason why I started using it; it's more efficient compared to synchronous programming environments. By "efficient" I do not mean not speed of processing, but more flexibility in programming style.

As a good example of this flexibility, I can share some solutions I recently programmed for a Redis adapter for the Jugglingdb ORM[4]. Problem: during peaks of website usage, we are running a lot of db queries to serve pages, and most of the queries are the same. The obvious solution is to cache results of the queries, but this solution requires additional coding and some logic for cache invalidation. We've come with a better solution: cache queries and not results. When a query comes, we don't execute it immediately; instead, we wait for some time, collect identical queries, then execute query once and run multiple callbacks to serve all clients. This solution is simple and requires no additional logic. As a result, we have flat db usage even during peaks. This solution is natural in Node.js, and that's why Node.js rocks!

---

[1] http://storify.com

[2] http://livefyre.com

[3] http://compoundjs.com

[4] https://github.com/1602/jugglingdb

Life after discovering Node.js was great, full of interesting challenges and work, but one thing was annoying: each time I start a new project, I have to do almost the same work to organize code. For me as a Rails developer, it was really great to be able to create well-structured MVC applications fast, generate scaffolding controllers / views and other stuff. But this kind of tool was missing in Node.js and that's why I spent my Christmas holidays writing it; the project was called express-on-railway at first, then RailwayJS, then CompoundJS.

The main goal of this project was to bring structure to an Express.js application, add the ability to extend applications in a standard way, and generate application code. So, it was not a new framework, but just Express.js with decent MVC structure, which is good for developers who don't need to learn anything but Express.js to be able to understand what's going on in CompoundJS application. And it was a kind of piggybacking on Express.js and Rails experience: the idea was to get best ideas from rails and bring to node platform, and Express.js was selected as the base because it is the most popular framework for Node.js and has a relatively big community, so I won't be alone with my "new framework". It was the start of my open-source years, which completely changed my attitude to programming and all matters, but this is another story.

And what can I say to conclude: web development in Node.js started with Express.js. It is a minimalistic and a robust framework which gives you all you need to build decent web applications. Even if you decide to move to some more advanced frameworks at some point, Express.js knowledge is a basic skill you have to learn. In addition, this book contains everything you need to know to start using Express.js and clearly explains all concepts and answers to most of the questions that newcomers usually ask. For these reasons, this book is a must read!

–

Anatoliy Chakkaev,

Creator of CompoundJS[5] and JugglingDB[6]

[5] http://compoundjs.com

[6] http://jugglingdb.co/

Acknowledgment

This book would not be possible without the existence of my parents, the Internet and JavaScript. Also, words cannot express my gratitude to Tony Phillips, Shawn Drost, Douglas Calhoun and Marcus Phillips for founding HackReactor and helping many people venture into new careers.

In addition to that special thanks to General Assembly, pariSOMA and Marakana for giving me opportunities to test my instructions; to Peter Armstrong for LeanPub; to Sahil Lavingia for Gumroad; to Daring Fireball for Markdown; to Metaclassy for Byword; to Fred Zirdung for advices; to Janet Williams for editing; and to Rachmad Adv for the splendid cover!

Introduction

In this part, we'll go over why this book was written, who might benefit from the book and how to use it most efficiently.

Why

Express.js is the most popular Node.js web framework yet. As of this writing (September of 2013), there are no books that are solely dedicated to it. Its official website has bits of insights for advanced Node.js programmers. However, I found that many people — including those who go through HackReactor[7] program and come to my Node.js classes at General Assembly and pariSOMA — are interested in a comprehensive resource. The one that would cover all the different components of Express.js work together in a real production-like application. The goal of Express.js Guide is to become such resource.

What This Book is

Express.js Guide is a concise book on **one** particular library. This book contains Express.js API 3.3.5[8] description, the best practices on code organization and patterns, real-world examples of web apps. The topics include but not limited to middleware, command-line interface and scaffolding, rendering templates, extracting params from dynamic URLs, parsing payloads and cookies, managing authentication with sessions, error handling and prepping apps for production.

For more details and for what exactly the book covers, please refer to the **Table of Content**.

What This Book is Not

This book is **not** an introduction to Node.js. Nor is it a book that covers all aspects of building a modern day web application, e.g., websockets, databases and (of course) front-end development. Keep in mind that readers also **won't find** in Express.js Guide a resource for learning programming and/or JavaScript fundamentals.

You might want to take a look at Rapid Prototyping with JS[9] for the introduction to Node.js, MongoDB and front-end development with Backbone.js.

[7] http://hackreactor.com

[8] https://github.com/visionmedia/express/tree/3.3.5

[9] http://rpjs.co

In the real-world and especially in Node.js development, due to its modularized philosophy, we seldom use just a single framework. In the book, we have tried to stick only to Express.js and leave everything else out as much as possible, without compromising the usefulness of examples. Therefore, we intentionally left out some important chunks of web developments, for example databases, authentication and testing. Although these elements are present in tutorials and examples, they're not explained in details. For those materials, you could check books in the related reading list section at the end of the book.

Who This Book is For

This book is for people fluent in programming and front-end JavaScript. In addition, to get the most benefits, readers must be familiar with basic Node.js concepts like `process` and `global`, and know core modules, including streams, clusters and buffer type.

If you're thinking of starting a Node.js app, or of re-writing an existing one, and your weapon of choice is Express.js — this guide is for you! It will answer most of your "how" and "why" questions.

Notation

The code blocks will look like this: `console.log('Hello reader!');`. The terminal and console commands will have prefixes `$` and `›` correspondingly.

If we mention and use a specific file or a folder, those names will be monospaced as well: `index.js`.

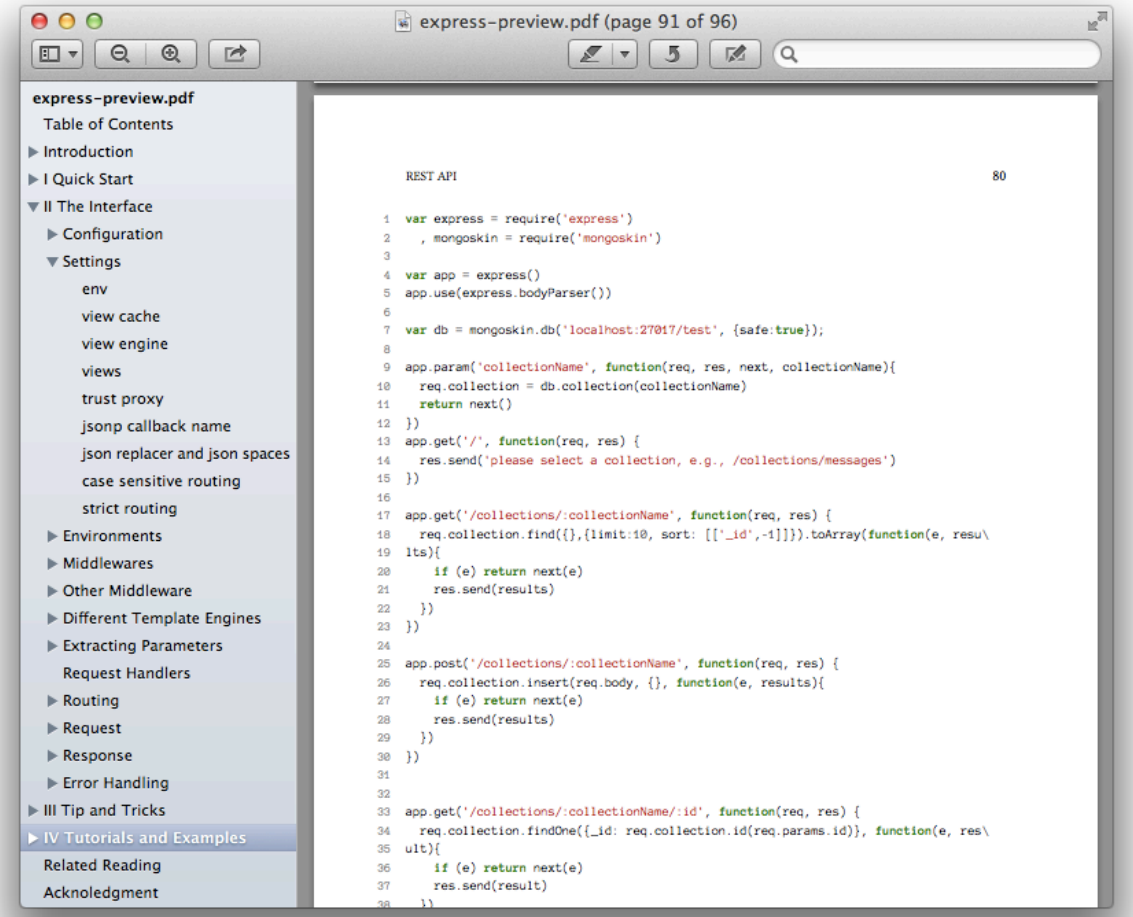
If there is a class or a module name in the text, it'll be emboldened, e.g., **superagent**.

The new resources will be hot-linked only the first time we mention them.

Navigation

In PDF, EPUB/iPad and Mobi/Kindle versions, you could use the Table of Content, which is in the beginning of the book and has internal links, to jump to the most interesting parts or chapters.

For faster navigation between parts, chapters and sections of the book, please use the book's navigation pane which is based on the Table of Contents (the screenshot is below).



The Table of Contents pane in the Mac OS X Preview app.

PDF version of the book is suitable for printing on US Letter paper because all links are in the footnotes.

How to Use the Book

If you're contemplating using Express.js or writing an app with it, read the whole book from top to bottom. However, if you're somewhat familiar with the framework, jump straight to the question at hand. You can also skim through them later and more advanced chapters that deal with code organization and getting apps production.

Examples

The book contains code snippets and run-ready examples. The latter is available in GitHub repository at github.com/azat-co/expressjsguide[10]. Additional examples are in http://github.com/azat-co/todo-express[11], github.com/azat-co/sfy-gallery[12] and github.com/azat-co/hackhall[13].

The provided examples were written and tested with the specific versions of dependencies only. Because Node.js and its ecosystem of modules are being developed rapidly, please pay attention whether the new versions have breaking changes or not. Here is the list of versions that we used:

- Express.js v3.3.5
- Google Chrome Version 28.0.1500.95
- Node.js v0.10.12
- MongoDB v2.2.2
- Redis v2.6.7
- NPM v1.2.32

[10] http://github.com/azat-co/expressjsguide
[11] http://github.com/azat-co/todo-express
[12] http://github.com/azat-co/sfy-gallery
[13] http://github.com/azat-co/expressjsguide

About the Author

Azat Mardanov: a software engineer, an author and a yogi.

Azat Mardanov has over 12 years of experience in web, mobile and software development. With a Bachelor's Degree in Informatics and a Master of Science in Information Systems Technology degree, Azat possesses deep academic knowledge as well as extensive practical experience.

Currently, Azat works as a Senior Software Engineer at DocuSign[14], where his team re-builds 50 million user product (DocuSign web app) using the ech stack of Node.js, Express.js, Backbone.js, CoffeeScript, Jade, Stylus and Redis.

Recently, he worked as an engineer at the curated social media news aggregator website, Storify.com[15] (acquired by LiveFyre[16]). Before that, Azat worked as a CTO/co-founder at Gizmo[17] — an en-

[14] http://docusign.com
[15] http://storify.com
[16] http://livefyre.com
[17] http://www.crunchbase.com/company/gizmo

terprise cloud platform for mobile marketing campaigns, and has undertaken the prestigious 500 Startups[18] business accelerator program. Previously, he was developing mission-critical applications for government agencies in Washington, DC: National Institutes of Health[19], National Center for Biotechnology Information[20], Federal Deposit Insurance Corporation[21], and Lockheed Martin[22]. Azat is a frequent attendee at Bay Area tech meet-ups and hackathons (AngelHack[23] hackathon '12 finalist with team FashionMetric.com[24]).

In addition, Azat teaches technical classes at General Assembly[25] and Hack Reactor[26], pariSOMA[27] and Marakana[28] (acquired by Twitter) to much acclaim.

In his spare time, he writes about technology on his blog: webAppLog.com[29] which is number one[30] in "express.js tutorial" Google search results. Azat is also the author of Express.js Guide[31], Rapid Prototyping with JS[32] and Oh My JS![33].

Contact us

Let's be Friends on the Internet!

- Tweet Node.js question on Twitter: @azat_co[34]
- Follow on Facebook: facebook.com/azat.mardanov[35]
- Website: expressjsguide.com[36]
- GitHub: github.com/azat-co/rpjs[37]

Other Ways to Reach Us

- Email Azat directly: hi@azat.co[38]

[18] http://500.co/
[19] http://nih.gov
[20] http://ncbi.nlm.nih.gov
[21] http://fdic.gov
[22] http://lockheedmartin.com
[23] http://angelhack.com
[24] http://fashionmetric.com
[25] http://generalassemb.ly
[26] http://hackreactor.com
[27] http://parisoma.com
[28] http://marakana.com
[29] http://webapplog.com
[30] http://expressjsguide.com/assets/img/expressjs-tutorial.png
[31] http://expressjsguide.com
[32] http://rpjs.co
[33] http://leanpub.com/ohmyjs
[34] https://twitter.com/azat_co
[35] https://www.facebook.com/azat.mardanov
[36] http://expressjsguide.com/
[37] https://github.com/azat-co/expressjsguide
[38] mailto:hi@azat.co

- Google Group: rpjs@googlegroups.com[39] and https://groups.google.com/forum/#!forum/rpjs
- Blog: webapplog.com[40]
- HackHall[41]: community for hackers, hipsters and pirates

Share on Twitter with ClickToTweet link: http://clicktotweet.com/HDUx0, or just click:

> "I'm reading Express.js Guide — the most popular Node.js framework's manual by @azat_co #RPJS"[42]

[39] mailto:rpjs@googlegroups.com
[40] http://webapplog.com
[41] http://hackhall.com
[42] http://clicktotweet.com/HDUx0

I Quick Start

In this part, we'll briefly go over what Express.js is, install it, build a few simple applications and begin to touch on configurations.

1 What is Express.js?

Express.js is a web framework which is based on the core Node.js **http** module and Connect[1] components. Those components are called middleware. They are the cornerstone of the framework's philosophy, i.e., *configuration over convention*. In other words, developers are free to pick whatever libraries they need for a particular project which provides them with flexibility and high customization.

If you wrote any serious apps using only core Node.js modules, you most likely found yourself re-inventing the wheel by constantly writing same code for the similar tasks, such as

- parsing of HTTP request bodies
- parsing of cookies
- managing sessions
- organizing routes with a chain of `if` condition based on URL paths and HTTP methods of the requests
- determining proper response headers based on data types

Express.js solves these and many other problems, as well as providing an MVC-like structure for your web apps. Those apps could vary from bare-bone back-end-only REST APIs to full-blown highly scalable full-stack (with jade-browser[2] and Socket.IO[3]) real-time web apps.

Some developers familiar with Ruby compare Express.js to Sinatra which has a very different approach to the Ruby on Rails framework.

[1] http://www.senchalabs.org/connect/

[2] https://npmjs.org/package/jade-browser

[3] http://socket.io

2 How Express.js Works

Express.js usually has an entry point, a.k.a., a main file. In that file, we do the following steps:

1. Include third-party dependencies as well as our own modules such as controllers, utilities, helpers and models
2. Configure Express.js app settings such as template engine and its files extension
3. Define middlewares such as error handlers, static files folder, cookies and other parsers
4. Define routes
5. Connect to databases such as MongoDB, Redis or MySQL
6. Start the app

In the advanced setup, we might have to expose the app as a module, and use up-time[1] module. We can also utilize recently added clusters (as outline in the Tips and Tricks part) to spawn multiple workers.

Tip

Here is a small nice tutorial on **up-time** module Staying up with Node.JS[2].

When Express.js app is running, it listens to requests. Each incoming request is processed according to defined chain of middlewares and routes starting from top to bottom. This aspect is **important** in controlling the execution flow. For example, we can have multiple functions handling each request. Some of those functions will be in the middle, hence the name middleware:

1. Parse cookie information and go to the next step when done
2. Parse parameters from the URL and go to the next step when done
3. Get the information from the database based on the value of the parameter if user is authorized (cookie/session), and go to the next step if there is a match
4. Display the data and end the response

---

[1] https://npmjs.org/package/up-time

[2] http://www.devthought.com/2012/01/29/staying-up-with-node-js/

3 Installation

Express.js comes in two versions:

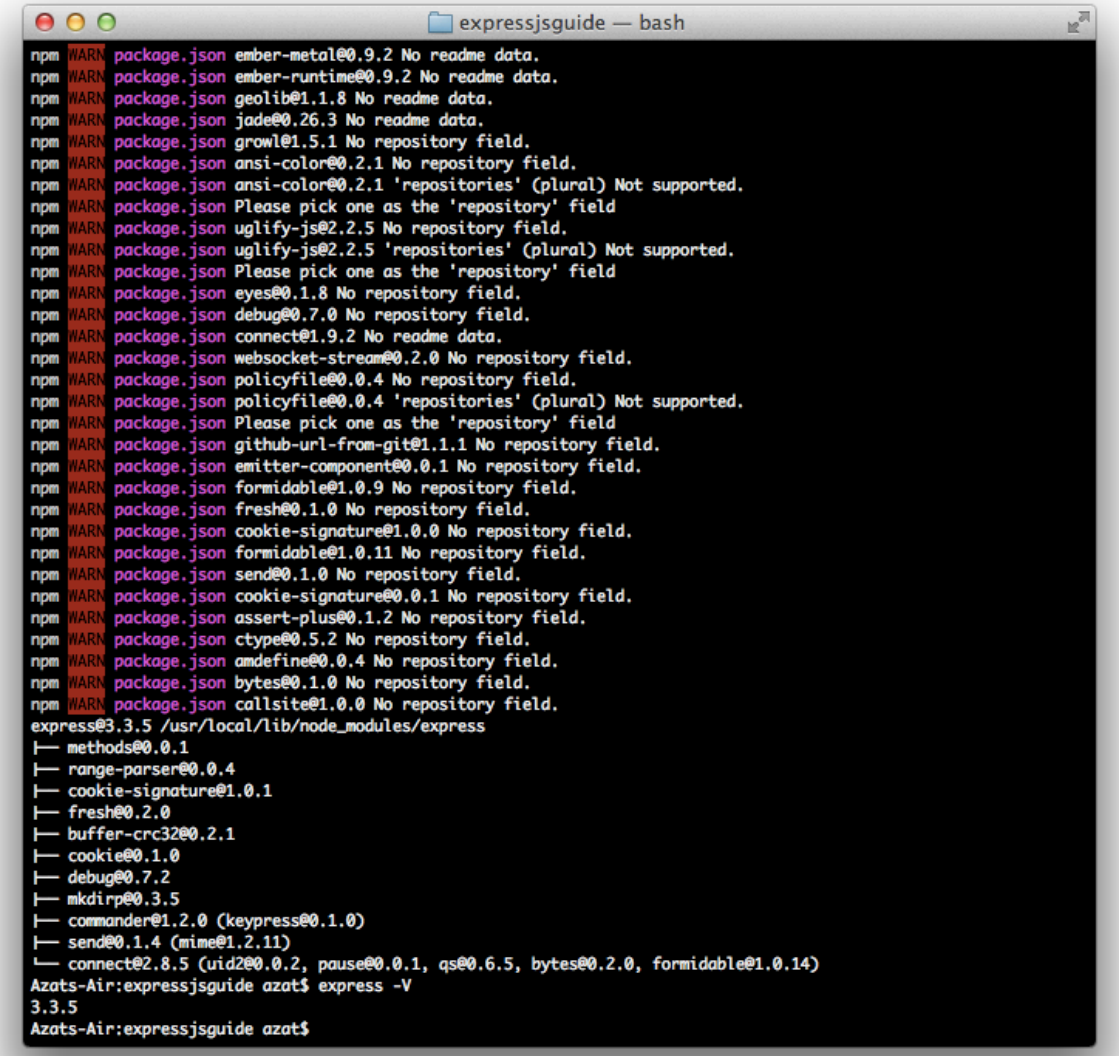
1. Command line tool for scaffolding
2. Module in your Node.js app, i.e., actual dependency

To install the former, run `$ npm install -g express` from anywhere on your Mac/Linux machine. This will download and link `$ express` terminal command to the proper path so that we can later access its CLI for creation of new apps.

Of course, we can be more specific and tell NPM to install v3.3.5: `$ npm install -g express@3.3.5`.

Note

Most likely, your system will require root/administrator right to write to the folder. In this case, you'll need `$ sudo npm install -g express`.

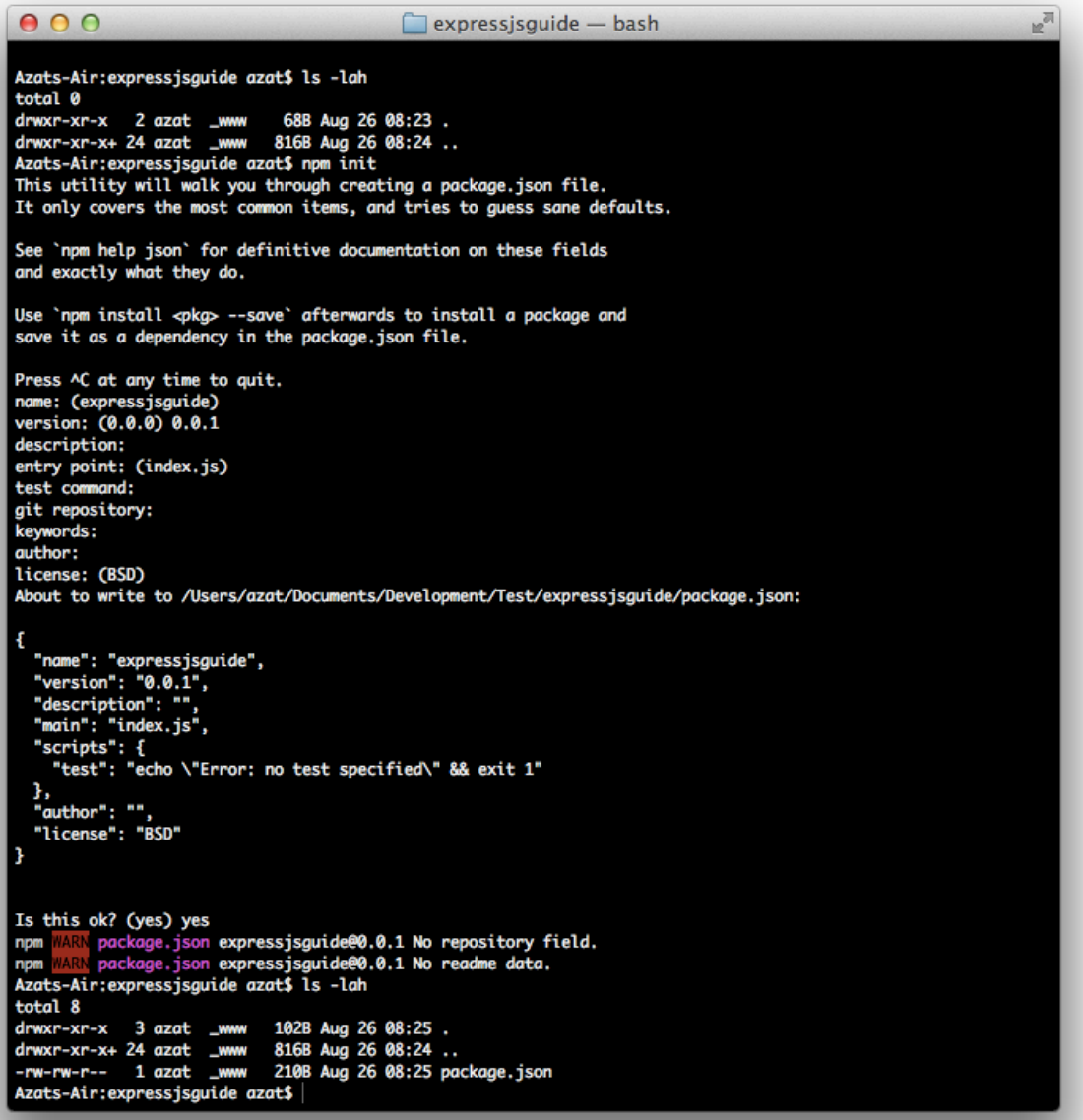


The result of running NPM with `-g` and `$ express -V`.

Please notice the path in the figure above which is `/usr/local/lib/node_modules/express`.

For the latter, i.e., local Express.js module installation as a dependency, let's create a new folder `$ mkdir expressjsguide`. This will be our project folder for the book. Now, we can open it with `$ cd expressjsguide`.

Once we are inside the project folder, we can create `package.json` manually in a text editor or with the `$ npm init` terminal command.



The result of running `$ npm init`

Example of the `package.json` file with vanilla ‘ $ npm init‘ options :

```
{
  "name": "expressjsguide",
  "version": "0.0.1",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "BSD"
}
```

Finally, we can install the module utilizing NPM:

```
$ npm install express
```

Or, if we want to be specific which is a good idea:

```
$ npm install express@3.3.5
```

Note

If you attempt to run the aforementioned `$ npm install express` command without `package.json` file or `node_modules` folder, the *smart* NPM will traverse up the directory tree to the folder that has either of those two things. This behavior somewhat mimics Git's logic. For more information on NPM installation algorithm, please refer to the official documentation[1].

Alternatively, we can update the `package.json` file by specifying the dependency (`"express": "3.3.x"` or `"express": "3.3.5"`) and run $ npm install‘.

The `package.json` file with an added Express.js v3.3.5 dependency:

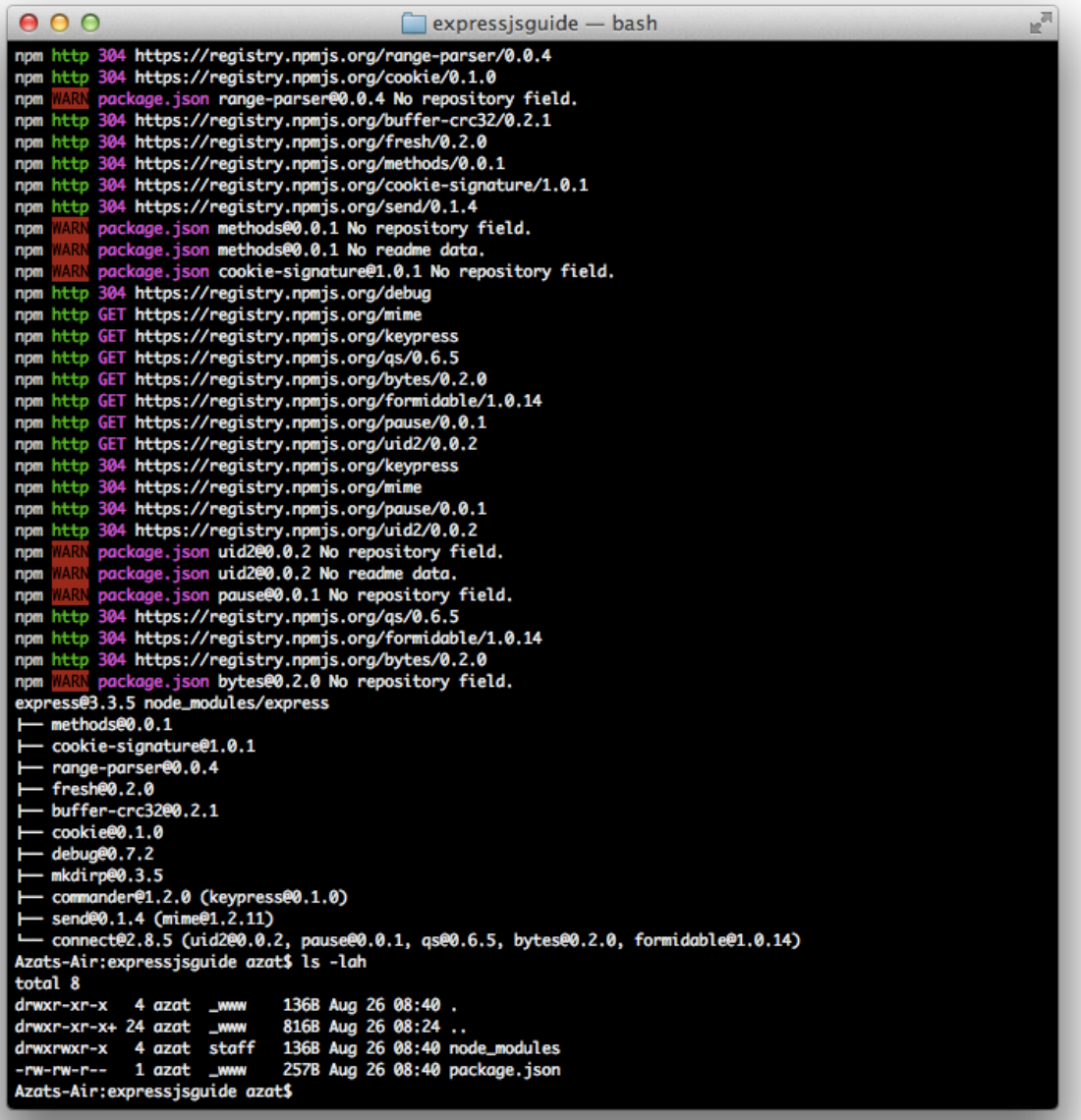
```
{
  "name": "expressjsguide",
  "version": "0.0.1",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "dependencies": {
    "express": "3.3.5"
```

[1] https://npmjs.org/doc/folders.html

```
  },
  "author": "",
  "license": "BSD"
}
```

```
$ npm install
```

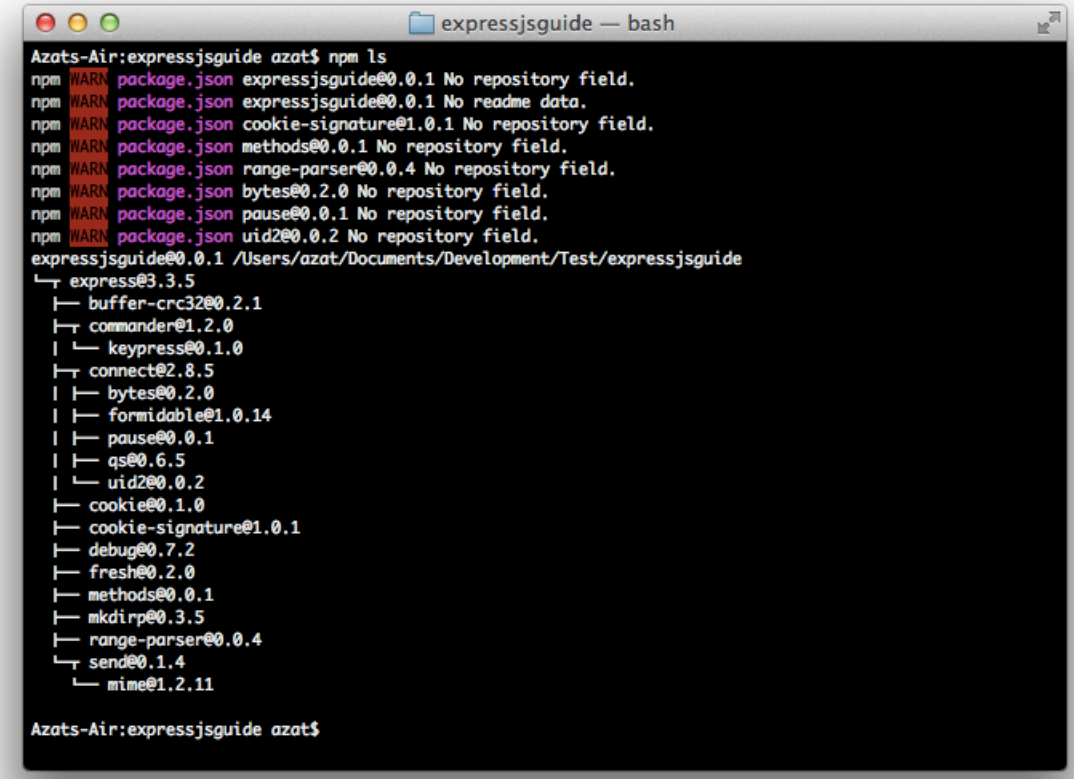
Please notice the path after the `express@3.3.5` string in this figure:



The result of running `$ npm install`.

If you want to install Express.js to an existing project and save the dependency (smart thing to do!) into the `package.json` file — which is already present in that project's folder — run `$ npm install express --save`.

To double check the installation of Express.js and its dependencies, we can run `$ npm ls` command:



The result of running `$ npm ls`.

4 Hello World Example

In this chapter, to get our feet wet, we'll build the quintessential programming example the "Hello World" app. If you've done so already via online tutorial, feel free to skip to the later chapters of the book (The Interface).

In the folder `expressjsguide`, create a `hello.js` file. Use your favorite text editor such as VIM, Emacs, Sublime Text 2 or TextMate. The file `hello.js` server will utilize Express.js; therefore, let's include this library:

```
1  var express = require('express');
```

Now we can create an application:

```
1  var app = express();
```

The application is a web server that will run locally on port 3000:

```
1  var port = 3000;
```

Let's define *a wildcard route* (`*`) with `app.get()` function:

```
1  app.get('*', function(req, res){
2    res.end('Hello World');
3  });
```

The `app.get()` function above accepts regular expressions[1] of the URL patterns in a string format. In our example, we're processing all URLs with the wildcard `*` character.

The second parameter to the `app.get()` is *a request handler*. A typical Express.js request handler is similar to the one we pass as a callback to the native/core Node.js `http.createServer()` method.

For those unfamiliar with the core **http** module, request handler is a function that will be executed every time the server receives a particular request, usually defined by HTTP method, e.g., `GET`, and the URL path, i.e., URL without the protocol, host and port.

Express.js request handler needs at least two parameters — more on this later in the Error Handling section — request or simply `req`, and response or just `res`. Similarly to their core counterparts, we can utilize readable and writable streams interfaces[2] via `res.pipe()` and/or `res.on('data', function(chunk) {...})`.

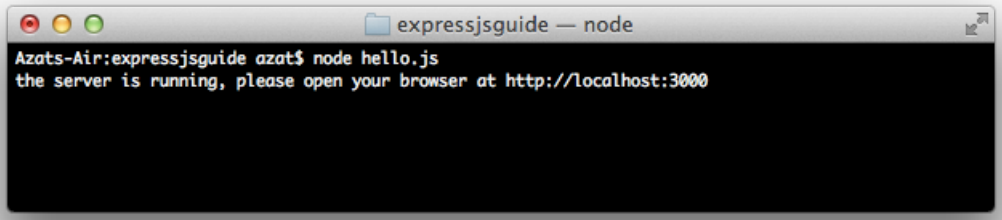
Lastly, we start the Express.js web server and output a user-friendly terminal message in a callback:

[1] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions

[2] http://nodejs.org/api/stream.html

```
app.listen(port, function(){
  console.log('The server is running, ' +
    ' please, open your browser at http://localhost:%s',
    port);
});
```

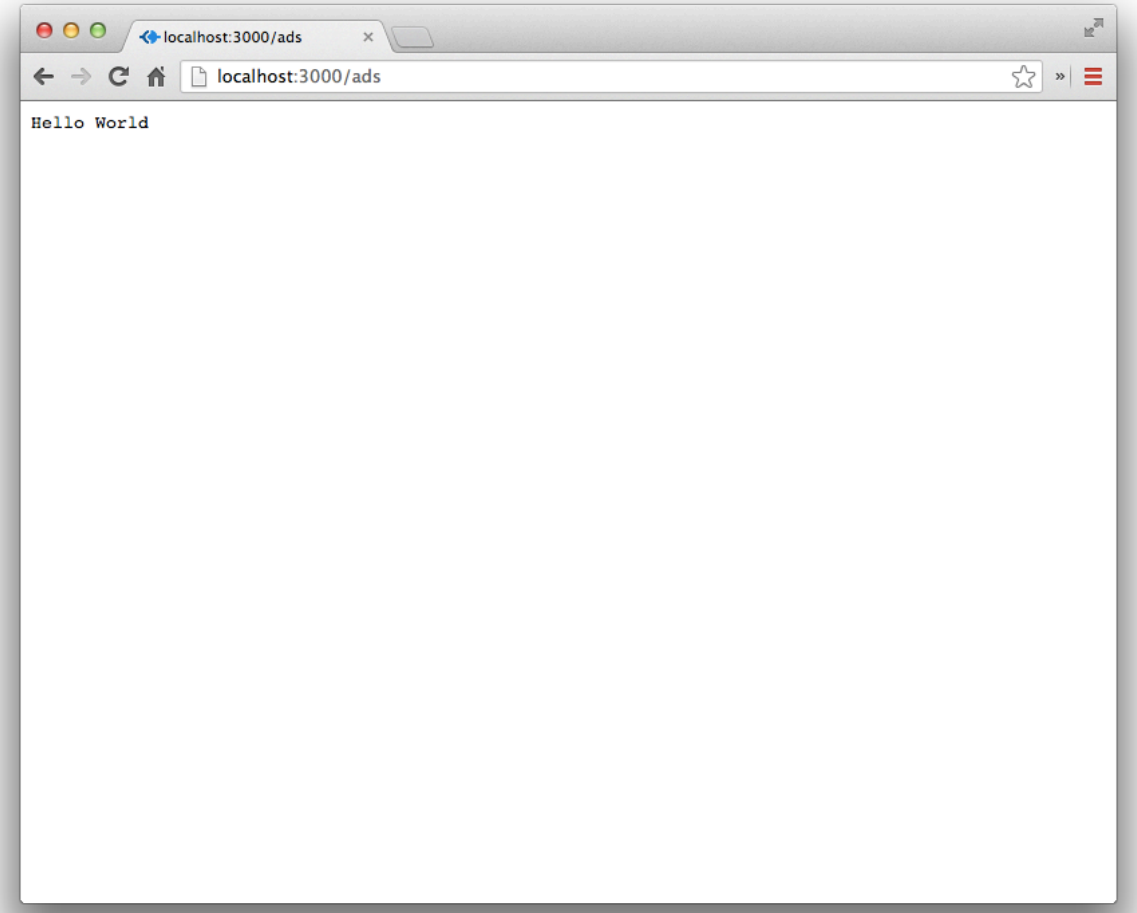
To run the script, we execute `$ node hello.js` from the project folder:



The result of running `$ node hello.js`

Now, if you open your browser at http://localhost:3000 (same as http://127.0.0.1:3000[3] or http://0.0.0.0:3000/[4]), you should see the "Hello World" message no matter what the URL path is:

[3] http://127.0.0.1:3000
[4] http://0.0.0.0:3000



Browser opened at http://localhost:3000/ads.

The full code of `hello.js` file for your reference as well as in the GitHub repository[5]:

```
1  var express = require('express');
2  var port = 3000;
3  var app = express();
4
5  app.get('*', function(req, res){
6    res.end('Hello World');
7  });
8
9  app.listen(port, function(){
10   console.log('The server is running, ' +
```

[5] https://github.com/azat-co/expressjsguide/blob/master/hello.js

```
    ' please open your browser at http://localhost:%s',
     port);
});
```

We can make our example a bit more interactive by echoing the name that we provide to the server along with the "Hello" phrase. To do so, we can copy `hello.js` file with `cp hello.js hello-name.js` and add the following route **before** the all-encompassing route from the previous example:

```
app.get('/name/:user_name', function(req,res) {
  res.status(200);
  res.set('Content-type', 'text/html');
  res.send('<html><body>' +
    '<h1>Hello ' + req.params.user_name + '</h1>' +
    '</body></html>'
  );
});
```

Inside of the `/name/:name_route` route, we set the proper HTTP status code (200 means *okay*), HTTP response headers and wrap our dynamic text in HTML `body` and `h1` tags.

Note

`res.send()` is a special Express.js method that conveniently goes beyond what our old friend from core **http** module `res.end()` does. For example, the former automatically adds `Content-Length` HTTP header for us. It also augments `Content-Type` based on the data provided to it — more on this later in the Response chapter.

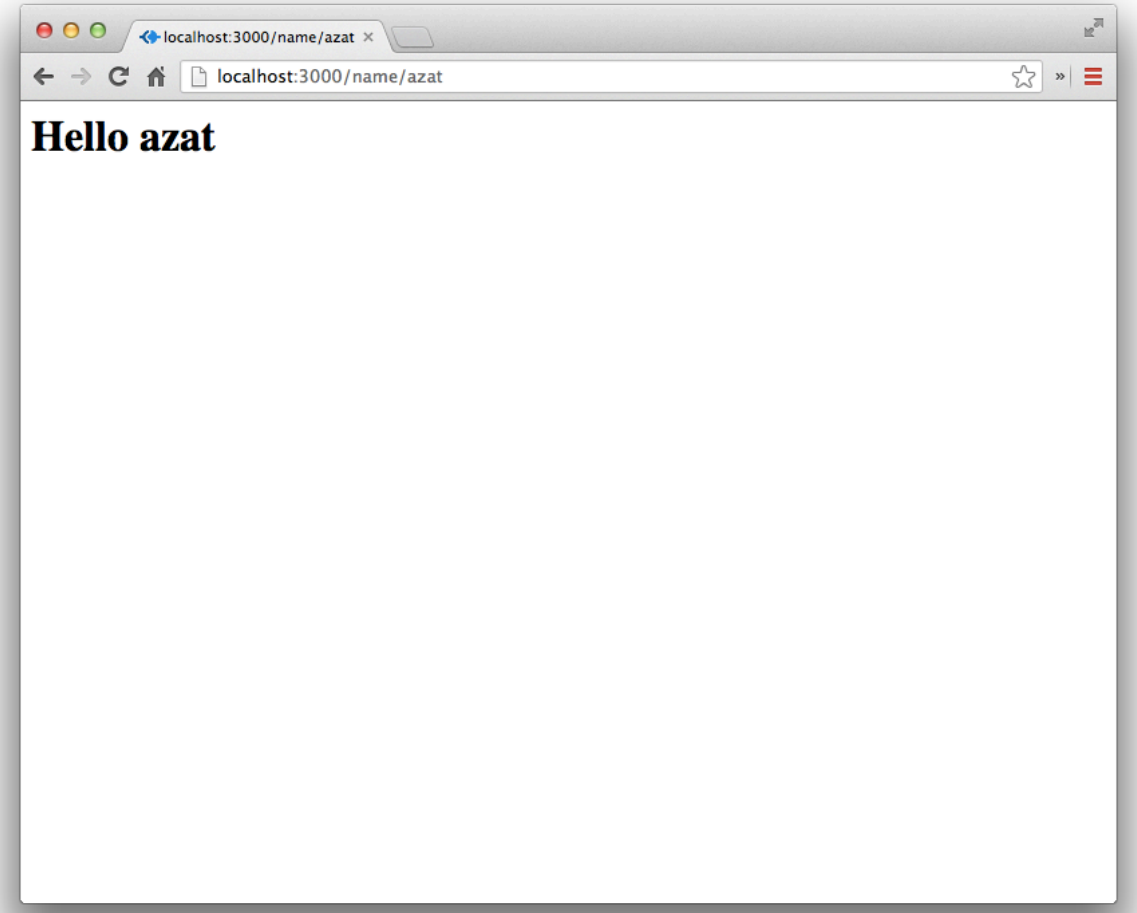
The full source code of the `hello-name.js` file which is also in GitHub[6]:

```
var express = require('express');
var port = 3000;
var app = express();

app.get('/name/:user_name', function(req,res) {
  res.status(200);
  res.set('Content-type', 'text/html');
  res.end('<html><body>' +
    '<h1>Hello ' + req.params.user_name + '</h1>' +
    '</body></html>'
  );
```

[6] https://github.com/azat-co/expressjsguide/blob/master/hello-name.js

```
});

app.get('*', function(req, res){
  res.end('Hello World');
});

app.listen(port, function(){
  console.log('The server is running, ' +
    ' please open your browser at http://localhost:%s',
     port);
});
```

After shutting down the previous server and launching `hello-name.js` script, you'll be able to see the dynamic response, e.g., entering http://localhost:3000/name/azat in your browser yields:



Dynamic Hello User example

5 CLI

Comparably to Ruby on Rails and many other web frameworks, Express.js comes with a command-line interface for jump-starting your development process. The CLI generates a basic foundation for the most common cases.

If you followed the global installation instructions in the Installation chapter, you should be able to see the version number if you run `$ express -V` form **anywhere** on your machine. If we type `$ express -h` or `$ express --help`, we'll get the list of available options and their usage:

To generate a skeleton Express.js app, we need to run a terminal command: `express [options] [dir|appname]` where options are:

- `-s` or `--sessions` adds session support
- `-e` or `--ejs` adds EJS[1] engine support, by default jade[2] is used
- `-J` or `--jshtml` adds JSHTML[3] engine support, by default jade is used
- `-H` or `--hogan` adds hogan.js engine support
- `-c <engine>` or `--css <engine>` adds stylesheet `<engine>` support (e.g., LESS[4] or Stylus[5]), by default plain CSS is used
- `-f` or `--force` forces app generation on non-empty directory

If the dir/appname option is omitted, Express.js will create files using the current folder as the base for the project. Otherwise, the application will be under the specified directory.

For the sake of experimenting, let's run this command: `$ express -s -e -c less -f cli-app`. Here is what will be generated:

---

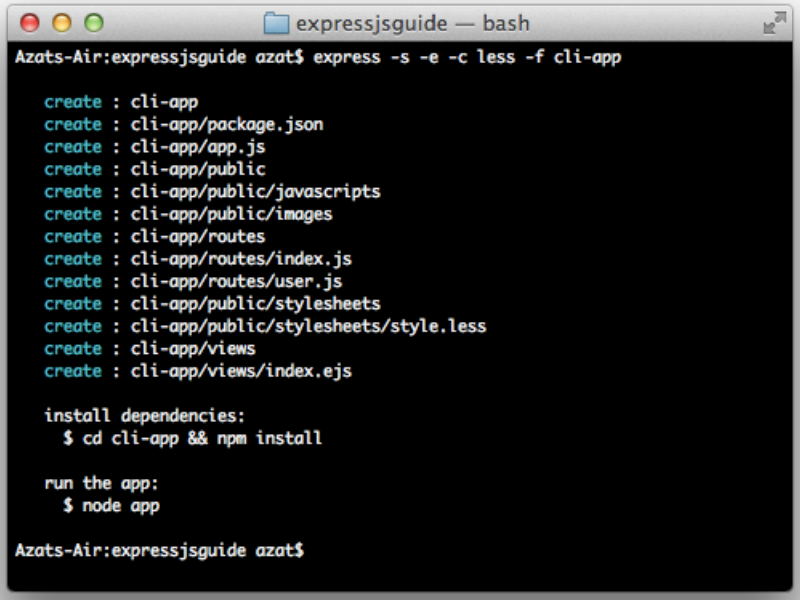
[1] http://embeddedjs.com/

[2] http://jade-lang.com/tutorial/

[3] http://james.padolsey.com/javascript/introducing-jshtml/

[4] http://lesscss.org/

[5] http://learnboost.github.io/stylus/



The result of running `$ express -s -e -c less -f cli-app`.

As you can see, Express.js provides a robust command-line tool for spawning boilerplates rapidly. The downside is that this approach is not immensely configurable, e.g., it's possible to use Handlebars template engine (and many others) creating apps manually, not just Hogan, jade, JSHTML or EJS provided by CLI.

Let's briefly examine the application structure. We have three folder:

1. `public` for static assets
2. `views` for templates
3. `routes` for request handlers

The `public` folder has three folders on its own:

1. `images` for storing images
2. `javascripts` for front-end JavaScript files
3. `stylesheets` for CSS or in our example LESS files (the `-c less` options)

Open the main web server file `app.js` in your favorite text editor. We'll briefly go through the auto-generated code and what it does, before diving deeper into each one of those configurations later in the book.

We include module dependencies:

```
var express = require('express');
var routes = require('./routes');
var user = require('./routes/user');
var http = require('http');
var path = require('path');
```

Create the Express.js app object:

```
var app = express();
```

Define configurations, you can probably guess their meaning based on their names, i.e., instruct app about web server port number, where to get template files from and what template engine to use. More on these parameters in the Configuration chapter of The Interface part.

```
app.set('port', process.env.PORT || 3000);
app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
```

Define middlewares (more on them later in the book) to serve favicon, log events, parse request body, support old browsers' HTTP methods, parse cookies and utilize routes:

```
app.use(express.favicon());
app.use(express.logger('dev'));
app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(express.cookieParser('your secret here'));
app.use(express.session());
app.use(app.router);
```

These take care of serving static assets from `public` folder:

```
app.use(require('less-middleware')({ src: __dirname + '/public' }));
app.use(express.static(path.join(__dirname, 'public')));
```

Express.js gets its env variable from `process.env.NODE_ENV` which is passed as `NODE_ENV=production`, for example either when the server is started or in the machine's configs. With this condition, we enable more explicit error handler for the development environment:

```
if ('development' == app.get('env')) {
  app.use(express.errorHandler());
}
```

The routes are defined as a module in a separate file, so we just pass the functions' expressions instead of defining them right here as anonymous request handlers:

```
app.get('/', routes.index);
app.get('/users', user.list);
```

There is another way to start up the server:

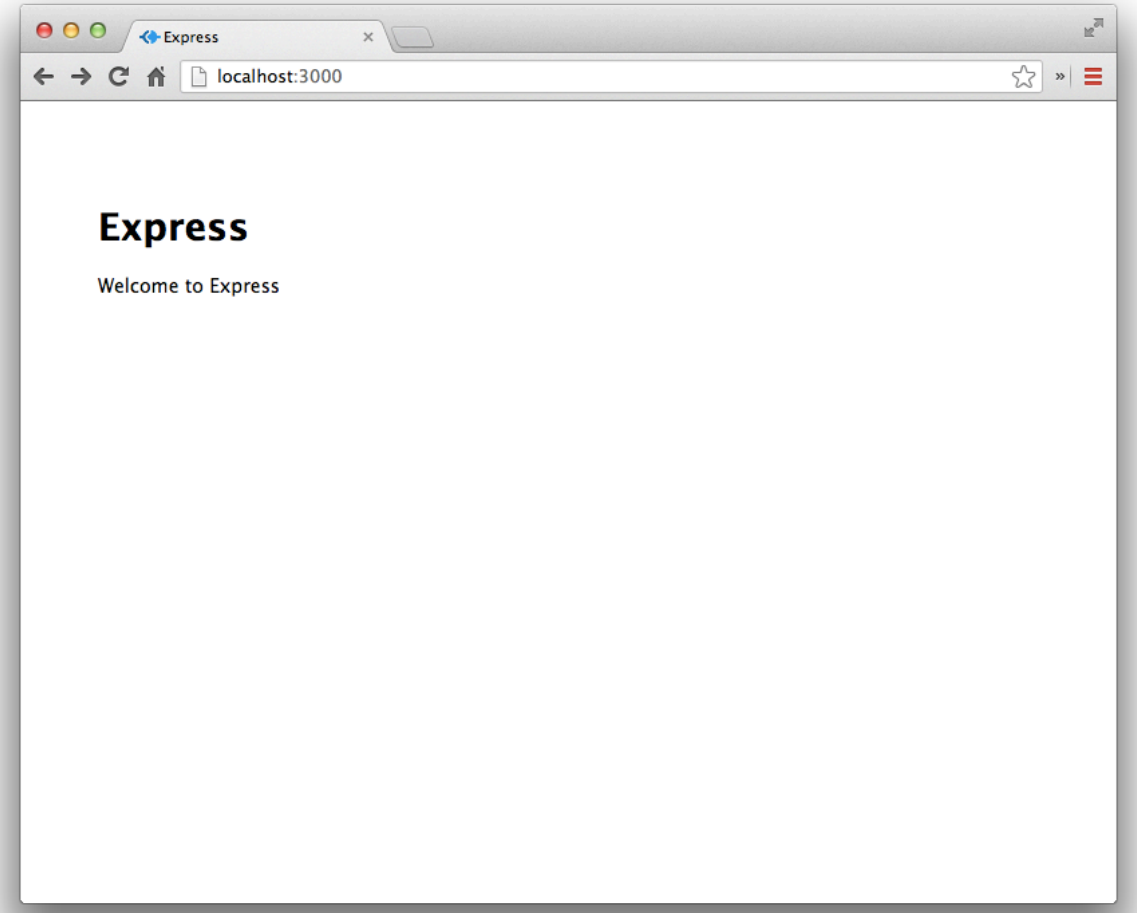
```
http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

For your reference, the full code for `app.js` :

```
/**
 * Module dependencies.
 */

var express = require('express');
var routes = require('./routes');
var user = require('./routes/user');
var http = require('http');
var path = require('path');

var app = express();

// all environments
app.set('port', process.env.PORT || 3000);
app.set('views', __dirname + '/views');
app.set('view engine', 'ejs');
app.use(express.favicon());
app.use(express.logger('dev'));
app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(express.cookieParser('your secret here'));
app.use(express.session());
app.use(app.router);
```

```
app.use(require('less-middleware')({ src: __dirname + '/public' }));
app.use(express.static(path.join(__dirname, 'public')));

// development only
if ('development' == app.get('env')) {
  app.use(express.errorHandler());
}

app.get('/', routes.index);
app.get('/users', user.list);

app.get('/admin', function(req, res, next) {
  if (!req.query._token) return next(new Error('no token provided'));
  }, function(req, res, next) {
  res.render('admin');
});


http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

If you `$ cd cli-app` into the project folder and run `$ npm start` or `$ node app`, you should see common response at http://localhost:3000:



Boilerplate Express.js app.

6 Watching for File Changes

This topic is little bit out of Express.js area, but we thought it is so important that it's worth mentioning. Node.js applications are stored in memory, and if we make changes to the source code, we need to restart the process, i.e., node.

There are brilliant tools that can leverage watch[1] method from the core Node.js **fs** module and restart our servers when we save changes from an editor:

- supervisor[2] (GitHub[3])
- up[4] (GitHub[5])
- node-dev[6] (GitHub[7])
- nodemon[8] (GitHub[9])

Tip

It's good to know that Express.js reloads a template file for every new request by default. So, no server restart is necessary. However, we can cache templates by enabling `view cache` setting which we'll cover at length later in the book.

---

[1] http://nodejs.org/docs/latest/api/fs.html#fs_fs_watch_filename_options_listener
[2] https://npmjs.org/package/supervisor
[3] https://github.com/isaacs/node-supervisor
[4] https://npmjs.org/package/up
[5] https://github.com/LearnBoost/up
[6] https://npmjs.org/package/node-dev
[7] https://github.com/fgnass/node-dev
[8] https://npmjs.org/package/nodemon
[9] https://github.com/remy/nodemon

7 MVC Structure and Modules

Express.js is a highly configurable framework, which means that we can apply any structure we find suitable. As we've observed in the previous chapter CLI, this tool generates a few folders for us right out of the bat: `public`, `views` and `routes`. What is lacking to adhere to the general MVC paradigm is the model. If you use something like Mongoose, you might want to create a folder `models` and put the Schema objects there. More advanced applications might have a nested folder structure similar to this:

```
.
├── app.js
├── package.json
├── views
    └── *.jade
├── routes
    └── *.js
├── models
    └── *.js
├── config
    └── *.js
├── public
    ├── javascripts
        └── *.js
    ├── images
        └── *.png, *.jpg
    └── stylesheets
        └── *.less, *.styl
├── test
    └── *.js
└── logs
    └── *.log
```

The best practice is to have static assets under a special folder. Those assets could also be written in compilable languages like CoffeeScript or LESS.

In case you prefer to rename the folders, just make sure you update the corresponding code in your `app.js` file (or the other main script file, if you're creating an app from scratch with a different filename). For example, if I want to serve my user-related routes from folder `controllers`, I would update my `app.js` file like this:

```
var user = require('./controllers/user');
```

The patterns for the modules don't have to be complicated. From the main file, we include the object with `require()` function, and inside of that module file, we apply `exports` global keyword to attach a method that we want to expose (and use later in the main file):

```
/*
 * GET users listing.
 */

exports.list = function(req, res){
  res.send("respond with a resource");
};
```

One caveat here — or a feature depending on how you look at it — is if we omit the file name and require a folder, e.g., `var routes = require('./routes');` from our previous example, Node.js will grab the `index.js` file from that folder if one exists. This might come in handy when declaring some helpers or utility functions that you might want to share across the files of that particular folder.

Later, in the Multi-Threading with Clusters chapter, we'll discuss how to make your application a module in itself, so we can spawn multiple processes, i.e., workers, in a production environment.

Similar approach is applicable to a template folder. If we decide that we want to have `templates` instead of `views`, we need to change a settings line to:

```
app.set('views', __dirname + '/templates');
```

In this line, the first parameter to `app.set()` function is the name of the setting, i.e., `views` and the second is the value that is dynamically prefixed with a global `__dirname` variable[1]. The `__dirname` returns the system path to the module being executed.

Let's look at the most important setting available in Express.js configuration in the next chapter Configuration.

[1] http://nodejs.org/docs/latest/api/globals.html#globals_dirname

II The Interface

This part can be used as a stand alone reference for the Express.js API, because it contains description of all the main methods and properties, as well as short examples and explanations.

8 Configuration

8.1 app.set() and app.get()

The method `app.set(name, value)` accepts two parameters: name and value, and as you might guess from its name, it sets the value for the name. For example, we often want to store the value of the port on which we plan to start our server:

```
app.set('port', 3000);
```

Or, for a more advanced and realistic use case, we can grab the port from system environment variable PORT:

```
app.set('port', process.env.PORT || 3000);
```

The name value could be an Express.js setting or an arbitrary string.

To get the value, we can use `app.set(name)` with a **single** parameter or more explicit method `app.get(name)`, for example:

```
console.log('Express server listening on port ' + app.get('port'));
```

The `app.set()` also exposes variable to templates app-wide, e.g.,

```
app.set('appName', 'HackHall');
```

Will be available in **all** templates, e.g., in a jade template layout, this would be valid:

```
doctype 5
html
  head
    title= appName
  body
    block content
```

8.2 app.enable() and app.disable()

There are some settings that, instead of strings, can only be set to boolean false or true. For such flags, there are shorthands, so alternatively to the `app.set(name, true)` and `app.set(name, false)` functions, we can use the concise `app.enable(name)` and `app.disable(name)` calls correspondingly.

For example:

```
app.disable('etag');
```

8.3 app.enabled() and app.disabled()

To check whether the aforementioned values equal true or false, we can call methods `app.enabled(name)` and `app.disabled(name)`. For example,

```
app.disable('etag');
console.log(app.disabled('etab'));
```

will output true in the context of the Express.js app.

9 Settings

This is one of the places where Express.js shines! All the configurations are very self-explanatory and easy to read and understand.

9.1 env

This variable is used to store the current environment mode for this particular Node.js process. The value is automatically set by Express.js from process.env.NODE_ENV (which is fed to Node.js through environmental variable of the executing machine) or if that is not set to the `development` value.

The other most common values for `env` setting are:

- `test`
- `stage`
- `preview`
- `production`

We can augment the `env` param by adding `app.set('env', 'preview');` or `process.env.NODE_ENV=preview` in our code. However, the better way is to start an app with `$ NODE_ENV=preview node app` or set NODE_ENV variable on the machine.

Knowing in what mode the application runs is very important because logic related to error handling, compilation of stylesheets and rendering of the templates can differ dramatically. Needless to say, databases and hostnames are different from environment to environment.

9.2 view cache

This flag if set to false, which is the default, and allows for a painless development because templates are read each time the server requests them. On the other hand, if `view cache` is true, it facilitates templates compilation caching which is a desired behavior in production.

If the previous setting `env` is `production`, then this is enabled by default.

9.3 view engine

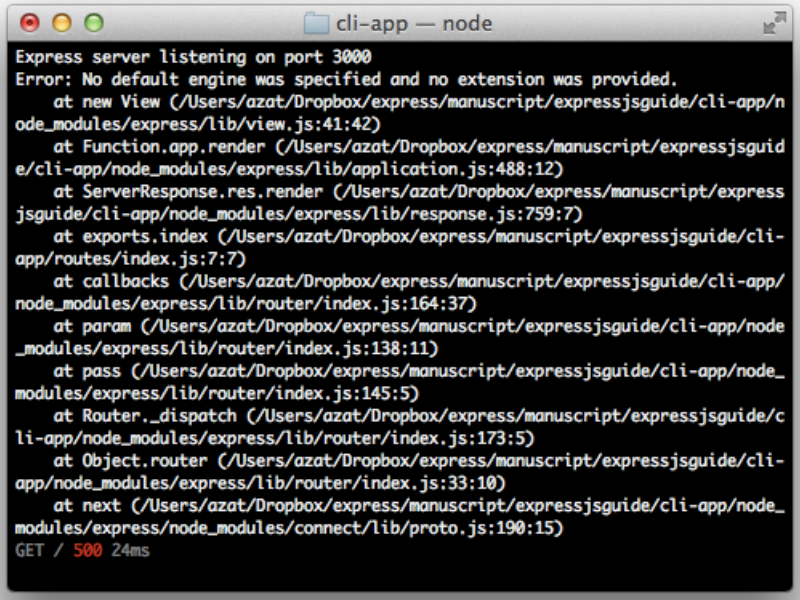
`view engine` holds the template file extension, e.g., 'ext' or 'jade', to utilize if the file extension is not passed to the `res.render()` function inside of the request handler.

For example, if we comment out this line from `cli-app/app.js` example:

```
// app.set('view engine', 'ejs');
```

The server won't be able to locate the file because our instructions in `cli-app/routes/index.js` are too ambiguous:

```
/*
 * GET home page.
 */

exports.index = function(req, res){
  res.render('index', { title: 'Express' });
};
```



The result of not having a proper template extension set.

We can effortlessly fix this by adding an extension to the `cli-app/routes/index.js` file:

```
exports.index = function(req, res){
  res.render('index.ejs', { title: 'Express' });
};
```

For more information on how to apply different template engines, please refer to the chapter **Different Template Engines**.

9.4 views

The `views` setting is an absolute templates' directory path. This setting defaults to `./views` relative to the main application file, e.g., `app.js`, or the file where the `__dirname` global is called.

As we mentioned above in the *MVC Structure and Modules* chapter, changing template folder name is trivial, e.g.,

```
1 app.set('views', __dirname + '/templates');
```

9.5 trust proxy

Set `trust proxy` to true if your Node.js app is working behind reverse proxy such as Varnish or Nginx. This will permit trusting in the `X-Forwarded-*` headers, e.g., `X-Forwarded-Proto` (req.protocol) or `X-Forwarder-For` (req.ips).

`trust proxy` is disabled by default.

Examples:

```
1 app.set('trust proxy', true);
```

9.6 jsonp callback name

If you're building an application that serves front-end clients from different domains and you don't want to apply cross-origin resource sharing[1] (CORS) mechanism, then JSONP is the way to go along with the `res.jsonp()` method of Express.js.

The default callback name which is a prefix for our JSONP response is usually provided in the query string of the request with the name `callback`, e.g., `?callback=updateView`. However, if you want to use something different, just set the setting `jsonp callback name` to that value, e.g., for the requests with a query string param `?cb=updateView`, we can use this:

```
1 app.set('jsonp callback name', 'cb');
```

so that our responses would be wrapped in `updateView && updateView(body);` JavaScript code (with the proper `Content-Type` header of course).

In most cases, we don't want to alter this value because the `callback` value is somewhat standardized by jQuery JSONP functions.

[1] http://en.wikipedia.org/wiki/Cross-origin_resource_sharing

9.7 json replacer and json spaces

Likewise, when we use Express.js method `res.json()`, we can apply special parameters: replacer and spaces, to `JSON.stringify()` function[2] in the scope of the application.

Replacer acts like a filer. You can read more about it at Mozilla Developer Network[3] (MDN).

Express.js uses `null` as the default value for `json replacer`.

The `spaces` parameter is in essence an indentation size. Its value defaults to `2` in development and to `0` in production. In most cases, we leave these settings alone.

9.8 case sensitive routing

The `case sensitive routing` flag should be self explanatory. We disregard case of the URL paths when it's false which is the default value, and do otherwise when the value is set to true. For example, if we have `app.enable('case sensitive routing');`, then `/users` and `/Users` won't be the same. It's best to have this option disabled for the sake of avoiding confusion.

9.9 strict routing

The last but not least flag `strict routing` deals with cases of trailing slashes in the URLs. With the `strict routing` enabled, e.g., `app.set('strict routing', true');`, the paths will be treated differently, e.g., `/users` and `/users/` would be completely separate routes.

By default, this parameter is set to false.

9.10 x-powered-by

Sets the HTTP response header `X-Powered-By` to `Express` value. This option is enabled by default.

9.11 etag

ETag[4] or entity tag is one of the caching tools. If someone doesn't know what it is or how to use it, it's better to leave it on. Otherwise, to disable it: `app.disable('etag');`.

[2] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/stringify
[3] https://developer.mozilla.org/en-US/docs/Using_native_JSON#The_replacer_parameter
[4] http://en.wikipedia.org/wiki/HTTP_ETag

9.12 subdomain offset

The `subdomain offset` setting controls the value returned by `req.subdomains` property. This setting is useful when the app is deployed on multiple subdomains, such as `http://ncbi.nlm.nih.gov`. By default, the extreme two parts in the domain are dropped and the rest are returned in reversed order in the `req.subdomains`, so for our example, the result is `['nlm', 'ncbi']`. However, if the app has `subdomain offset` set to three by `app.set('subdomain offset', 3);`, the result of `req.subdomains` will be just `['ncbi']`.

10 Environments

Of course, we can write up some `if else` statements based on the `process.env.NODE_ENV` value, for example:

```
if ('development' === process.env.NODE_ENV) {
  //connect to development database
} else if ('production' === process.env.NODE_ENV) {
  //connect to production database
 }; //continue for stag and preview environments
```

or using Express.js `env` param (please refer to the section *env* above):

```
//assuming that app has reference to Express.js instance
if ('development' === app.get('env')) {
  //connect to development database
} else if ('production' === app.get('env')) {
  //connect to production database
 }; //continue for stag and preview environments
```

However, the same algorithm could be written more elegantly with Express.js sugarcoating method `app.configure()`.

10.1 app.configure()

Now that we know how to set application-wide settings, we might find ourselves wanting to apply the whole sets of such settings. For example, for development and production, we want to use different database URIs. This could be smartly achieved with `app.configure()` method which takes one parameter for all environments, e.g., if we want to set admin email, we can write:

```
app.configure(function(){
  app.set('adminEmail', 'hi@azat.co');
});
```

However, if we pass two parameters (or more) and the last one is still a function, the code will be called only when the app in those modes:

```
app.configure('development', function() {
  app.set('dbUri', 'mongodb://localhost:27017/db');
});
app.configure('stage', 'production', function() {
  app.set('dbUri', process.env.MONGOHQ_URL);
});
```

The callback is called immediately, so make sure you have all of the objects used inside of the callback available. The `app.configure()`function returns an instance of `app` so it's suitable for chaining.

Tip

Express.js **often** uses the difference in the number of input parameters and their types to direct functions' behavior. Therefore, please pay close attention to how you invoke your methods.

11 Applying Middleware

To set up a middleware, we utilize `app.use()` method.

11.1 app.use()

The method `app.use()` has one optional string parameter path and one mandatory function parameter callback. For example, to implement a logger with a date, time, request method and URL:

```
1 //instantiate the Express.js app
2 app.use(function(req, res, next) {
3   console.log('%s %s - %s', (new Date).toString(), req.method, req.url);
4   return next();
5 });
6 //implement server routes
```

On the other hand, if we want to prefix the middleware, a.k.a., *mounting*, we can use the `path` parameter which restricts the use of this particular middleware to only the routes that have such prefix. For example, to limit the logging only to admin dashboard route `/admin`, we can write:

```
1 //instantiate the Express.js app
2 app.use('/admin', function(req, res, next) {
3   console.log('%s %s - %s', (new Date).toString(), req.method, req.url);
4   return next();
5 });
6
7 //actually implement the /admin route
```

Writing everything from scratch as obvious as logging and serving of the static files is not much fun. Therefore, Express.js comes with these (and many other) middlewares build-in:

```
var express = require('express');
//instantiate and configure the app
app.use(express.logger());
app.use(express.static(__dirname + '/public'));
//implement server routes
```

More advanced use example where we restrict assets to their respective folders:

```
app.use('/css', express.static(__dirname + '/public/css'));
app.use('/img', express.static(__dirname + '/public/images'));
app.use('/js', express.static(__dirname + '/public/javascripts'));
```

12 Types of Middleware

As you've seen in the previous chapter, middleware is nothing more than a function that takes `req` and `res` objects. Express provides a few more middlewares right out of the box with most of them coming from Sencha's Connect library[1] (NPM[2], GitHub[3]).

The main thing to remember when using middlewares is that the order in which they're declared via `app.use()` function **matters**. For example, session must follow cookie while csrf requires session.

12.1 express.compress()

This middleware allows for gzipping of transferred data and is usually put in the very beginning of Express.js app configuration to precede the other middleware and routes. The `express.compress()` is good without extra params, but here they are just in case (gzip uses core Node.js module zlib[4] and just passes these options to it):

- chunkSize (default: 16*1024)
- windowBits
- level
- memLevel
- strategy
- filter: function that by default tests for the `Content-Type` header to be json, text or javascript

For more information on these options, please see the Zlib docs[5].

12.2 express.logger()

This middleware keeps track of all the requests. It takes either **options** object or a **format** string, e.g.,

[1] http://www.senchalabs.org/connect/
[2] https://npmjs.org/package/connect
[3] https://github.com/senchalabs/connect
[4] http://nodejs.org/api/zlib.html#zlib_options
[5] http://zlib.net/manual.html#Advanced

```
app.use(express.logger()); //vanilla logger
app.use(express.logger('short'));
app.use(express.logger('dev')
```

Supported options:

- `format`: a string with an output format, see token string
- `stream`: the output stream to use, defaults to stdout but could be anything else, e.g., a file or a another stream
- `buffer` number of millisecond for the buffer interval, defaults to 1000ms if not set or not a number
- `immediate` boolean value which when set to true makes logger to write log line on request instead of response

Available format string params or tokens:

- `:req[header]`, e.g., `:req[Accept]`
- `:res[header]`, e.g., `:res[Content-Length]`
- `:http-version`
- `:response-time`
- `:remote-addr`
- `:date`
- `:method`
- `:url`
- `:referrer`
- `:user-agent`
- `:status`

Pre-defined formats/tokens that come with Express.js/Connect:

- `default` is same as `:remote-addr - - [:date] ":method :url HTTP/:http-version" :status :res[content-length] ":referrer" ":user-agent"`
- `short` is same as `:remote-addr - :method :url HTTP/:http-version :status :res[content-length] - :response-time ms`
- `tiny` is same as `:method :url :status :res[content-length] - :response-time ms`
- `dev` short and colored development output with response statuses

You could also define your own formats. For more info, please refer to the Connect documentation[6].

[6] http://www.senchalabs.org/connect/logger.html

12.3 express.json()

If the request has MIME type of `application/json`, this middleware will try to parse the request payload as JSON. The result will be put in `req.body` object and passed to the next middlewares and routes.

We can pass the following options as properties:

- strict: boolean true or false, if it's true (default), then 400 status error will be passed to `next()` callback when first character is not `[` or `{`
- reviver: is a second parameter to `JSON.parse()` function which transforms the output; more info at MDN[7]
- limit: max byte size; disabled by default

For example, if you need to skip the private methods/properties (by conventions they begin with the underscore _ symbol), apply non-strict parsing and have a limit of 5000 bytes:

```
app.use(express.json({
  strict: false,
  reviver: function(key, value) {
    if (key.substr(0,1) === '_') {
      return undefined;
    } else {
      return value;
    }
  },
  limit: 5000
}));
```

12.4 express.urlencoded()

This `express.urlencoded()` middleware parses **only** requests with the `x-ww-form-urlencoded` header. It utilizes qs[8] module's `querystirng.parse()` function and puts the resulting JS object into `req.body`.

We can also pass the limit parameter similar to `express.json()` middleware above, e.g.,

```
app.use(express.urlencoded({limit:10000});
```

[7] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse
[8] https://npmjs.org/package/qs

12.5 express.multipart()

The `express.multipart()` middleware is very similar to the previous two, but deals with forms, i.e., requests with the `multipart/form-data` header. It ignores `GET` and `HEAD` HTTP methods. The middleware is build on top of Felix Geisendörfer's formidable[9] (GitHub[10]) and accepts its options, e.g., `uploadDir: '/tpm'`, in addition to:

- limit: string or number of bytes to limit the request body size to
- defer: boolean true or false; if true passes Formidable form object in `req.form`

Example:

```
1 app.use(express.multipart());
```

12.6 express.bodyParser()

The three middleware we went through previously are rarely used by themselves. Therefore, there's a wrapper `express.bodyParser()`. In other words, `app.use(express.bodyParser(options));` is equivalent to

```
1 app.use(express.json(options));
2 app.use(express.urlencoded(options));
3 app.use(express.multipart(options));
```

Where the **options** object is passed to all three calls and has the corresponding options (refer to the aforementioned middlewares for more info on supported options).

Warning

`bodyParser()` middleware is known to be prone to malfunctioning when handling large file uploads. The exact problem is described in this article[11] and TJ's response. Future versions of Express.js might drop support for `bodyParser()`.

12.7 express.cookieParser()

The `express.cookieParser()` allows us to access user cookie values from the `req.cookie` object in request handlers. The method takes a string which is used for signing cookies. Usually, it's some clever pseudo random sequence, e.g., `very secret string`.

Use it like this:

[9] https://npmjs.org/package/formidable
[10] https://github.com/felixge/node-formidable
[11] http://andrewkelley.me/post/do-not-use-bodyparser-with-express-js.html

```
app.use(express.cookieParser());
```

or with the string:

```
app.use(express.cookieParser('cats and dogs));
```

Warning

Never store any information in the cookies, especially sensitive user-related information. In most cases, cookies are just used to store a unique and hard to guess key that matches against a value on the server to retrieve a user session.

12.8 express.session()

This middleware must have `express.cookeParser()` enabled before its definition. The `express.session()` takes these options:

Options

- `key`: cookie name defaulting to `connect.sid`
- `store`: session store instance, usually Redis object (more about it in the Redis chapter)
- `secret`: session cookie is signed with this secret to prevent tampering, usually just a random string
- `cookie`: session cookie settings, defaulting to { path: '/', httpOnly: true, maxAge: null }
- `proxy`: boolean that tells to trust the reverse proxy when setting secure cookies (via "x-forwarded-proto")

By default, sessions are stored in the memory. However, we can use Redis for persistance and for sharing sessions between multiple machines. For more information on Express.js session, please refer to the Tips and Tricks part.

12.9 express.csrf()

Cross-site request forgery[12] (CSRF) protection is handled by Express.js by putting `_csrf` token in the session (`req.session._csrf`) and validating that value against values in `req.body`, `req.query` and `X-CSRF-Token` header. If the values don't match, the 403 Forbidden[13] HTTP status code is returned.

This middleware doesn't check `GET`, `HEAD` or `OPTIONS` methods.

You can override the default function that checks the value by passing a callback function in a `value` property; for example, to use a different name and check against **only** the request body:

[12] http://en.wikipedia.org/wiki/Cross-site_request_forgery
[13] http://en.wikipedia.org/wiki/HTTP_403

```
app.use(express.csrf({
  value: function (req) {
    return (req.body && req.body.cross_site_request_forgery_value);
  }
}));
```

The `csrf` middleware must be after `session`, `cookieParser` and optionally `bodyParser` and `query` middlewares:

```
app.use(express.bodyParser());
app.use(express.query());
app.use(express.cookieParser());
app.use(express.csrf());
```

12.10 express.static()

Already familiar with `express.static(path, options)` method that serves files from a specified **root** path to the folder, e.g.,

```
app.use(express.static(path.join(__dirname, 'public')));
```

or

```
app.use(express.static(__dirname + '/public'));
```

The express.static(path, options) method takes these options:

- `maxAge`: number of milliseconds to set for browser cache maxAge which defaults to `0`
- `hidden`: boolean true or false which enables serving of hidden files; defaults to false for security reasons
- `redirect`: boolean true or false (default is true) that allows for a redirect to a trailing slash "/" when the URL pathname is a directory

Advanced use:

```
app.use(express.static(__dirname + '/public', {
  maxAge: 86400000,
  redirect: false,
  hidden: true
}));
```

12.11 express.basicAuth()

Very basic HTTP authentication, e.g.,

```
app.use(express.basicAuth(function(user, pass){
  if (user === 'azat' && pass === 'expressjs' ) {
    return true;
  } else {
    return false;
  }
}));
```

Please see **Tips and Tricks** and **Tutorials and Examples** for more advanced and realistic scenarios.

12.12 Other Express.js/Connect Middlewares

- directory[14]
- compress[15]
- errorHandler[16]
- favicon[17]
- limit[18]
- logger[19]
- methodOverride[20]
- responseTime[21]
- staticCache[22]

[14] http://www.senchalabs.org/connect/directory.html
[15] http://www.senchalabs.org/connect/compress.html
[16] http://www.senchalabs.org/connect/errorHandler.html
[17] http://www.senchalabs.org/connect/favicon.html
[18] http://www.senchalabs.org/connect/limit.html
[19] http://www.senchalabs.org/connect/logger.html
[20] http://www.senchalabs.org/connect/methodOverride.html
[21] http://www.senchalabs.org/connect/responseTime.html
[22] http://www.senchalabs.org/connect/staticCache.html

- vhost[23]
- subdomains[24]
- cookieSession[25]

[23] http://www.senchalabs.org/connect/vhost.html
[24] http://www.senchalabs.org/connect/subdomains.html
[25] http://www.senchalabs.org/connect/cookieSession.html

13 Different Template Engines

To use something different, make sure you install that module with NPM preferably by adding it to `package.json` as well.

13.1 app.engine()

By default, Express.js will try to require template engine based on the extension provided. That is when we call `res.render('index.jade')` (more on this method later) with `index.jade` file name, the framework is calling ‘ app.engine(‘jade’, require(‘jade’).__express);‘ internally.

The `app.engine()` method example:

```
1 //decleare dependencies
2 //instantiate the app
3 //configure the app
4 app.engine('jade', require('jade').__express);
5 //define the routes
```

Therefore, to map file extensions to different template engines, please take a look at the `app.engine()` method. The callback, i.e., the second parameter to the method, must be in a special format. To insure that the template library supports this format, we call `__express` on it.

Here is the list of the libraries that support Express.js without any modification (taken from the Express.js Wiki page[1]):

- Jade[2] – Haml inspired template engine
- Haml.js[3] – Haml implementation
- EJS[4] – Embedded JavaScript template engine
- hbs[5] – adapter for Handlebars.js, an extension of Mustache.js template engine
- h4e[6] – adapter for Hogan.js, with support for partials and layouts
- hulk-hogan[7] – adapter for Twitter’s Hogan.js[8] (Mustache syntax), with support for Partials

[1] https://github.com/visionmedia/express/wiki#template-engines
[2] http://github.com/visionmedia/jade
[3] http://github.com/visionmedia/haml.js
[4] http://github.com/visionmedia/ejs
[5] http://github.com/donpark/hbs
[6] https://github.com/tldrio/h4e
[7] https://github.com/quangv/hulk-hogan
[8] http://twitter.github.com/hogan.js/

- combyne.js[9] – A template engine that hopefully works the way you'd expect.
- swig[10] – fast, Django-like template engine
- whiskers[11] – small, fast, mustachioed
- Blade[12] – HTML Template Compiler, inspired by Jade & Haml
- Haml-Coffee[13] – Haml templates where you can write inline CoffeeScript.
- Webfiller[14] – plain-html5 dual-side rendering, self-configuring routes, organized source tree, 100% js.
- express-hbs[15] – Handlebars with layouts, partials and blocks for express 3 from Barc[16]
- express3-handlebars[17] – A Handlebars view engine for Express which doesn't suck.

In case the template engine of your choice does not provide `__express()` method, consider the consolidate library[18] (GitHub[19]). More on how to use it with Express.js in the Tips and Tricks part.

[9] http://github.com/tbranyen/combyne.js
[10] https://github.com/paularmstrong/swig
[11] https://github.com/gsf/whiskers.js
[12] https://github.com/bminer/node-blade
[13] https://github.com/netzpirat/haml-coffee
[14] https://github.com/haraldrudell/webfiller
[15] https://github.com/barc/express-hbs
[16] http://barc.com
[17] https://github.com/ericf/express3-handlebars
[18] https://npmjs.org/package/consolidate
[19] https://github.com/visionmedia/consolidate.js

14 Extracting Parameters

To extract parameters from the URL, we can manually apply same logic to many routes. For example, imagine that we need user information on user profile page (`/user/:username`) and on admin page (`/admin/:username`), this is how we can implement it:

```
var findUserByUsername = function (username, callback) {
  //perform database query that calls callback when its done
};

app.get('/users/:username', function(req, res, next) {
  var username = '';
  //parse req.url to get the username
  fundUserByUsername(username, function(e, user) {
    if (e) return next(e);
    return res.render(user);
  });
});

app.get('/admin/:username', function(req, res, next) {
  var username = '';
  //parse req.url to get the username
  fundUserByUsername(username, function(e, user) {
    if (e) return next(e);
    return res.render(user);
  });
});
```

Even with abstracting the bulk of code into the `findUserByUsername()` function, we still ended up with ineloquent code. Meet the `app.param()` method!

14.1 app.param()

Anytime the given string is present in the URL, the callback will be triggered, e.g., `app.param('username', function(req, res, next, username){...})`.

The `app.param()` is very similar to `app.use()`, but provides the value as the fourth, last parameter (username is our example):

```
app.param('username', function (req, res, next, username) {
  //username has the value!
  //perform database query that
  //stores the user object in the req object
  //and calls next() when it's done
});

app.get('/users/:username', function(req, res, next) {
  //no need for extra lines of code
  //we have req.user already!
  return res.render(req.user);
});

app.get('/admin/:username', function(req, res, next) {
  //same thing, req.user is available!
  return res.render(user);
});
```

In other words, parameters are values passed in a query string of a URL of the request. If we didn't have Express.js or similar library, and had to use just the core Node.js modules, we'd had to extract parameters from HTTP.request[1] object via some `require('querystring').parse(url)` or `require('url').parse(url, true)` functions trickery.

Here is another example of how we can plug param middleware in our app:

```
app.param('id', function(req,res, next, id){
  //do something with id
  //store id or other info in req object
  //call next when done
  next();
});

app.get('/api/v1/stories/:id',function(req,res){
  //param middleware will be execute before and
  //we expect req object already have needed info
  //output something
  res.send(data);
});
```

Or in the application that has a Mongoskin/Monk-like database connection in `req.db`:

[1] http://nodejs.org/api/http.html#http_http_request_options_callback

```
app.param('id', function(req,res, next, id){
  req.db.get('stories').findOne({_id:id}, function (e, story){
    if (e) return next(e);
    if (!story) return next(new Error('Nothing is found'));
    req.story = story;
    next();
  });
});

app.get('/api/v1/stories/:id',function(req,res){
  res.send(req.story);
});
```

Or we can use multiple request handlers but the concept remains the same: we can expect to have `req.story` object or an error thrown prior to the execution of this code so we abstract common code/logic of getting parameters and their respective objects:

```
app.get('/api/v1/stories/:id', function(req,res, next) {
  //do authorization
  }),
  //we have an object in req.story so no work is needed here
  function(req,res) {
  //output the result of the database search
  res.send(story);
});
```

Authorization and input sanitation are also good candidates for residing in the middlewares.

Function `param()` is especially cool because we can combine different variables in the routes, e.g.:

```
app.param('storyId', function(req, res, next, storyId) {
  //fetch the story by its ID
});
app.param('elementId', function(req, res, next, elementId) {
  //fetch the element by its ID
  //narrow down the search when req.story is provided
});
app.get('/api/v1/stories/:storyId/elements/:elementId',function(req,res){
  res.send(req.element);
});
```

15 Routing

Express.js is a node.js framework that among other things provides a way to organize routes.

15.1 app.VERB()

Each route is defined via a method call on an application object with a URL pattern as the first parameter — Regular Expressions[1] (RegExps) are also supported — that is `app.METHOD(path, [callback...], callback)`, for example:

```
app.get('api/v1/stories/', function(res, req){
  ...
})
```

or, for a POST method:

```
app.post('/api/v1/stories', function(req,res){
  ...
})
```

It's needless to say that DELETE, PUT and other methods are supported as well[2].

The callbacks that we pass to `get()` or `post()` methods are called request handlers (more info on them in the Request Handlers chapter), because they take requests (`req`), process them and write to response (`res`) objects. For example:

```
app.get('/about', function(req,res){
  res.send('About Us: ...');
});
```

We can have multiple request handlers, hence the name *middleware*. They accept a third parameter/function `next`, calling which (`next()`) will switch the execution flow to the next handler:

[1] http://en.wikipedia.org/wiki/Regular_expression
[2] http://expressjs.com/api.html#app.VERB

```
app.get('/api/v1/stories/:id', function(req,res, next) {
  //do authorization
  //if not authorized or there is an error
  // return next(error);
  //if authorized and no errors
  return next();
}), function(req,res, next) {
  //extract id and fetch the object from the database
  //assuming no errors, save story in the request object
  req.story = story;
  return next();
}), function(req,res) {
  //output the result of the database search
  res.send(res.story);
});
```

Where ID of a story in URL pattern is a query string parameter which we need for finding matching items in the database.

We can have multiple request handlers defined which are extremely helpful in re-using the code for authentication, validation and loading of resources:

```
app.get('/admin',
  function(req,res,next) {
    //check active session, i.e.,
    //make sure the request has cookies associated with a valid user session
    //check if the user has administrator privileges
    return next();
  },  function(req,res,next){
    //load the information required for admin dashboard
    //such as user list, preferences, sensitive info
    return next();
  }, function(req, res) {
    //render the information with proper templates
    //finish response with a proper status
    res.end();
   })
```

or the same thing, but much *cleaner* with named functions:

```
var authAdmin = function (req, res, next) {
  ...
  return next();
}
var getUsers = function (req, res, next) {
  ...
  return next();
}
var renderUsers = function (req, res) {
  res.end();
}
app.get('/admin', authAdmin,  getUsers, renderUsers);
```

Another useful technique is to pass callbacks as items of an array, thanks to the inner workings of `arguments` JavaScript mechanism[3]:

```
var authAdmin = function (req, res, next) {
  ...
  return next();
}
var getUsers = function (req, res, next) {
  ...
  return next();
}
var renderUsers = function (req, res) {
  res.end();
}
var admin = [authAdmin, getUsers, renderUsers];
app.get('/admin', authAdmin,  getUsers, renderUsers);
```

One distinct difference between request handlers in routes and middleware is that we can bypass the rest of the callbacks in the chain by calling `next('route');`. This might come in handy if, in the example above, where we have `/admin` route, a request fails authentication in the first callback, then there's no need to proceed.

Please note that if the first parameter we pass to `app.VERB()` contains query strings, e.g., `/?debug=true`, that information is disregarded by Express.js. For example, `app.get('/?debug=true', routes.index);` will be treated exactly as `app.get('/', routes.index);`.

Here is the list of most commonly used Representational State Transfer[4] (REST) server architecture HTTP methods:

[3] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions_and_function_scope/arguments

[4] http://en.wikipedia.org/wiki/Representational_state_transfer

- `app.get()`: retrieves entity or a list of entities
- `app.head()`: same as GET only without the body
- `app.post()`: submits a new entity
- `app.put()`: updates an entity by complete replacement
- `app.path()`: updates an entity partially
- `app.del()`: deletes existing entity
- `app.options()`: retrieves the capabilities of the server

For more information on HTTP methods, please refer to RFC2616[5] and its Method Definitions[6].

15.2 app.all()

The `app.all()` method allows the execution of specified request handlers on a particular path no matter what the HTTP method of the request is. This procedure might be a life saver when defining *global* or namespaced logic, e.g.,

```
1 app.all('*', userAuth);
2 ...
3 app.all('/api/*', apiAuth);
```

15.3 Trailing Slashes

Paths with trailing slashes at the end are treated as their normal counterparts by default. To turn off this feature, use `app.enable('strict routing');` or `app.set('strict routing', true);`. More about setting options in the Configuration and Settings chapters.

[5] http://tools.ietf.org/html/rfc2616
[6] http://tools.ietf.org/html/rfc2616#page-51

16 Request Handlers

Request handlers in Express.js are strikingly similar to callbacks in the core Node.js `http.createServer()` method, because they're just functions (anonymous, named or methods) with `req` and `res` parameters:

```
var ping = function(req, res) {
  console.log('ping');
  res.end(200);
};

app.get('/', ping);
```

In addition, we can utilize the third parameter `next()` for control flow. It's closely related to the error handling topic which we will cover later. Here is a simple example of two request handlers ping and pong:

```
var ping = function(req, res, next) {
  console.log('ping');
  return next();
};
var pong = function(req, res) {
  console.log('pong');
  res.end(200);
};
app.get('/', ping, pong);
```

When a request comes on the `/` route, Express.js calls `ping()` which acts as a middleware in this case (because it's in the middle!). Ping in turn, when it's done, calls pong that finished response with `res.end()`.

The `return` keyword is also very important. For example, we don't want to continue processing the request if the authentication has failed in the first middleware:

```
//instantiate app and configure error handling

//authentication middleware
var checkUserIsAdmin = function (req, res, next) {
  if (req.session && req.session._admin !== true) {
    return next (401);
  }
  return next();
};

//admin route that fetches users and call render function
var admin = {
  main: function (req, res, next) {
    req.db.get('users').find({}, function(e, users) {
      if (e) return next(e);
      if (!users) return next(new Error('No users to display.'));
      res.render('admin/index.html', users);
   });
  }
};

//display list of users for admin dashboard
app.get('/admin', checkUserIsAdmin, admin.main);
```

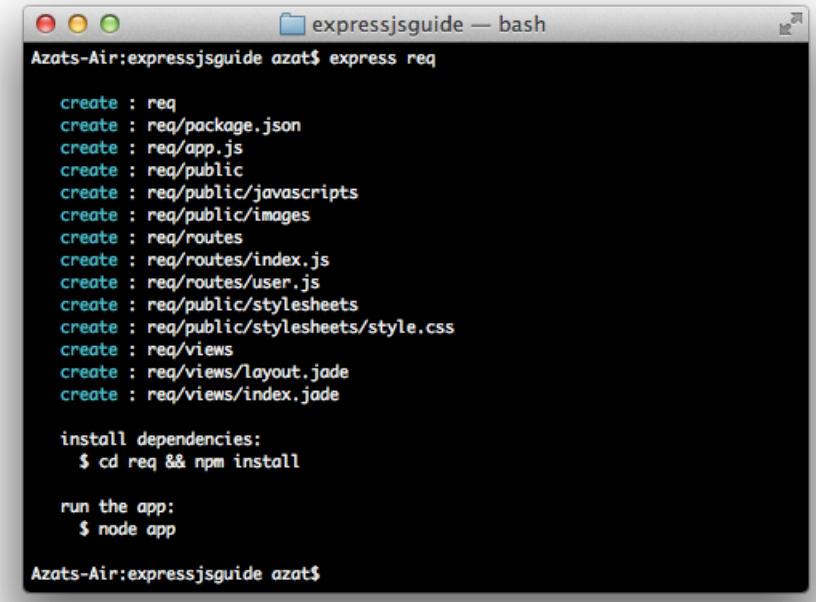
Please notice that `return` keyword is essential because if we don't use it for `next(e)` call, the application will try to render (`res.render()`) even when there is an error and/or we don't have any users. For example, this is probably **a bad idea**:

```
var admin = {
  main: function (req, res, next) {
    req.db.get('users').find({}, function(e, users) {
      if (e) next(e);
      if (!users) next(new Error('No users to display.'));
      res.render('admin/index.html', users);
   });
  }
};
```

17 Request

Express request object is a wrapper of core Node.js **http.request** object. Its has some neat functionality, but in its essence supports everything that the native **http.request **can do.

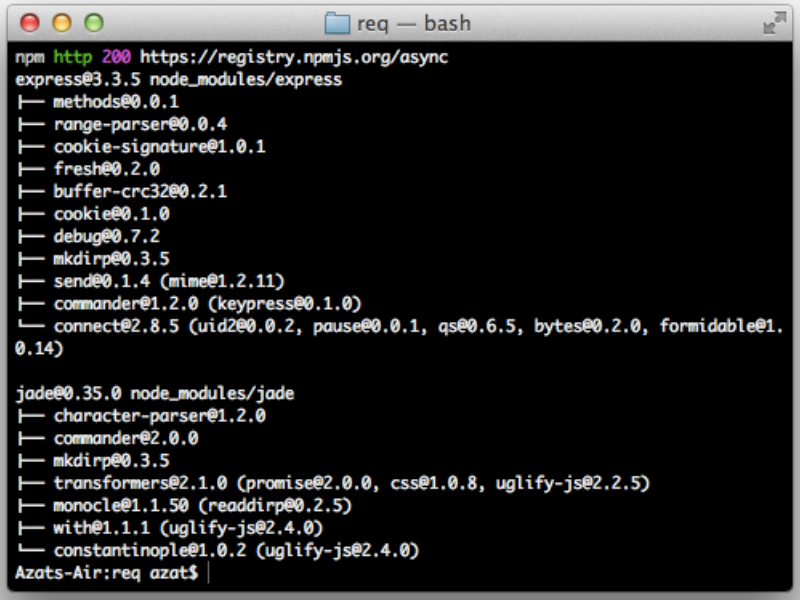
To better understand the request object, let's create a brand new Express.js app with `express req` command from our `expressjsguide` folder:



The result of executing a vanilla `$ express` command.

Now we can install dependencies:

```
1 $ cd req
2 $ npm install
```



The result of running `$ npm install`.

17.1 query

Query string is everything to the left of the question mark in a given URL, e.g., in the URL `https://twitter.com/search?q=js&src=typd` the query string is `q=js&src=typd` and the JS object would be `{q:'js', src:'typd'}`.

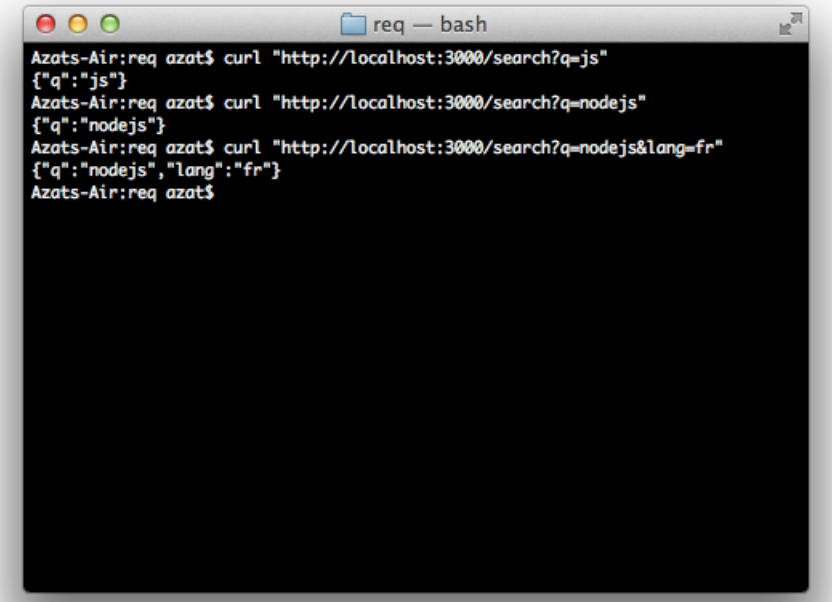
The `connect.query()` is utilized by Express.js behind the scenes without any extra configurations. It is resembling `express.bodyParser()` and `express.cookieParser()` in a way that it puts a property (`query` in this case) on a request object `req` that is passed to the next middlewares and routes.

We can add this route to any Express.js server such as the one we've created with the CLI, i.e., `expresjsguide/req`:

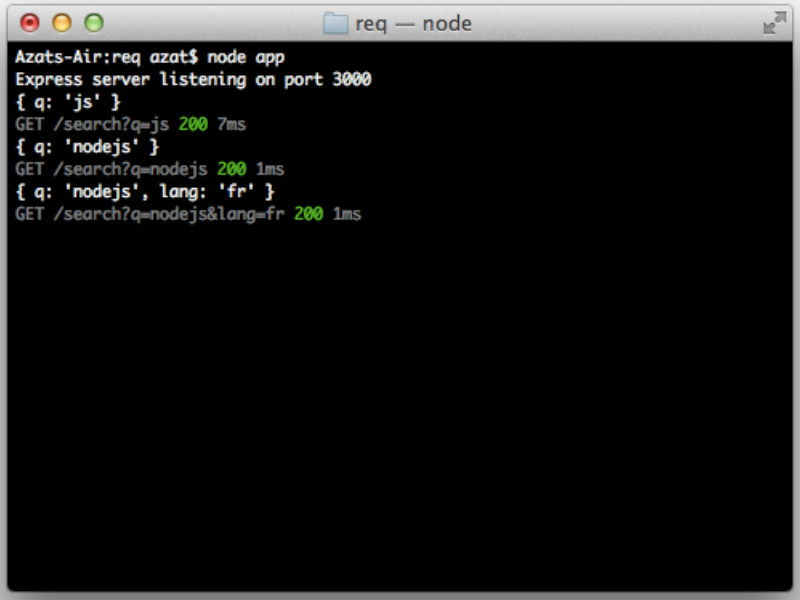
```
1  app.get('/search', function(req, res) {
2    console.log(req.query)
3    res.end(JSON.stringify(req.query)+'/n');
4  })
```

And make get requests with CURL:

```
curl "http://localhost:3000/search?q=js"
curl "http://localhost:3000/search?q=nodejs"
curl "http://localhost:3000/search?q=nodejs&lang=fr"
```



The client side results of running CURL commands with the query string params.



The server-side results of running CURL commands with the query string params.

The full source code of the modified `req` server (also available under `/expressjsguide/req/app.js`) where you can see that `req.query` is available by default in Expess.js apps:

```
1  /**
2   * Module dependencies.
3   */
4
5  var express = require('express');
6  var routes = require('./routes');
7  var user = require('./routes/user');
8  var http = require('http');
9  var path = require('path');
10
11 var app = express();
12
13 // all environments
14 app.set('port', process.env.PORT || 3000);
15 app.set('views', __dirname + '/views');
16 app.set('view engine', 'jade');
17 app.use(express.favicon());
18 app.use(express.logger('dev'));
```

```
19  app.use(express.bodyParser());
20  app.use(express.methodOverride());
21  app.use(app.router);
22  app.use(express.static(__dirname + '/public'));
23
24  // development only
25  if ('development' == app.get('env')) {
26    app.use(express.errorHandler());
27  }
28
29
30  app.get('/search', function(req, res) {
31    console.log(req.query);
32    res.end(JSON.stringify(req.query)+'\n');
33  });
34
35  app.get('/params/:role/:name/:status', function(req, res) {
36    console.log(req.params);
37    console.log(req.route);
38    res.end();
39  });
40
41  app.post('/body', function(req, res){
42    console.log(req.body);
43    res.end(JSON.stringify(req.body)+'\n');
44  });
45
46  app.post('/upload', function(req, res){
47    console.log(req.files.archive);
48    //read req.files.archive.path
49    //process the data
50    //save the data
51    res.end();
52  })
53
54  app.get('/route', function(req, res){
55    console.log(req.route);
56    res.end();
57  })
58  app.get('/', routes.index);
59  app.get('/users', user.list);
60
```

```
61 http.createServer(app).listen(app.get('port'), function(){
62   console.log('Express server listening on port ' + app.get('port'));
63 });
```

17.2 req.params

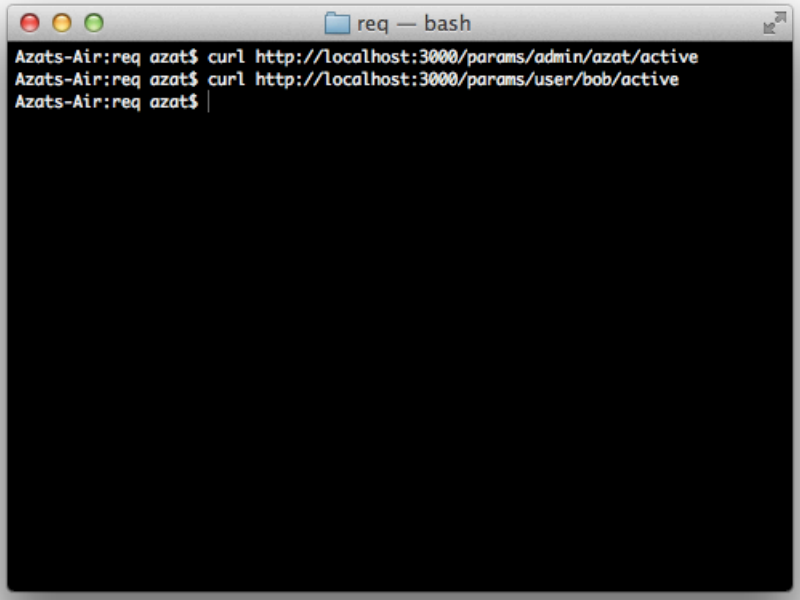
In the preceding chapter **Extracting Parameters**, we covered how to set up a middleware to process data taken from the URLs of the requests. However, sometimes it's more convenient just to get such values from within a specific request handler directly. For this, there's a `req.params` object which is an array with key value pairs.

We can add the following route to the `req/app.js`:

```
1 app.get('/params/:role/:name/:status', function(req, res) {
2   console.log(req.params)
3   res.end();
4 })
```

And after running CURL terminal commands:

```
1 $ curl http://localhost:3000/params/admin/azat/active
2 $ curl http://localhost:3000/params/user/bob/active
```



Sending GET requests with CURL.

We rightfully see these server logs of `req.params` object:

```
1  [ role: 'admin', name: 'azat', status: 'active' ]
2  [ role: 'user', name: 'bob', status: 'active' ]
```

The result of processing req.params.

17.3 req.body

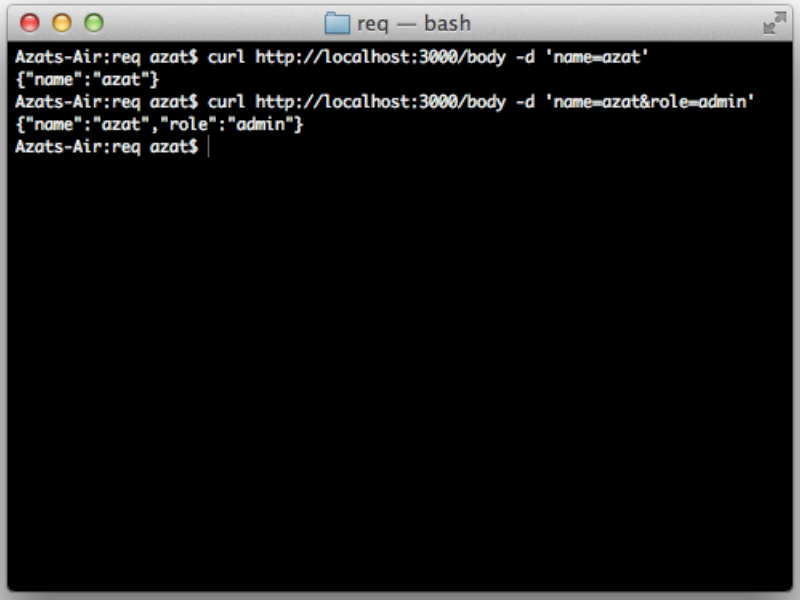
The `req.body` object is a magic that's provided to us via enabling `express.bodyParser()` middleware (or either one of its children, more on it in the Type of Middleware chapter above).

Again, let's re-use our previous project and add this route to see how `req.body` object works, remembering that `app.use(express.bodyParser());` is in the code already:

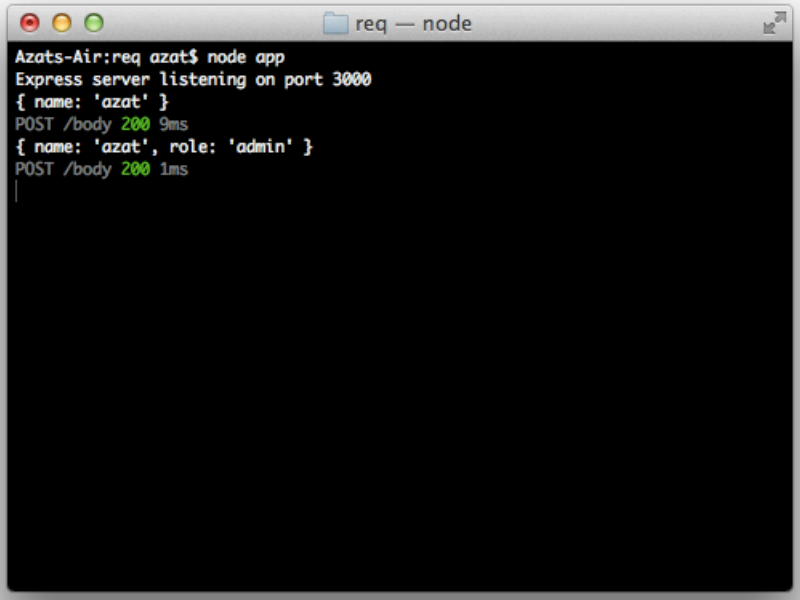
```
app.post('/body', function(req, res){
  console.log(req.body);
  res.end(JSON.stringify(req.body)+'\n');
});
```

CURL-ing with `curl http://localhost:3000/body -d 'name=azat'` and `curl http://localhost:3000/body -d 'name=azat&role=admin'` will yield `req.body`:

```
{ name: 'azat' }
{ name: 'azat', role: 'admin' }
```



Sending POST requests with CURL.



The result of processing req.body.

17.4 req.files

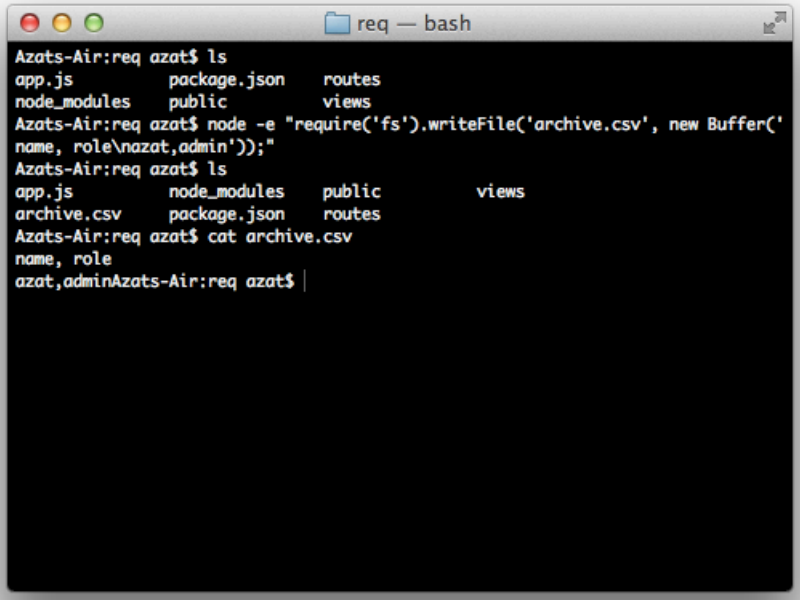
The `req.files` is used to handle file uploads. It is similar to `req.body` and is turned on either by `express.bodyParser()` or `express.multipart()` middlewares. Express.js (and other modules behind the scene) process the request data which is usually a form and give us extensive information in the `req.files.FIELD_NAME` object.

To illustrate how `req.files` works, let's add this route to the req project:

```
app.post('/upload', function(req, res){
  console.log(req.files.archive);
  //read req.files.archive.path
  //process the data
  //save the data
  res.end();
})
```

We can create a file for an upload with `$ node -e`. Let's say we have a legacy system archive of user names and roles in a CSV format:

```
$ node -e "require('fs').writeFile('archive.csv', new Buffer('name, role\nazat,ad\
min'));"
```

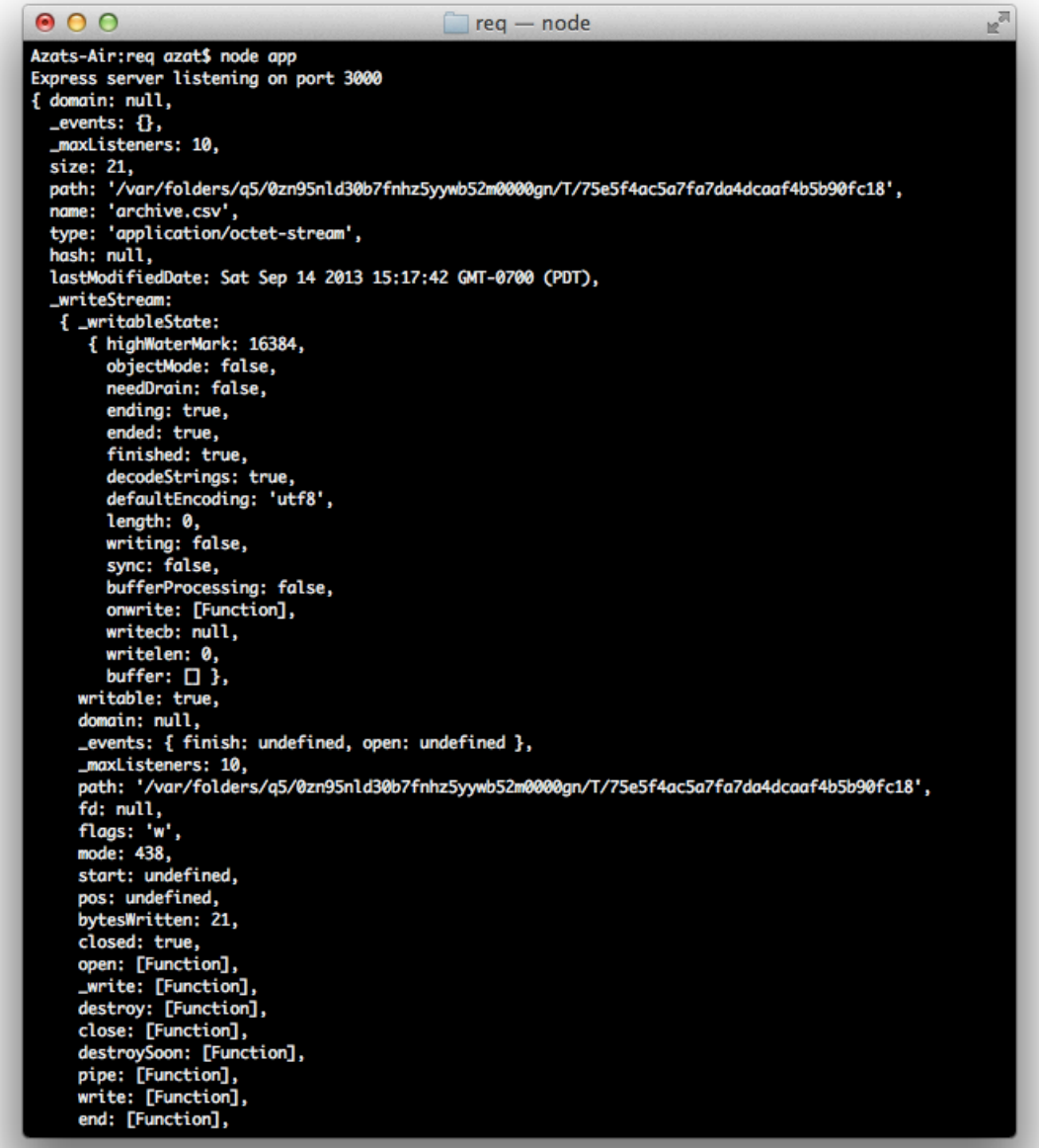


Creating a file for an upload with Node.js.

Now since we have a file, we can upload it by sending a POST request. Please note that `archive` in this command must match the property name you're reading on the server, i.e., `req.files.archive`:

```
$ curl -F archive=@./archive.csv http://localhost:3000/upload/
```

As a result of our POST CURL command, Node.js will save the file on the server, and output the `req.files.archive` to us:



The path and name properties of the `req.files.archive` object.

The `req.files.archive` will have a lot of useful attributes and methods. The most important of them are:

- `name`: file name
- `path`: path to the file on the server
- `type`: file type

- `size`: size of the file

In addition to that, `req.files.archive` could be processed as a stream. This is important when we need to handle large files such as data, video/audio and we want to start operating with it without waiting until the last byte of the file is loaded.

17.5 req.route

The `req.route` object simply has the current route's information such as:

- `path`: original URL pattern of the request
- `method`: HTTP method of the request
- `keys`: list of parameters in the URL pattern (i.e., values prefixed with `:`)
- `regexp`: Express.js-generated pattern for the path
- `params`: req.params object

We can add the `console.log(req.route);` statement to our `req.params` route from the example above like this:

```
app.get('/params/:role/:name/:status', function(req, res) {
  console.log(req.params);
  console.log(req.route);
  res.end();
});
```

We might expect to see server logs of `req.route` object:

```
[ role: 'admin', name: 'azat', status: 'active' ]
{ path: '/params/:role/:name/:status',
  method: 'get',
  callbacks: [ [Function] ],
  keys:
   [ { name: 'role', optional: false },
     { name: 'name', optional: false },
     { name: 'status', optional: false } ],
  regexp: /^\/params\/(?:([^\/]+?))\/(?:([^\/]+?))\/(?:([^\/]+?))\/?$/i,
  params: [ role: 'admin', name: 'azat', status: 'active' ] }
```

17.6 req.cookies

When the `express.cookieParser()` middleware is enabled (please refer to the Type of Middleware for more information), we get access to the request cookies (user-agent cookies) via `req.cookies` object. Cookies are automatically presented as a JavaScript object, e.g.,

```
req.cookies.session_id
```

17.7 req.signedCookies

The `req.signedCookies` is akin to aforementioned `req.cookies`; however, it's used when the secret string is passed to the `express.cookierParser('some secret string');` method.

Warning

Singed cookies **does not** mean that the cookie is hidden or encrypted. It's a simple way to prevent tampering by applying a private value.

17.8 req.header() and reg.get()

The `req.header` and `req.get` methods are identical and allow for retrieval of the request headers by their names. The naming is case-insensitive:

```
app.get('Content-type');
app.header('content-type');
```

17.9 Other Attributes and Methods

There are plenty of other objects in the Express.js Request:

- `req.accepts()`: true if passed string (single or comma separated values) or an array of MIME types (or extensions) matches against the request **Accept** header, false if there's no match (API[1])
- `req.accepted`: an array of accepted MIME types (API[2])
- `req.is()`: true if a passed MIME type string matches the **Content-Type** header types, false if there's no match (API[3])
- `req.ip`: the IP address of the request, please see `trust proxy` configuration for proxy situations (API[4])
- `req.ips`: an array of IPs when `trust proxy` config is enabled (API[5])

[1] http://expressjs.com/api.html#req.accepts
[2] http://expressjs.com/api.html#req.accepted
[3] http://expressjs.com/api.html#req.is
[4] http://expressjs.com/api.html#req.ip
[5] http://expressjs.com/api.html#req.ips

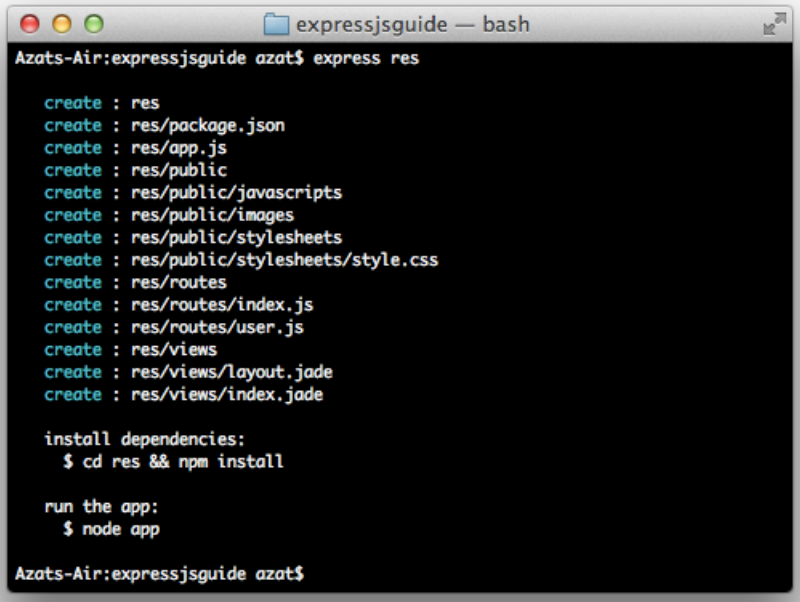
- `req.path`: string with a URL path of the request (API[6])
- `req.host`: value from the **Host** header of the request (API[7])
- `req.fresh`: true if request is *fresh* based on **Last-Modified** and **ETag** headers, false otherwise (API[8])
- `req.stale`: opposite of `req.fresh` (API[9])
- `req.xhr`: true if the request is an AJAX call via **X-Requested-With** header and its **XMLHttpRequest** value (API[10])
- `req.protocol`: request protocol value, e.g., http or https (API[11])
- `req.secure`: true if the request protocol is `https` (API[12])
- `req.subdomains`: array of subdomains from the **Host** header (API[13])
- `req.originalUrl`: unchangeable value of the request URL (API[14])
- `req.acceptedLanguages`: array of language code (e.g., en-us, en) from the request's **Accept-Language** header (API[15])
- `req.acceptsLanguage()`: true if a passed language code is in the request header (API[16])
- `req.acceptedCharsets`: array of charsets (e.g., iso-8859-5) from the request's **Accept-Charset** header (API[17])
- `req.acceptsCharset()`: true if a passed charset is in the request header (API[18])

[6] http://expressjs.com/api.html#req.path
[7] http://expressjs.com/api.html#req.host
[8] http://expressjs.com/api.html#req.fresh
[9] http://expressjs.com/api.html#req.stale
[10] http://expressjs.com/api.html#req.xhr
[11] http://expressjs.com/api.html#req.protocol
[12] http://expressjs.com/api.html#req.secure
[13] http://expressjs.com/api.html#req.subdomains
[14] http://expressjs.com/api.html#req.originalUrl
[15] http://expressjs.com/api.html#req.acceptedLanguages
[16] http://expressjs.com/api.html#req.acceptsLanguage
[17] http://expressjs.com/api.html#req.acceptedCharsets
[18] http://expressjs.com/api.html#req.acceptsCharset

18 Response

Express.js response object — which we get inside of the request handler callbacks — is the same good old Node.js `http.response` object[1] but on steroids. If somebody ever wrote a web server with only core Node.js modules, there's only the `res.end()` method[2] that finishes the request. The Express.js response object contains many convenient wrappers, like `res.json()` or `res.send()`.

Let's create a brand new Express.js app again with the `$ express res` terminal command.



The result of running `$ express res` from the project folder.

Obviously, now we need to run `$ cd res && npm install` to download the dependencies. For a screenshot, please refer to the `expressjsguide/req` example in the previous chapter.

18.1 res.render()

The `res.render()` is the staple of Express.js. From our previous examples and from the function's name, you could guess that it has something to do with generating HTML out of templates (such as

[1] http://nodejs.org/api/http.html#http_class_http_serverresponse
[2] http://nodejs.org/api/http.html#http_response_end_data_encoding

jade, Handlebars, or EJS) and data.

The method takes three parameters, but only one is mandatory: **template name** in a string format. The template name can be with or without extension. For more information on template engine extensions, please refer to the chapter **Different Template Engines**.

Here is an example of such a simple setup for the home page route in the `res/app.js` file:

```
app.get('/render', function(req, res) {
  res.render('render');
});
```

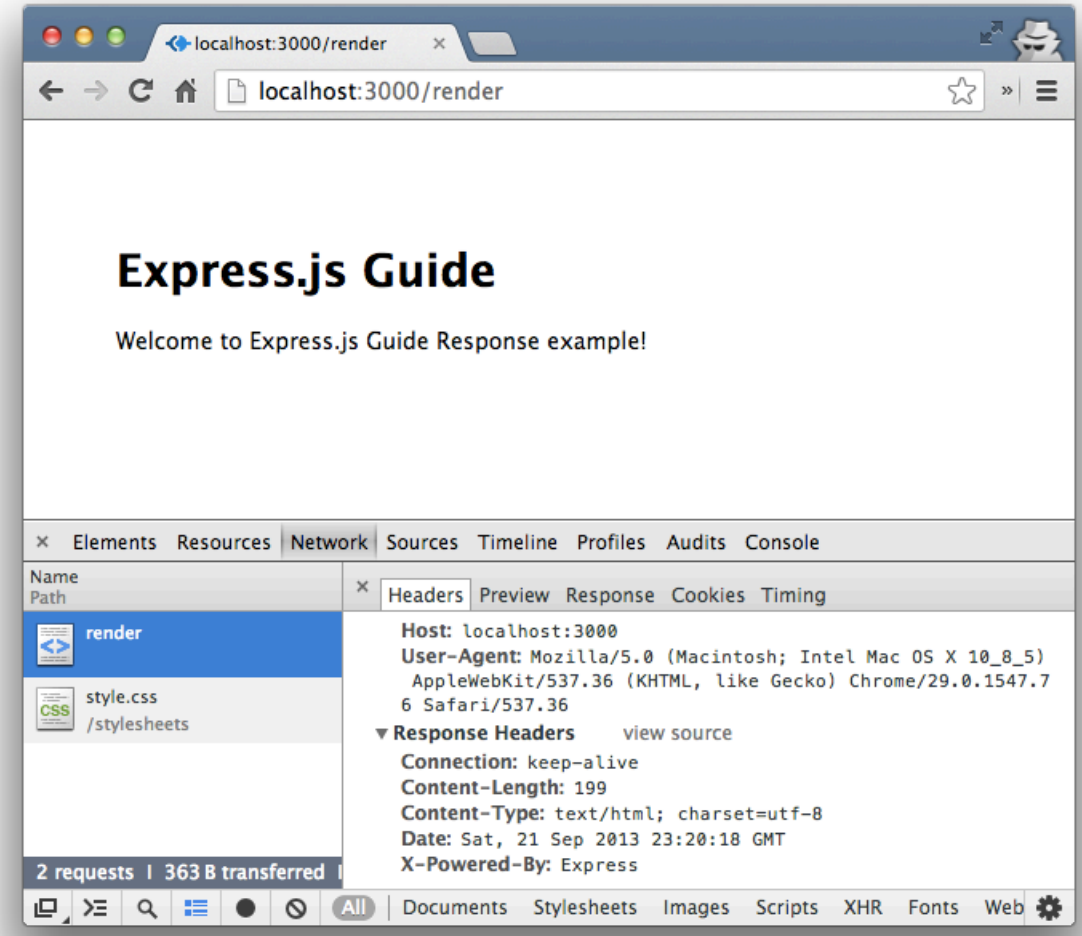
The new `render.jade` file looks static for now:

```
extends layout

block content
  h1= 'Express.js Guide'
  p Welcome to Express.js Guide Response example!
```

Note

Jade uses Python/Haml like syntax which takes into account white spaces and tabs — be careful with the markup. We can use = as a *print* command (h1 tag) or nothing (p tag). For more information, please visit the official documentation[3].

[3] http://jade-lang.com/



The result of plain `res.render()` call without params.

The full source code for this chapter's example is under `expressjsguide/res` folder and on GitHub[4]. This is how `res/app.js` file looks like (including other examples):

[4] http://github.com/azat-co/expressjsguide/

```
/**
 * Module dependencies.
 */

var express = require('express');
var routes = require('./routes');
var user = require('./routes/user');
var http = require('http');
var path = require('path');

var app = express();

// all environments
app.set('port', process.env.PORT || 3000);
app.set('views', __dirname + '/views');
app.set('view engine', 'jade');
app.use(express.favicon());
app.use(express.logger('dev'));
app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(app.router);
app.use(express.static(path.join(__dirname, 'public')));

// development only
if ('development' == app.get('env')) {
  app.use(express.errorHandler());
}

app.get('/', routes.index);
app.get('/users', user.list);

app.get('/render', function(req, res) {
  res.render('render');
});

app.get('/render-title', function(req, res) {
  res.render('index', {title: 'Express.js Guide'});
});

app.get('/locals', function(req, res){
  res.locals = { title: 'Express.js Guide' };
  res.render('index');
```

```
});

app.get('/set-html', function(req, res) {
  //some code
  res.set('Content-type', 'text/html');
  res.end('<html><body>' +
    '<h1>Express.js Guide</h1>' +
    '</body></html>');
});

app.get('/set-csv', function(req, res) {
  var body = 'title, tags\n' +
    'Express.js Guide, node.js express.js\n' +
    'Rapid Prototyping with JS, backbone.js, node.js, mongodb\n' +
    'JavaScript: The Good Parts, javascript\n'
  res.set({'Content-Type': 'text/plain',
    'Content-Length': body.length,
    'Set-Cookie': ['type=reader', 'language=javascript']});
  res.end(body);
});

app.get('/status', function(req, res) {
  res.status(200).send('alive');
});

app.get('/send-ok', function(req, res) {
  res.send(200, {message: 'Data was submitted successfully.'});
});

app.get('/send-err', function(req, res) {
  res.send(500, {message: 'Oops, the server is down.'});
});

app.get('/send-buf', function(req, res) {
  res.set('Content-Type', 'text/plain');
  res.send(new Buffer('CSV data in text format'));
});

app.get('/json', function(req, res) {
  res.json(200, [{title: 'Express.js Guide', tags: 'node.js express.js'},
    {title: 'Rapid Prototyping with JS', tags: 'backbone.js, node.js, mongodb'},
    {title: 'JavaScript: The Good Parts', tags: 'javascript'}
```

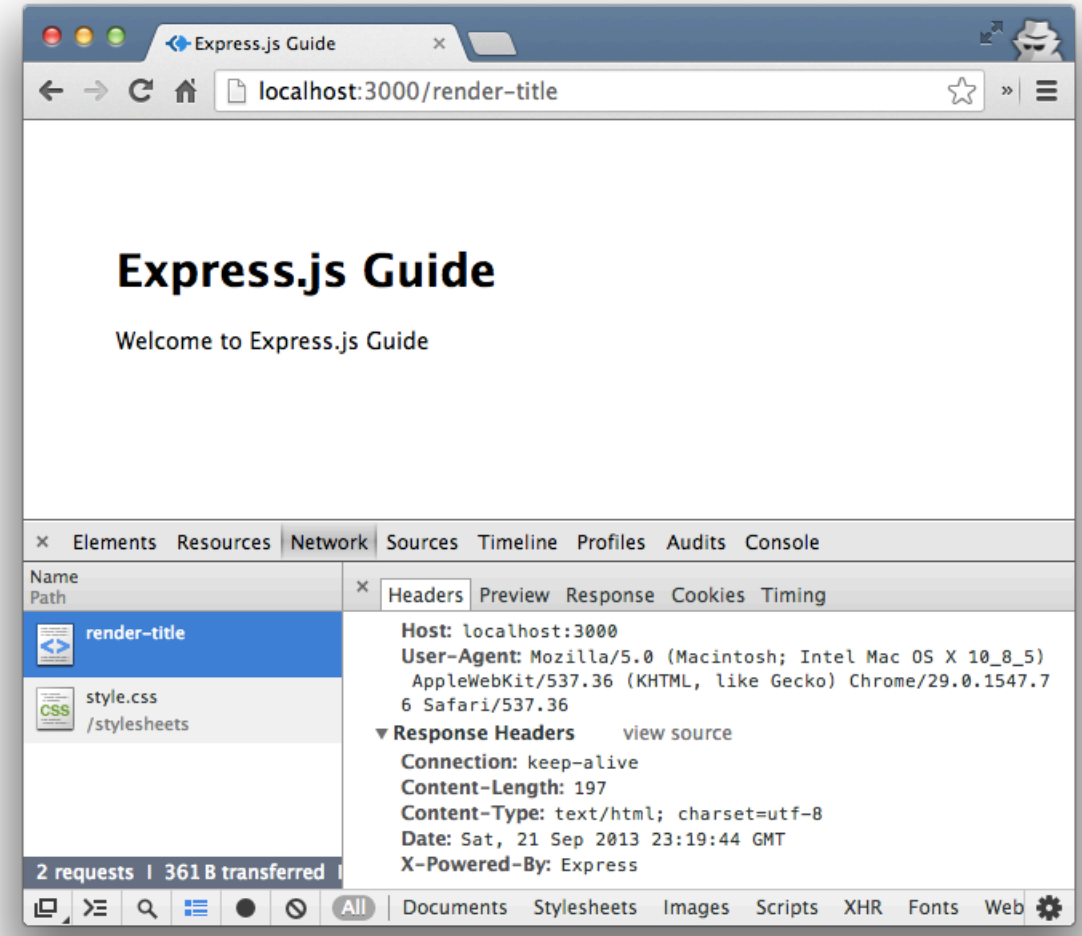
```
  ]);
});

app.get('/jsonp', function (req, res) {
  res.jsonp(200, [{title: 'Express.js Guide', tags: 'node.js express.js'},
    {title: 'Rapid Prototyping with JS', tags: 'backbone.js, node.js, mongodb'},
    {title: 'JavaScript: The Good Parts', tags: 'javascript'}
  ]);
});
http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

Two more parameters are optional and they are **data** and **callback**. The former makes templates more dynamic than static HTML files, and allows us to update the output.

```
app.get('/render-title', function(req, res) {
  res.render('index', {title: 'Express.js Guide'});
});
```

The `index.jade` file remains the same:

```
extends layout

block content
  h1= title
  p Welcome to #{title}
```



The `res.render()` example with the data parameter that has title property.

The latter, i.e., the third `res.render()` param callback, accepts two parameters itself: error and an HTML string (the output). This example is not in the `res/app.js` project, but shows how to pass callbacks to `res.render()`:

```
app.get('/render-title', function(req, res) {
  res.render('index', {title: 'Express.js Guide'}, function (error, html) {
    //do something
  });
});
```

Warning

The properties of the **data** parameter are your **locals** in the template. In other words, if you want to access a value of title inside of your template, the data object must contain such key and value pair. Nested object are supported by most of the template engines.

The callback can take the place of the data due to Express.js' ability to determine the type of the parameter. This example is not in the `res/app.js`, but shows how to pass callbacks with our data:

```
app.get('/render-title', function(req, res) {
  res.render('index', function (error, html) {
    //do something
  });
});
```

Behind the scene, `res.render()` calls `res.send()` (covered later) for successful compilation of HTML strings or `req.next(error)` for failure, **if the callback is not provided**. In other words, the default callback to `res.render()` is (GitHub[5]):

```
  // default callback to respond
  fn = fn || function(err, str){
    if (err) return req.next(err);
    self.send(str);
  };
```

18.2 res.locals()

The `res.locals()` object is another way to pass data to the templates so they both can be compiled into HTML. The first way is to pass data as a parameter to `res.render()` method as outlined above:

```
app.get('/render-title', function(req, res) {
  res.render('index', {title: 'Express.js Guide'});
});
```

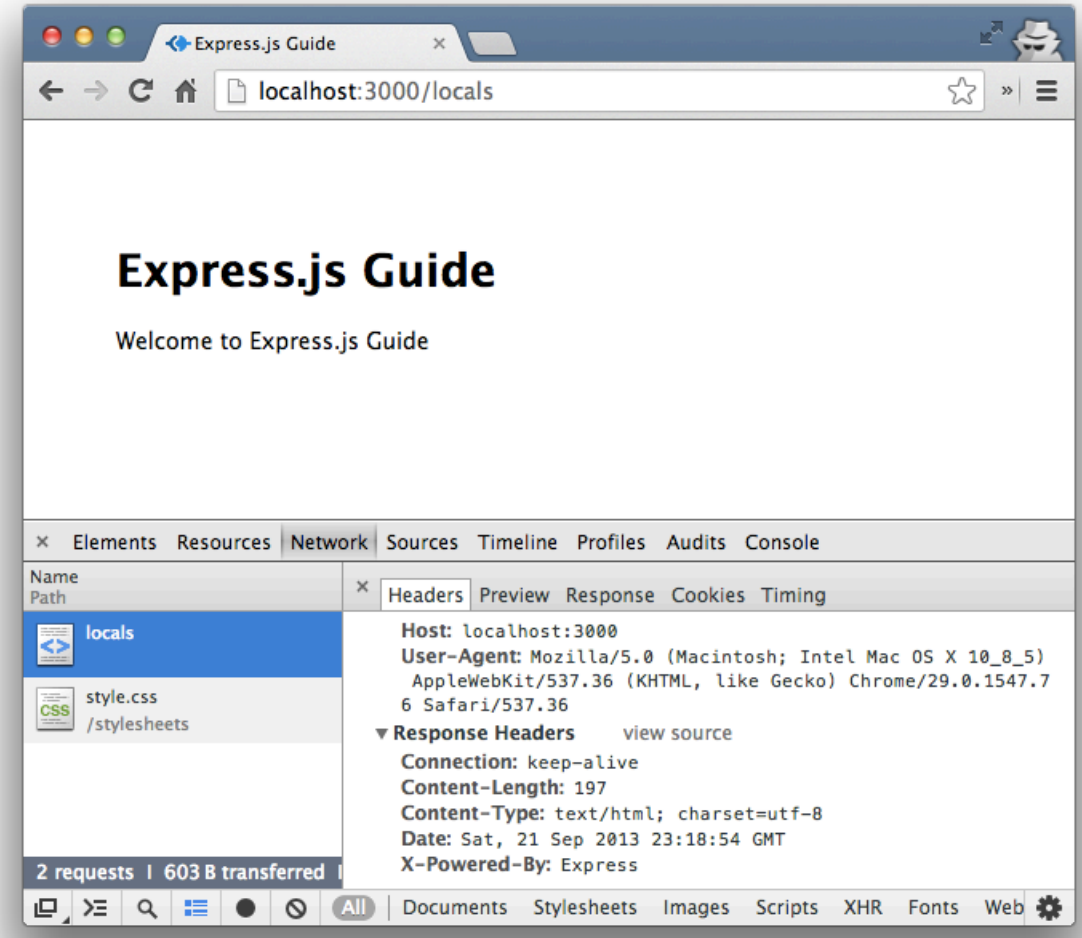
However, with `res.locals`, we can achieve the same thing. Our object will be available inside of the template:

[5] https://github.com/visionmedia/express/blob/3.3.5/lib/response.js#L753

```
app.get('/locals', function(req, res){
  res.locals = { title: 'Express.js Guide' };
  res.render('index');
});
```

Again, the `index.jade` jade template remains the same:

```
extends layout

block content
  h1= title
  p Welcome to #{title}
```



The `res.locals` example renders the same page as `res.render()` example.

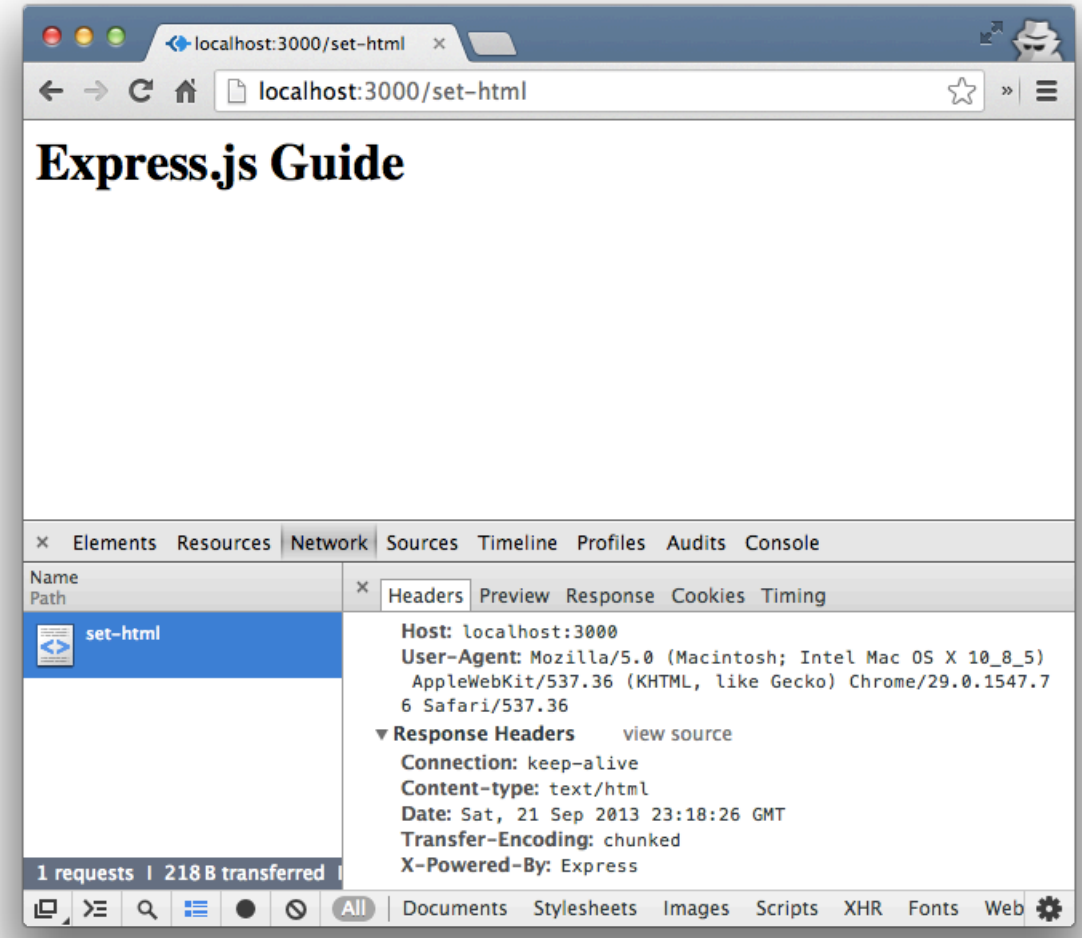
18.3 res.set()

The `res.set()` method is an alias of `res.header()` (or the other way around) and serves as a wrapper for the Node.js http core module's `response.setHeader()` function[6]. The main difference is that `res.set()` is smart enough to call itself recursively when we pass to it multiple header-value pair in the form of an object.

Here is an example of setting a single response header:

[6] http://nodejs.org/api/http.html#http_response_setheader_name_value

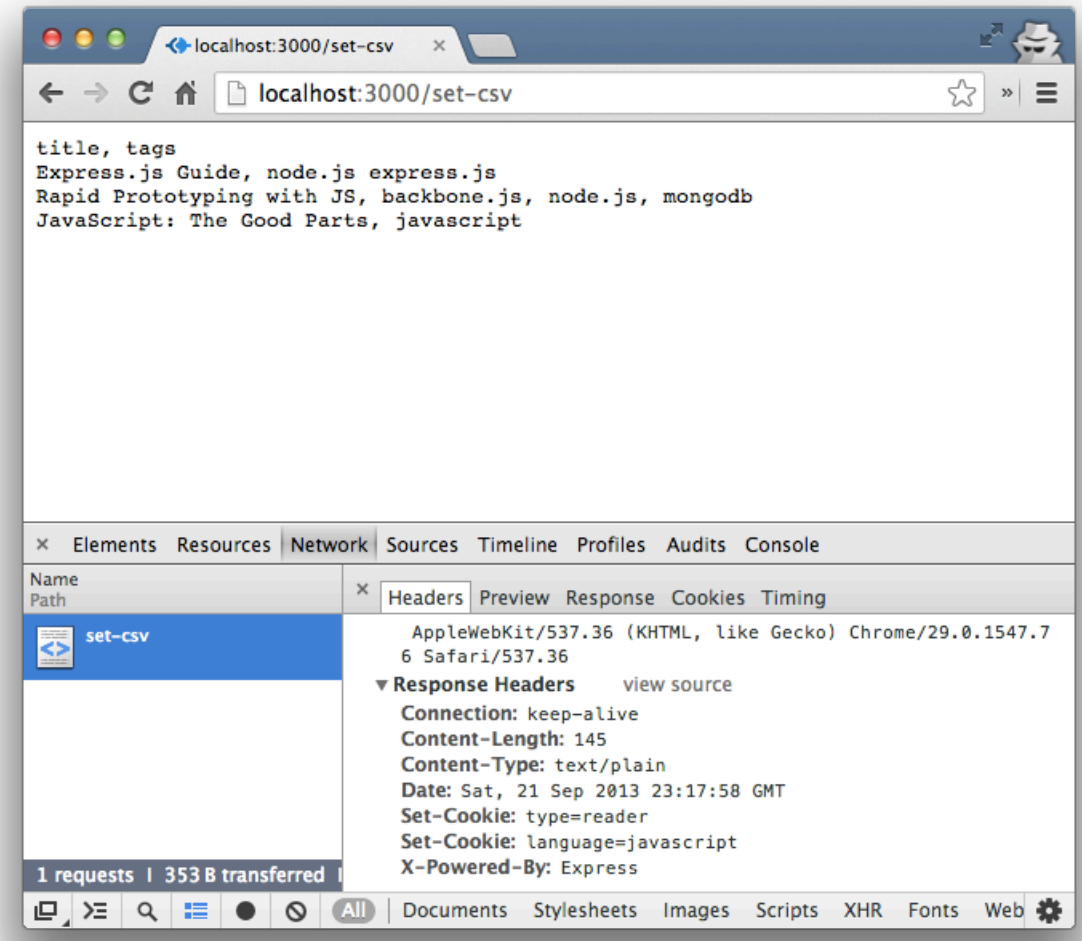
```
app.get('/set-html', function(req, res) {
  //some code
  res.set('Content-type', 'text/html');
  res.end('<html><body>' +
    '<h1>Express.js Guide</h1>' +
    '</body></html>');
});
```



The `res.set()` example rendering HTML.

Often times though, our servers need to provide more than one header so all the different browsers and other HTTP clients process it properly. Imagine that the service we are building sends out comma-separated values (CVS) files with books' titles and their tags. This is how we can implement this route:

```
app.get('/set-csv', function(req, res) {
  var body = 'title, tags\n' +
    'Express.js Guide, node.js express.js\n' +
    'Rapid Prototyping with JS, backbone.js, node.js, mongodb\n' +
    'JavaScript: The Good Parts, javascript\n'
  res.set({'Content-Type': 'text/plain',
    'Content-Length': body.length,
    'Set-Cookie': ['type=reader', 'language=javascript']});
  res.end(body);
});
```



The `res.set()` example rendering Content-Type, Content-Length and Set-Coookie headers with CSV data.

Exercise

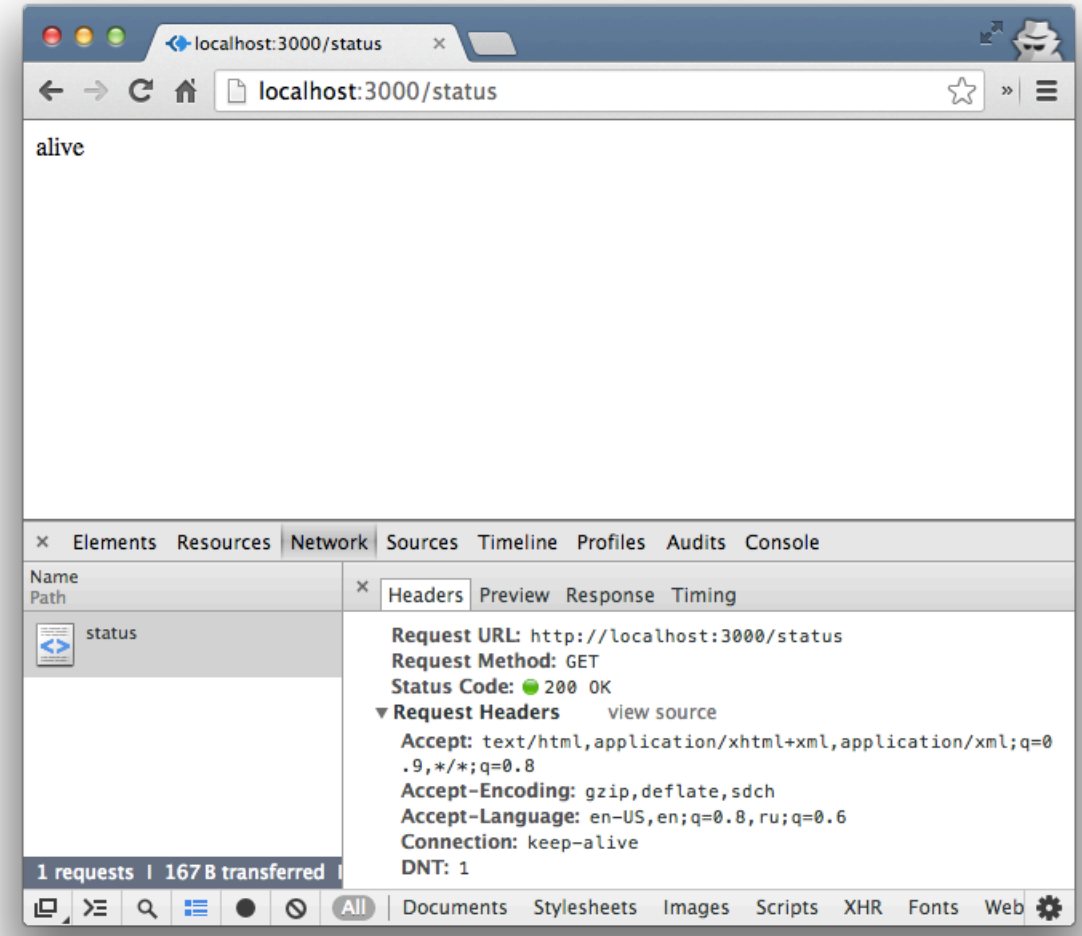
The screen shot of the `res.set()` example above was taken after adding the route to the `app.js` file of the `res` project. Readers are encouraged to try doing this on their own.

18.4 res.status()

The `res.status()` accepts HTTP status code[7] number and sends it in response. The only difference between its core counterpart[8] is that `res.status()` is chainable:

```
1 app.get('/status', function(req, res) {
2   res.status(200).send('alive');
3 });
```

[7] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html
[8] http://nodejs.org/api/http.html#http_response_statuscode



The `res.status()` example response.

18.5 res.send()

The `res.send()` method lies somewhere in between high-level `res.render()` and low-level `res.end()`. The `res.send()` conveniently outputs any data application thrown at it (such as strings, JavaScript objects, and even Buffers) with automatically generated proper HTTP headers, e.g., Content-Length, ETag or Cache-Control.

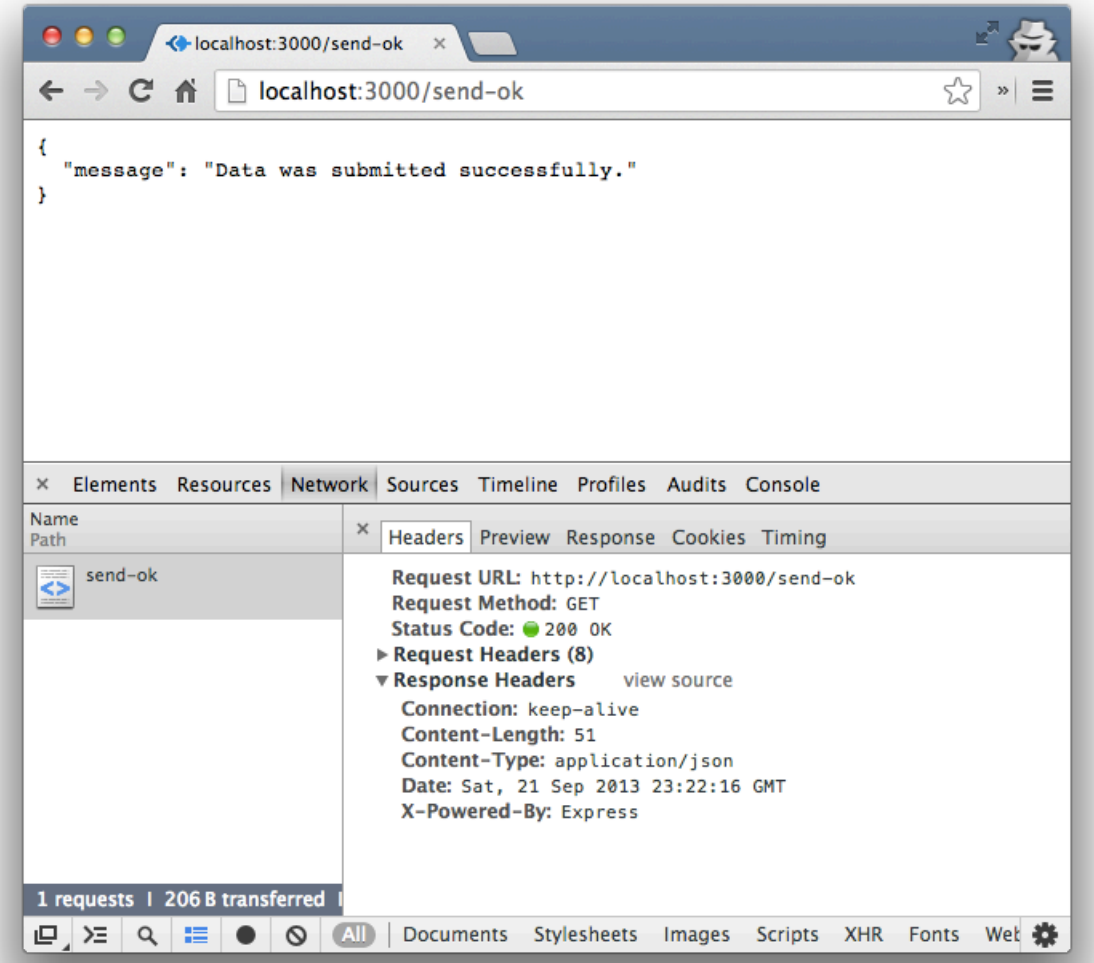
Due to its omnivorous behavior (cause by `arguments.length`), `res.send()` can be used in countless ways:

- Status code number: `res.send(500);` for Server Error or `res.send(200);` for OK

- String: `res.send('success');`
- Object: `res.send({message: 'success'});` or `res.send({message: 'error'});`
- Array: `res.send([{title: 'Express.js Guide'}, {title: 'Rapid Prototyping with JS'}]);`
- Buffer: `res.send(new Buffer('Express.js Guide'));`

The status code and data parameters can be combined, for example:

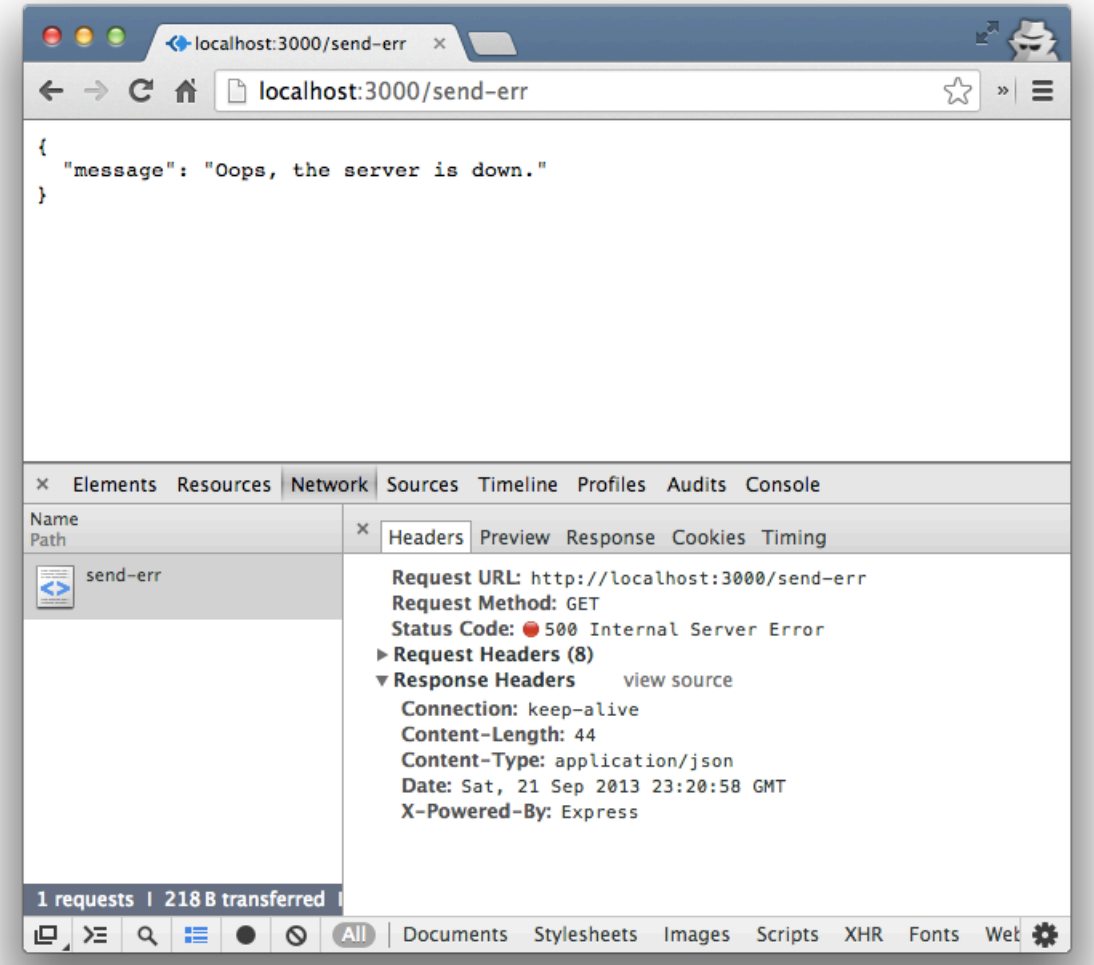
```
app.get('/send-ok', function(req, res) {
  res.send(200, {message: 'Data was submitted successfully.'});
});
```



The `res.send()` 200 status code example response.

Or, for those error cases:

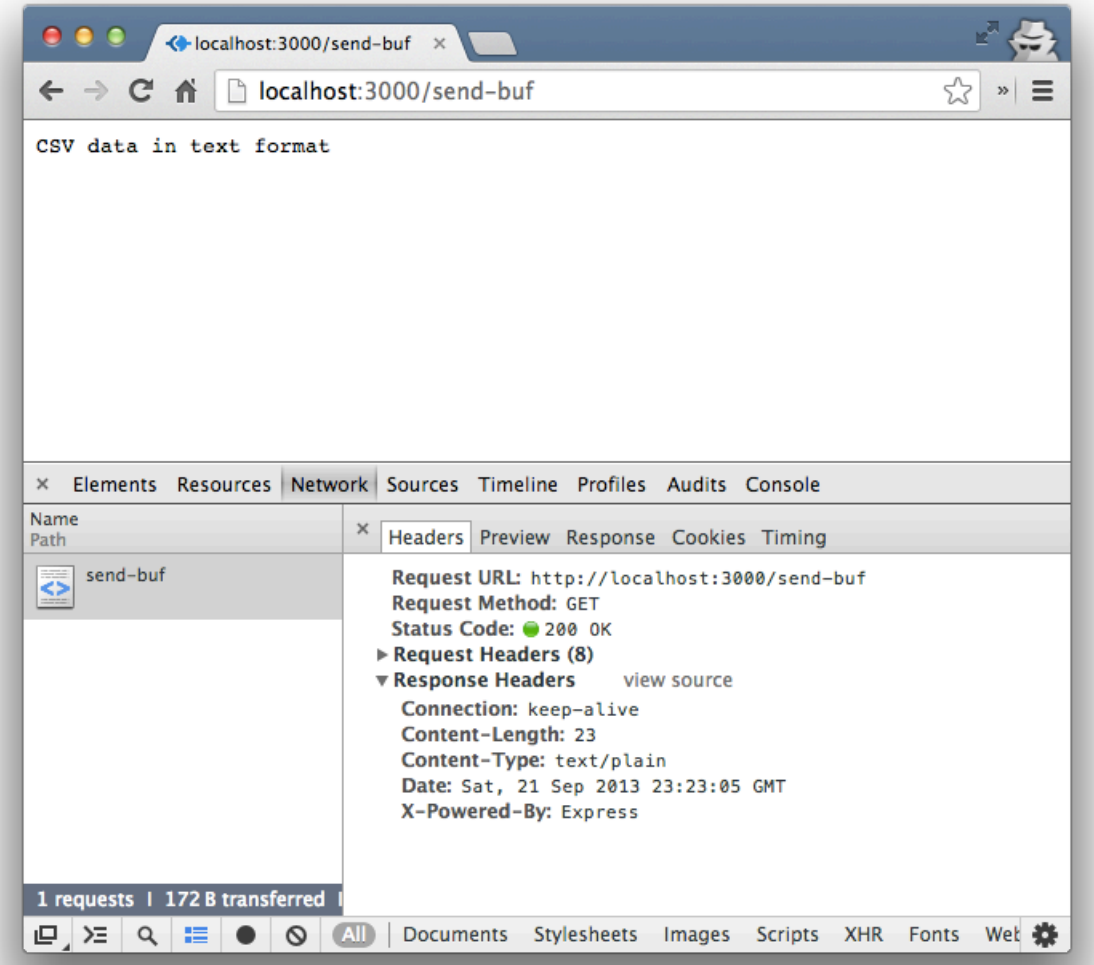
```
1  app.get('/send-err', function(req, res) {
2    res.send(500, {message: 'Oops, the server is down.'});
3  });
```



The `res.send()` 500 status code example response.

The headers generated by `res.send()` might be overwritten if specified explicitly before. For example, Buffer type will have Content-Type as `application/octet-stream` but we can change it to `text/plain` with:

```
app.get('/send-buf', function(req, res) {
  res.set('Content-Type', 'text/plain');
  res.send(new Buffer('CSV data in text format'));
});
```



The `res.send()` Buffer example response.

Note

Please note that virtually all of core Node.js methods (and ConnectJS as well) are available in Express.js objects. Therefore, we have access to `res.end()` and other methods in Express.js response API.

18.6 res.json()

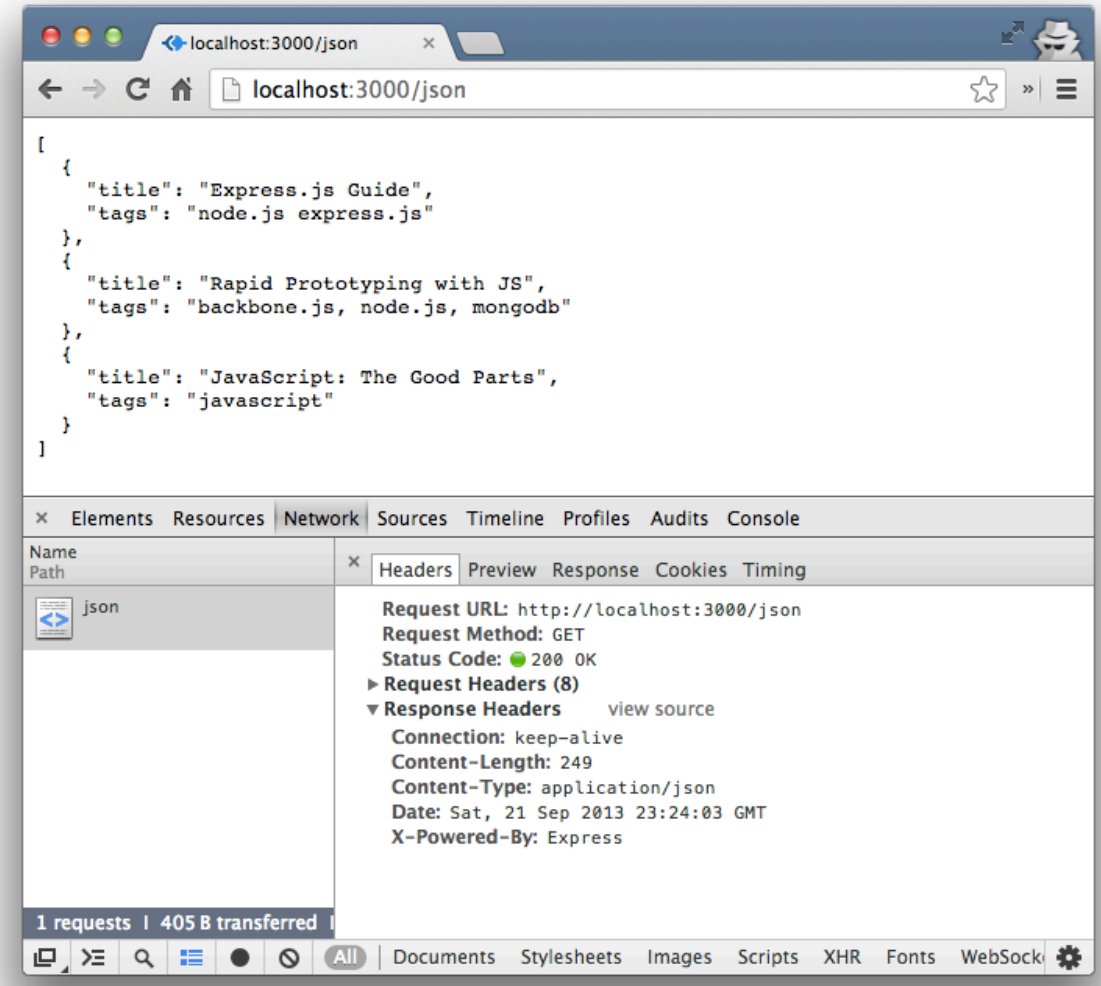
The `res.json()` method is a convenient way of sending JSON data. It's equivalent to `res.send()` when data passed is of type Array or Object. In other cases, `res.json()` forces data conversion

with `JSON.stringify()`. By default, the header Content-Type is set to `application/json`, but can be overwritten prior to `res.json()` with `res.set()`.

If you remember our old friends from the Settings chapter, the `json replacer` and `json spaces`, that's where they're taken into account.

Most common use of `res.json()` is with appropriate status codes:

```
app.get('/json', function(req, res) {
  res.json(200, [{title: 'Express.js Guide', tags: 'node.js express.js'},
    {title: 'Rapid Prototyping with JS', tags: 'backbone.js, node.js, mongodb'},
    {title: 'JavaScript: The Good Parts', tags: 'javascript'}
  ]);
});
```



The result of using `res.json()`: automatically generated headers.

Exercise

The screen shot of the `res.json()` example above was taken after adding the route to the `res/app.js` file of the `expressjsguide/res` project. Readers are encouraged to try doing this on their own.

Other uses of `res.json()` possible as well, for example with no status code:

```
app.get('/api/v1/stories/:id', function(req,res){
  res.json(req.story);
});
```

Assuming `req.story` is an array or an object, the following code would produce similar results to the snippet above (no need to set the header to `application/json` in either case):

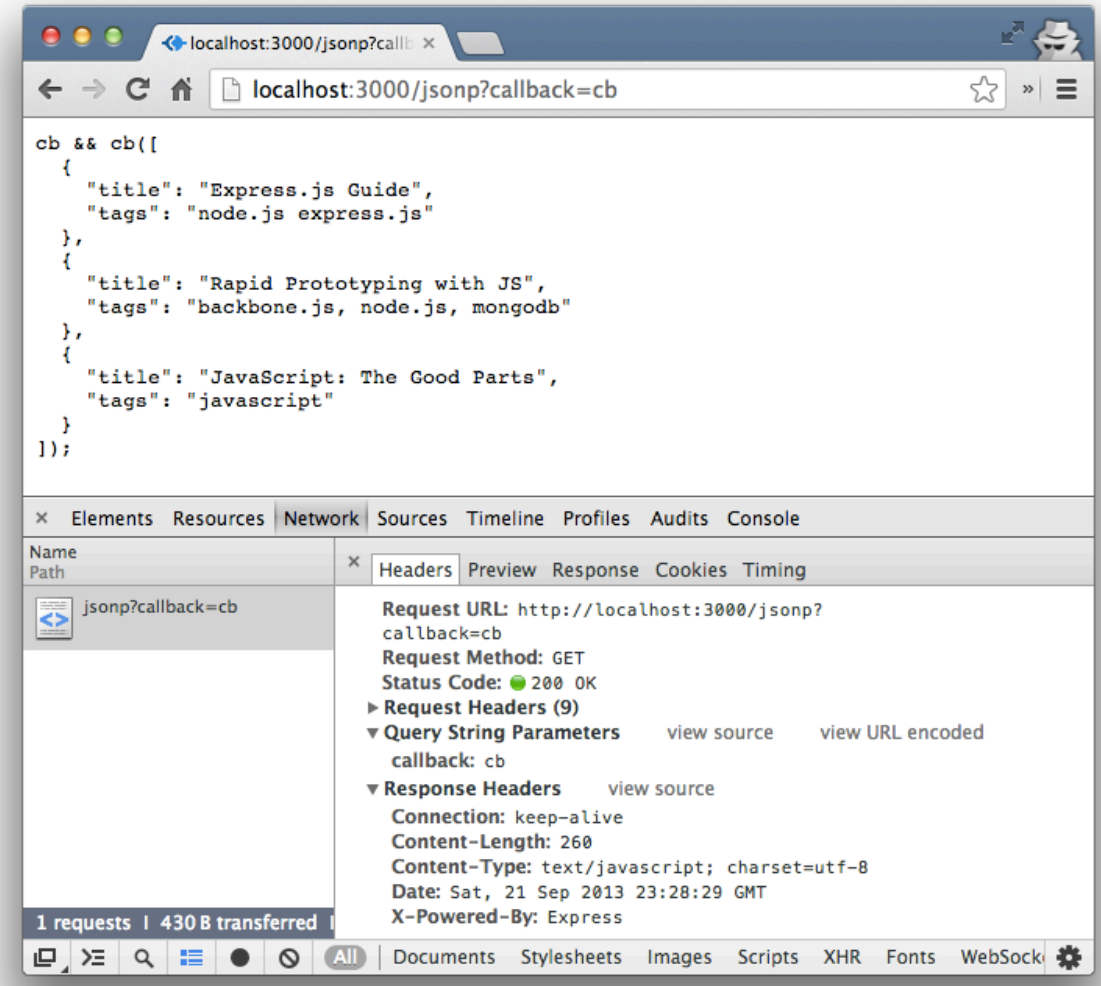
```
app.get('/api/v1/stories/:id', function(req,res){
  res.send(req.story);
});
```

18.7 res.jsonp()

The `res.jsonp()` method is similar to `res.json()`, but provides JSONP response. That is, the JSON data is wrapped in a JavaScript function call, e.g., `processResponse({...});`. This is usually used for cross-domain calls support. By default, Express.js uses `callback` name to extract the name of the callback function. It's possible to override this value with `json callback name` settings (more on it in the Settings chapter). If there is no proper callback specified in the query string of the request (e.g., `?callback=cb`), then the response is simply JSON.

Assume that we need to serve CSV data to front-end request via JSONP:

```
app.get('/', function (req, res) {
  res.jsonp(200, [{title: 'Express.js Guide', tags: 'node.js express.js'},
    {title: 'Rapid Prototyping with JS', tags: 'backbone.js, node.js, mongodb'},
    {title: 'JavaScript: The Good Parts', tags: 'javascript'}
  ]);
});
```



The result of `res.jsonp()` and ?callback=cb is a `text/javascript` header and JavaScript function prefix.

Exercise

The screen shot of the `res.json()` example above was taken after adding the route to the `index.js` file of the `cli-app` project. Readers are encouraged to try doing this on their own.

18.8 res.redirect()

Sometimes it's needed to just redirect users/requests to another route. We can use absolute, relative or full paths:

```
res.redirect('/admin');
res.redirect('../users');
res.redirect('http://rapidprototypingwithjs.com');
```

By default, `res.redirect()` sends 302: Found/Temporarily Moved status code[9]. Of course we can configure it to our liking in the same manner as `res.send()`, i.e., passing first status code number as the first parameter:

```
res.redirect(301, 'http://rpjs.co');
```

18.9 Other Response Methods and Properties

Most of these methods and properties are convenient alternatives to the methods covered already in the book. In other words, we can accomplish most of the logic with the main methods, but knowing these shortcuts can save developers a few keystrokes and improve readability. For example, `res.type()` is a niche case of `res.header()` for Content-Type only header.

- `res.get()`: string value of response header for a passed header type (API[10])
- `res.cookie()`: takes cookie key-value pair and sets it on response (API[11])
- `res.clearCookie()`: takes cookie key/name and optional path parameter to clear the cookies (API[12])
- `res.location()`: takes relative, absolute or full paths as a string and sets that value to Location response header (API[13])
- `res.charset`: the charset value of the response (API[14])
- `res.type()`: takes a string and sets it as a value of Content-Type header (API[15])
- `res.format()`: takes an object as a mapping of types and responses and executes them according to Accepted request header (API[16])
- `res.attachment()`: takes optional filename as a string and sets Content-Disposition and if filename provided Content-Type headers to `attachment` and file type accordingly (API[17])
- `res.sendfile()`: takes path to a file on the server and optional options and callback parameters, and sends the file to the requester (API[18])

[9] http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html
[10] http://expressjs.com/api.html#res.get
[11] http://expressjs.com/api.html#res.cookie
[12] http://expressjs.com/api.html#res.clearCookie
[13] http://expressjs.com/api.html#res.location
[14] http://expressjs.com/api.html#res.charset
[15] http://expressjs.com/api.html#res.type
[16] http://expressjs.com/api.html#res.format
[17] http://expressjs.com/api.html#res.attachment
[18] http://expressjs.com/api.html#res.sendfile

- `res.download()`: takes same params as `res.sendfile()`, and sets Content-Disposition and calls `res.sendfile()` (API[19])
- `res.links()`: takes an object of URLs to populate Links response header (API[20])

[19] http://expressjs.com/api.html#res.download
[20] http://expressjs.com/api.html#res.links

19 Error Handling

Due to asynchronous nature of Node.js and callback pattern, the state in which errors happen is not that trivial to catch and log for future analysis. In the Tips and Tricks chapter, we cover the use of domains with Express.js apps. It's a more advanced technique and for most implementations right out of the box framework, error handling might prove sufficient.

We can start with the basic `errorHandler()` middleware from our `cli-app` example which is enabled by this code:

```
1  // development only
2  if ('development' == app.get('env')) {
3    app.use(express.errorHandler());
4  }
```

This makes sense because error handling is typically used across the whole application, therefore it's best to implement it as a middleware.

For custom error handler implementations, the middleware is the same as any other except that it has one more param, `error` or `err` for short:

```
1  //main middlewares
2  app.use(function(err, req, res, next) {
3    //do logging and user-friendly error message display
4    console.error(err);
5    res.send(500);
6  });
7  //routes
```

In fact, the response can be anything: JSON, text, redirect to a static page or something else.

For most front-end and other JSON clients, the preferred format is of course JSON:

```
1  app.use(function(err, req, res, next) {
2    //do logging and user-friendly error message display
3    console.error(err);
4    res.send(500, {status:500, message: 'internal error', type:'internal'});
5  })
```

Note

Developers can use `req.xhr` property or check if Accepted request header has `application/json` value.

The most straightforward way is to just send a text:

```
app.use(function(err, req, res, next) {
  //do logging and user-friendly error message display
  console.error(err);
  res.send(500, 'internal server error');
})
```

Or if we know that it's secure to output the error message:

```
app.use(function(err, req, res, next) {
  //do logging and user-friendly error message display
  console.error(err);
  res.send(500, 'internal server error: ' + err);
})
```

To just render a static error page with the name `500` and default extension:

```
app.use(function(err, req, res, next) {
  //do logging and user-friendly error message display
  console.error(err);
  //assuming that template engine is plugged in
  res.render('500');
})
```

Or, for a full filename of `500.html`:

```
app.use(function(err, req, res, next) {
  //do logging and user-friendly error message display
  console.error(err);
  //assuming that template engine is plugged in
  res.render('500.html');
})
```

We can also use `res.redirect()`:

```
app.use(function(err, req, res, next) {
  //do logging and user-friendly error message display
  res.redirect('/public/500.html');
})
```

It is always recommended to use the proper HTTP response statuses such as 401, 400, 500, etc.:

```
app.use(function(err, req, res, next) {
  //do logging and user-friendly error message display
  res.end(500);
})
```

To trigger an error from within your request handlers and middleware, you can just call:

```
app.get('/', function(req, res, next){
  next(error);
});
```

Or, if we want to pass a specific error message:

```
app.get('/', function(req,res,next){
  next(new Error('Something went wrong :-(');
});
```

It would be a good idea to use **return** keyword for processing multiple error prone cases and combine both approaches above:

```
app.get('/users', function(req, res, next) {
  db.get('users').find({}, function(error, users) {
    if (error) return next(error);
    if (!users) return next(new Error('No users found.'));
    //do something, if fail the return next(error);
    res.send(users);
});
```

For complex apps, it's best to use multiple error handlers, for example one for XHR/AJAX requests, one for normal requests and one generic catch-everything else. It's also a good idea to use named functions (and organize them in modules) instead of anonymous ones.

For an example of such advanced error handling, please refer to the HackHall example in the Tutorial and Examples part.

20 Running an App

20.1 app.locals

The `app.locals` object is similar to aforementioned `res.locals` in the sense that it exposes data to templates. However, there's a main difference: `app.locals` makes its properties available in **all** in templates rendered by app, while `res.locals` restricts them **only** to that request. Therefore, developers needs to be careful not to reveal any sensitive information via `app.locals`. The best use case for which is app-wide settings like locales, URLs, contact info, etc., e.g.,

```
app.locals.lang = 'en';
app.locals.appName = 'HackHall';
```

The `app.locals` can also be invoked like a function:

```
app.locals([
  author: 'Azat Mardanov',
  email: 'hi@azat.co',
  website: 'http://expressjsguide.com'
]);
```

20.2 app.render()

The `app.render()` method is invoked either with a view name and a callback or with a view name, data and a callback. For example, the system might have an email template for a *Thank you for signing up* message and another for *Reset your password*:

```
var sendgrid  = require('sendgrid')(api_user, api_key);

var sendThankYouEmail = function (userEmail) {
  app.render('emails/thank-you', function(err, html){
    if (err) return console.error(err);
    sendgrid.send({
      to:       userEmail,
      from:     app.get('appEmail'),
      subject:  'Thank you for signing up',
```

```
      html:  html
    }, function(err, json) {
      if (err) { return console.error(err); }
      console.log(json);
    });
  });
}

var resetPasswordEmail = function (userEmail) {
  app.render('emails/reset-password', {token: generateResetToken()}, function(err\
, html){
    if (err) return console.error(err);
    sendgrid.send({
      to:      userEmail,
      from:     app.get('appEmail'),
      subject:  'Reset your password',
      html:  html
    }, function(err, json) {
      if (err) { return console.error(err); }
      console.log(json);
    });
  });
}
```

Note

The sendgrid module used in the example available at NPM[1] and GitHub[2].

20.3 app.routes

The `app.routes` object merely contains Express.js app routes where properties are HTTP methods. For example, `expressjsguide/res` from the Response chapter has the following `app.routes` object:

[1] http://npmjs.org/sendgrid

[2] https://github.com/sendgrid/sendgrid-nodejs

```
{ get:
   [ { path: '/',
       method: 'get',
       callbacks: [Object],
       keys: [],
       regexp: /^\/\/?$/i },
     { path: '/users',
       method: 'get',
       callbacks: [Object],
       keys: [],
       regexp: /^\/users\/?$/i },
     { path: '/render',
       method: 'get',
       callbacks: [Object],
       keys: [],
       regexp: /^\/render\/?$/i },
     { path: '/render-title',
       method: 'get',
       callbacks: [Object],
       keys: [],
       regexp: /^\/render-title\/?$/i },
     { path: '/locals',
       method: 'get',
       callbacks: [Object],
       keys: [],
       regexp: /^\/locals\/?$/i },
     { path: '/set-html',
       method: 'get',
       callbacks: [Object],
       keys: [],
       regexp: /^\/set-html\/?$/i },
     { path: '/set-csv',
       method: 'get',
       callbacks: [Object],
       keys: [],
       regexp: /^\/set-csv\/?$/i },
     { path: '/status',
       method: 'get',
       callbacks: [Object],
       keys: [],
       regexp: /^\/status\/?$/i },
     { path: '/send-ok',
```

```
43        method: 'get',
44        callbacks: [Object],
45        keys: [],
46        regexp: /^\/send-ok\/?$/i },
47      { path: '/send-err',
48        method: 'get',
49        callbacks: [Object],
50        keys: [],
51        regexp: /^\/send-err\/?$/i },
52      { path: '/send-buf',
53        method: 'get',
54        callbacks: [Object],
55        keys: [],
56        regexp: /^\/send-buf\/?$/i },
57      { path: '/json',
58        method: 'get',
59        callbacks: [Object],
60        keys: [],
61        regexp: /^\/json\/?$/i },
62      { path: '/jsonp',
63        method: 'get',
64        callbacks: [Object],
65        keys: [],
66        regexp: /^\/jsonp\/?$/i } ] }
```

This object can be used by developers who write Node.js frameworks on top of Express.js to augment routes, e.g., delete some of them.

20.4 app.listen()

The Express.js `app.listen()` method is akin `server.listen()`[3] from the core Node.js core http module. This method is cornerstone for starting Express.js apps. There are two ways to do it.

The first way is to spin up Express.js app direclty on a particular port:

[3] http://nodejs.org/api/http.html#http_server_listen_port_hostname_backlog_callback

```
var express = require('express');
var app = express();
//configuration
//routes
app.listen(3000);
```

This approach was used in `expressjsguide/hello.js` and `expressjsguide/hell-name.js` examples in the first part of this book.

The second way is to apply Express.js app to the core Node.js server function. This might be useful for spawning HTTP and HTTPS server with the same code base:

```
var express = require('express');
var https = require('https');
var http = require('http');
var app = express();
var ops = require('conf/ops');
//configuration
//routes
http.createServer(app).listen(80);
https.createServer(ops, app).listen(443);
```

The second approach is used by Express.js generator and readers observed it in `expressjsguide/cli-app`, `expressjsguide/req` and `expressjsguide/res` examples.

III Tip and Tricks

In this part we'll cover some common patterns for Node.js/Express.js development such as code organization, streams, Redis, clusters, authentication, Stylus, LESS and SASS, domains, security and Socket.IO.

21 Abstraction

Middleware concept provides flexibility. Software engineers can use anonymous or named functions as middlewares. However, the best thing is to abstract them into external modules based on the functionality. Let's say we have a REST API with stories, elements and users. We can separate request handlers into files accordingly, so that `routes/stories.js` has:

```
module.exports.findStories = function(req, res, next) {
...
};
module.exports.createStory = function(req, res, next) {
...
};
```

The `routes/users.js` holds the logic for user entities:

```
module.exports.findUser = function(req, res, next){
...
};
module.exports.updateUser = function(req, res, next){
...
};
module.exports.removeUser = function(req, res, next){
...
};
```

The main app file can use the modules above in this manner:

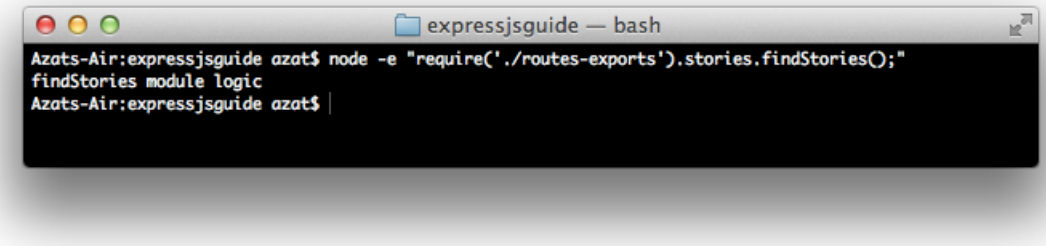
```
...
var stories = require.('./routes/stories');
var users = require.('./routes/users');
...
app.get('/stories', stories.findStories);
app.post('/stories', stories.createStory);
app.get('/users/:user_id', users.findUser);
app.put('/users/:user_id', users.updateUser);
app.del('/users/:user_id', users.removeUser);
...
```

In the example with `var stories = require('./routes/stories');`, the `stories` is a file `stories.js` with omitted (optional) `.js` extension.

Please notice that the same thing repeats itself over and over with each line/module, i.e., developers have to duplicate code by importing same modules (e.g., users). Imagine that we need to include these three modules in each other file! To avoid this, there's a clever way to include multiple files. That is to put `index.js` file inside the `stories` folder and let that file include all the routes.

For example, `require('./routes').stories.findStories();` will access `index.js` with `exports.stories = require('./find-stories.js')` which in turn reads `find-stories.js` with `exports.findStories = function(){...}`.

For a working example, you can run `node -e "require('./routes-exports').stories.findStories();"` from the `expressjsguide` folder to see a string output by `console.log` from the module:



The importing of modules via folder and index.js file.

To illustrate another approach of code re-use, assume that there's an app with these routes:

```
app.get('/admin', function(req, res, next) {
  if (!req.query._token) return next(new Error('no token provided'));
  }, function(req, res, next) {
  res.render('admin');
});
//middleware that applied to all /api/* calls
app.use('/api/*', function(req, res, next) {
  if (!req.query.api_key) return next(new Error('no api key provided'));
});
```

Wouldn't it be slick to have something like a function that returns a function instead:

```
var requiredParam = function (param) {
  //do something with the param, e.g.,
  //create a private attribute paramName based on the value of param variable
  var paramName = '';
  if (param === '_token') paramName = 'token';
  else if (param === 'api_key') paramName = 'API key'
  return function (req,res, next) {
    //use paramName, e.g.,
    //if query has no such parameter, proceed next() with error using paramName
    if (!req.query[param]) return next(new Error('no ' + paramName +' provided'));
    next();
  });
}

app.get('/admin', requiredParam('_token'), function(req, res, next) {
  res.render('admin');
});
//middleware that applied to all /api/* calls
app.use('/api/*', requiredParam('api_key'));
```

This is a very basic example and in most cases, developers don't worry about mapping for proper error text message. Nevertheless, you can use this pattern for anything like restricting output, permissions, switching between different blocks.

Note

The `__dirname` global provides an absolute path to the file that uses it. On the contrary, `./` returns current working directory which might be different depending from where we execute Node.js script (e.g, `$ node ~/code/app/index.js` vs. `$ node index.js`). One exception to `./` rule is when it's used in `require()` function, e.g., `conf = require('./config.json');`, then it acts as `__dirname`.

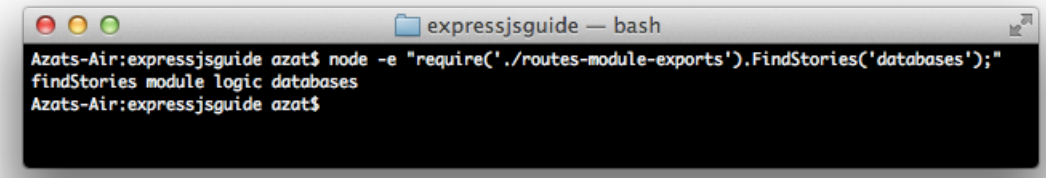
As you can see, middleware/request handlers is a powerful concept for keeping code organized. The best practice is to keep router lean and thin by moving all of the logic into corresponding external modules/files. This way, important server configuration parameters will be neatly in one place, right where and when you need them! :-)

The globals `module.exports` and `exports` are not quite the same when it comes to the assigning of new values to each of them. While in the example above `module.exports = function(){...}` works fine and makes total sense, the `exports = function() {...}` or even `exports = someObject;` will fail miserably.

This is due to JavaScript fundamentals: the properties of the objects can be replaced without losing the reference to that object, but when we replace the whole object, i.e., `exports = ...`, we're losing the link to *the outside world* that exposes our functions.

This behavior also referred to as objects being mutable and being primitives (strings, numbers, boolean are immutable in JS). Therefore, `exports` only works by creating and assigning properties to it, e.g., `exports.method = function() {...};`.

For example, we can run `$ node -e "require('./routes-module-exports').FindStories('databases');"` and see out nested structure reduced by one level.



The result of using module.exports.

Take a look at this article for more examples: Node.js, Require and Exports[1].

For a working example, take a look at the HackHall chapter.

[1] http://openmymind.net/2012/2/3/Node-Require-and-Exports

22 Using Databases in Modules

With the native Node.js MongoDB driver, Express.js server needs to wait for the connection to be created before it can use the database:

```
...
var mongodb = require ('mongodb');
var Db = mongodb.Db;
var db=new Db ('test', new Server(dbHost, dbPort, {}));
...
db.open(function(error, dbConnection){
  var app = express();
  app.get(...);
  ...
  app.listen(3000);
});
```

Thanks for more advanced libraries like Mongoskin, Monk and Mongoose that have buffering, developers' tasks can be as easy as:

```
var express = require('express'),
  mongoskin = require('mongoskin');

var app = express();
app.use(express.bodyParser());

var db = mongoskin.db('localhost:27017/test', {safe:true});
...
app.get('/', function(req, res) {
  res.send('please select a collection, e.g., /collections/messages')
});
...
app.listen(3000);
```

In case we have routes that need access to database objects, like connection, models, etc. — but those routes aren't in the main server file — all we need to do is attach the required object to the request, i.e., `req`:

```
app.use(function(req, res, next) {
  req.db = db;
  next();
}
```

Another way is to pass/accept the needed variables in the constructor. The example of the `routes.js` module:

```
module.exports = function(app){
  console.log (app.get('db')); //has everything we need!
  ...
  return {
    findUsers: function(req, res) {
    ...
    }
}
```

And the main file:

```
var app = express();
...
app.set('db',db);
routes = require('./routes.js')(app);
...
app.get('/users', routes.findUser);
...
```

Or, we can refactor in a variable:

```
var app = express();
...
app.set('db',db);
Routes = require('./routes.js');
routes = Routes(app);
...
app.get('/users', routes.findUser);
...
```

Readers can try passing data to modules themselves with the example in the `expressjsguide/routes` folder. For that, simply run `$ node -e "require('./routes-module-exports').FindStories('databases');"` from the main folder, i.e., `expressjsguide`.

For a real-life example on passing Mongoose database object to routes via `req` object, take a look at the HackHall chapter.

23 Keys and Passwords

A typical web service is most likely to require connections to other services via usernames, passwords in case of databases, and API keys and secrets/tokens in case of third-party APIs. As readers might guess, it's not a good idea to store these sensitive data in the source code! The two most widespread approaches to solving this issue are

1. JSON file
2. Environmental variables

The JSON file approach is as easy as it sounds. All we need is a JSON file. For example, suppose we have a local database and two external services, HackHall and Twitter in `conf/keys.json`:

```
{
  "db": {
    "host": "http://localhost",
    "port": 27017,
    "username": "azat",
    "password": "CE0E08FE-E486-4AE0-8441-2193DF8D5ED0"
  },
  "hackhall": {
    "api_key": "C7C211A6-D8A7-4E41-99E6-DA0EB95CD864"
  },
  "twitter": {
    "consumer_key": "668C68E1-B947-492E-90C7-F69F5D32B42E",
    "consumer_secret": "4B5EE783-E6BB-4F4E-8B05-2A746056BEE1"
  }
}
```

The latest versions of Node.js allow developers to import JSON files with `require()` function. Hurray for not messing around with the `fs` module!Therefore, the main application file might use these statements:

```
var configurations = require('/conf/keys.json');
var twitterConsumerKey = configurations.twitter.consumer_key;
```

Alternatively, we can just read the file manually with the `fs` module and parse the stream into a JavaScript object. Try this on your own.

As far as access to `configurations` goes, it's even better if we can share this configuration object globally:

```
app.set('configuratons', configurations);
```

Or, using a middleware propagate to every request that comes **after**:

```
app.use(function(req, res, next) {
  req.configurations = configurations;
});
```

Adding `conf/keys.json` to `.gitignore` prevents from tracking and exposing the file:

```
conf/keys.json
```

The problem is still present with the delivery of JSON configuration file to the server. This can be done via SSH and `scp` command: `scp [options] username1@source_host:directory1/filename1 username2@destination_host:directory2/filename2`.

The second approach involves the use of environmental variables. The easiest way to illustrate env vars is to start a script with a `key=value` prefix, for example `$ NODE_ENV=test node app`. This will populate `process.env.NODE_ENV`. Try this yourself:

```
NODE_ENV=test node -e 'console.log (process.env.NODE_ENV)'
```

To deliver/deploy these vars into a remote server, we can use Ubuntu's `/etc/init/nodeprogram.conf`. More details are in this neat tutorial Run Node.js as a Service on Ubuntu[1].

Furthermore, there is a Nodejitsu[2] tool to deamonize node processes forever[3] (GitHub[4]).

For Heroku, the process of synching env vars with cloud is even simpler: locally, we put vars into `.env` file in the project folder for Foreman[5](comes with Heroku toolbelt), and then push them to cloud with heroku-config[6] CLI. More information is at Heroku Dev Center[7].

For a working example, obviously sans sensitive info, take a look at the HackHall chapter.

[1] http://kvz.io/blog/2009/12/15/run-nodejs-as-a-service-on-ubuntu-karmic/
[2] http://nodejitsu.com
[3] http://npmjs.org/forever
[4] https://github.com/nodejitsu/forever
[5] https://devcenter.heroku.com/articles/procfile#developing-locally-with-foreman
[6] https://github.com/ddollar/heroku-config
[7] https://devcenter.heroku.com/articles/config-vars

24 Streams

The Express.js request and response objects are readable and writable Node.js streams respectably. For those vaguely familiar with streams, they're powerful tools for processing chunks of data before particular process (reading, receiving, writing, sending) actually ends. This makes streams useful when dealing with huge data such as audio or video. Another case for streams is when doing big database migrations.

Tip

For more information on how to use streams, there are amazing resources by substack[1]: stream-handbook[2] and stream-adventure[3].

Here is an example of piping a stream to a normal response from `expressjsguide/streams-http-res.js`:

```
1  var http = require('http');
2  var fs = require('fs');
3  var server = http.createServer(function (req, res) {
4      fs.createReadStream('users.csv').pipe(res);
5  });
6  server.listen(3000);
```

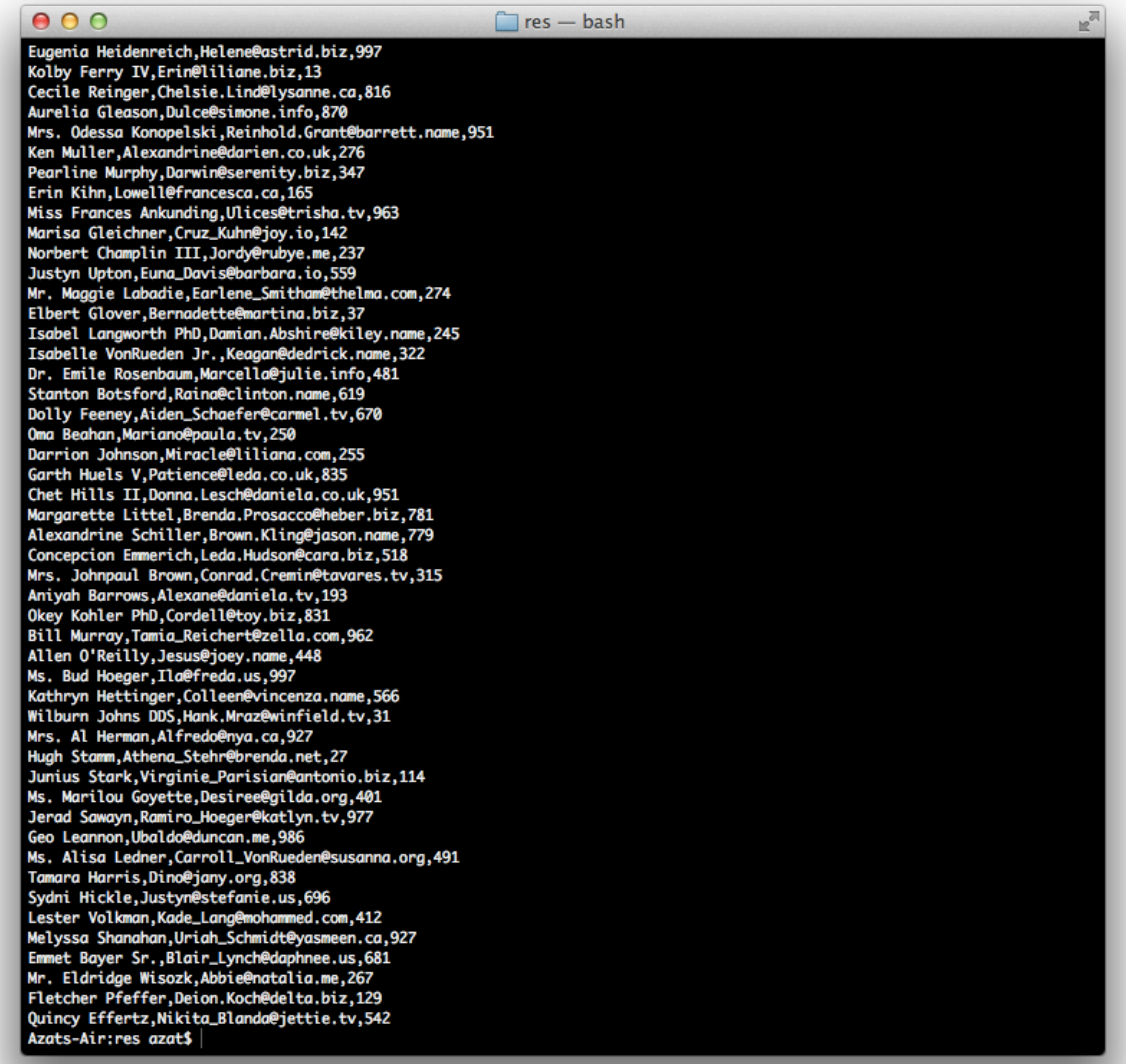
The GET request with CURL from terminal looks like this:

```
1  $ curl http://localhost:3000
```

The line above will make the server to output the content of the file `users.csv`, e.g.,

[1] http://substack.net/
[2] https://github.com/substack/stream-handbook
[3] https://npmjs.org/package/stream-adventure

```
...
Stanton Botsford,Raina@clinton.name,619
Dolly Feeney,Aiden_Schaefer@carmel.tv,670
Oma Beahan,Mariano@paula.tv,250
Darrion Johnson,Miracle@liliana.com,255
Garth Huels V,Patience@leda.co.uk,835
Chet Hills II,Donna.Lesch@daniela.co.uk,951
Margarette Littel,Brenda.Prosacco@heber.biz,781
Alexandrine Schiller,Brown.Kling@jason.name,779
Concepcion Emmerich,Leda.Hudson@cara.biz,518
Mrs. Johnpaul Brown,Conrad.Cremin@tavares.tv,315
Aniyah Barrows,Alexane@daniela.tv,193
Okey Kohler PhD,Cordell@toy.biz,831
Bill Murray,Tamia_Reichert@zella.com,962
Allen O'Reilly,Jesus@joey.name,448
Ms. Bud Hoeger,Ila@freda.us,997
Kathryn Hettinger,Colleen@vincenza.name,566
...
```



The result of running stream response from the `users.csv` file.

If you want to create your own test file such as `users.csv`, you can install Faker.js[4] (GitHub[5]) and re-run `seed-users.js` file:

```
$ npm install Faker.js
$ node seed-users.js
```

The Express.js implementation is strikingly similar, `expressjsguide/stream-express-res.js`:

[4] https://npmjs.org/package/Faker
[5] https://github.com/marak/Faker.js/

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

var app = express()

app.get('*', function (req, res) {
    fs.createReadStream('users.csv').pipe(res);
});

app.listen(3000);
```

Keeping in mind that request is a readable stream, and response is a writable one, we can implement a server that saves POST requests into a file. Here is the content of `expressjsguide/stream-http-req.js`:

```
var http = require('http');
var fs = require('fs');
var server = http.createServer(function (req, res) {
    if (req.method === 'POST') {
        req.pipe(fs.createWriteStream('ips.txt'));
    }
    res.end('\n');
});
server.listen(3000);
```

We call Faker.js to generate test data consisting of names, domains, IP addresses, latitudes and longitudes. This time, we won't save the data to file, but pipe it to CURL instead.

Here is the trivial Faker.js script that outputs JSON object of 1000 records to the stdout from `expressjsguide/seed-ips.js`:

```
var Faker = require('Faker');
var fs = require('fs');
var body = [];

for (var i = 0; i < 1000; i++) {
  body.push({
    'name': Faker.Name.findName(),
    'domain': Faker.Internet.domainName(),
    'ip': Faker.Internet.ip(),
    'latitude': Faker.Address.latitude(),
    'longitude': Faker.Address.longitude()
```

```
12    });
13  }
14  process.stdout.write(JSON.stringify(body));
```

To test our `stream-http-req.js`, let's run `node seed-ips.js | curl -d@- http://localhost:3000`.

The beginning of the file written by Node.js server.

Once more, let's convert this example into Express.js app in `expressjsguide/stream-express-req.js`:

```
var http = require('http');
var fs = require('fs');
var express = require('express');
var app = express();

app.post('*', function (req, res) {
   req.pipe(fs.createWriteStream('ips.txt'));
   res.end('\n');
});
app.listen(3000);
```

Tip

In some cases, it's good to have a pass through logic that doesn't consume too much resources. For that check out module through[6] (GitHub[7]). Another useful module is concat-stream[8] (GitHub[9]). It allows concatenation of streams.

[6] https://npmjs.org/package/through
[7] https://github.com/dominictarr/through
[8] https://npmjs.org/package/concat-stream
[9] https://github.com/maxogden/node-concat-stream

25 Redis

Redis is often used in Express.js applications for sessions persistence, because storing sessions in a physical storage allows apps not to lose users' data during times when systems are restarted or re-deployed. It also enables the use of multiple RESTful servers due to the fact that they can connect to the same Redis. In addition, Redis can be used for queues and scheduling tasks (e.g., email jobs).

To download Redis 2.6.7, follow these simple steps:

```
$ wget http://download.redis.io/releases/redis-2.6.7.tar.gz
$ tar xzf redis-2.6.7.tar.gz
$ cd redis-2.6.7
$ make
```

Or for more Redis instructions, you can visit redis.io/download[1].

To start Redis:

```
$ src/redis-server
```

To stop it, just press `control + c`.

To access the Redis command-line interface:

```
$ src/redis-cli
```

Below is the uncomplicated illustration on how to use Redis to manage Express.js sessions.

Firstly, to access Redis, we can utilize `connect-redis` driver. Someone can do so with familiar dependency key-value pair in `package.json`:

[1] http://redis.io/download

```
{
  "name": "redis-example",
  "dependencies": {
    "express": "3.3.5",
    "connect-redis": "1.4.5"
  }
}
```

To use Redis as a session store in an Express.js server:

```
var express = require('express');
var app = express();
var RedisStore = require('connect-redis')(express);

app.use(express.cookieParser());
app.use(express.session({
  store: new RedisStore({
    host: 'localhost',
    port: 6379,
    db: 'redis-example',
    pass: 'B7E0C34F-F40D-46FE-926F-EE947562436A'
  }),
  secret: '0FFD9D8D-78F1-4A30-9A4E-0940ADE81645',
  cookie: { path: '/', maxAge: 3600000 }
}));
```

Just for your information, the connect-redis is backed by the redis[2] (GitHub[3]) module. With this module, the Redis can be used as a flat standalone database. Interestingly, Redis supports four types of data: strings, lists, sets and hashes.

Tip

For a deeper Redis learning, an interactive tutorial is available at try.redis.io[4].

[2] https://npmjs.org/package/redis
[3] https://github.com/mranney/node_redis
[4] http://try.redis.io/

26 Authentication

The most common authentication is username and password combination. We can check for the match against our database and then store `authenticated=true` flag in the session. The session data is automatically stored by the Express.js for **every** other request by that agent:

```
app.use(function(req, res, next) {
  if (req.session && req.session.authenticated)
     return next();
  else {
    return res.redirect('/login');
  }
}
```

In case we need additional user information, it also can be stored in the session:

```
app.post('/login', function(req, res) {
  // check the database for the username and password combination
  db.findOne({username: req.body.username,
    password: req.body.password},
    function(error, user) {
      if (error) return next();
      if (!user) return next(new Error('Bad username/password');
      req.session.user = user;
      res.redirect ('/protected_area');
    }
  );
});
```

Authentication with third-parties is often done via OAuth. For a working example of the oauth[1] (GitHub[2]) module (to which the author, Azat, contributed some docs), take a look at the HackHall chapter.

OAuth 1.0/2.0 require callback routes for the user redirect back to our sites. This is effortless with Express.js. In addition, there are fully automated solutions that take care of everything (database, signing, routing, etc.): Everyauth[3] and Passport[4].

[1] https://github.com/ciaranj/node-oauth
[2] https://github.com/ciaranj/node-oauth
[3] https://npmjs.org/package/everyauth
[4] https://npmjs.org/package/passport

27 Multi-Threading with Clusters

There are a lot of arguments out there against Node.js, which are rooted on the myth that Node.js-based systems **have** to be single threaded. Nothing can be further from the truth, and with `cluster` module we can skillfully utilize all machines' CPUs.

Here is a working example of Express.js app that runs on four processes. At the beginning of the file, we import dependencies:

```
var cluster = require('cluster');
var http = require('http');
var numCPUs = require('os').cpus().length;
var express = require('express');
```

The cluster module has a property that tells us if the process is master or child. We use it to spawn four workers (the default workers will use the same file, but this can be overwritten with setupMaster[1]). In addition to that, we can attach event listeners and receive messages from workers (e.g., 'kill').

```
if (cluster.isMaster) {
  console.log (' Fork %s worker(s) from master', numCPUs)
  for (var i = 0; i < numCPUs; i++) {
    cluster.fork();
  };
  cluster.on('online', function(worker) {
    console.log ('worker is running on %s pid', worker.process.pid)
  });
  cluster.on('exit', function(worker, code, signal) {
    console.log('worker with %s is closed', worker.process.pid );
  });
}
```

The worker code is just an Express.js app with a twist — we're getting the process PID:

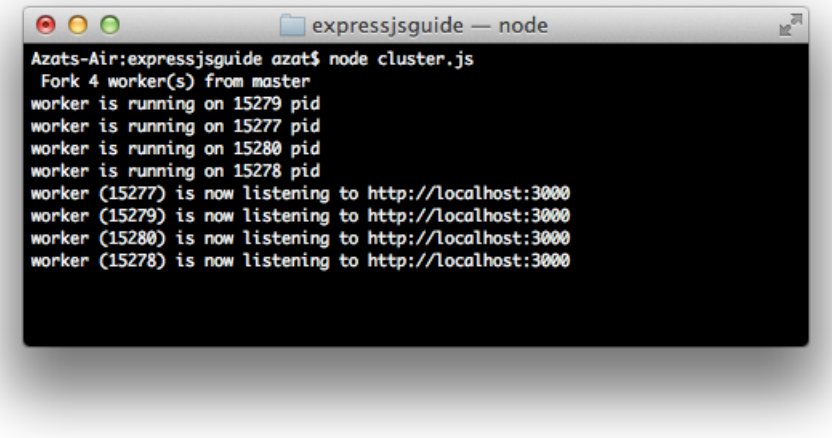
[1] http://nodejs.org/docs/v0.9.0/api/cluster.html#cluster_cluster_setupmaster_settings

```
} else if (cluster.isWorker) {
  var port = 3000;
  console.log('worker (%s) is now listening to http://localhost:%s',
    cluster.worker.process.pid, port);
  var app = express();
  app.get('*', function(req, res) {
    res.send(200, 'cluser '
      + cluster.worker.process.pid
      + ' responded \n');
  })
  app.listen(port);
}
```

The full source code of `expressjsguide/cluster.js`:

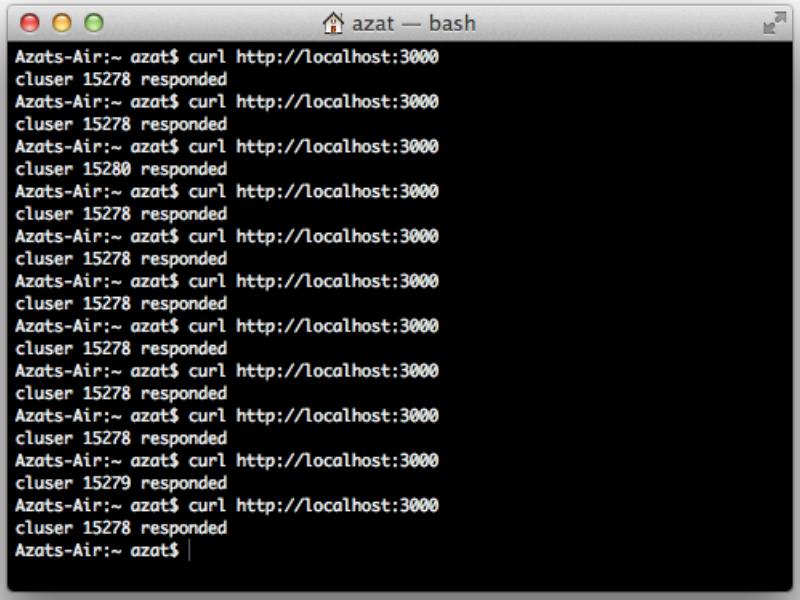
```
var cluster = require('cluster');
var http = require('http');
var numCPUs = require('os').cpus().length;
var express = require('express');

if (cluster.isMaster) {
  console.log (' Fork %s worker(s) from master', numCPUs)
  for (var i = 0; i < numCPUs; i++) {
    cluster.fork();
  };
  cluster.on('online', function(worker) {
    console.log ('worker is running on %s pid', worker.process.pid)
  });
  cluster.on('exit', function(worker, code, signal) {
    console.log('worker with %s is closed', worker.process.pid );
  });
} else if (cluster.isWorker) {
  var port = 3000;
  console.log('worker (%s) is now listening to http://localhost:%s', cluster.work\
er.process.pid, port);
  var app = express();
  app.get('*', function(req, res) {
    res.send(200, 'cluser ' + cluster.worker.process.pid + ' responded \n');
  })
  app.listen(port);

}
```

As usual, to start an app, run `$ node cluster`. There should be four (or two depending on your machine's architecture) processes:



Starting of four processes with cluster.

When we CURL with `$ curl http://localhost:3000`, there are different processes that listen to the **same** port and respond to us:



Server response is rendered by different processes.

Tip

If someone prefers ready solutions to low-level libraries (like cluster), check out the real-world production library created and used by eBay: cluster2[2] (GitHub[3]).

[2] https://npmjs.org/package/cluster2
[3] https://github.com/ql-io/cluster2

28 Consolidate.js

The consolidate library[1] (GitHub[2]) streamlines and generalizes a few dozen of template engine modules so they play nicely with Express.js.

The Consolidate.js example:

```
var express = require('express');
var consolidate = require('consolidate');

var app = express();

//configure template engine:
app.engine('html', consolidate.handlebars);
app.set('view engine', 'html');
app.set('views', __dirname + '/views');
```

That's it, `res.render()` is ready to use Handlebars!

All the template engines that consolidate support as of this writing (taken from its GitHub page[3]):

- atpl[4]
- dust[5] (website)[6]
- eco[7]
- ect[8] (website)[9]
- ejs[10]
- haml[11] (website)[12]
- haml-coffee[13] (website)[14]

[1] https://npmjs.org/package/consolidate
[2] https://github.com/visionmedia/consolidate.js
[3] https://github.com/visionmedia/consolidate.js/blob/master/Readme.md
[4] https://github.com/soywiz/atpl.js
[5] https://github.com/akdubya/dustjs
[6] http://akdubya.github.com/dustjs/
[7] https://github.com/sstephenson/eco
[8] https://github.com/baryshev/ect
[9] http://ectjs.com/
[10] https://github.com/visionmedia/ejs
[11] https://github.com/visionmedia/haml.js
[12] http://haml-lang.com/
[13] https://github.com/9elements/haml-coffee
[14] http://haml-lang.com/

- handlebars[15] (website)[16]
- hogan[17] (website)[18]
- jade[19] (website)[20]
- jazz[21]
- jqtpl[22] (website)[23]
- JUST[24]
- liquor[25]
- mustache[26]
- QEJS[27]
- swig[28] (website)[29]
- templayed[30]
- toffee[31]
- underscore[32] (website)[33]
- walrus[34] (website)[35]
- whiskers[36]

[15] https://github.com/wycats/handlebars.js/
[16] http://handlebarsjs.com/
[17] https://github.com/twitter/hogan.js
[18] http://twitter.github.com/hogan.js/
[19] https://github.com/visionmedia/jade
[20] http://jade-lang.com/
[21] https://github.com/shinetech/jazz
[22] https://github.com/kof/node-jqtpl
[23] http://api.jquery.com/category/plugins/templates/
[24] https://github.com/baryshev/just
[25] https://github.com/chjj/liquor
[26] https://github.com/janl/mustache.js
[27] https://github.com/jepso/QEJS
[28] https://github.com/paularmstrong/swig
[29] http://paularmstrong.github.com/swig/
[30] http://archan937.github.com/templayed.js/
[31] https://github.com/malgorithms/toffee
[32] https://github.com/documentcloud/underscore
[33] http://documentcloud.github.com/underscore/
[34] https://github.com/jeremyruppel/walrus
[35] http://documentup.com/jeremyruppel/walrus/
[36] https://github.com/gsf/whiskers.js/tree/

29 Stylus, LESS and SASS

CSS is tedious to write and manage in complex projects. Stylus, LESS and SASS bring so much needed sanity of re-usability (mixins, extend, vars, etc.) to stylesheets.

29.1 Stylus

Stylus is a sibling of Express.js and the most often used CSS framework. To install Stylus, type and run `$ npm install stylus --save`. Then, to apply `static` middleware, include this into your server file:

```
...
app.use(require('stylus').middleware(__dirname + '/public'));
app.use(express.static(path.join(__dirname, 'public')));
...
```

Put `*.styl` file(s) inside of the folder that we expose (e.g., `public/css`) and include them in jade templates with `*.css` **extension**:

```
...
  head
    link(rel='stylesheet', href='/stylesheets/style.css')
...
```

Or, in any other templates of your choice, or in plain HTML file(s):

```
<link rel="stylesheet" href="/stylesheets/style.css"/>
```

For the project created from scratch, someone can use `$ express -c stylus express-app` command.

The stylus-enabled project is in the `expressjsguide/stylus` folder as well as on GitHub[1].

29.2 LESS

To use LESS with Express.js v3.x, we'll need less-middleware[2] which is an external NPM module:

[1] https://github.com/azat-co/expressjsguide/tree/master/stylus

[2] http://npmjs.org/less-middleware

```
$ npm install less-middleware --save
```

Then, we need to add less-middleware **before** the `static` and router ones:

```
...
app.use(require('less-middleware')({
  src: __dirname + '/public',
  compress: true
}));
app.use(express.static(path.join(__dirname, 'public')));

app.use(app.router);
...
//your routes
```

Assuming the LESS file(s) are in `public/css`, we can link `*.css` file(s) and the rest will be handled automatically, e.g., jade template might use this:

```
...
  head
    link(rel='stylesheet', href='/stylesheets/style.css')
...
```

Or, in any other template of your choice, or in plain HTML file(s):

```
<link rel="stylesheet" href="/stylesheets/style.css"/>
```

For the project created from scratch, someone can use `$ express -c less express-app` command.

The less-enabled project is in the `expressjsguide/less` folder as well as on GitHub[3].

29.3 SASS

To use SASS with Express.js v 3.x, we need node-sass[4](GitHub[5]) which is an external NPM module:

```
$ npm install node-sass --save
```

This is our Express.js plug-in for SASS:

[3] https://github.com/azat-co/expressjsguide/tree/master/less
[4] https://npmjs.org/package/node-sass
[5] https://github.com/andrew/node-sass

```
...
app.use(app.router);
app.use(require('node-sass').middleware({
  src: __dirname + '/public',
  dest: __dirname + '/public',
  debug: true,
  outputStyle: 'compressed'
}));
app.use(express.static(path.join(__dirname, 'public')));
...
```

The Jade template also imports `*.css` file:

```
    link(rel='stylesheet', href='/stylesheets/style.css')
```

The sass-enabled project is in the `expressjsguide/sass` folder as well as on GitHub[6].

[6] https://github.com/azat-co/expressjsguide/tree/master/sass

30 Security

30.1 CSRF

CSRF middleware was briskly covered in The Interface part. It does most of the job of matching incoming values from requests for us. However, we still need to expose it in responses and pass it back to server in templates (or JavaScript XHRs). One of the ways of implementation is to use custom middleware:

```
app.use(function (req, res, next) {
  res.locals.csrftoken = req.session._csrf;
  next();
});
```

And then render value in the template (this is jade language):

```
doctype 5
html
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    form(method="post", action="/login")
      input(type="hidden", name="_csrf", value="#{csrftoken}")
      input(type="text", name="username", placeholder="Username")
      input(type="password", name="password", placeholder="Password")
      button(type="submit") Login
```

30.2 Permissions

Plainly, it's usually a bad idea to run web services as a root. Operations developers can utilize Ubuntu's authbind[1] to bind to privileged ports (e.g., 80 for HTTP and 443 for HTTPS) without giving root access.

Alternatively, it's possible to drop privileges after biding to a port. Here is an example of doing it by setting GID and UID[2] with properties from `process.env.GID` and `process.evn.UID` environmental vars:

[1] http://manpages.ubuntu.com/manpages/hardy/man1/authbind.1.html

[2] http://www.gnu.org/software/coreutils/manual/html_node/Directory-Setuid-and-Setgid.html

```
...
var app = express();
...
http.createServer(app).listen(app.get('port'), function(){
    console.log("Express server listening on port "
    + app.get('port'));
    process.setgid(process.env.GID);
    process.setuid(process.env.UID);
});
```

30.3 Headers

There is an Express.js middleware called helmet[3] (GitHub[4]) that provides most of the security headers described in Seven Web Server HTTP Headers that Improve Web Application Security for Free[5].

In addition to that, programmers can use express-secure-skeleton[6] for jump starting their secure app.

[3] https://npmjs.org/package/helmet

[4] https://github.com/evilpacket/helmet

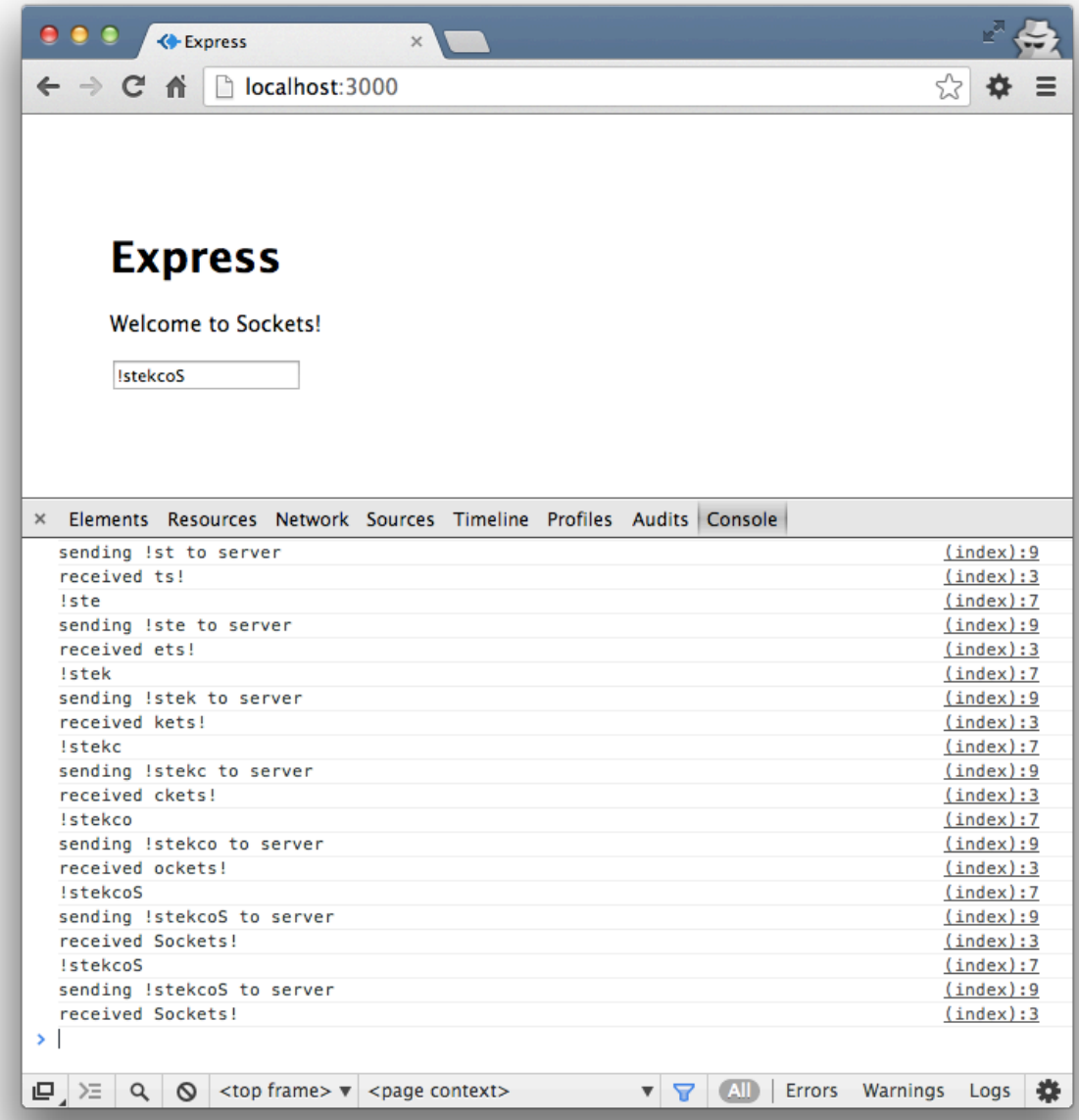
[5] http://recxltd.blogspot.com/2012/03/seven-web-server-http-headers-that.html

[6] https://github.com/evilpacket/express-secure-skeleton

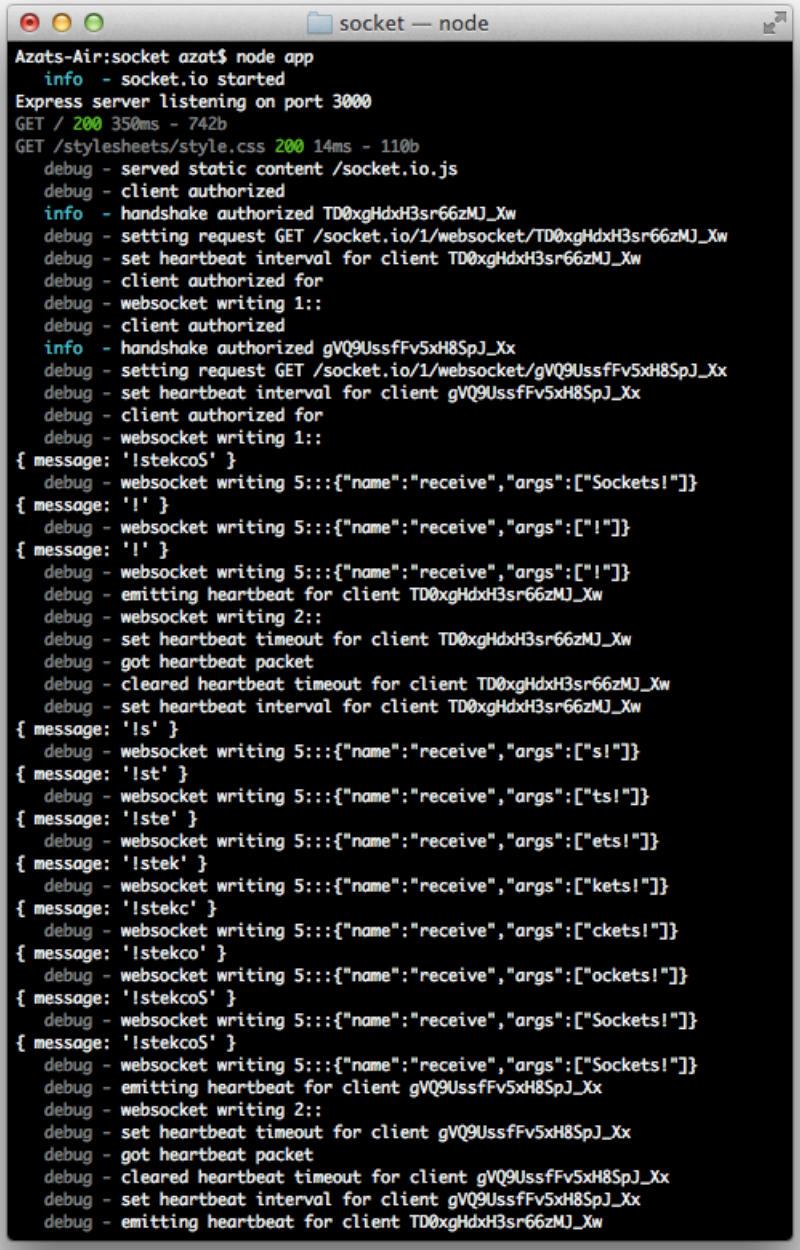
31 Socket.IO

The full coverage of the Socket.IO[1] library deserves its own book. However, it's so cool and easy to start using with Express.js that we included a facile example. The example will echo (browser to server and back) our input in reverse in real time.

[1] http://socket.io



The input of !stekcoS yelds Sockets!



Express.js server is catching and processing input in real-time.

We start with a fresh Express.js app created by `$ express socket`. Then, we install dependencies with `$ cd socket && npm install`.

To include Socket.io, we can use `$ npm install socket.io --save' or manually define it in`

package.json‘:

```
{
  "name": "application-name",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "3.3.5",
    "jade": "*"
  }
}
```

Socket.IO in some way might be considered another server. This is how we refactor auto-generated Express.js code:

```
var server = http.createServer(app);
var io = require('socket.io').listen(server);

...
server.listen(app.get('port'), function(){
  console.log('Express server listening on port '
  + app.get('port'));
});
```

Inside goes our reversal logic:

```
io.sockets.on('connection', function (socket) {
  socket.on('messageChange', function (data) {
    console.log(data);
    socket.emit('receive',
    data.message.split('').reverse().join('') );
  })
});
```

The full content of `expressjsguide/socket/app.js`:

```
/**
 * Module dependencies.
 */

var express = require('express');
var routes = require('./routes');
var user = require('./routes/user');
var http = require('http');
var path = require('path');

var app = express();

// all environments
app.set('port', process.env.PORT || 3000);
app.set('views', __dirname + '/views');
app.set('view engine', 'jade');
app.use(express.favicon());
app.use(express.logger('dev'));
app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(app.router);
app.use(express.static(path.join(__dirname, 'public')));

// development only
if ('development' == app.get('env')) {
  app.use(express.errorHandler());
}

app.get('/', routes.index);
app.get('/users', user.list);

var server = http.createServer(app);
var io = require('socket.io').listen(server);

io.sockets.on('connection', function (socket) {
  socket.on('messageChange', function (data) {
    console.log(data);
    socket.emit('receive', data.message.split('').reverse().join('') );
  })
});

server.listen(app.get('port'), function(){
```

```
43   console.log('Express server listening on port ' + app.get('port'));
44 });
```

Lastly, our app needs some front-end love in `index.jade`:

```
1  extends layout
2
3  block content
4    h1= title
5    p Welcome to
6      span.received-message #{title}
7    input(type='text', class='message', placeholder='what is on your mind?', onkeyu\
8  p='send(this)')
9    script(src="/socket.io/socket.io.js")
10   script.
11     var socket = io.connect('http://localhost');
12     socket.on('receive', function (message) {
13       console.log('received %s', message);
14       document.querySelector('.received-message').innerText = message;
15     });
16     var send = function(input) {
17       console.log(input.value)
18       var value = input.value;
19       console.log('sending %s to server', value);
20       socket.emit('messageChange', {message: value});
21     }
```

For more Socket.IO examples, go to socket.io/#how-to-use[2].

[2] http://socket.io/#how-to-use

32 Domains

Contrary to its more popular homonym, domain is a core Node.js module[1]. It aids developers in tracking and isolating errors which could be a juggernaut task. Think about domains as a smarter version of try/catch statements[2].

When it comes to Express.js, we can apply domains in an error prone routes. Before the routes, we need to define custom handlers to catch errors from domains:

```
var domain = require('domain');
var defaultHandler = express.errorHandler();
app.use(function (error, req, res, next) {
  if (domain.active) {
    console.info('caught with domain')
    domain.active.emit("error", error);
  } else {
    console.info('no domain')
    defaultHandler(error, req, res, next);
  }
});
```

Here is a crashy route:

```
app.get("/e", function (req, res, next) {
  var d = domain.create();
  d.on("error", function (error) {
    console.error(error.stack)
    res.send(500, {"error": error.message});
  });
  d.run(function () {
    //error prone code goes here
    throw new Error("Database is down.");
  });
});
```

On the other hand, we can call next with an error object (e.g., when error variable comes from other nested calls):

[1] http://nodejs.org/api/domain.html

[2] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/try...catch

```
app.get("/e", function (req, res, next) {
  var d = domain.create();
  d.on("error", function (error) {
    console.error(error.stack)
    res.send(500, {"error": error.message});
  });
  d.run(function () {
    //error prone code goes here
    next(new Error("Database is down."));
  });
});
```

The working (or should we write crashing example) is in the `expressjsguide/domains` folder and on GitHub[3].

The full content of `expressjsguide/domains/app.js`:

```
/**
 * Module dependencies.
 */

var express = require('express');
var routes = require('./routes');
var user = require('./routes/user');
var http = require('http');
var path = require('path');

var app = express();

// all environments
app.set('port', process.env.PORT || 3000);
app.set('views', __dirname + '/views');
app.set('view engine', 'jade');
app.use(express.favicon());
app.use(express.logger('dev'));
app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(app.router);
app.use(express.static(path.join(__dirname, 'public')));

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

[3] https://github.com/azat-co/expressjsguide/tree/master/domains

```
var defaultHandler = express.errorHandler();
app.use(function (error, req, res, next) {
  if (domain.active) {
    console.info('caught with domain', domain.active)
    domain.active.emit("error", error);
  } else {
    console.info('no domain')
    defaultHandler(error, req, res, next);
  }
});
app.get("/e", function (req, res, next) {
  var d = domain.create();
  d.on("error", function (error) {
    console.error(error.stack)
    res.send(500, {"error": error.message});
  });
  d.run(function () {
    //error prone code goes here
    throw new Error("Database is down.");
    // next(new Error("Database is down."));
  });
});

app.get('/', routes.index);
app.get('/users', user.list);

http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

For more ways to apply domains with Express.js, take a look at this amazing presentation[4].

Warning

Domain module is in *experimental* stage. This means that it's likely that methods and behavior will change. Therefore, stay updated and use exact versions in `package.json` file.

[4] http://othiym23.github.io/nodeconf2013-domains/#/4/1

IV Tutorials and Examples

33 Instagram Gallery

This tutorial will demonstrate how to use Express.js external third-party services. In this tutorial, in addition to Express.js, we'll use Instagram photos via Storify API, superagent, consolidate and handlebars modules.

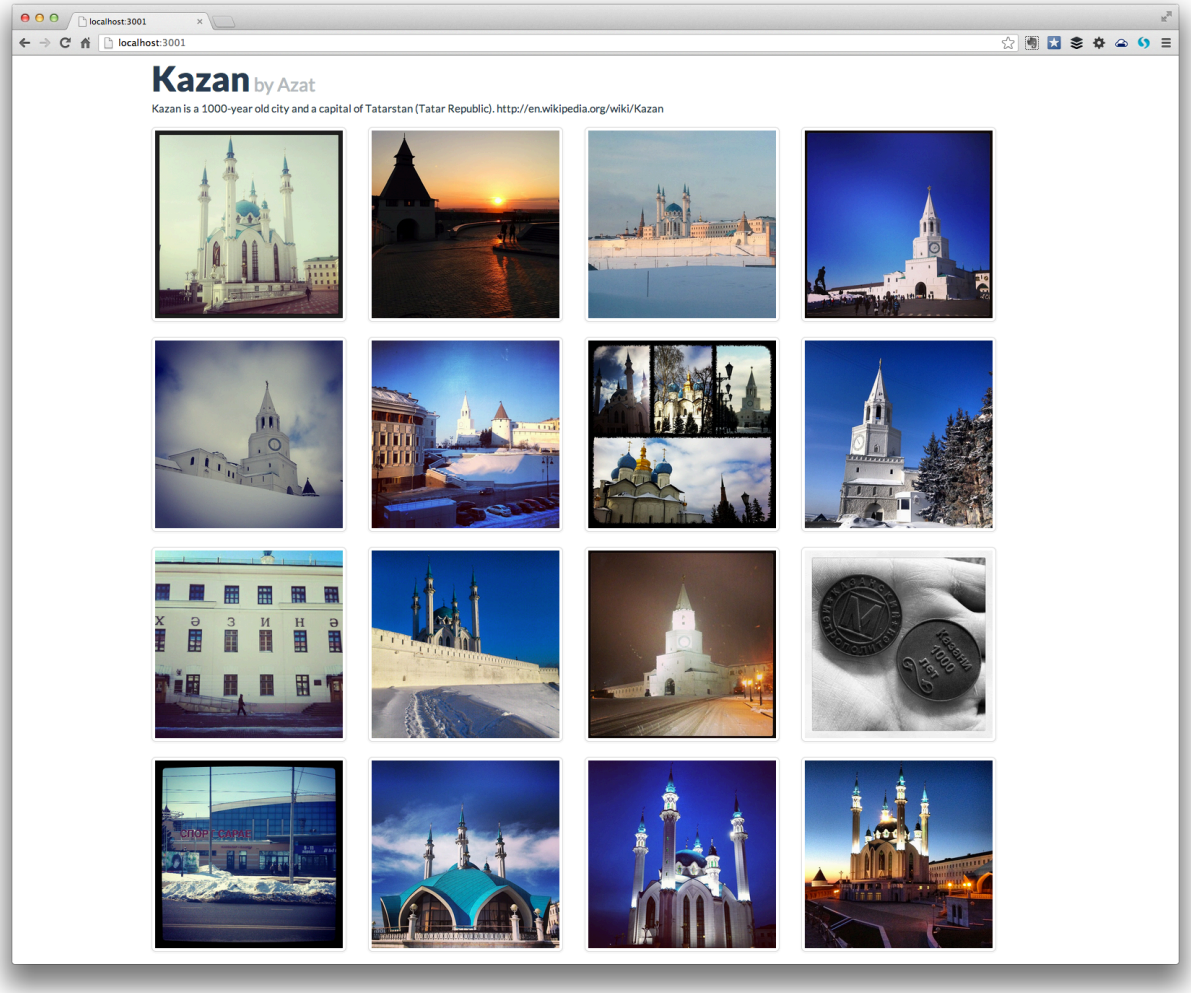
Example

The full source code of this example is available at https://github.com/azat-co/sfy-gallery.

Storify[1] runs on Node.js[2] and Express.js[3]; therefore, why not use these technologies to write an application that demonstrates how to build apps that rely on third-party APIs and HTTP requests to them?

The Instagram Gallery app will fetch a story object and display its title, description, and a gallery of the elements/images like this:

[1] http://storify.com
[2] http://nodejs.org
[3] http://expressjs.com



Instagram Gallery: Storify API + Node.js = <3

33.1 A File Structure

```
- index.js
- package.json
- views/index.html
- css/bootstrap-responsive.min.css
- css/flatly-bootstrap.min.css
```

Where `index.js` is our main Node.js file and `index.html` the Handlebars template.

33.2 Dependencies

Our dependencies include:

- express: 3.2.5 for Express.js framework
- superagent: 0.14.6 for making HTTP requests
- consolidate: 0.9.1 for using Handlebars with Express.js
- handlebars: 1.0.12 for using Handlebars template engine

The `package.json` file:

```
{
  "name": "sfy-demo",
  "version": "0.0.0",
  "description": "Simple Node.js demo app on top of Storify API",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "dependencies": {
    "express": "3.2.5",
    "superagent": "0.14.6",
    "consolidate": "0.9.1",
    "handlebars": "1.0.12"
  },
  "repository": "",
  "author": "Azat Mardanov",
  "license": "BSD"
}
```

33.3 Node.js Server

At the beginning of the file, we require dependencies:

```
var express = require('express');
var superagent = require('superagent');
var consolidate = require('consolidate');

var app = express();
```

Then, configure template engine:

```
app.engine('html', consolidate.handlebars);
app.set('view engine', 'html');
app.set('views', __dirname + '/views');
```

Set up a static folder middleware:

```
app.use(express.static(__dirname + '/public'));
```

If you want to serve any other story, feel free to do so. All you need is the username of the author and the story slug, for my gallery of the capitol of Tatarstan, Kazan[4], leave the following:

```
var user = 'azat_co';
var story_slug = 'kazan';
```

Paste your values, i.e., Storify API key, username and _token if you have one. As of this writing, Storify API is public meaning there is no need for the authentication. In case this changes in the future, please obtain your sign up for an API key at http://dev.storify.com/request or follow the official documentation at dev.storify.com[5]:

```
var api_key = "";
var username = "";
var _token = "";
```

Let's define the home route:

```
app.get('/',function(req, res){
```

Now we'll fetch elements from Storify API the route's callback:

[4] http://en.wikipedia.org/wiki/Kazan
[5] http://dev.storify.com

```
  superagent.get("http://api.storify.com/v1/stories/"
    + user + "/" + story_slug)
    .query({api_key: api_key,
      username: username,
      _token: _token})
    .set({  Accept: 'application/json' })
    .end(function(e, storifyResponse){
      if (e) next(e);
```

To render the template with the story object which is in the HTTP response bodyâ€™s content property:

```
      return res.render('index', storifyResponse.body.content);
    })

})

app.listen(3001);
```

33.4 Handlebars Template

The `views/index.html` file:

```
<!DOCTYPE html lang="en">
<html>
  <head>
    <link type="text/css"
      href="css/flatly-bootstrap.min.css"
      rel="stylesheet" />
    <link type="text/css"
      href="css/bootstrap-responsive.min.css"
      rel="stylesheet"/>
  </head>

  <body class="container">
    <div class="row">
      <h1>{{title}}<small> by {{author.name}}</small></h1>
      <p>{{description}}</p>
    </div>
    <div class="row">
      <ul class="thumbnails">
```

```
19        {{#each elements}}
20          <li class="span3">
21            <a class="thumbnail" href="{{permalink}}"
22            target="_blank">
23              <img src="{{data.image.src}}"
24                title="{{data.image.caption}}" />
25            </a>
26          </li>
27        {{/each}}
28        </ul>
29      </div>
30    </body>
31
32  </html>
```

33.5 Conclusion

Express.js and superagent allow developers to retrieve and manipulate data provided by third-party services such as Storify, Twitter and Facebook in just a few lines of code.

Note

In most cases, service providers (such as Google, Facebook and Twitter) require authentication (which is not the case with Storify API as of this writing). To make OAuth 1.0, OAuth 2.0 and OAuth Echo requests, consider oauth[6] (GitHub[7]), everyauth[8] (GitHub[9]) and/or Passport[10](website[11] and GitHub[12]).

[6] https://npmjs.org/package/oauth
[7] https://github.com/ciaranj/node-oauth
[8] https://npmjs.org/package/everyauth
[9] https://github.com/bnoguchi/everyauth
[10] http://passportjs.org/
[11] http://passportjs.org/
[12] https://github.com/jaredhanson/passport

34 Todo App

> Todo apps are considered to be quintessential in showcasing frameworks akin to famous Todomvc.com[1] for front-end JavaScript frameworks. In this example, we'll use Jade, forms, LESS, AJAX/XHR and CSRF.

In our Todo app, we'll intentionally not use Backbone.js or Angular to demonstrate how to build *traditional* websites with the use of forms and redirects. In addition to that, we'll explain how to plug-in CSRF and LESS.

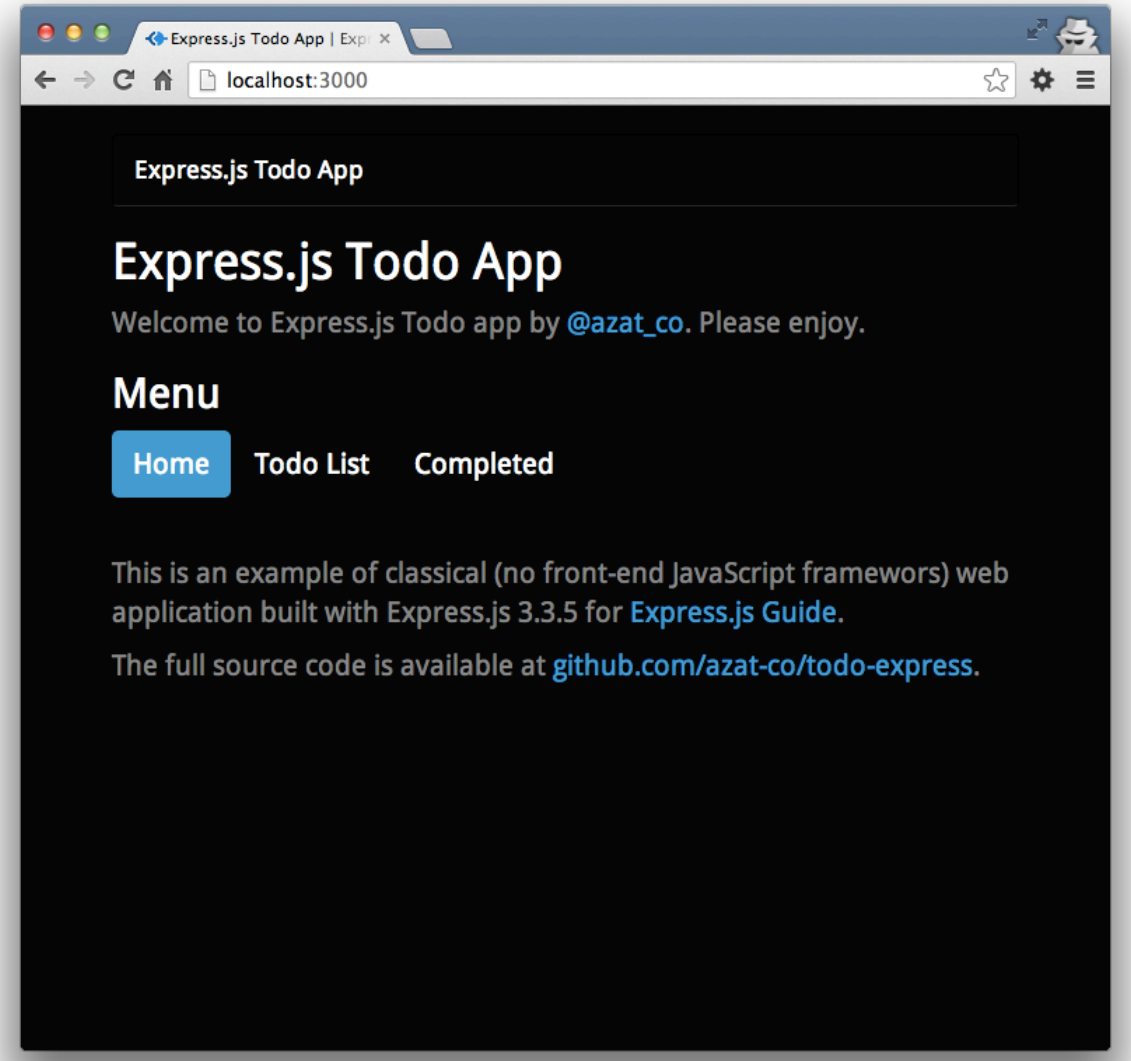
Example

All the source code is in the github.com/azat-co/todo-express[2] for your convenience.

Here are some screenshots of Todo app in which we start from a home page:

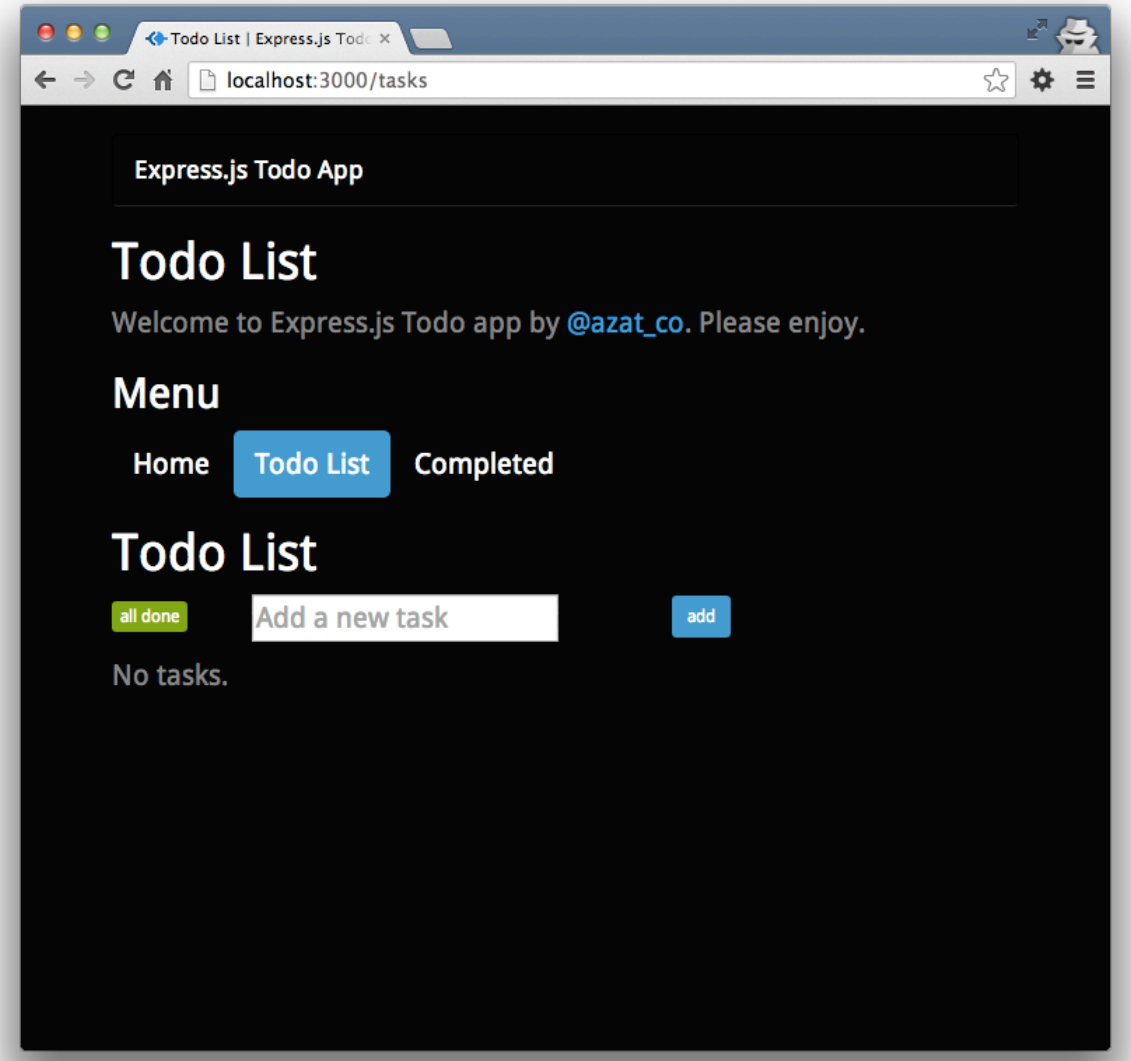
[1] http://todomvc.com

[2] http://github.com/azat-co/todo-express



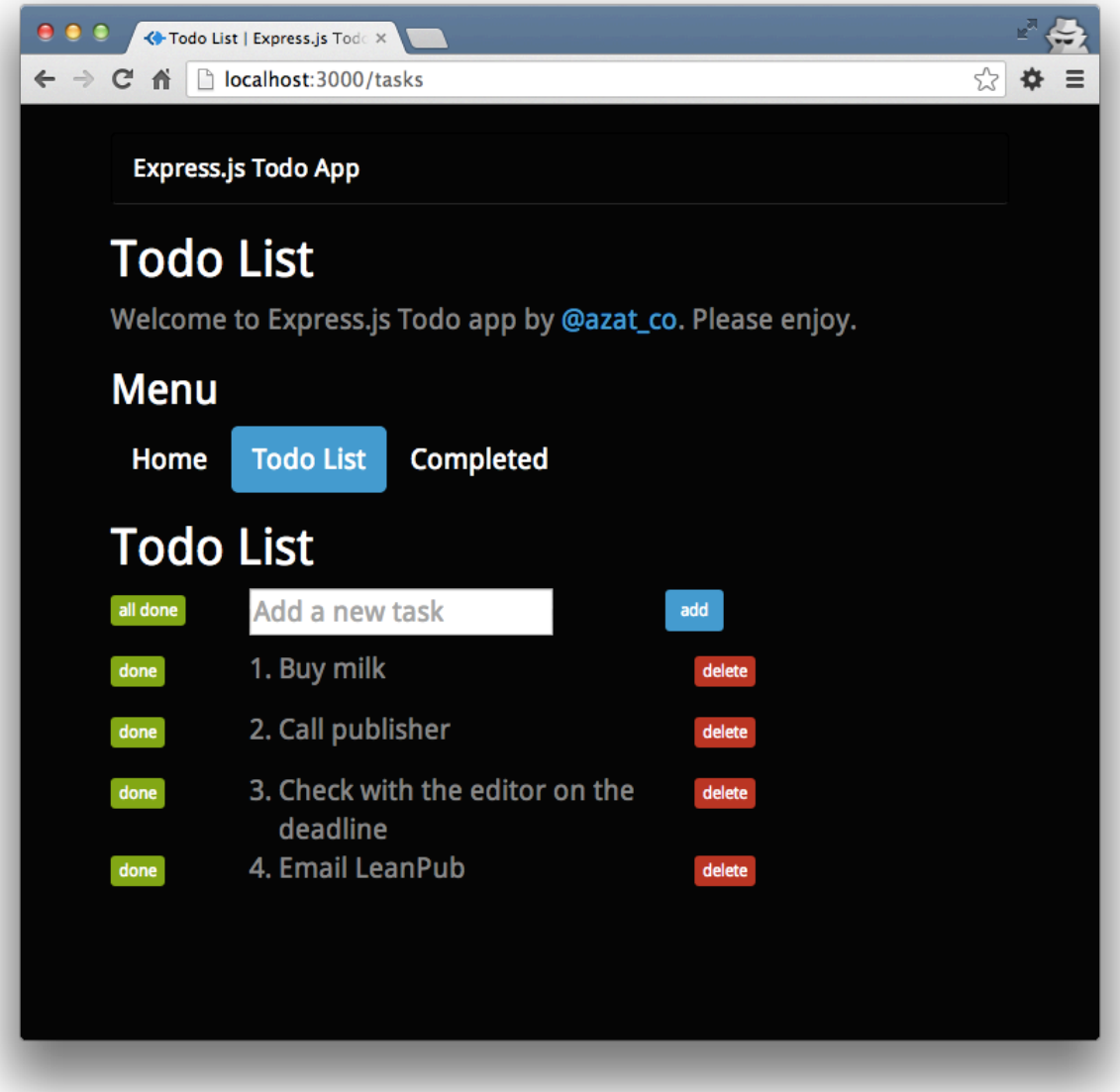
The Todo app home page.

There's an empty list (unless you played with this app before):



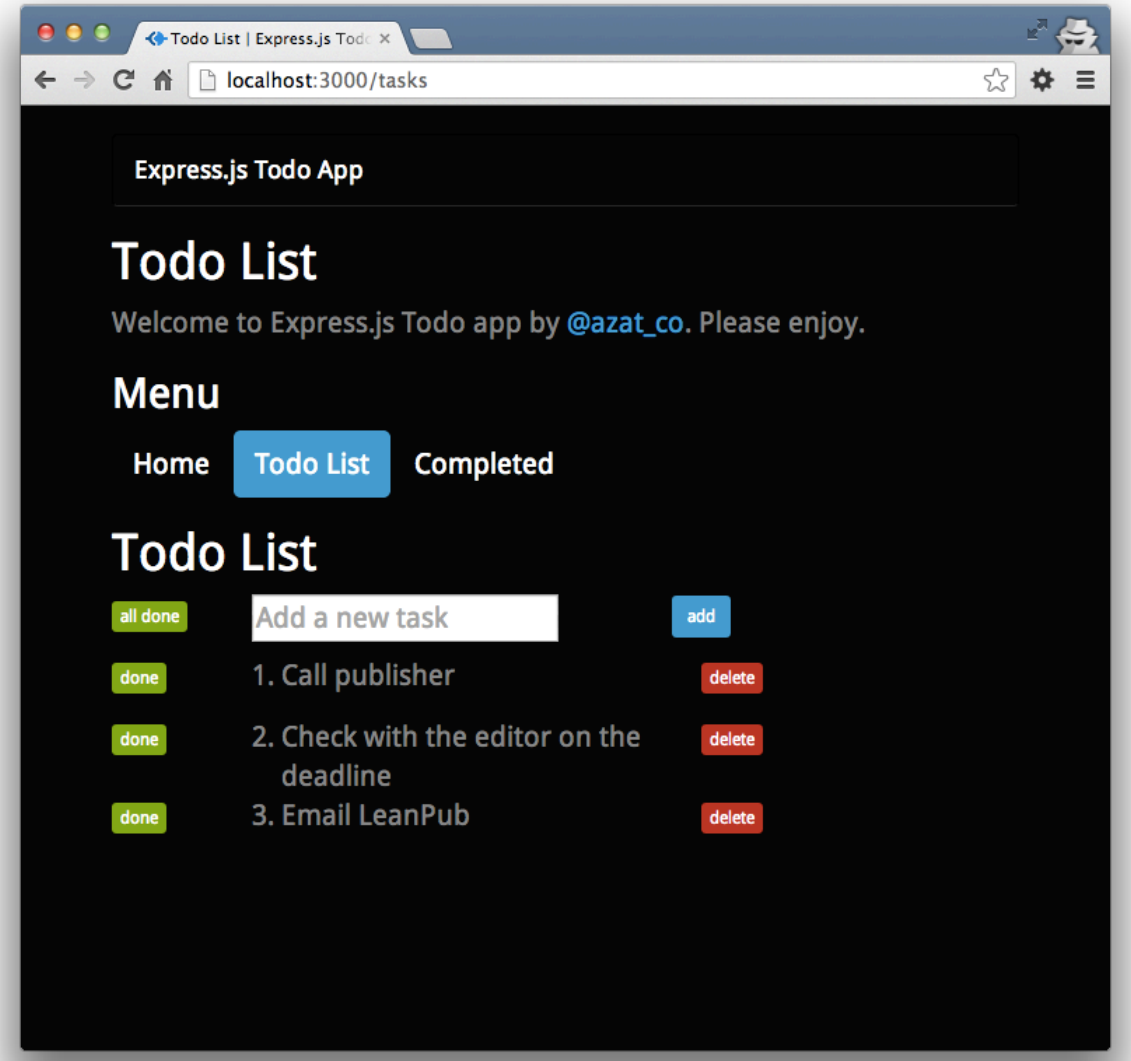
The empty Todo List page.

Now we can add four items to the Todo List:



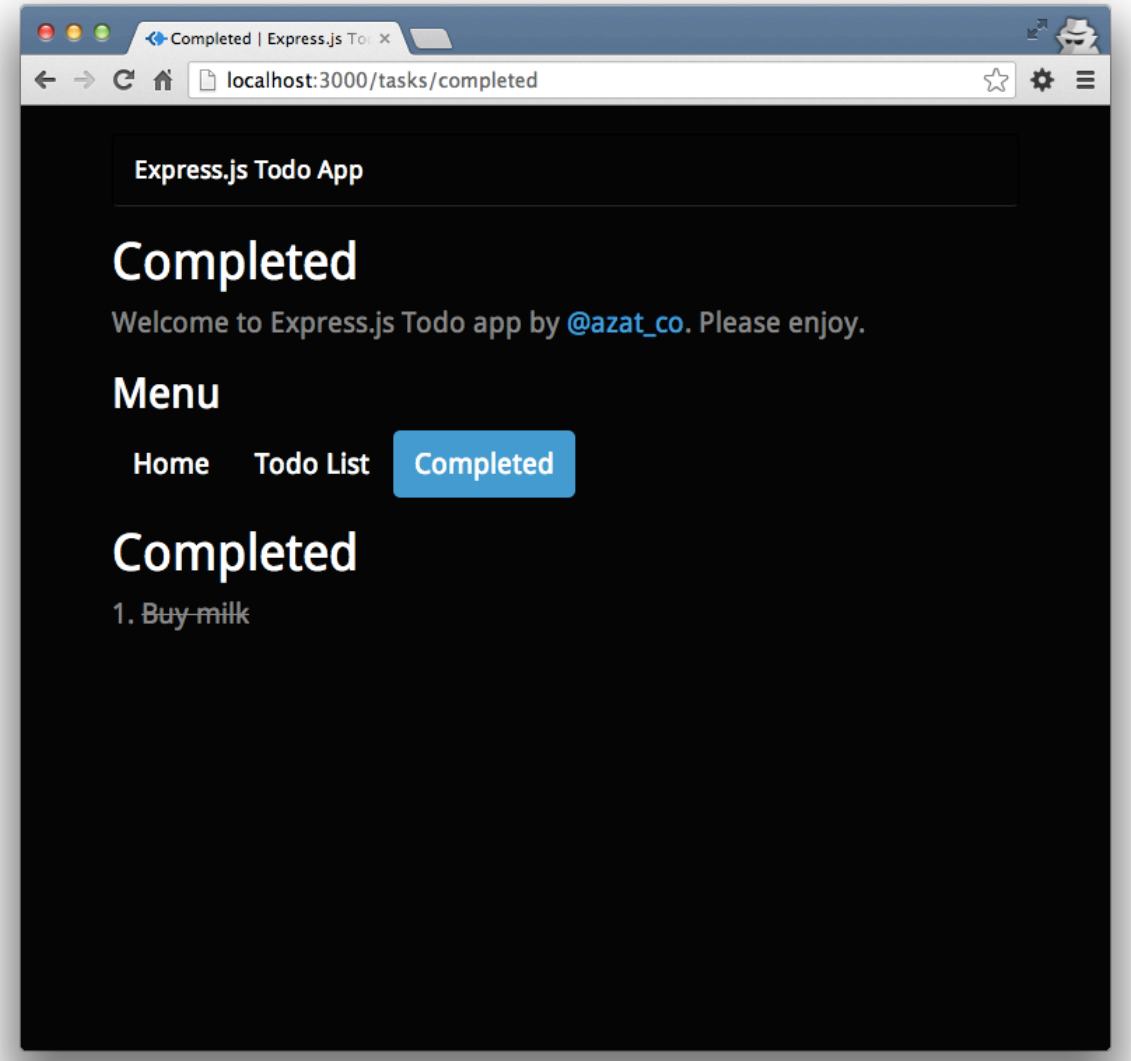
The Todo List page with added items.

Mark one of the tasks as “done”, e.g.. “Buy milk”:



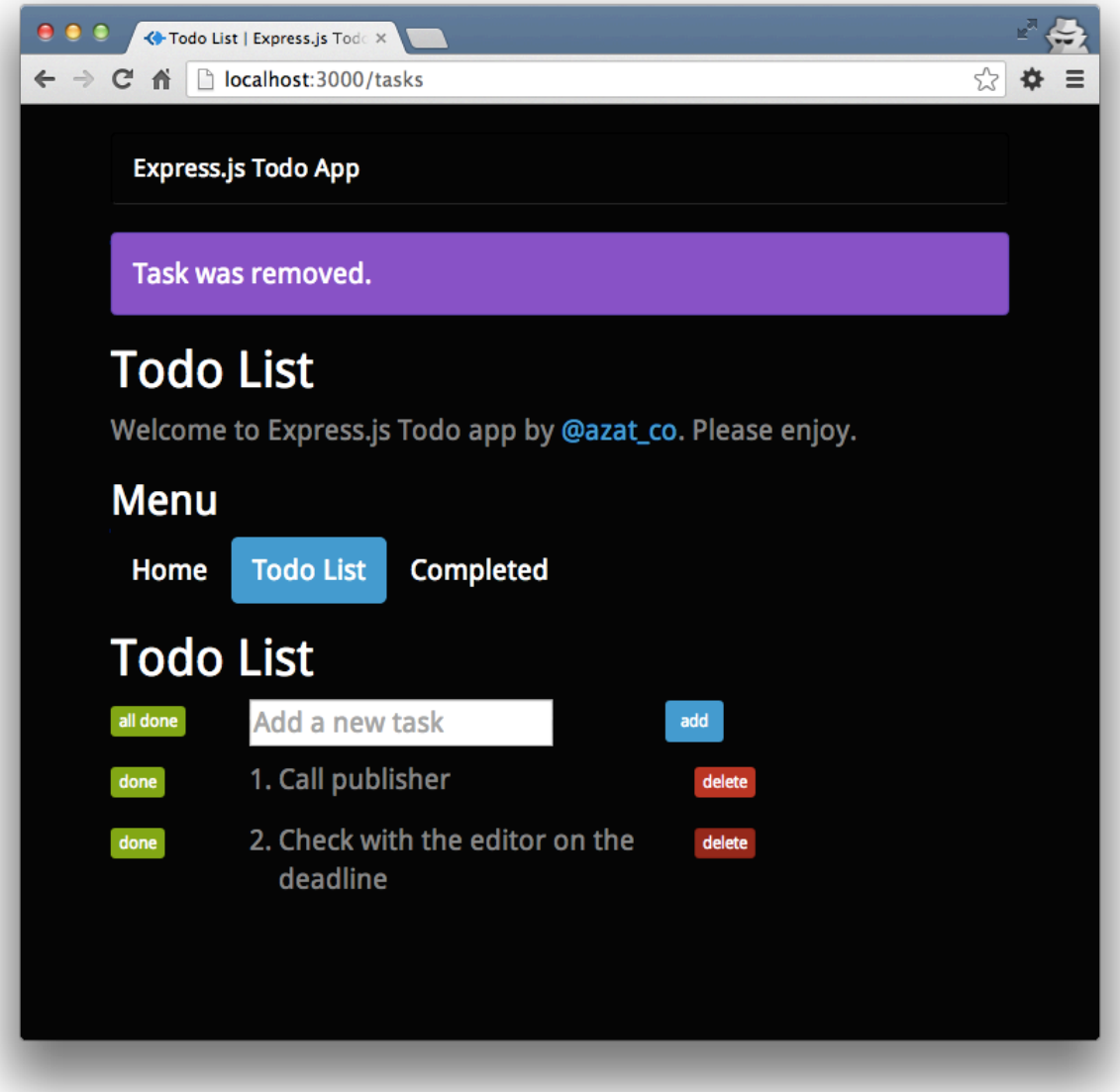
The Todo List page with one item marked done.

Going to the Complete page reveals this *done* item:



The Todo app Completed page.

Deletion of an item from the Todo list is the only action performed via AJAX/XHR request. The rest of the logic is done via GETs and POSTs (by forms).



The Todo List page with a removed tasks.

34.1 Scaffolding

As usual, we start by running

```
$ express todo-express
$ cd todo-express
$ npm install
```

This will give us the basic Express.js application.

We'll need to add two extra dependencies to `package.json`, the less-middleware and Mongoskin libraries:

```
$ npm install less-middleware --save
$ npm install mongoskin --save
```

Changing the name to `todo-express` is optional:

```
{
  "name": "todo-express",
  "version": "0.0.1",
  "private": true,
  "scripts": {
    "start": "node app.js"
  },
  "dependencies": {
    "express": "3.3.5",
    "jade": "*",
    "mongoskin": "~0.6.0",
    "less-middleware": "~0.1.12"
  }
}
```

34.2 MongoDB

Install MongoDB if you don't have it already.

```
$ brew update
$ brew install mongodb
$ mongo --version
```

For more flavors of MongoDB installation, check out the official docs[3].

34.3 Structure

The final version of the app has the following folder/file structure (GitHub[4]):

[3] http://docs.mongodb.org/manual/tutorial/install-mongodb-on-os-x/

[4] http://github.com/azat-co/todo-express

```
/todo-express
  /public
    /bootstrap
      *.less
    /images
    /javascripts
      main.js
      jquery.js
    /stylesheets
      style.css
      main.less
  /routes
    tasks.js
    index.js
  /views
    tasks_completed.jade
    layout.jade
    index.jade
    tasks.jade
  app.js
  readme.md
  package.json
```

The `*.less` in `bootstrap` folder means there are bunch of Twitter Bootstrap[5] (the CSS framework) source files. They're available at GitHub[6].

34.4 app.js

This is a break down of the Express.js generated `app.js` file with addition of routes, database, session, LESS and param middlewares.

Firstly, we import dependencies with Node.js global `require()` function:

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

Similarly, we get access to our own modules which are app's routes:

[5] http://getbootstrap.com/

[6] https://github.com/twbs/bootstrap/tree/master/less

```
var routes = require('./routes');
var tasks = require('./routes/tasks');
```

The core `http` and `path` modules will be needed as well:

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

Mongoskin is a better alternative to the native MongoDB driver:

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

One line is all we need to get the database connection object. The first param follows standard URI convention of `protocol://username:password@host:port/database`:

```
var db = mongoskin.db('mongodb://localhost:27017/todo?auto_reconnect', {safe:true\
});
```

The app itself:

```
var app = express();
```

In this middleware, we *export* the database object to all middlewares. By doing so, we'll be able to perform database operations in the routes modules:

```
app.use(function(req, res, next) {
  req.db = {};
```

We just store the tasks collection in every request:

```
  req.db.tasks = db.collection('tasks');
  next();
})
```

This line allows us to access `appname` from within every jade template:

```
app.locals.appname = 'Express.js Todo App'
```

We set the server port to either the environment variable or if that's undefined to 3000:

```
app.set('port', process.env.PORT || 3000);
```

These statements tell Express.js where templates live and what file extension to prepend in case the extension is omitted during the `render` calls:

```
app.set('views', __dirname + '/views');
app.set('view engine', 'jade');
```

Display Express.js favicon (the graphic in the URL address bar of browsers):

```
app.use(express.favicon());
```

Out-of-the-box logger will print requests in the terminal window:

```
app.use(express.logger('dev'));
```

The `bodyParser()` middleware is needed for painlessly accessing incoming data:

```
app.use(express.bodyParser());
```

The `methodOverride()` middleware is a work around for HTTP methods that involve headers. It's not essential for this example, but we'll leave it here:

```
app.use(express.methodOverride());
```

To use CSRF, we need `cookieParser()` and `session()`:

```
app.use(express.cookieParser());
app.use(express.session({
  secret: '59B93087-78BC-4EB9-993A-A61FC844F6C9'
}));
```

The `csrf()` middleware itself. The order is important; in other words, `csrf()` **must** be preceded by `cookieParser()` and `session()`:

```
app.use(express.csrf());
```

To process LESS stylesheets into CSS ones, we utilize `less-middleware` in this manner:

```
app.use(require('less-middleware')({
  src: __dirname + '/public',
  compress: true
}));
```

The other static files are also in the `public` folder:

```
app.use(express.static(path.join(__dirname, 'public')));
```

Remember CSRF? This is how we expose it to templates:

```
app.use(function(req, res, next) {
  res.locals._csrf = req.session._csrf;
  return next();
})
```

The router plug-in is enabled by this statement. It's important to have this line **after** `less-middleware` and `csrf()` lines above:

```
app.use(app.router);
```

It's possible to configure different behavior based on environments:

```
if ('development' == app.get('env')) {
  app.use(express.errorHandler());
}
```

When there's a request that matches route/RegExp with `:task_id` in it, this block is executed:

```
app.param('task_id', function(req, res, next, taskId) {
```

The value of task ID is in `taskId` and we query the database to find that object:

```
  req.db.tasks.findById(taskId, function(error, task){
```

It's very important to check for errors and empty results:

```
    if (error) return next(error);
    if (!task) return next(new Error('Task is not found.'));
```

If there's data, store it in the request and proceed to next middleware:

```
    req.task = task;
    return next();
  });
});
```

Now it's time to define our routes. We start with home page:

```
app.get('/', routes.index);
```

The Todo List page:

```
app.get('/tasks', tasks.list);
```

This route will mark all tasks in the todo list as completed if the user presses *all done* button. In a RESP API, the HTTP method would be PUT but because we're building classical web apps with forms, we have to use POST:

```
app.post('/tasks', tasks.markAllCompleted)
```

The same URL for adding new tasks as for marking all tasks completed, but in the previous methods itself (`markAllCompleted`) you'll see how we handle flow control:

```
app.post('/tasks', tasks.add);
```

To mark a single task completed, we use aforementioned `:task_id` string in our URL pattern. In REST API, this should have been a PUT request:

```
app.post('/tasks/:task_id', tasks.markCompleted);
```

Unlike with the POST route above, we utilize Express.js param middleware with `:task_id` token:

```
app.del('/tasks/:task_id', tasks.del);
```

For our Completed page, we define this route:

```
app.get('/tasks/completed', tasks.completed);
```

In case of malicious attacks or mistyped URLs, it's a user-friendly thing to catch all requests with `*`. Keep in mind that if we had a match previously, the Node.js won't come to execute this block:

```
app.all('*', function(req, res){
  res.send(404);
})
```

Finally, we spin up our application with good 'ol `http` method:

```
http.createServer(app).listen(app.get('port'),
  function(){
    console.log('Express server listening on port '
      + app.get('port'));
  }
);
```

The full content of `app.js` file:

```
/**
 * Module dependencies.
 */

var express = require('express');
var routes = require('./routes');
var tasks = require('./routes/tasks');
var http = require('http');
var path = require('path');
var mongoskin = require('mongoskin');
var db = mongoskin.db('mongodb://localhost:27017/todo?auto_reconnect', {safe:true\
});
var app = express();
app.use(function(req, res, next) {
  req.db = {};
  req.db.tasks = db.collection('tasks');
  next();
})
app.locals.appname = 'Express.js Todo App'
// all environments

```

```
22
23  app.set('port', process.env.PORT || 3000);
24  app.set('views', __dirname + '/views');
25  app.set('view engine', 'jade');
26  app.use(express.favicon());
27  app.use(express.logger('dev'));
28  app.use(express.bodyParser());
29  app.use(express.methodOverride());
30  app.use(express.cookieParser());
31  app.use(express.session({secret: '59B93087-78BC-4EB9-993A-A61FC844F6C9'}));
32  app.use(express.csrf());
33
34  app.use(require('less-middleware')({ src: __dirname + '/public', compress: true }\
35  ));
36  app.use(express.static(path.join(__dirname, 'public')));
37  app.use(function(req, res, next) {
38    res.locals._csrf = req.session._csrf;
39    return next();
40  })
41  app.use(app.router);
42
43  // development only
44  if ('development' == app.get('env')) {
45    app.use(express.errorHandler());
46  }
47  app.param('task_id', function(req, res, next, taskId) {
48    req.db.tasks.findById(taskId, function(error, task){
49      if (error) return next(error);
50      if (!task) return next(new Error('Task is not found.'));
51      req.task = task;
52      return next();
53    });
54  });
55
56  app.get('/', routes.index);
57  app.get('/tasks', tasks.list);
58  app.post('/tasks', tasks.markAllCompleted)
59  app.post('/tasks', tasks.add);
60  app.post('/tasks/:task_id', tasks.markCompleted);
61  app.del('/tasks/:task_id', tasks.del);
62  app.get('/tasks/completed', tasks.completed);
63
```

```
app.all('*', function(req, res){
  res.send(404);
})
http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

34.5 Routes

There are only two files in `routes` folder. One of them serves the home page (e.g., http://localhost:3000/) and is straightforward:

```
/*
 * GET home page.
 */

exports.index = function(req, res){
  res.render('index', { title: 'Express.js Todo App' });
};
```

The remaining logic that deals with tasks itself has been placed in the `todo-express/routes/tasks.js`. Let's break it down a bit.

We start by exporting `list()` request handler that gives us list of incomplete tasks:

```
exports.list = function(req, res, next){
```

To do so, we perform a database search with completed=false query:

```
  req.db.tasks.find({
    completed: false
  }).toArray(function(error, tasks){
```

In the callback, we need to check for any errors: `javascript if (error) return next(error);`

Since we use `toArray()`, we can send the date directly to the template:

```
    res.render('tasks', {
      title: 'Todo List',
      tasks: tasks || []
    });
  });
};
```

Adding a new task requires us to check for the name parameter:

```
exports.add = function(req, res, next){
  if (!req.body || !req.body.name)
    return next(new Error('No data provided.'));
```

Thanks to our middleware, we already have a database collection in the `req` object, and the default value for the task is incomplete (`completed: false`):

```
  req.db.tasks.save({
    name: req.body.name,
    completed: false
  }, function(error, task){
```

Again, it's important to check for errors and propagate them with Express.js `next()` function:

```
    if (error) return next(error);
    if (!task) return next(new Error('Failed to save.'));
```

The logging is optional; however, it's useful for learning and debugging:

```
    console.info('Added %s with id=%s', task.name, task._id);
```

Lastly, we redirect back to the Todo List page when the saving operation is finished successfully:

```
    res.redirect('/tasks');
  })
};
```

This method marks all incomplete tasks as complete:

```
exports.markAllCompleted = function(req, res, next) {
```

Because we had to re-use POST route and since it's a good illustration of flow control, we check for the `all_done` parameter to decide if this request comes from the *all done* button or the *add* button:

```
  if (!req.body.all_done
    || req.body.all_done !== 'true')
    return next();
```

If the execution come this far, we perform db query with `multi: true`:`javascript req.db.tasks.update({ completed: false }, {$set: { completed: true }}, {multi: true}, function(error, count){`

Significant error handling, logging and redirection back to Todo List page:

```
    if (error) return next(error);
    console.info('Marked %s task(s) completed.', count);
    res.redirect('/tasks');
  })
};
```

The Completed route is akin to Todo List except for the completed flag value (true in this case):

```
exports.completed = function(req, res, next) {
  req.db.tasks.find({
    completed: true
  }).toArray(function(error, tasks) {
    res.render('tasks_completed', {
      title: 'Completed',
      tasks: tasks || []
    });
  });
};
```

This is the route that takes care of marking a single task as done. We use `updateById` but the same thing can be accomplished with a plain `update` method from Mongoskin/MongoDB API. The trick with `completed: req.body.completed === 'true'` is needed because the incoming value is a string and not a boolean.

```
exports.markCompleted = function(req, res, next) {
  if (!req.body.completed)
    return next(new Error('Param is missing'));
  req.db.tasks.updateById(req.task._id, {
    $set: {completed: req.body.completed === 'true'}},
    function(error, count) {
```

Once more, we perform error and results check (`update()` and `updateById()` don't return object, but the count of affected documents instead):

```
      if (error) return next(error);
      if (count !==1)
        return next(new Error('Something went wrong.'));
      console.info('Marked task %s with id=%s completed.',
        req.task.name,
        req.task._id);
      res.redirect('/tasks');
    }
  )
}
```

Delete is the single route called by an AJAX request. However, there's nothing special about its implementation. The only difference is that we don't redirect, but send status 200 back.

Just for your information, the `remove()` method can be used instead of `removeById()`.

```
exports.del = function(req, res, next) {
  req.db.tasks.removeById(req.task._id, function(error, count) {
    if (error) return next(error);
    if (count !==1) return next(new Error('Something went wrong.'));
    console.info('Deleted task %s with id=%s completed.',
      req.task.name,
      req.task._id);
    res.send(200);
  });
}
```

For your convenience, here's the full content of the `todo-express/routes/tasks.js` file:

```
/*
 * GET users listing.
 */

exports.list = function(req, res, next){
  req.db.tasks.find({completed: false}).toArray(function(error, tasks){
    if (error) return next(error);
    res.render('tasks', {
      title: 'Todo List',
      tasks: tasks || []
    });
  });
};

exports.add = function(req, res, next){
  if (!req.body || !req.body.name) return next(new Error('No data provided.'));
  req.db.tasks.save({
    name: req.body.name,
    completed: false
  }, function(error, task){
    if (error) return next(error);
    if (!task) return next(new Error('Failed to save.'));
    console.info('Added %s with id=%s', task.name, task._id);
    res.redirect('/tasks');
  })
};

exports.markAllCompleted = function(req, res, next) {
  if (!req.body.all_done || req.body.all_done !== 'true') return next();
  req.db.tasks.update({
    completed: false
  }, {$set: {
    completed: true
  }}, {multi: true}, function(error, count){
    if (error) return next(error);
    console.info('Marked %s task(s) completed.', count);
    res.redirect('/tasks');
  })
};

exports.completed = function(req, res, next) {
  req.db.tasks.find({completed: true}).toArray(function(error, tasks) {
```

```
43      res.render('tasks_completed', {
44        title: 'Completed',
45        tasks: tasks || []
46      });
47    });
48  };
49
50  exports.markCompleted = function(req, res, next) {
51    if (!req.body.completed) return next(new Error('Param is missing'));
52    req.db.tasks.updateById(req.task._id, {$set: {completed: req.body.completed ===\
53   'true'}}, function(error, count) {
54      if (error) return next(error);
55      if (count !==1) return next(new Error('Something went wrong.'));
56      console.info('Marked task %s with id=%s completed.', req.task.name, req.task.\
57  _id);
58      res.redirect('/tasks');
59    })
60  };
61
62  exports.del = function(req, res, next) {
63    req.db.tasks.removeById(req.task._id, function(error, count) {
64      if (error) return next(error);
65      if (count !==1) return next(new Error('Something went wrong.'));
66      console.info('Deleted task %s with id=%s completed.', req.task.name, req.task\
67  ._id);
68      res.send(200);
69    });
70  };
```

34.6 Jades

In the Todo app, we use four templates:

- `layout.jade`: the skeleton of HTML pages that is used on all pages
- `index.jade`: home page
- `tasks.jade`: Todo List page
- `tasks_completed.jade`: Completed page

Let's go through each file starting with `layout.jade`. It starts with doctype, html and head types:

```
doctype 5
html
  head
```

We should have `appname` variable set:

```
    title= title + ' | ' + appname
```

Next we include `*.css` files but underneath, Express.js will serve its contents from LESS files:

```
    link(rel="stylesheet", href="/stylesheets/style.css")
    link(rel="stylesheet", href="/bootstrap/bootstrap.css")
    link(rel="stylesheet", href="/stylesheets/main.css")
```

The body with Twitter Bootstrap structure consist of `.container` and `.navbar`. To read more about those and other classes, go to getbootstrap.com/css/[7]:

```
  body
    .container
      .navbar.navbar-default
        .container
          .navbar-header
            a.navbar-brand(href='/')= appname
      .alert.alert-dismissable
      h1= title
      p Welcome to Express.js Todo app by 
        a(href='http://twitter.com/azat_co') @azat_co
        |. Please enjoy.
```

This is the place where other jades (like `tasks.jade`) will be imported:

```
      block content
```

The last lines include front-end JavaScript files:

```
  script(src='/javascripts/jquery.js', type="text/javascript")
  script(src='/javascripts/main.js', type="text/javascript")
```

The full `layout.jade` file:

[7] http://getbootstrap.com/css/

```
doctype 5
html
  head
    title= title + ' | ' + appname
    link(rel="stylesheet", href="/stylesheets/style.css")
    link(rel="stylesheet", href="/bootstrap/bootstrap.css")
    link(rel="stylesheet", href="/stylesheets/main.css")

  body
    .container
      .navbar.navbar-default
        .container
          .navbar-header
            a.navbar-brand(href='/')= appname
      .alert.alert-dismissable
      h1= title
      p Welcome to Express.js Todo app by 
        a(href='http://twitter.com/azat_co') @azat_co
        |. Please enjoy.
      block content
  script(src='/javascripts/jquery.js', type="text/javascript")
  script(src='/javascripts/main.js', type="text/javascript")
```

The `index.jade` file is our home page and it's quite vanilla. The most interesting thing it had is the `nav-pills` menu:

```
extends layout

block content
  .menu
    h2 Menu
    ul.nav.nav-pills
      li.active
        a(href="/tasks") Home
      li
        a(href="/tasks") Todo List
      li
        a(href="/tasks") Completed
  .home
    p This is an example of classical (no front-end JavaScript frameworks) web ap\
plication built with Express.js 3.3.5 for
      a(href="http://expressjsguide.com") Express.js Guide
```

```
17        |.
18      p The full source code is available at
19        a(href='http://github.com/azat-co/todo-express') github.com/azat-co/todo-ex\
20 press
21        |.
```

The `tasks.jade` uses `extends layout`:

```
1  extends layout
2
3  block content
```

Then goes our main page specific content:

```
1    .menu
2      h2 Menu
3      ul.nav.nav-pills
4        li
5          a(href='/') Home
6        li.active
7          a(href='/tasks') Todo List
8        li
9          a(href="/tasks/completed") Completed
10   h1= title
```

The `div` with list class will hold the Todo List:

```
1    .list
2      .item.add-task
```

The form to mark all items as done has CSRF token in a hidden field and uses POST method pointed to `/tasks`:

```
1        div.action
2          form(action='/tasks', method='post')
3            input(type='hidden', value='true', name='all_done')
4            input(type='hidden', value=locals._csrf, name='_csrf')
5            input(type='submit', class='btn btn-success btn-xs', value='all done')
```

Similar CSRF enabled form is for new task creation:

```
      form(action="/tasks", method='post')
        input(type='hidden', value=locals._csrf, name='_csrf')
        div.name
          input(type="text", name="name", placeholder='Add a new task')
        div.delete
          input.btn.btn-primary.btn-sm(type="submit", value='add')
```

When we start the app for the first time (or clean the database), there are no tasks:

```
    if (tasks.length === 0)
      | No tasks.
```

Jade supports iterations with `each` command:

```
    each task, index in tasks
      .item
        div.action
```

This form submits data to individual task route:

```
          form(action='/tasks/#{task._id}', method='post')
            input(type='hidden', value=task._id.toString(), name='id')
            input(type='hidden', value='true', name='completed')
            input(type='hidden', value=locals._csrf, name='_csrf')
            input(type='submit', class='btn btn-success btn-xs task-done', value=\
'done')
```

The `index` variable is used to display order in the list of tasks:

```
        div.num
          span=index+1
            |. 
        div.name
          span.name=task.name
          //- no support for DELETE method in forms
          //- http://amundsen.com/examples/put-delete-forms/
          //- so do XHR request instead from public/javascripts/main.js
```

The delete button doesn't have anything fancy attached to it, because events are attached to these buttons from `main.js` front-end JavaScript file:

```
        div.delete
          a(class='btn btn-danger btn-xs task-delete', data-task-id=task._id.toSt\
ring(), data-csrf=locals._csrf) delete
```

The full source code of `tasks.jade`:

```
extends layout

block content

  .menu
    h2 Menu
    ul.nav.nav-pills
      li
        a(href='/') Home
      li.active
        a(href='/tasks') Todo List
      li
        a(href="/tasks/completed") Completed
  h1= title

  .list
    .item.add-task
      div.action
        form(action='/tasks', method='post')
          input(type='hidden', value='true', name='all_done')
          input(type='hidden', value=locals._csrf, name='_csrf')
          input(type='submit', class='btn btn-success btn-xs', value='all done')
      form(action="/tasks", method='post')
        input(type='hidden', value=locals._csrf, name='_csrf')
        div.name
          input(type="text", name="name", placeholder='Add a new task')
        div.delete
          input.btn.btn-primary.btn-sm(type="submit", value='add')
    if (tasks.length === 0)
      | No tasks.
    each task, index in tasks
      .item
        div.action
          form(action='/tasks/#{task._id}', method='post')
            input(type='hidden', value=task._id.toString(), name='id')
            input(type='hidden', value='true', name='completed')
```

```
37              input(type='hidden', value=locals._csrf, name='_csrf')
38              input(type='submit', class='btn btn-success btn-xs task-done', value=\
39  'done')
40          div.num
41            span=index+1
42              |. 
43          div.name
44            span.name=task.name
45            //- no support for DELETE method in forms
46            //- http://amundsen.com/examples/put-delete-forms/
47            //- so do XHR request instead from public/javascripts/main.js
48          div.delete
49            a(class='btn btn-danger btn-xs task-delete', data-task-id=task._id.toSt\
50  ring(), data-csrf=locals._csrf) delete
```

Last but not least, comes `tasks_completed.jade` which is just a striped down version of `tasks.jade` file:

```
1   extends layout
2
3   block content
4
5     .menu
6       h2 Menu
7       ul.nav.nav-pills
8         li
9           a(href='/') Home
10        li
11          a(href='/tasks') Todo List
12        li.active
13          a(href="/tasks/completed") Completed
14
15    h1= title
16
17    .list
18      if (tasks.length === 0)
19        | No tasks.
20      each task, index in tasks
21        .item
22          div.num
23            span=index+1
24              |. 
```

```
25          div.name.completed-task
26            span.name=task.name
```

34.7 LESS

As we've mentioned before, after applying proper middleware in `app.js` files, we can put `*.less` files anywhere under `public` folder. Express.js works by accepting request for some `.css` file and tries to match corresponding file by name. Therefore, we include `*.css` files in our jades.

Here is the content of the `todo-express/public/stylesheets/main.less` file:

```
1  * {
2    font-size:20px;
3  }
4  .btn {
5    // margin-left: 20px;
6    // margin-right: 20px;
7  }
8  .num {
9    // margin-right: 3px;
10 }
11 .item {
12   height: 44px;
13   width: 100%;
14   clear: both;
15   .name {
16     width: 300px;
17   }
18   .action {
19     width: 100px;
20   }
21   .delete {
22     width: 100px
23   }
24   div {
25     float:left;
26   }
27 }
28 .home {
29   margin-top: 40px;
30 }
31 .name.completed-task {
```

```
32    text-decoration: line-through;
33  }
```

34.8 Conclusion

The Todo app is considered classical because it doesn't rely on any front-end framework. This was done intentionally to show how easy it is to use Express.js for such tasks. In modern day development, people often leverage some sort of REST API server architecture with front-end client built with Backbone.js, Angular, Ember or something else[8]. Next examples dive into details about how to write such servers.

[8] http://todomvc.com

35 REST API

In this tutorial in addition to Express.js, we'll use MongoDB via Mongoskin.

This tutorial will walk you through writing test using the Mocha[1] and Super Agent[2] libraries and then use them in a test-driven development manner to build a Node.js[3] free JSON REST API server utilizing Express.js[4] framework and Mongoskin[5] library for MongoDB[6].

Example

All the source code is in the github.com/azat-co/rest-api-express[7] for your convenience.

In this REST API server, we'll perform **create**, **update**, **remove and delete** (CRUD) operations and harness Express.js middleware[8] concept with `app.param()` and `app.use()` methods.

35.1 Test Coverage

Before anything else, let's write functional tests that make HTTP requests to our soon-to-be-created REST API server. If you know how to use Mocha[9] or just want to jump straight to the Express.js app implementation, feel free to do so. You can use CURL terminal commands for testing too.

Assuming we already have Node.js, NPM[10] and MongoDB installed, let's create a *new* folder (or if you wrote the tests, use that folder):

```
1 mkdir rest-api
2 cd rest-api
```

We'll use Mocha[11], Expect.js[12] and Super Agent[13] libraries. To install them, run these commands from the project folder:

[1] http://visionmedia.github.io/mocha/
[2] http://visionmedia.github.io/superagent/
[3] http://nodejs.org
[4] http://expressjs.com/
[5] https://github.com/kissjs/node-mongoskin
[6] http://www.mongodb.org/
[7] https://github.com/azat-co/rest-api-express
[8] http://expressjs.com/api.html#middleware
[9] http://visionmedia.github.io/mocha/
[10] http://npmjs.org
[11] http://visionmedia.github.io/mocha/
[12] https://github.com/LearnBoost/expect.js/
[13] http://visionmedia.github.io/superagent/

```
$ npm install -g mocha
$ npm install expect.js
$ npm install superagent
```

Tip

Installing Mocha locally will give us the ability to use different versions at the same time. To run tests, simply point to `./node_modules/mocha/bin/mocha`. A better alternative would be to use Makefile as outlined in the HackHall example.

Now let's create **express.test.js** file in the same folder which will have six suites:

- creating a new object
- retrieving an object by its ID
- retrieving the whole collection
- updating an object by its ID
- checking an updated object by its ID
- removing an object by its ID

HTTP requests are just a breeze with Super Agent's chained functions which we'll put inside of each test suite.

Here is the full source code for the **rest-api-express/express.test.js** file:

```
var superagent = require('superagent')
var expect = require('expect.js')

describe('express rest api server', function(){
  var id

  it('post object', function(done){
    superagent.post('http://localhost:3000/collections/test')
      .send({ name: 'John'
        , email: 'john@rpjs.co'
      })
      .end(function(e,res){
        // console.log(res.body)
        expect(e).to.eql(null)
        expect(res.body.length).to.eql(1)
        expect(res.body[0]._id.length).to.eql(24)
        id = res.body[0]._id
```

```
18          done()
19        })
20    })
21
22    it('retrieves an object', function(done){
23      superagent.get('http://localhost:3000/collections/test/'+id)
24        .end(function(e, res){
25          // console.log(res.body)
26          expect(e).to.eql(null)
27          expect(typeof res.body).to.eql('object')
28          expect(res.body._id.length).to.eql(24)
29          expect(res.body._id).to.eql(id)
30          done()
31        })
32    })
33
34    it('retrieves a collection', function(done){
35      superagent.get('http://localhost:3000/collections/test')
36        .end(function(e, res){
37          // console.log(res.body)
38          expect(e).to.eql(null)
39          expect(res.body.length).to.be.above(0)
40          expect(res.body.map(function (item){return item._id})).to.contain(id)
41          done()
42        })
43    })
44
45    it('updates an object', function(done){
46      superagent.put('http://localhost:3000/collections/test/'+id)
47        .send({name: 'Peter'
48          , email: 'peter@yahoo.com'})
49        .end(function(e, res){
50          // console.log(res.body)
51          expect(e).to.eql(null)
52          expect(typeof res.body).to.eql('object')
53          expect(res.body.msg).to.eql('success')
54          done()
55        })
56    })
57
58    it('checks an updated object', function(done){
59      superagent.get('http://localhost:3000/collections/test/'+id)
```

```
60        .end(function(e, res){
61          // console.log(res.body)
62          expect(e).to.eql(null)
63          expect(typeof res.body).to.eql('object')
64          expect(res.body._id.length).to.eql(24)
65          expect(res.body._id).to.eql(id)
66          expect(res.body.name).to.eql('Peter')
67          done()
68        })
69    })
70    it('removes an object', function(done){
71      superagent.del('http://localhost:3000/collections/test/'+id)
72        .end(function(e, res){
73          // console.log(res.body)
74          expect(e).to.eql(null)
75          expect(typeof res.body).to.eql('object')
76          expect(res.body.msg).to.eql('success')
77          done()
78        })
79    })
80  })
```

To run the tests, we can use the `$ mocha express.test.js` command.

35.2 Dependencies

In this tutorial, we'll utilize Mongoskin[14], a MongoDB library which is a better alternative to the plain good old native MongoDB driver for Node.js[15]. In addition, Mongoskin is more light-weight than Mongoose and schema-less. For more insight, please check out Mongoskin comparison blurb[16].

Express.js[17] is a wrapper for the core Node.js HTTP module[18] objects. The Express.js framework is build on top of Connect[19] middleware and provided tons of convenience. Some people compare the framework to Ruby's Sinatra in terms of how it's non-opinionated and configurable.

If you've create a `rest-api` folder in the previous section *Test Coverage*, simply run these commands to install modules for the application:

[14] https://github.com/kissjs/node-mongoskin
[15] https://github.com/mongodb/node-mongodb-native
[16] https://github.com/kissjs/node-mongoskin#comparation
[17] http://expressjs.com/
[18] http://nodejs.org/api/http.html
[19] https://github.com/senchalabs/connect

```
npm install express
npm install mongoskin
```

35.3 Implementation

First things first, so let's define our dependencies:

```
var express = require('express')
  , mongoskin = require('mongoskin')
```

After version 3.x, Express streamlines the instantiation of its app instance, in a way that this line will give us a server object:

```
var app = express()
```

To extract params from the body of the requests, we'll use `bodyParser()` middleware which looks more like a configuration statement:

```
app.use(express.bodyParser())
```

Middleware (in this[20] and other forms[21]) is a powerful and convenient pattern in Express.js and Connect[22] to organize and re-use code.

As with the `bodyParser()` method that saves us from the hurdles of parsing a body object of HTTP request, Mongoskin makes it possible to connect to the MongoDB database in one effortless line of code:

```
var db = mongoskin.db('localhost:27017/test', {safe:true});
```

Note

If you wish to connect to a remote database, e.g., MongoHQ[23] instance, substitute the string with your username, password, host and port values. Here is the format of the URI string: `mongodb://[username:password@] host1[:port1][,host2[:port2],...[,hostN[:portN]]] [/[database][?options]]`

The `app.param()` method is another Express.js middleware. It basically says "*do something every time there is this value in the URL pattern of the request handler*". In our case, we select a particular collection when request pattern contains a string `collectionName` prefixed with a colon (you'll see it later in the routes):

[20] http://expressjs.com/api.html#app.use
[21] http://expressjs.com/api.html#middleware
[22] https://github.com/senchalabs/connect
[23] https://www.mongohq.com/home

```
app.param('collectionName', function(req, res, next, collectionName){
  req.collection = db.collection(collectionName)
  return next()
})
```

Merely, to be user-friendly, let's put a root route with a message:

```
app.get('/', function(req, res, next) {
  res.send('please select a collection, e.g., /collections/messages')
})
```

Now the real work begins, here is how we retrieve a list of items sorted by `_id` and which has a limit of 10:

```
app.get('/collections/:collectionName', function(req, res, next) {
  req.collection.find({},{
    limit:10, sort: [['_id',-1]]
  }).toArray(function(e, results){
    if (e) return next(e)
    res.send(results)
  })
})
```

Have you noticed a `:collectionName` string in the URL pattern parameter? This and the previous `app.param()` middleware is what gives us the `req.collection` object which points to a specified collection in our database.

The object creating endpoint is slightly easier to grasp since we just pass the whole payload to the MongoDB (method a.k.a. free JSON REST API):

```
app.post('/collections/:collectionName', function(req, res, next) {
  req.collection.insert(req.body, {}, function(e, results){
    if (e) return next(e)
    res.send(results)
  })
})
```

Single object retrieval functions are faster than `find()`, but they use different interface (they return object directly instead of a cursor), so please be aware of that. In addition, we're extracting the ID from `:id` part of the path with `req.params.id` Express.js magic:

```
app.get('/collections/:collectionName/:id', function(req, res, next) {
  req.collection.findOne({
    _id: req.collection.id(req.params.id)
  }, function(e, result){
    if (e) return next(e)
    res.send(result)
  })
})
```

PUT request handler gets more interesting because `update()` doesn't return the augmented object; instead, it returns a count of affected objects.

Also `{$set:req.body}` is a special MongoDB operator (operators tend to start with a dollar sign) that sets values.

The second ‘ {safe:true, multi:false}‘ parameter is an object with options that tell MongoDB to wait for the execution before running the callback function and to process only one (first) item.

```
app.put('/collections/:collectionName/:id', function(req, res, next) {
  req.collection.update({
      _id: req.collection.id(req.params.id)
    }, {$set:req.body}, {safe:true, multi:false},
    function(e, result){
      if (e) return next(e)
      res.send((result===1)?{msg:'success'}:{msg:'error'})
    }
  );
})
```

Finally, the DELETE method which also outputs a custom JSON message:

```
app.del('/collections/:collectionName/:id', function(req, res, next) {
  req.collection.remove({
      _id: req.collection.id(req.params.id)
    },
    function(e, result){
      if (e) return next(e)
      res.send((result===1)?{msg:'success'}:{msg:'error'})
    }
  );
})
```

Note: *The* `delete` *is an operator in JavaScript, so Express.js uses* `app.del` *instead.*

The last line that actually starts the server on port 3000 in this case:

```
app.listen(3000)
```

Just in case something is not working quite well, here is the full code of **rest-api-express/express.js** file :

```
var express = require('express')
  , mongoskin = require('mongoskin')

var app = express()
app.use(express.bodyParser())

var db = mongoskin.db('localhost:27017/test', {safe:true});

app.param('collectionName', function(req, res, next, collectionName){
  req.collection = db.collection(collectionName)
  return next()
})

app.get('/', function(req, res, next) {
  res.send('please select a collection, e.g., /collections/messages')
})

app.get('/collections/:collectionName', function(req, res, next) {
  req.collection.find({},{limit:10, sort: [['_id',-1]]}).toArray(function(e, resu\
lts){
    if (e) return next(e)
    res.send(results)
  })
})

app.post('/collections/:collectionName', function(req, res, next) {
  req.collection.insert(req.body, {}, function(e, results){
    if (e) return next(e)
    res.send(results)
  })
})


app.get('/collections/:collectionName/:id', function(req, res, next) {
  req.collection.findOne({_id: req.collection.id(req.params.id)}, function(e, res\
ult){
    if (e) return next(e)
```

```
    res.send(result)
  })
})
app.put('/collections/:collectionName/:id', function(req, res, next) {
  req.collection.update({_id: req.collection.id(req.params.id)}, {$set:req.body},\
 {safe:true, multi:false}, function(e, result){
    if (e) return next(e)
    res.send((result===1)?{msg:'success'}:{msg:'error'})
  })
})
app.del('/collections/:collectionName/:id', function(req, res, next) {
  req.collection.remove({_id: req.collection.id(req.params.id)}, function(e, resu\
lt){
    if (e) return next(e)
    res.send((result===1)?{msg:'success'}:{msg:'error'})
  })
})



app.listen(3000)
```

Exit your editor and run this command in your terminal:

```
$ node express.js
```

And in a different window (without closing the first one):

```
$ mocha express.test.js
```

If you really don't like Mocha and/or BDD, CURL is always there for you. :-)

For example, CURL data to make a POST request:

```
$ curl -d "" http://localhost:3000
```

GET requests also work in the browser, for example http://localhost:3000/test.

In this tutorial, our tests are longer than the app code itself so abandoning test-driven development might be tempting, but believe me, **the good habits of TDD will save you hours and hours** during any serious development when the complexity of the application you are working on is big.

35.4 Conclusion

The Express.js and Mongoskin libraries are great when you need to build a simple REST API server in a few lines of code. Later, if you need to expand the libraries, they also provide a way to configure and organize your code.

NoSQL databases like MongoDB are good at free-REST APIs where we don't have to define schemas and can throw any data and it'll be saved.

The full code for both test and app files: https://gist.github.com/azat-co/6075685.

If you would like to learn more about Express.js and other JavaScript libraries, take a look at the series Intro to Express.js tutorials[24].

Note: *In this example, I'm using semi-colon less style. Semi-colons in JavaScript are absolutely optional[25] except in two cases: in the for loop and before expressions/statements that start with parenthesis (e.g., Immediately-Invoked Function Expression[26]).

[24] http://webapplog.com/tag/intro-to-express-js/

[25] http://blog.izs.me/post/2353458699/an-open-letter-to-javascript-leaders-regarding

[26] http://en.wikipedia.org/wiki/Immediately-invoked_function_expression

36 HackHall

This project was written using Backbone.js and Underscore for the front-end app, and Express.js, MongoDB via Mongoose for the back-end REST API server.

Example

The HackHall source code is in the public GitHub repository[1].

The live demo is accessible at hackhall.com[2] either with AngelList or pre-filled email (1@1.com) and password (1).

36.1 What is HackHall

HackHall (ex-Accelerator.IO) is an open-source invite-only social network and collaboration tool for hackers, hipsters, entrepreneurs and pirates (just kidding). HackHall is akin to Reddit, plus Hacker News, plus Facebook Groups with curation.

The HackHall project is in its early stage and roughly a beta. We plan to extend the code base in the future and bring in more people to share skills, wisdom and passion for programming.

In this chapter, we'll cover the 1.0 release[3] which has:

- OAuth 1.0 with oauth modules and AngelList API
- Email and password authentication
- Mongoose models and schemas
- Express.js structure with routes in modules
- JSON REST API
- Express.js error handling
- Front-end client Backbone.js app (for more info on Backbone.js, download/read online our Rapid Prototyping with JS[4] tutorials)
- Environmental variables with Foreman's `.env`
- TDD with Mocha
- Basic Makefile setup

[1] http://github.com/azat-co/hackhall
[2] http://hackhall.com
[3] https://github.com/azat-co/hackhall/releases/tag/1.0
[4] http://rpjs.co

36.2 Running HackHall

To get the source code, you can navigate to `hackhall` folder or clone from GitHub:

```
$ git clone git@github.com:azat-co/hackhall
$ git checkout 1.0
$ npm install
```

If you plan to test an AngelList (optional), HackHall is using Heroku and Foreman setup for AngelList API keys storing them in environmental variables, so we need to add `.evn` file like this (below are fake values):

```
ANGELLIST_CLIENT_ID=254C0335-5F9A-4607-87C0
ANGELLIST_CLIENT_SECRET=99F5C1AC-C5F7-44E6-81A1-8DF4FC42B8D9
```

The keys are obtainable at angel.co/api[5] after someone creates and registers his/her AngelList app.

Download and install MongoDB if you don't have it already. The databases and third-party libraries are out of scope of this book. However, you can find enough materials online[6] and in Rapid Prototyping with JS[7].

To start MongoDB server, open a new terminal window and run:

```
$ mongod
```

Go back to the project folder and run:

```
$ foreman start
```

After MongoDB is running on localhost with default port 27017, it's possible to seed the database `hackhall` with default admin user by running `seed.js` mongo script:

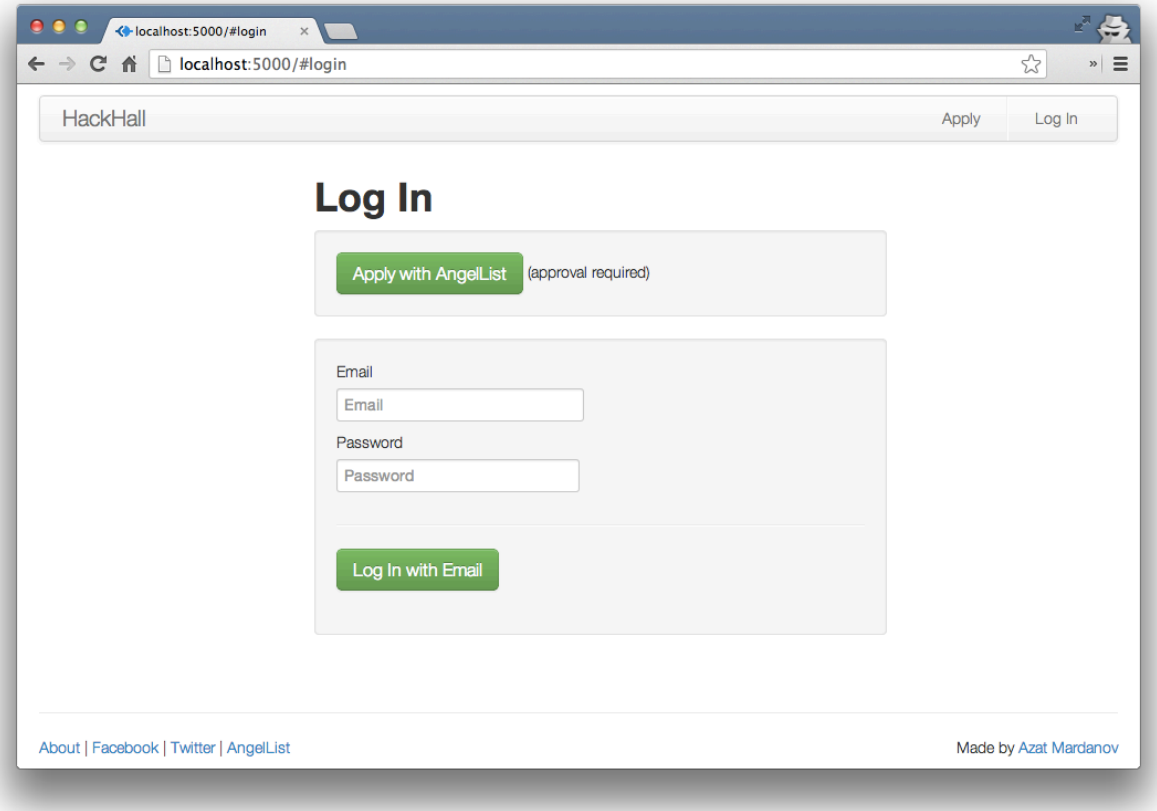
```
$ mongo localhost:27017/hackhall seed.js
```

Feel free to modify `seed.js` to your liking (beware that it erases all previous data!):

[5] https://angel.co/api
[6] http://webapplog.com
[7] http://rpjs.co

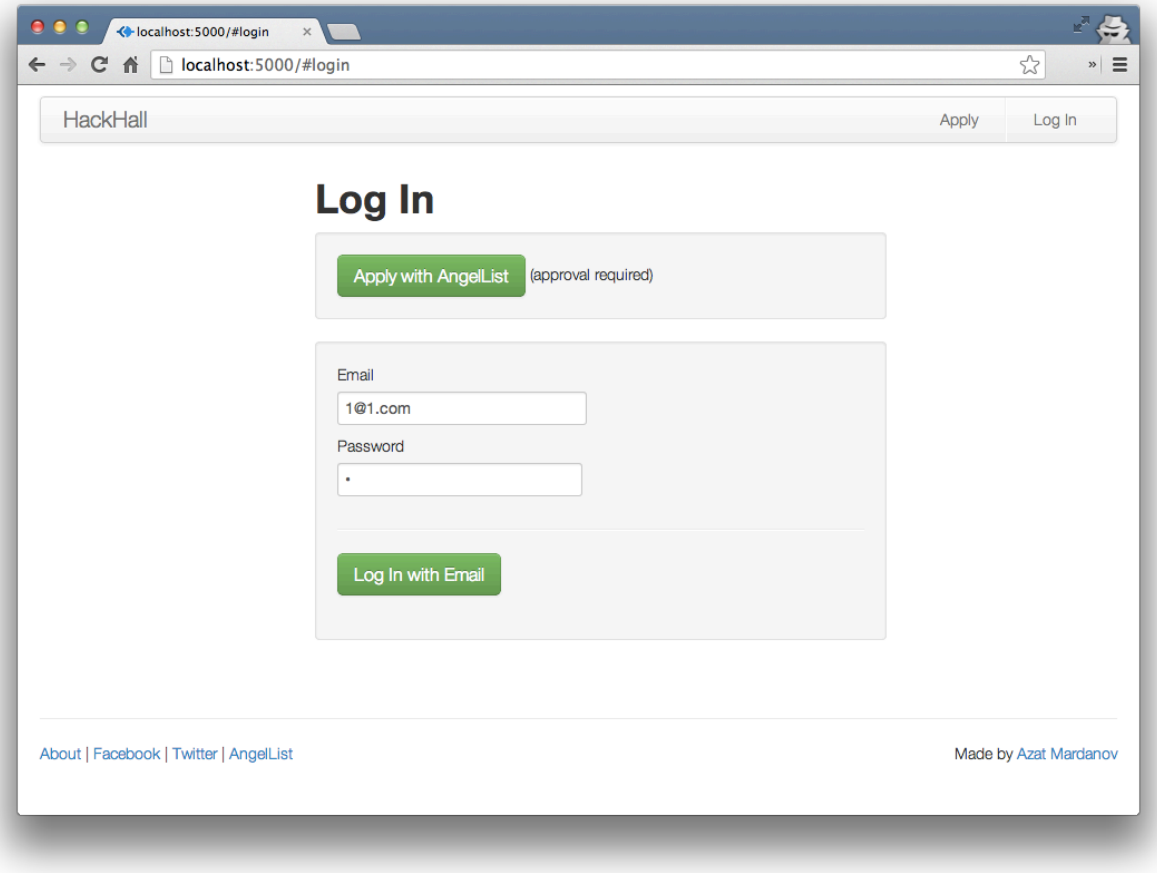
```
db.dropDatabase();
var seedUser ={
  firstName:'Azat',
  lastName:"Mardanov",
  displayName:"Azat Mardanov",
  password:'1',
  email:'1@1.com',
  role:'admin',
  approved: true,
  admin: true
};
db.users.save(seedUser);
```

If you open your browser at http://localhost:5000, you should see the login screen.



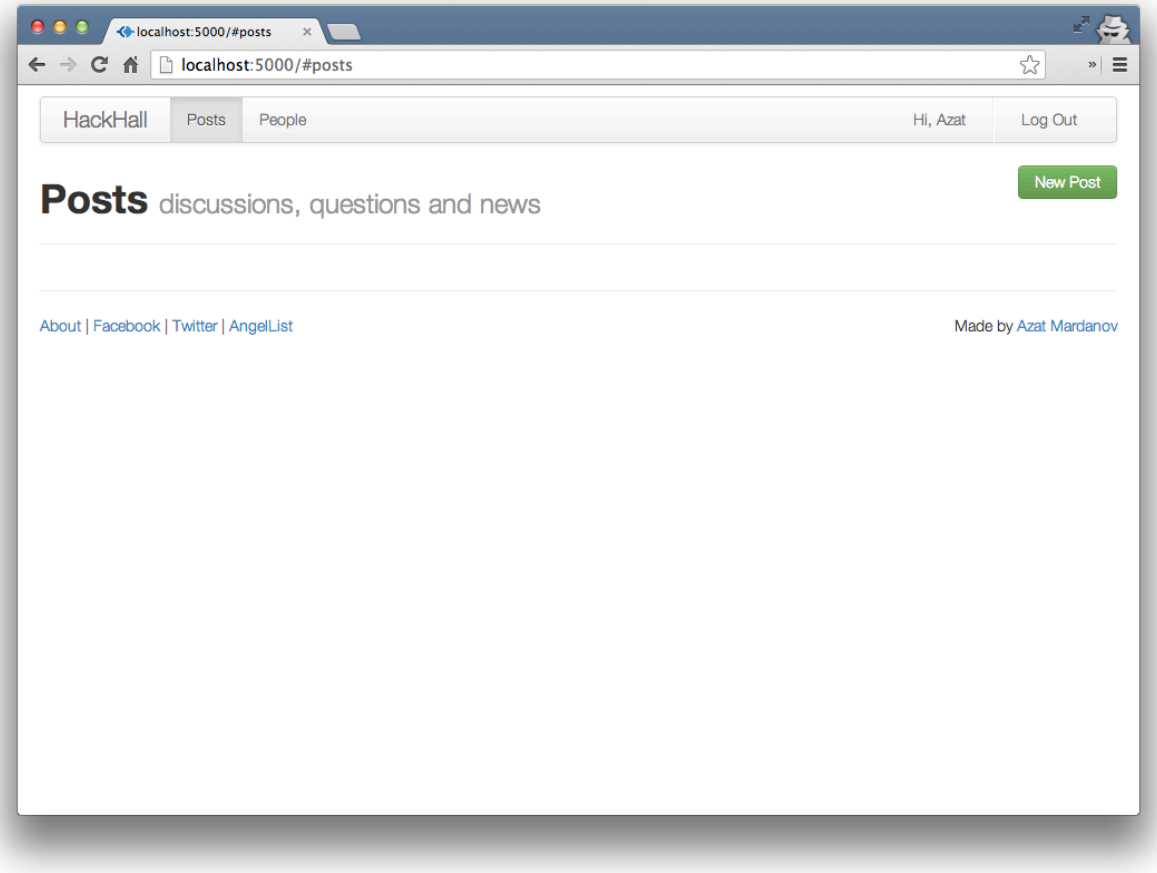
The HackHall login page running locally.

Enter username and password to get in (the ones from the `seed.js` file).



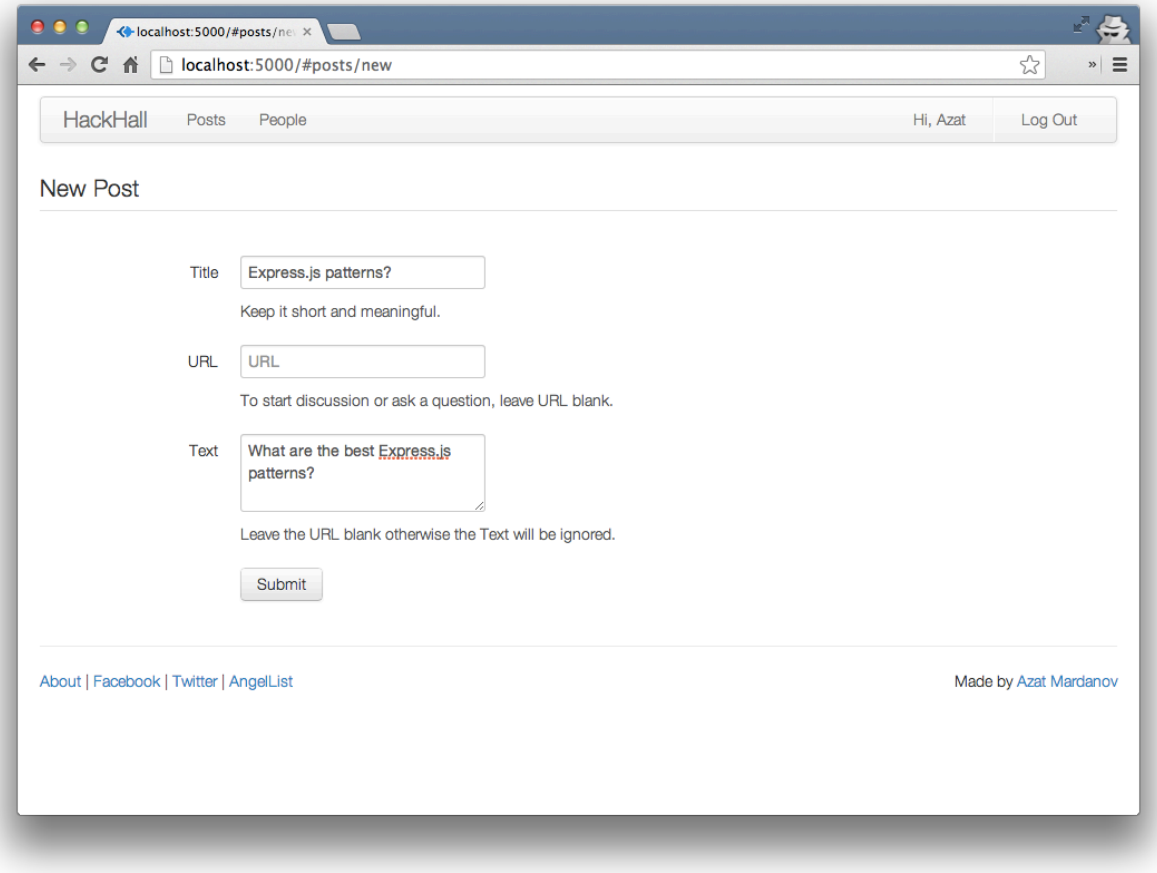
The HackHall login page with seed data.

After successful authentication, users are redirected to the Posts page:



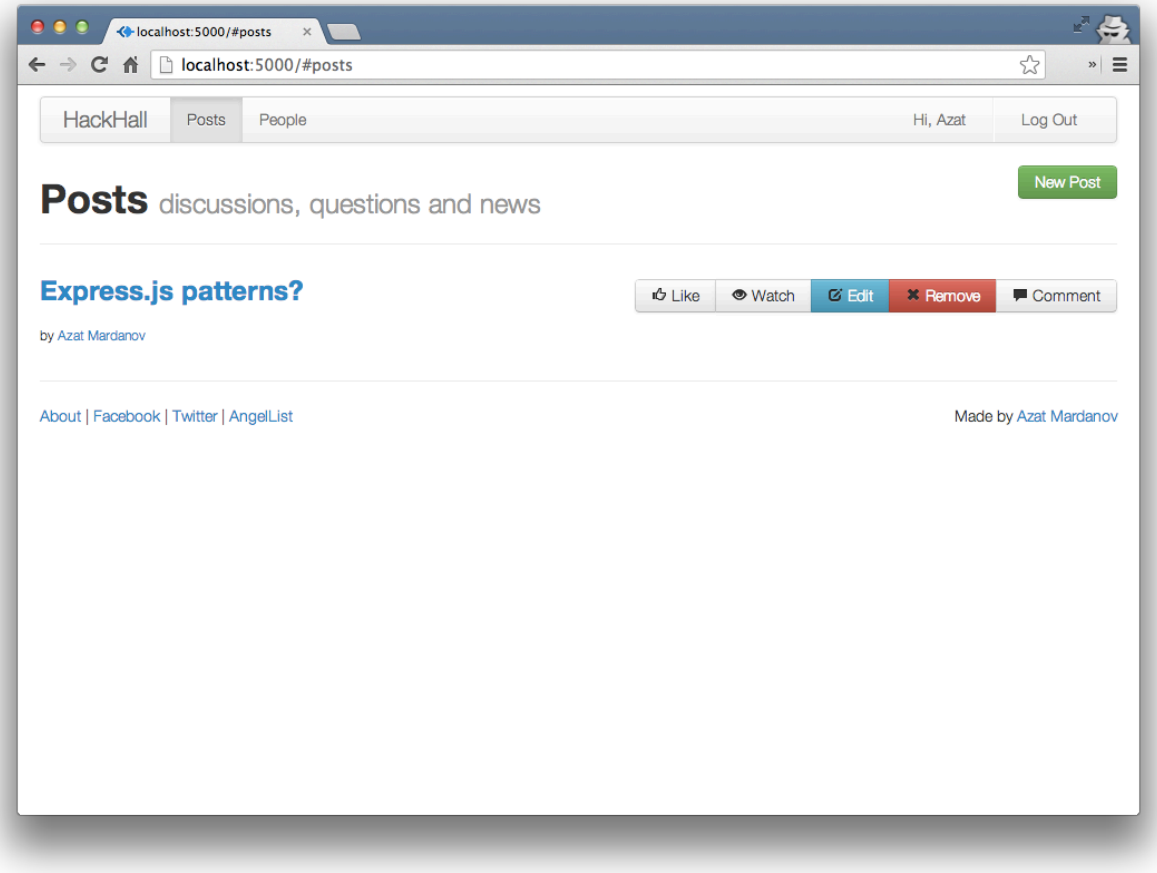
The HackHall Posts page.

where they can create a post (e.g., a question):



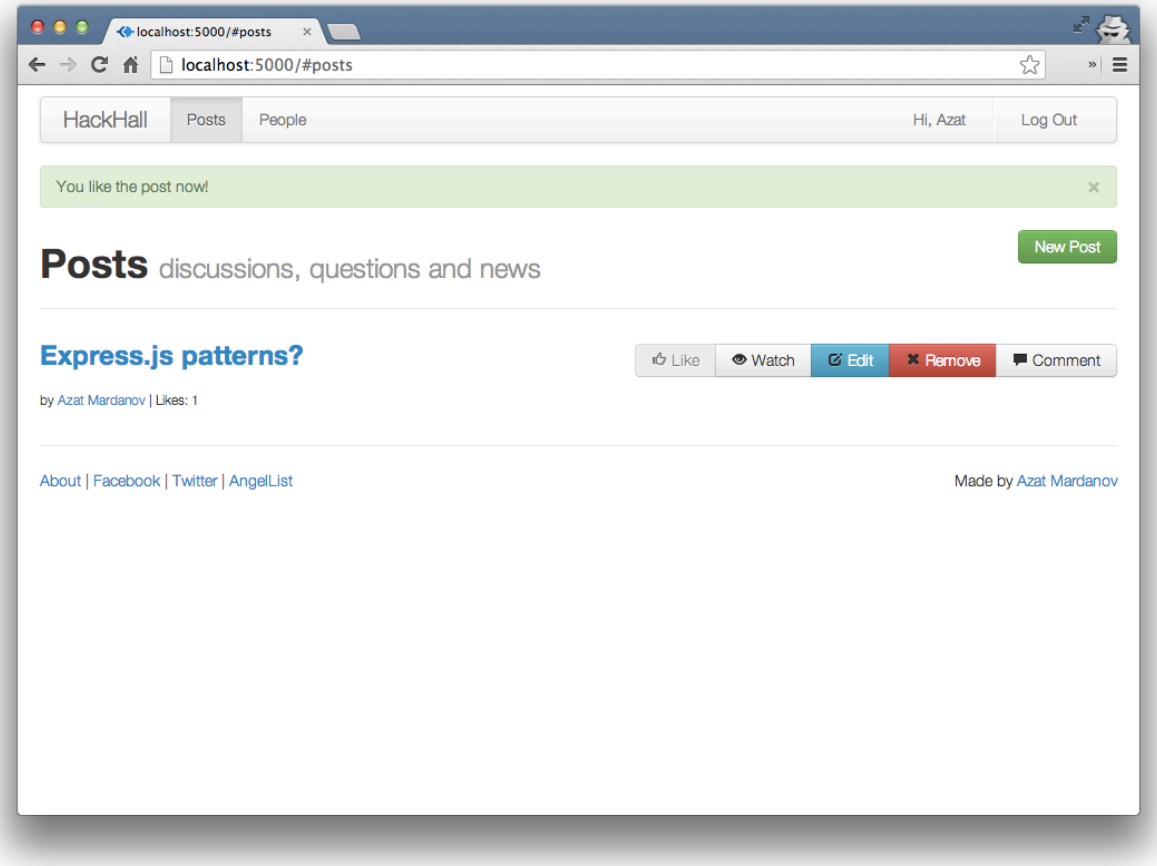
The HackHall Posts page.

Save the post:



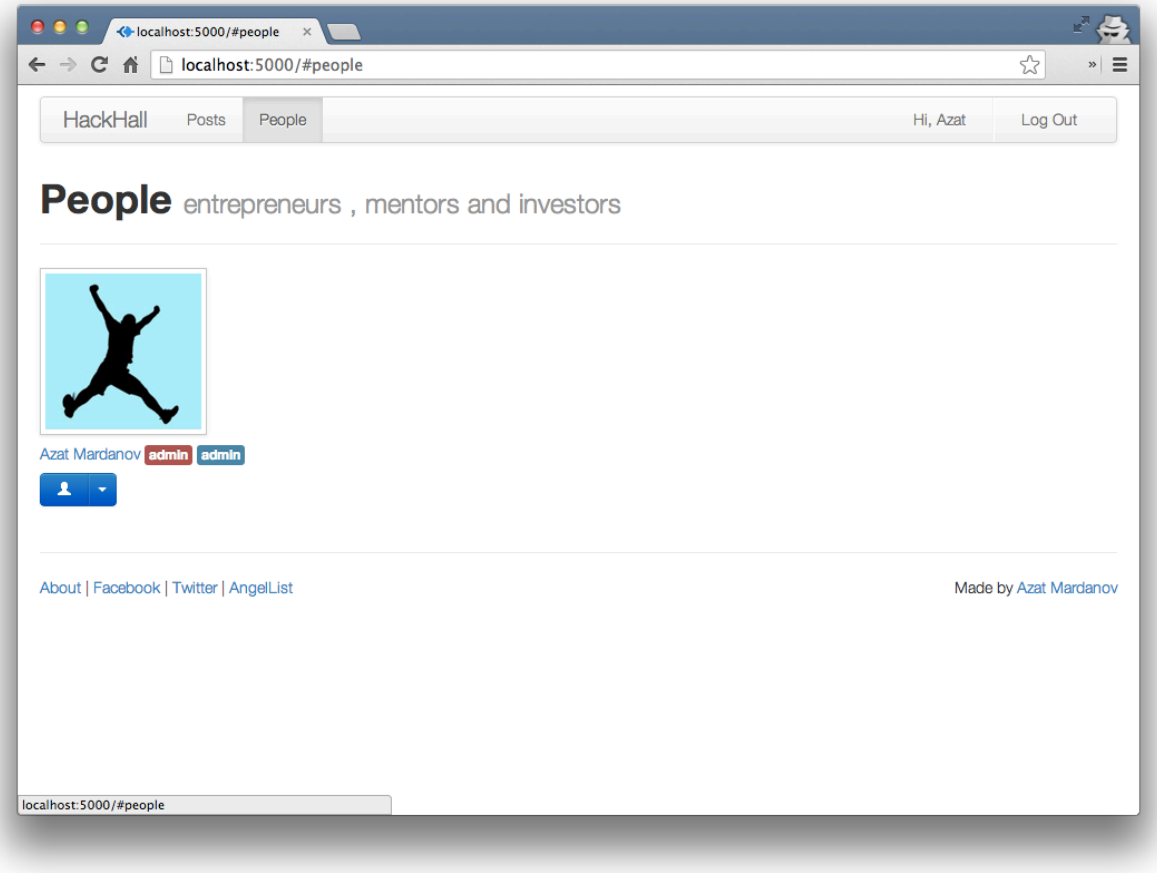
The HackHall Posts page with a saved post.

Like posts:



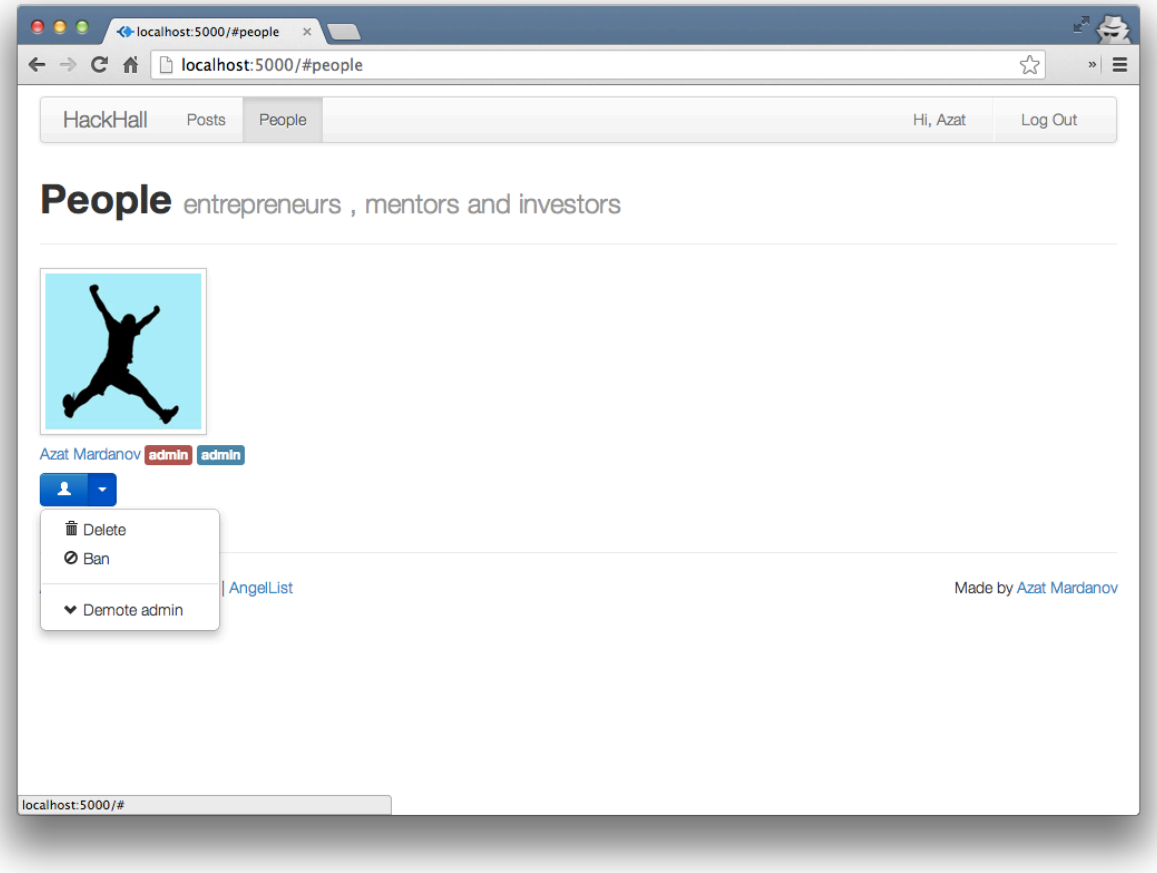
The HackHall Posts page with a liked post.

Visit other users' profiles:



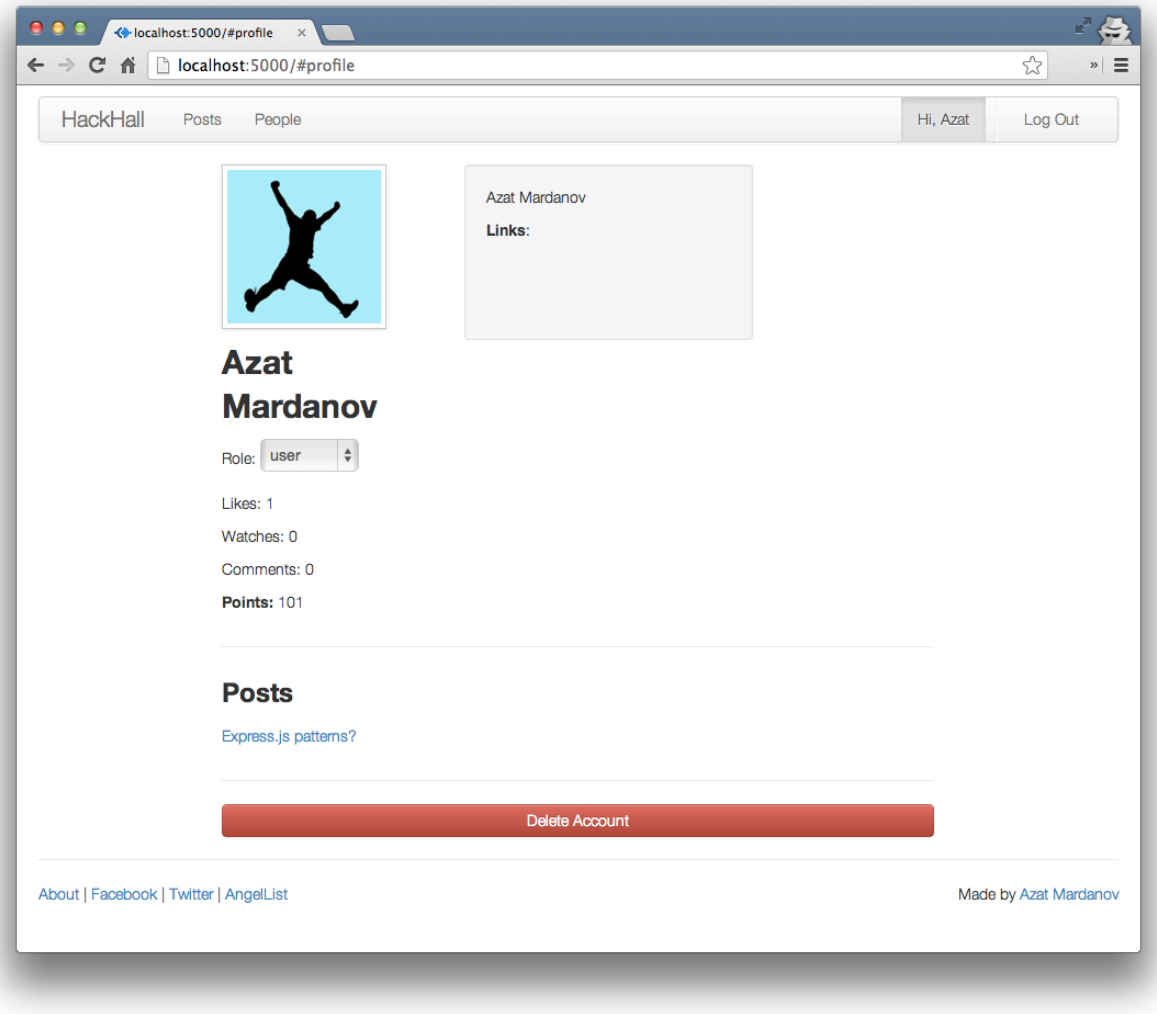
The HackHall People page.

If they have admin rights, users can approve applicants:



The HackHall People page with admin rights.

and manage their account on Profile page:



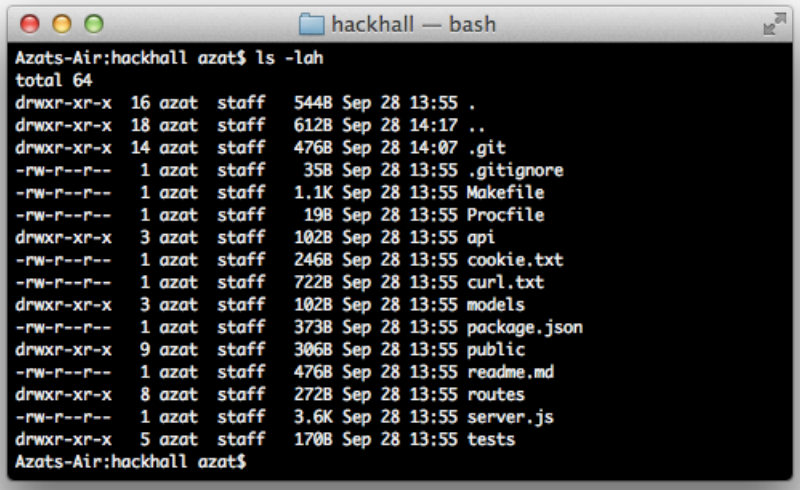
The HackHall Profile page.

36.3 Structure

Here are what each of the folders and files have:

- `/api`: app-shared routes
- `/models`: Mongoose models
- `/public`: Backbone app, static files like front-end JavaScript, CSS, HTML
- `/routes`: REST API routes
- `/tests`: Mocha tests
- `.gitignore`: list of files that git should ignore
- `Makefile`: make file to run tests

- `Procfile`: Cedar stack file needed for Heroku deployment
- `package.json`: NPM dependencies and HackHall meta data
- `readme.md`: description
- `server.js`: main HackHall server file



Content of the HackHall base folder.

36.4 Express.js App

Let's jump straight to the `server.js` file and learn how it's implemented. Firstly, we declare dependencies:

```
1  var express = require('express'),
2    routes = require('./routes'),
3    http = require('http'),
4    util = require('util'),
5    oauth = require('oauth'),
6    querystring = require('querystring');
```

Then, we initialize app and configure middlewares. The `process.env.PORT` is populated by Heroku, and in case of a local setup fallsback on 3000.

```
var app = express();
app.configure(function(){
  app.set('port', process.env.PORT || 3000  );
  app.use(express.favicon());
  app.use(express.logger('dev'));
  app.use(express.bodyParser());
  app.use(express.methodOverride());
```

The values passed to cookieParser and session middlewares are needed for authentication. Obviously, session secrets are supposed to be private:

```
  app.use(express.cookieParser('asd;lfkajs;ldfkj'));
  app.use(express.session({
    secret: '<h1>WHEEYEEE</h1>',
    key: 'sid',
    cookie: {
      secret: true,
      expires: false
    }
  }));
```

This is how we serve our front-end client Backbone.js app and other static files like CSS:

```
  app.use(express.static(__dirname + '/public'));
  app.use(app.router);
});
```

Error handling is broken down into three functions with `clientErrorHandler` dedicated to AJAX/XHR requests from the Backbone.js app (responds with JSON):

```
app.configure(function() {
  app.use(logErrors);
  app.use(clientErrorHandler);
  app.use(errorHandler);
});

function logErrors(err, req, res, next) {
  console.error('logErrors', err.toString());
  next(err);
}

```

```
function clientErrorHandler(err, req, res, next) {
  console.error('clientErrors ', err.toString());
  res.send(500, { error: err.toString()});
  if (req.xhr) {
    console.error(err);
    res.send(500, { error: err.toString()});
  } else {
    next(err);
  }
}

function errorHandler(err, req, res, next) {
  console.error('lastErrors ', err.toString());
  res.send(500, {error: err.toString()});
}
```

In the same way that we determine `process.env.PORT` and fallback on local setup value 3000, we do a similar thing with MongoDB connection string:

```
var dbUrl = process.env.MONGOHQ_URL
  || 'mongodb://@127.0.0.1:27017/hackhall';
var mongoose = require('mongoose');
var connection = mongoose.createConnection(dbUrl);
connection.on('error', console.error.bind(console,
  'connection error:'));
```

Sometime it's a good idea to log on the connection open event:

```
connection.once('open', function () {
  console.info('connected to database')
});
```

The Mongoose models live in the `models` folder:

```
var models = require('./models');
```

This middleware will provide access to two collections within our routes methods:

```
function db (req, res, next) {
  req.db = {
    User: connection.model('User', models.User, 'users'),
    Post: connection.model('Post', models.Post, 'posts')
  };
  return next();
}
```

Just a new name for imported auth functions:

```
checkUser = routes.main.checkUser;
checkAdmin = routes.main.checkAdmin;
checkApplicant = routes.main.checkApplicant;
```

AngelList OAuth routes:

```
app.get('/auth/angellist', routes.auth.angelList);
app.get('/auth/angellist/callback',
  routes.auth.angelListCallback,
  routes.auth.angelListLogin,
  db,
  routes.users.findOrAddUser);
```

Main application routes including `api/profile` return user session if user is logged in:

```
//MAIN
app.get('/api/profile', checkUser, db, routes.main.profile);
app.del('/api/profile', checkUser, db, routes.main.delProfile);
app.post('/api/login', db, routes.main.login);
app.post('/api/logout', routes.main.logout);
```

The POST requests for creating users and posts:

```
//POSTS
app.get('/api/posts', checkUser, db, routes.posts.getPosts);
app.post('/api/posts', checkUser, db, routes.posts.add);
app.get('/api/posts/:id', checkUser, db, routes.posts.getPost);
app.put('/api/posts/:id', checkUser, db, routes.posts.updatePost);
app.del('/api/posts/:id', checkUser, db, routes.posts.del);

//USERS
app.get('/api/users', checkUser, db, routes.users.getUsers);
app.get('/api/users/:id', checkUser, db,routes.users.getUser);
app.post('/api/users', checkAdmin, db, routes.users.add);
app.put('/api/users/:id', checkAdmin, db, routes.users.update);
app.del('/api/users/:id', checkAdmin, db, routes.users.del);
```

These routes are for new members that haven't been approved yet:

```
//APPLICATION
app.post('/api/application',
  checkAdmin,
  db,
  routes.application.add);
app.put('/api/application',
  checkApplicant,
  db,
  routes.application.update);
app.get('/api/application',
  checkApplicant,
  db,
  routes.application.get);
```

The catch all else route:

```
app.get('*', function(req, res){
  res.send(404);
});
```

The `require.main === module` is a clever trick to determine if this file is being executed as a stand alone or as an imported module:

```
http.createServer(app);
if (require.main === module) {
  app.listen(app.get('port'), function(){
    console.info('Express server listening on port '
      + app.get('port'));
  });
}
else {
  console.info('Running app as a module')
  exports.app = app;
}
```

The full source code for `hackhall/server.js`:

```
var express = require('express'),
  routes = require('./routes'),
  http = require('http'),
  util = require('util'),
  oauth = require('oauth'),
  querystring = require('querystring');

var app = express();
app.configure(function(){
  app.set('port', process.env.PORT || 3000  );
  app.use(express.favicon());
  app.use(express.logger('dev'));
  app.use(express.bodyParser());
  app.use(express.methodOverride());
  app.use(express.cookieParser('asd;lfkajs;ldfkj'));
  app.use(express.session({
    secret: '<h1>WHEEYEEE</h1>',
    key: 'sid',
    cookie: {
      secret: true,
      expires: false
    }
  }));
  // app.use(express.csrf());
  // app.use(function(req, res, next) {
    // res.locals.csrf = req.session._cstf;
    // return next();
  // });
```

```
29    app.use(express.static(__dirname + '/public'));
30    app.use(app.router);
31  });
32
33  app.configure(function() {
34    app.use(logErrors);
35    app.use(clientErrorHandler);
36    app.use(errorHandler);
37  });
38
39  function logErrors(err, req, res, next) {
40    console.error('logErrors', err.toString());
41    next(err);
42  }
43
44  function clientErrorHandler(err, req, res, next) {
45    console.error('clientErrors ', err.toString());
46    res.send(500, { error: err.toString()});
47    if (req.xhr) {
48      console.error(err);
49      res.send(500, { error: err.toString()});
50    } else {
51      next(err);
52    }
53  }
54
55  function errorHandler(err, req, res, next) {
56    console.error('lastErrors ', err.toString());
57    res.send(500, {error: err.toString()});
58  }
59
60  var dbUrl = process.env.MONGOHQ_URL || 'mongodb://@127.0.0.1:27017/hackhall';
61  var mongoose = require('mongoose');
62  var connection = mongoose.createConnection(dbUrl);
63  connection.on('error', console.error.bind(console, 'connection error:'));
64  connection.once('open', function () {
65    console.info('connected to database')
66  });
67
68  var models = require('./models');
69  function db (req, res, next) {
70    req.db = {
```

```
    User: connection.model('User', models.User, 'users'),
    Post: connection.model('Post', models.Post, 'posts')
  };
  return next();
}
checkUser = routes.main.checkUser;
checkAdmin = routes.main.checkAdmin;
checkApplicant = routes.main.checkApplicant;

app.get('/auth/angellist', routes.auth.angelList);
app.get('/auth/angellist/callback',
  routes.auth.angelListCallback,
  routes.auth.angelListLogin,
  db,
  routes.users.findOrAddUser);

//MAIN
app.get('/api/profile', checkUser, db, routes.main.profile);
app.del('/api/profile', checkUser, db, routes.main.delProfile);
app.post('/api/login', db, routes.main.login);
app.post('/api/logout', routes.main.logout);

//POSTS
app.get('/api/posts', checkUser, db, routes.posts.getPosts);
app.post('/api/posts', checkUser, db, routes.posts.add);
app.get('/api/posts/:id', checkUser, db, routes.posts.getPost);
app.put('/api/posts/:id', checkUser, db, routes.posts.updatePost);
app.del('/api/posts/:id', checkUser, db, routes.posts.del);

//USERS
app.get('/api/users', checkUser, db, routes.users.getUsers);
app.get('/api/users/:id', checkUser, db,routes.users.getUser);
app.post('/api/users', checkAdmin, db, routes.users.add);
app.put('/api/users/:id', checkAdmin, db, routes.users.update);
app.del('/api/users/:id', checkAdmin, db, routes.users.del);

//APPLICATION
app.post('/api/application', checkAdmin, db, routes.application.add);
app.put('/api/application', checkApplicant, db, routes.application.update);
app.get('/api/application', checkApplicant, db, routes.application.get);

app.get('*', function(req, res){
```

```
  res.send(404);
});

http.createServer(app);
if (require.main === module) {
  app.listen(app.get('port'), function(){
    console.info('Express server listening on port ' + app.get('port'));
  });
}
else {
  console.info('Running app as a module')
  exports.app = app;
}
```

36.5 Routes

The HackHall routes reside in `hackhall/routes` folder and are groped into several modules:

- `hackhall/routes/index.js`: bridge between `server.js` and other routes in the folder
- `hackhall/routes/auth.js`: routes that handle OAuth dance with AngelList API
- `hackhall/routes/main.js`: login, logout and other routes
- `hackhall/routes/users.js`: routes related to users REST API
- `hackhall/routes/application.js`: submission of application to become a user
- `hackhall/routes/posts.js`: routes related to posts REST API

36.5.1 index.js

Let's peak into `hackhall/routes/index.js` where we include other modules:

```
exports.posts = require('./posts');
exports.main = require('./main');
exports.users = require('./users');
exports.application = require('./application');
exports.auth = require('./auth');
```

36.5.2 auth.js

In this module, we'll handle OAuth *dance* with AngelList API. To do so, we'll have to rely on `https` library:

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

The AngelList API client ID and client secret are obtained at angel.co/api[8] website and stored in environment variables:

```
var angelListClientId = process.env.ANGELLIST_CLIENT_ID;
var angelListClientSecret = process.env.ANGELLIST_CLIENT_SECRET;
```

The method will redirect users to angel.co website for authentication:

```
exports.angelList = function(req, res) {
  res.redirect('https://angel.co/api/oauth/authorize?client_id=' + angelListClien\
tId + '&scope=email&response_type=code');
}
```

After users allow our app to access their information, AngelList sends them back to this route for us to make a new (HTTPS) request to retrieve the token:

```
exports.angelListCallback = function(req, res, next) {
  var token;
  var buf = '';
  var data;
  // console.log('/api/oauth/token?client_id='
  //+ angelListClientId
  //+ '&client_secret='
  //+ angelListClientSecret
  //+ '&code='
  //+ req.query.code
  //+ '&grant_type=authorization_code');
  var angelReq = https.request({
      host: 'angel.co',
      path: '/api/oauth/token?client_id='
        + angelListClientId
        + '&client_secret='
        + angelListClientSecret
        + '&code='
        + req.query.code
        + '&grant_type=authorization_code',
      port: 443,
      method: 'POST',
```

[8] http://angel.co/api

```
      headers: {
        'content-length': 0
      }
    },
    function(angelRes) {
      angelRes.on('data', function(buffer) {
        buf += buffer;
      });
      angelRes.on('end', function() {
        try {
          data = JSON.parse(buf.toString('utf-8'));
        } catch (e) {
          if (e) return res.send(e);
        }
        if (!data || !data.access_token) return res.send(500);
        token = data.access_token;
        req.session.angelListAccessToken = token;
        if (token) next();
        else res.send(500);
      });
    });
  angelReq.end();
  angelReq.on('error', function(e) {
    console.error(e);
    next(e);
  });
}
```

Directly call AngleList API with the token from the previous middleware to get user information:

```
exports.angelListLogin = function(req, res, next) {
  token = req.session.angelListAccessToken;
  httpsRequest = https.request({
      host: 'api.angel.co',
      path: '/1/me?access_token=' + token,
      port: 443,
      method: 'GET'
    },
    function(httpsResponse) {
      httpsResponse.on('data', function(buffer) {
        data = JSON.parse(buffer.toString('utf-8'));
        if (data) {
```

```
          req.angelProfile = data;
          next();
        }
      });
    }
  );
  httpsRequest.end();
  httpsRequest.on('error', function(e) {
    console.error(e);
  });
};
```

The full source code for `hackhall/routes/auth.js` files:

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

var angelListClientId = process.env.ANGELLIST_CLIENT_ID;
var angelListClientSecret = process.env.ANGELLIST_CLIENT_SECRET;

exports.angelList = function(req, res) {
  res.redirect('https://angel.co/api/oauth/authorize?client_id=' + angelListClien\
tId + '&scope=email&response_type=code');
}
exports.angelListCallback = function(req, res, next) {
  var token;
  var buf = '';
  var data;
  // console.log('/api/oauth/token?client_id=' + angelListClientId + '&client_sec\
ret=' + angelListClientSecret + '&code=' + req.query.code + '&grant_type=authoriz\
ation_code');
  var angelReq = https.request({
      host: 'angel.co',
      path: '/api/oauth/token?client_id=' + angelListClientId + '&client_secret='\
 + angelListClientSecret + '&code=' + req.query.code + '&grant_type=authorization\
_code',
      port: 443,
      method: 'POST',
      headers: {
        'content-length': 0
      }
    },
    function(angelRes) {
```

```
29
30        angelRes.on('data', function(buffer) {
31          buf += buffer;
32        });
33        angelRes.on('end', function() {
34          try {
35            data = JSON.parse(buf.toString('utf-8'));
36          } catch (e) {
37            if (e) return res.send(e);
38          }
39          if (!data || !data.access_token) return res.send(500);
40          token = data.access_token;
41          req.session.angelListAccessToken = token;
42          if (token) next();
43          else res.send(500);
44        });
45      });
46    angelReq.end();
47    angelReq.on('error', function(e) {
48      console.error(e);
49      next(e);
50    });
51  }
52  exports.angelListLogin = function(req, res, next) {
53    token = req.session.angelListAccessToken;
54    httpsRequest = https.request({
55        host: 'api.angel.co',
56        path: '/1/me?access_token=' + token,
57        port: 443,
58        method: 'GET'
59      },
60      function(httpsResponse) {
61        httpsResponse.on('data', function(buffer) {
62          data = JSON.parse(buffer.toString('utf-8'));
63          if (data) {
64            req.angelProfile = data;
65            next();
66          }
67        });
68      }
69    );
70    httpsRequest.end();
```

```
71    httpsRequest.on('error', function(e) {
72      console.error(e);
73    });
74  };
```

36.5.3 main.js

The `hackhall/routes/main.js` file might be interesting as well.

The `checkAdmin()` function performs authentication for admin privileges. If the session object doesn't carry proper flag, we call Express.js `next()` function with an error object:

```
1  exports.checkAdmin = function(request, response, next) {
2    if (request.session
3      && request.session.auth
4      && request.session.userId
5      && request.session.admin) {
6      console.info('Access ADMIN: ' + request.session.userId);
7      return next();
8    } else {
9      next('User is not an administrator.');
10   }
11 };
```

Similarly, we can check only for the user without checking for admin rights:

```
1  exports.checkUser = function(req, res, next) {
2    if (req.session && req.session.auth && req.session.userId
3      && (req.session.user.approved || req.session.admin)) {
4      console.info('Access USER: ' + req.session.userId);
5      return next();
6    } else {
7      next('User is not logged in.');
8    }
9  };
```

Application is just an unapproved user object and we can also check for that:

```
exports.checkApplicant = function(req, res, next) {
  if (req.session && req.session.auth && req.session.userId
    && (!req.session.user.approved || req.session.admin)) {
    console.info('Access USER: ' + req.session.userId);
    return next();
  } else {
    next('User is not logged in.');
  }
};
```

In the login function, we search for email and password matches in the database. On success, we store the user object in the session and proceed; otherwise the request fails:

```
exports.login = function(req, res, next) {
  req.db.User.findOne({
      email: req.body.email,
      password: req.body.password
    },
    null, {
      safe: true
    },
    function(err, user) {
      if (err) return next(err);
      if (user) {
        req.session.auth = true;
        req.session.userId = user._id.toHexString();
        req.session.user = user;
        if (user.admin) {
          req.session.admin = true;
        }
        console.info('Login USER: ' + req.session.userId);
        res.json(200, {
          msg: 'Authorized'
        });
      } else {
        next(new Error('User is not found.'));
      }
    });
};
```

The logging out process removes any session information:

```
exports.logout = function(req, res) {
  console.info('Logout USER: ' + req.session.userId);
  req.session.destroy(function(error) {
    if (!error) {
      res.send({
        msg: 'Logged out'
      });
    }
  });
};
```

This route is used for both the Profile page as well as by Backbone.js for user authentication:

```
exports.profile = function(req, res, next) {
  req.db.User.findById(req.session.userId, 'firstName lastName'
      + 'displayName headline photoUrl admin'
      + 'approved banned role angelUrl twitterUrl'
      + 'facebookUrl linkedinUrl githubUrl', function(err, obj) {
    if (err) next(err);
    if (!obj) next(new Error('User is not found'));
    req.db.Post.find({
      author: {
        id: obj._id,
        name: obj.displayName
      }
    }, null, {
      sort: {
        'created': -1
      }
    }, function(err, list) {
      if (err) next(err);
      obj.posts.own = list || [];
      req.db.Post.find({
        likes: obj._id
      }, null, {
        sort: {
          'created': -1
        }
```

This logic finds Posts and Comments made by the user:

```
      }, function(err, list) {
        if (err) next(err);
        obj.posts.likes = list || [];
        req.db.Post.find({
          watches: obj._id
        }, null, {
          sort: {
            'created': -1
          }
        }, function(err, list) {
          if (err) next(err);
          obj.posts.watches = list || [];
          req.db.Post.find({
            'comments.author.id': obj._id
          }, null, {
            sort: {
              'created': -1
            }
          }, function(err, list) {
            if (err) next(err);
            obj.posts.comments = [];
            list.forEach(function(value, key, list) {
              obj.posts.comments.push(
                value.comments.filter(
                  function(el, i, arr) {
                    return (el.author.id.toString() == obj._id.toString());
                  }
                )
              );
            });
            res.json(200, obj);
          });
        });
      });
    });
  });
};
```

It's important to allow users to delete their profiles:

```
exports.delProfile = function(req, res, next) {
  console.log('del profile');
  console.log(req.session.userId);
  req.db.User.findByIdAndRemove(req.session.user._id, {},
    function(err, obj) {
      if (err) next(err);
      req.session.destroy(function(error) {
        if (err) {
          next(err)
        }
      });
      res.json(200, obj);
    }
  );
};
```

The full source code of `hackhall/routes/main.js` files:

```
exports.checkAdmin = function(request, response, next) {
  if (request.session && request.session.auth && request.session.userId && reques\
t.session.admin) {
    console.info('Access ADMIN: ' + request.session.userId);
    return next();
  } else {
    next('User is not an administrator.');
  }
};

exports.checkUser = function(req, res, next) {
  if (req.session && req.session.auth && req.session.userId && (req.session.user.\
approved || req.session.admin)) {
    console.info('Access USER: ' + req.session.userId);
    return next();
  } else {
    next('User is not logged in.');
  }
};

exports.checkApplicant = function(req, res, next) {
  if (req.session && req.session.auth && req.session.userId && (!req.session.user\
.approved || req.session.admin)) {
    console.info('Access USER: ' + req.session.userId);
```

```
25      return next();
26    } else {
27      next('User is not logged in.');
28    }
29  };
30
31  exports.login = function(req, res, next) {
32    req.db.User.findOne({
33        email: req.body.email,
34        password: req.body.password
35      },
36      null, {
37        safe: true
38      },
39      function(err, user) {
40        if (err) return next(err);
41        if (user) {
42          req.session.auth = true;
43          req.session.userId = user._id.toHexString();
44          req.session.user = user;
45          if (user.admin) {
46            req.session.admin = true;
47          }
48          console.info('Login USER: ' + req.session.userId);
49          res.json(200, {
50            msg: 'Authorized'
51          });
52        } else {
53          next(new Error('User is not found.'));
54        }
55      });
56  };
57
58  exports.logout = function(req, res) {
59    console.info('Logout USER: ' + req.session.userId);
60    req.session.destroy(function(error) {
61      if (!error) {
62        res.send({
63          msg: 'Logged out'
64        });
65      }
66    });
```

```
67  };
68
69  exports.profile = function(req, res, next) {
70    req.db.User.findById(req.session.userId, 'firstName lastName displayName headli\
71  ne photoUrl admin approved banned role angelUrl twitterUrl facebookUrl linkedinUr\
72  l githubUrl', function(err, obj) {
73      if (err) next(err);
74      if (!obj) next(new Error('User is not found'));
75      req.db.Post.find({
76        author: {
77          id: obj._id,
78          name: obj.displayName
79        }
80      }, null, {
81        sort: {
82          'created': -1
83        }
84      }, function(err, list) {
85        if (err) next(err);
86        obj.posts.own = list || [];
87        req.db.Post.find({
88          likes: obj._id
89        }, null, {
90          sort: {
91            'created': -1
92          }
93        }, function(err, list) {
94          if (err) next(err);
95          obj.posts.likes = list || [];
96          req.db.Post.find({
97            watches: obj._id
98          }, null, {
99            sort: {
100             'created': -1
101           }
102         }, function(err, list) {
103           if (err) next(err);
104           obj.posts.watches = list || [];
105           req.db.Post.find({
106             'comments.author.id': obj._id
107           }, null, {
108             sort: {
```

```
109               'created': -1
110             }
111           }, function(err, list) {
112             if (err) next(err);
113             obj.posts.comments = [];
114             list.forEach(function(value, key, list) {
115               obj.posts.comments.push(value.comments.filter(function(el, i, arr) {
116                 return (el.author.id.toString() == obj._id.toString());
117               }));
118             });
119             res.json(200, obj);
120           });
121         });
122       });
123     });
124   });
125 };
126 
127 exports.delProfile = function(req, res, next) {
128   console.log('del profile');
129   console.log(req.session.userId);
130   req.db.User.findByIdAndRemove(req.session.user._id, {}, function(err, obj) {
131     if (err) next(err);
132     req.session.destroy(function(error) {
133       if (err) {
134         next(err)
135       }
136     });
137     res.json(200, obj);
138   });
139 };
```

36.5.4 users.js

The full source code for `hackhall/routes/users.js` files:

```
objectId = require('mongodb').ObjectID;

exports.getUsers = function(req, res, next) {
  if (req.session.auth && req.session.userId) {
    req.db.User.find({}, 'firstName lastName displayName headline photoUrl admin \
approved banned role angelUrl twitterUrl facebookUrl linkedinUrl githubUrl', func\
tion(err, list) {
      if (err) next(err);
      res.json(200, list);
    });
  } else {
    next('User is not recognized.')
  }
}

exports.getUser = function(req, res, next) {
  req.db.User.findById(req.params.id, 'firstName lastName displayName headline ph\
otoUrl admin approved banned role angelUrl twitterUrl facebookUrl linkedinUrl git\
hubUrl', function(err, obj) {
    if (err) next(err);
    if (!obj) next(new Error('User is not found'));
    req.db.Post.find({
      author: {
        id: obj._id,
        name: obj.displayName
      }
    }, null, {
      sort: {
        'created': -1
      }
    }, function(err, list) {
      if (err) next(err);
      obj.posts.own = list || [];
      req.db.Post.find({
        likes: obj._id
      }, null, {
        sort: {
          'created': -1
        }
      }, function(err, list) {
        if (err) next(err);
        obj.posts.likes = list || [];
```

```
43          req.db.Post.find({
44            watches: obj._id
45          }, null, {
46            sort: {
47              'created': -1
48            }
49          }, function(err, list) {
50            if (err) next(err);
51            obj.posts.watches = list || [];
52            req.db.Post.find({
53              'comments.author.id': obj._id
54            }, null, {
55              sort: {
56                'created': -1
57              }
58            }, function(err, list) {
59              if (err) next(err);
60              obj.posts.comments = [];
61              list.forEach(function(value, key, list) {
62                obj.posts.comments.push(value.comments.filter(function(el, i, arr) {
63                  return (el.author.id.toString() == obj._id.toString());
64                }));
65              });
66              res.json(200, obj);
67            });
68          });
69        });
70      });
71    });
72  };
73
74  exports.add = function(req, res, next) {
75    var user = new req.db.User(req.body);
76    user.save(function(err) {
77      if (err) next(err);
78      res.json(user);
79    });
80  };
81
82  exports.update = function(req, res, next) {
83    var obj = req.body;
84    obj.updated = new Date();
```

```
85    delete obj._id;
86    req.db.User.findByIdAndUpdate(req.params.id, {
87      $set: obj
88    }, {
89      new: true
90    }, function(err, obj) {
91      if (err) next(err);
92      res.json(200, obj);
93    });
94  };
95
96  exports.del = function(req, res, next) {
97    req.db.User.findByIdAndRemove(req.params.id, function(err, obj) {
98      if (err) next(err);
99      res.json(200, obj);
100   });
101 };
102
103 exports.findOrAddUser = function(req, res, next) {
104   data = req.angelProfile;
105   req.db.User.findOne({
106     angelListId: data.id
107   }, function(err, obj) {
108     console.log('angelListLogin4');
109     if (err) next(err);
110     console.warn(obj);
111     if (!obj) {
112       req.db.User.create({
113         angelListId: data.id,
114         angelToken: token,
115         angelListProfile: data,
116         email: data.email,
117         firstName: data.name.split(' ')[0],
118         lastName: data.name.split(' ')[1],
119         displayName: data.name,
120         headline: data.bio,
121         photoUrl: data.image,
122         angelUrl: data.angellist_url,
123         twitterUrl: data.twitter_url,
124         facebookUrl: data.facebook_url,
125         linkedinUrl: data.linkedin_url,
126         githubUrl: data.github_url
```

```
127       }, function(err, obj) { //remember the scope of variables!
128         if (err) next(err);
129         console.log(obj);
130         req.session.auth = true;
131         req.session.userId = obj._id;
132         req.session.user = obj;
133         req.session.admin = false; //assing regular user role by default         \
134
135         res.redirect('/#application');
136         // }
137       });
138     } else { //user is in the database
139       req.session.auth = true;
140       req.session.userId = obj._id;
141       req.session.user = obj;
142       req.session.admin = obj.admin; //false; //assing regular user role by defau\
143 lt
144       if (obj.approved) {
145         res.redirect('/#posts');
146       } else {
147         res.redirect('/#application');
148       }
149     }
150   })
151 }
```

36.5.5 applications.js

In the current version, submitting and approving an application won't trigger an email notification. Therefore, users have to comeback to the website to check their status.

Merely add a user object (with approved=false by default) to the database:

```
1  exports.add = function(req, res, next) {
2    req.db.User.create({
3      firstName: req.body.firstName,
4      lastName: req.body.lastName,
5      displayName: req.body.displayName,
6      headline: req.body.headline,
7      photoUrl: req.body.photoUrl,
8      password: req.body.password,
9      email: req.body.email,
10     angelList: {
```

```
      blah: 'blah'
    },
    angelUrl: req.body.angelUrl,
    twitterUrl: req.body.twitterUrl,
    facebookUrl: req.body.facebookUrl,
    linkedinUrl: req.body.linkedinUrl,
    githubUrl: req.body.githubUrl
  }, function(err, obj) {
    if (err) next(err);
    if (!obj) next('Cannot create.')
    res.json(200, obj);
  })
};
```

Let the users update information in their applications:

```
exports.update = function(req, res, next) {
  var data = {};
  Object.keys(req.body).forEach(function(k) {
    if (req.body[k]) {
      data[k] = req.body[k];
    }
  });
  delete data._id;
  req.db.User.findByIdAndUpdate(req.session.user._id, {
    $set: data
  }, function(err, obj) {
    if (err) next(err);
    if (!obj) next('Cannot save.')
    res.json(200, obj);
  });
};
```

Select a particular object with the `get()` function:

```
exports.get = function(req, res, next) {
  req.db.User.findById(req.session.user._id,
    'firstName lastName photoUrl headline displayName'
      + 'angelUrl facebookUrl twitterUrl linkedinUrl'
      + 'githubUrl', {}, function(err, obj) {
      if (err) next(err);
      if (!obj) next('cannot find');
      res.json(200, obj);
    })
};
```

The full source code of `hackhall/routes/applications.js` files:

<<(hackhall/routes/applications.js)

36.5.6 posts.js

The last routes module that we bisect is `hackhall/routes/posts.js`. It takes care of adding, editing and removing posts, as well as commenting, watching and liking.

We use object ID for conversion from HEX strings to proper objects:

```
objectId = require('mongodb').ObjectID;
```

The coloring is nice for logging but of course optional. We accomplish it with escape sequences:

```
var red, blue, reset;
red   = '\u001b[31m';
blue  = '\u001b[34m';
reset = '\u001b[0m';
console.log(red + 'This is red' + reset + ' while ' + blue + ' this is blue' + re\
set);
```

The default values for the pagination of posts:

```
var LIMIT = 10;
var SKIP = 0;
```

The `add()` function handles creation of new posts:

```
exports.add = function(req, res, next) {
  if (req.body) {
    req.db.Post.create({
      title: req.body.title,
      text: req.body.text || null,
      url: req.body.url || null,
      author: {
        id: req.session.user._id,
        name: req.session.user.displayName
      }
    }, function(err, docs) {
      if (err) {
        console.error(err);
        next(err);
      } else {
        res.json(200, docs);
      }

    });
  } else {
    next(new Error('No data'));
  }
};
```

To retrieve the list of posts:

```
exports.getPosts = function(req, res, next) {
  var limit = req.query.limit || LIMIT;
  var skip = req.query.skip || SKIP;
  req.db.Post.find({}, null, {
    limit: limit,
    skip: skip,
    sort: {
      '_id': -1
    }
  }, function(err, obj) {
    if (!obj) next('There are not posts.');
    obj.forEach(function(item, i, list) {
      if (req.session.user.admin) {
        item.admin = true;
      } else {
        item.admin = false;
```

```
      }
      if (item.author.id == req.session.userId) {
        item.own = true;
      } else {
        item.own = false;
      }
      if (item.likes
        && item.likes.indexOf(req.session.userId) > -1) {
        item.like = true;
      } else {
        item.like = false;
      }
      if (item.watches
        && item.watches.indexOf(req.session.userId) > -1) {
        item.watch = true;
      } else {
        item.watch = false;
      }
    });
    var body = {};
    body.limit = limit;
    body.skip = skip;
    body.posts = obj;
    req.db.Post.count({}, function(err, total) {
      if (err) next(err);
      body.total = total;
      res.json(200, body);
    });
  });
};
```

For the individual post page, we need the `getPost()` method:

```
exports.getPost = function(req, res, next) {
  if (req.params.id) {
    req.db.Post.findById(req.params.id, {
      title: true,
      text: true,
      url: true,
      author: true,
      comments: true,
      watches: true,
```

```
      likes: true
    }, function(err, obj) {
      if (err) next(err);
      if (!obj) {
        next('Nothing is found.');
      } else {
        res.json(200, obj);
      }
    });
  } else {
    next('No post id');
  }
};
```

The `del()` function removes specific posts from the database. The `findById()` and `remove()` methods from Mongoose are used in this snippet. However, the same thing can be accomplished with just `remove()`.

```
exports.del = function(req, res, next) {
  req.db.Post.findById(req.params.id, function(err, obj) {
    if (err) next(err);
    if (req.session.admin || req.session.userId === obj.author.id) {
      obj.remove();
      res.json(200, obj);
    } else {
      next('User is not authorized to delete post.');
    }
  })
};
```

To like the post, we update the post item by prepending `post.likes` array with the ID of the user:

```
function likePost(req, res, next) {
  req.db.Post.findByIdAndUpdate(req.body._id, {
    $push: {
      likes: req.session.userId
    }
  }, {}, function(err, obj) {
    if (err) {
      next(err);
    } else {
      res.json(200, obj);
```

```
    }
  });
};
```

Likewise, when a user performs the watch action, the system adds a new ID to the `post.watches` array:

```
function watchPost(req, res, next) {
  req.db.Post.findByIdAndUpdate(req.body._id, {
    $push: {
      watches: req.session.userId
    }
  }, {}, function(err, obj) {
    if (err) next(err);
    else {
      res.json(200, obj);
    }
  });
};
```

The `updatePost()` is what calls like or watch functions based on the action flag sent with request. In addition, the `updatePost()` processes the changes to the post and comments:

```
exports.updatePost = function(req, res, next) {
  var anyAction = false;
  if (req.body._id && req.params.id) {
    if (req.body && req.body.action == 'like') {
      anyAction = true;
      likePost(req, res);
    }
    if (req.body && req.body.action == 'watch') {
      anyAction = true;
      watchPost(req, res);
    }
    if (req.body && req.body.action == 'comment'
      && req.body.comment && req.params.id) {
      anyAction = true;
      req.db.Post.findByIdAndUpdate(req.params.id, {
        $push: {
          comments: {
            author: {
              id: req.session.userId,
```

```
              name: req.session.user.displayName
            },
            text: req.body.comment
          }
        }
      }, {
        safe: true,
        new: true
      }, function(err, obj) {
        if (err) throw err;
        res.json(200, obj);
      });
    }
    if (req.session.auth && req.session.userId && req.body
      && req.body.action != 'comment' &&
      req.body.action != 'watch' && req.body != 'like' &&
      req.params.id && (req.body.author.id == req.session.user._id
      || req.session.user.admin)) {
      req.db.Post.findById(req.params.id, function(err, doc) {
        if (err) next(err);
        doc.title = req.body.title;
        doc.text = req.body.text || null;
        doc.url = req.body.url || null;
        doc.save(function(e, d) {
          if (e) next(e);
          res.json(200, d);
        });
      })
    } else {
      if (!anyAction) next('Something went wrong.');
    }

  } else {
    next('No post ID.');
  }
};
```

The full source code for the `hackhall/routes/posts.js` file:

```
objectId = require('mongodb').ObjectID;
var red, blue, reset;
red = '\u001b[31m';
blue = '\u001b[34m';
reset = '\u001b[0m';
console.log(red + 'This is red' + reset + ' while ' + blue + ' this is blue' + re\
set);

var LIMIT = 10;
var SKIP = 0;

exports.add = function(req, res, next) {
  if (req.body) {
    req.db.Post.create({
      title: req.body.title,
      text: req.body.text || null,
      url: req.body.url || null,
      author: {
        id: req.session.user._id,
        name: req.session.user.displayName
      }
    }, function(err, docs) {
      if (err) {
        console.error(err);
        next(err);
      } else {
        res.json(200, docs);
      }

    });
  } else {
    next(new Error('No data'));
  }
};

exports.getPosts = function(req, res, next) {
  var limit = req.query.limit || LIMIT;
  var skip = req.query.skip || SKIP;
  req.db.Post.find({}, null, {
    limit: limit,
    skip: skip,
    sort: {
```

```
      '_id': -1
    }
  }, function(err, obj) {
    if (!obj) next('There are not posts.');
    obj.forEach(function(item, i, list) {
      if (req.session.user.admin) {
        item.admin = true;
      } else {
        item.admin = false;
      }
      if (item.author.id == req.session.userId) {
        item.own = true;
      } else {
        item.own = false;
      }
      if (item.likes && item.likes.indexOf(req.session.userId) > -1) {
        item.like = true;
      } else {
        item.like = false;
      }
      if (item.watches && item.watches.indexOf(req.session.userId) > -1) {
        item.watch = true;
      } else {
        item.watch = false;
      }
    });
    var body = {};
    body.limit = limit;
    body.skip = skip;
    body.posts = obj;
    req.db.Post.count({}, function(err, total) {
      if (err) next(err);
      body.total = total;
      res.json(200, body);
    });
  });
};


exports.getPost = function(req, res, next) {
  if (req.params.id) {
    req.db.Post.findById(req.params.id, {
```

```
      title: true,
      text: true,
      url: true,
      author: true,
      comments: true,
      watches: true,
      likes: true
    }, function(err, obj) {
      if (err) next(err);
      if (!obj) {
        next('Nothing is found.');
      } else {
        res.json(200, obj);
      }
    });
  } else {
    next('No post id');
  }
};

exports.del = function(req, res, next) {
  req.db.Post.findById(req.params.id, function(err, obj) {
    if (err) next(err);
    if (req.session.admin || req.session.userId === obj.author.id) {
      obj.remove();
      res.json(200, obj);
    } else {
      next('User is not authorized to delete post.');
    }
  })
};

function likePost(req, res, next) {
  req.db.Post.findByIdAndUpdate(req.body._id, {
    $push: {
      likes: req.session.userId
    }
  }, {}, function(err, obj) {
    if (err) {
      next(err);
    } else {
      res.json(200, obj);
```

```
    }
  });
};

function watchPost(req, res, next) {
  req.db.Post.findByIdAndUpdate(req.body._id, {
    $push: {
      watches: req.session.userId
    }
  }, {}, function(err, obj) {
    if (err) next(err);
    else {
      res.json(200, obj);
    }
  });
};

exports.updatePost = function(req, res, next) {
  var anyAction = false;
  if (req.body._id && req.params.id) {
    if (req.body && req.body.action == 'like') {
      anyAction = true;
      likePost(req, res);
    }
    if (req.body && req.body.action == 'watch') {
      anyAction = true;
      watchPost(req, res);
    }
    if (req.body && req.body.action == 'comment' && req.body.comment && req.param\
s.id) {
      anyAction = true;
      req.db.Post.findByIdAndUpdate(req.params.id, {
        $push: {
          comments: {
            author: {
              id: req.session.userId,
              name: req.session.user.displayName
            },
            text: req.body.comment
          }
        }
      }, {
```

```
169         safe: true,
170         new: true
171       }, function(err, obj) {
172         if (err) throw err;
173         res.json(200, obj);
174       });
175     }
176     if (req.session.auth && req.session.userId && req.body && req.body.action != \
177 'comment' &&
178       req.body.action != 'watch' && req.body != 'like' &&
179       req.params.id && (req.body.author.id == req.session.user._id || req.session\
180 .user.admin)) {
181       req.db.Post.findById(req.params.id, function(err, doc) {
182         if (err) next(err);
183         doc.title = req.body.title;
184         doc.text = req.body.text || null;
185         doc.url = req.body.url || null;
186         doc.save(function(e, d) {
187           if (e) next(e);
188           res.json(200, d);
189         });
190       })
191     } else {
192       if (!anyAction) next('Something went wrong.');
193     }
194
195   } else {
196     next('No post ID.');
197   }
198 };
```

36.6 Mogoose Models

Ideally, in a big application, we would break down each model into a separate file. Right now in the HackHall app, we have them all in `hackhall/models/index.js`.

As always, our dependencies look better at the top:

```
1 var mongoose = require('mongoose');
2 var Schema = mongoose.Schema;
3 var roles = 'user staff mentor investor founder'.split(' ');
```

The Post model represents post with its likes, comments and watches.

```
exports.Post = new Schema ({
  title: {
    required: true,
    type: String,
    trim: true,
    // match: /^([[:alpha:][:space:][:punct:]]{1,100})$/
    match: /^([\w ,.!?]{1,100})$/
  },
  url: {
    type: String,
    trim: true,
    max: 1000
  },
  text: {
    type: String,
    trim: true,
    max: 2000
  },
  comments: [{
    text: {
      type: String,
      trim: true,
      max:2000
    },
    author: {
      id: {
        type: Schema.Types.ObjectId,
        ref: 'User'
      },
      name: String
    }
  }],
  watches: [{
    type: Schema.Types.ObjectId,
    ref: 'User'
  }],
  likes: [{
    type: Schema.Types.ObjectId,
    ref: 'User'
  }],
  author: {
    id: {
```

```
      type: Schema.Types.ObjectId,
      ref: 'User',
      required: true
    },
    name: {
      type: String,
      required: true
    }
  },
  created: {
    type: Date,
    default: Date.now,
    required: true
  },
  updated:  {
    type: Date,
    default: Date.now,
    required: true
  },
  own: Boolean,
  like: Boolean,
  watch: Boolean,
  admin: Boolean,
  action: String
});
```

The User model can also serve as an application object (when `approved=false`):

```
exports.User = new Schema({
  angelListId: String,
  angelListProfile: Schema.Types.Mixed,
  angelToken: String,
    firstName: {
    type: String,
    required: true,
    trim: true
  },
  lastName: {
    type: String,
    required: true,
    trim: true
  },
```

```
15    displayName: {
16      type: String,
17      required: true,
18      trim: true
19    },
20      password: String,
21      email: {
22      type: String,
23      required: true,
24      trim: true
25    },
26    role: {
27      type:String,
28      enum: roles,
29      required: true,
30      default: roles[0]
31    },
32    approved: {
33      type: Boolean,
34      default: false
35    },
36    banned: {
37      type: Boolean,
38      default: false
39    },
40    admin: {
41      type: Boolean,
42      default: false
43    },
44    headline: String,
45    photoUrl: String,
46    angelList: Schema.Types.Mixed,
47    created: {
48      type: Date,
49      default: Date.now
50    },
51    updated:  {
52      type: Date, default: Date.now
53    },
54    angelUrl: String,
55    twitterUrl: String,
56    facebookUrl: String,
```

```
  linkedinUrl: String,
  githubUrl: String,
  own: Boolean,
  posts: {
    own: [Schema.Types.Mixed],
    likes: [Schema.Types.Mixed],
    watches: [Schema.Types.Mixed],
    comments: [Schema.Types.Mixed]
  }
});
```

The full source code for `hackhall/models/index.js`:

```
var mongoose = require('mongoose');
var Schema = mongoose.Schema;
var roles = 'user staff mentor investor founder'.split(' ');

exports.Post = new Schema ({
  title: {
    required: true,
    type: String,
    trim: true,
    // match: /^([[:alpha:][:space:][:punct:]]{1,100})$/
    match: /^([\w ,.!?]{1,100})$/
  },
  url: {
    type: String,
    trim: true,
    max: 1000
  },
  text: {
    type: String,
    trim: true,
    max: 2000
  },
  comments: [{
    text: {
      type: String,
      trim: true,
      max:2000
    },
    author: {
```

```
      id: {
        type: Schema.Types.ObjectId,
        ref: 'User'
      },
      name: String
    }
  }],
  watches: [{
    type: Schema.Types.ObjectId,
    ref: 'User'
  }],
  likes: [{
    type: Schema.Types.ObjectId,
    ref: 'User'
  }],
  author: {
    id: {
      type: Schema.Types.ObjectId,
      ref: 'User',
      required: true
    },
    name: {
      type: String,
      required: true
    }
  },
  created: {
    type: Date,
    default: Date.now,
    required: true
  },
  updated:  {
    type: Date,
    default: Date.now,
    required: true
  },
  own: Boolean,
  like: Boolean,
  watch: Boolean,
  admin: Boolean,
  action: String
});
```

```
72
73  exports.User = new Schema({
74    angelListId: String,
75    angelListProfile: Schema.Types.Mixed,
76    angelToken: String,
77      firstName: {
78      type: String,
79      required: true,
80      trim: true
81    },
82    lastName: {
83      type: String,
84      required: true,
85      trim: true
86    },
87    displayName: {
88      type: String,
89      required: true,
90      trim: true
91    },
92      password: String,
93      email: {
94      type: String,
95      required: true,
96      trim: true
97    },
98    role: {
99      type:String,
100     enum: roles,
101     required: true,
102     default: roles[0]
103   },
104   approved: {
105     type: Boolean,
106     default: false
107   },
108   banned: {
109     type: Boolean,
110     default: false
111   },
112   admin: {
113     type: Boolean,
```

```
    default: false
  },
  headline: String,
  photoUrl: String,
  angelList: Schema.Types.Mixed,
  created: {
    type: Date,
    default: Date.now
  },
  updated:  {
    type: Date, default: Date.now
  },
  angelUrl: String,
  twitterUrl: String,
  facebookUrl: String,
  linkedinUrl: String,
  githubUrl: String,
  own: Boolean,
  posts: {
    own: [Schema.Types.Mixed],
    likes: [Schema.Types.Mixed],
    watches: [Schema.Types.Mixed],
    comments: [Schema.Types.Mixed]
  }
});
```

36.7 Mocha Tests

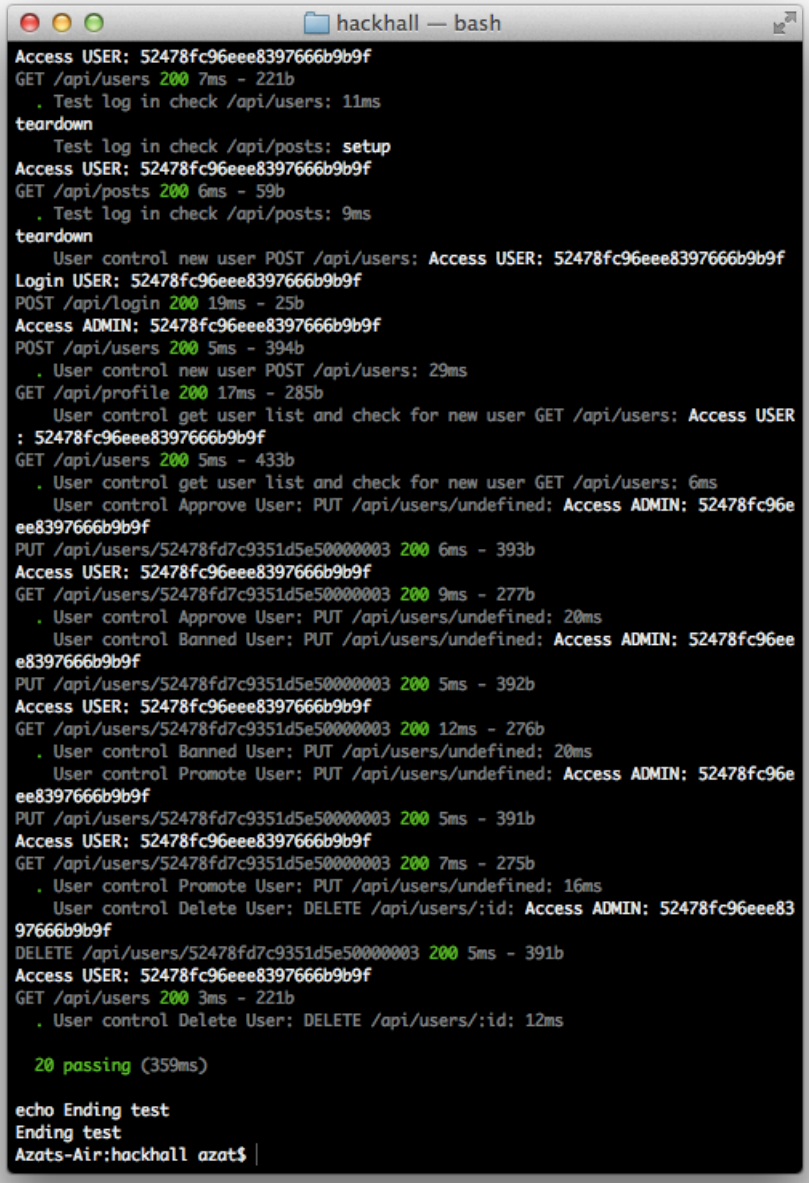
One of the benefits of using REST API server architecture is that each route and the application as a whole become very testable. The assurance of the passed tests is a wonderful supplement during development â€” the so called test-driven development approach.

To run tests, we utilize Makefile:

```
REPORTER = list
MOCHA_OPTS = --ui tdd --ignore-leaks

test:
        clear
        echo Starting test ************************************************************
        ./node_modules/mocha/bin/mocha \
        --reporter $(REPORTER) \
        $(MOCHA_OPTS) \
        tests/*.js
        echo Ending test

test-w:
        ./node_modules/mocha/bin/mocha \
        --reporter $(REPORTER) \
        --growl \
        --watch \
        $(MOCHA_OPTS) \
        tests/*.js

users:
        mocha tests/users.js --ui tdd --reporter list --ignore-leaks

posts:
        clear
        echo Starting test ************************************************************
        ./node_modules/mocha/bin/mocha \
        --reporter $(REPORTER) \
        $(MOCHA_OPTS) \
        tests/posts.js
        echo Ending test

application:
        mocha tests/application.js --ui tdd --reporter list --ignore-leaks

.PHONY: test test-w posts application
```

Therefore, we can start tests with: `$ make` command.

All the 20 tests should pass:



The results of running Mocha tests.

The HackHall tests live in `tests` folder and consist of:

- `hackhall/tests/application.js`: functional tests for unapproved users information
- `hackhall/tests/posts.js`: functional tests for posts
- `hackhall/tests/users.js`: functional tests for users

Test use a library called superagent[9](GitHub[10]).

The full content for `hackhall/tests/application.js`:

```
var app = require ('../server').app,
  assert = require('assert'),
  request = require('superagent');

app.listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});

var user1 = request.agent();
var port = 'http://localhost:'+app.get('port');
var userId;

suite('APPLICATION API', function (){
  suiteSetup(function(done){
    done();
  });
  test('log in as admin', function(done){
    user1.post(port+'/api/login').send({email:'1@1.com',password:'1'}).end(functi\
on(res){
        assert.equal(res.status,200);
      done();
    });
  });
  test('get profile for admin',function(done){
    user1.get(port+'/api/profile').end(function(res){
        assert.equal(res.status,200);
      done();
    });
  });
  test('submit applicaton for user 3@3.com', function(done){
    user1.post(port+'/api/application').send({
        firstName: 'Dummy',
      lastName: 'Application',
      displayName: 'Dummy Application',
        password: '3',
        email: '3@3.com',
      headline: 'Dummy Appliation',
```

[9] https://npmjs.org/package/superagent

[10] https://github.com/visionmedia/superagent

```
38        photoUrl: '/img/user.png',
39        angelList: {blah:'blah'},
40        angelUrl: 'http://angel.co.com/someuser',
41        twitterUrl: 'http://twitter.com/someuser',
42        facebookUrl: 'http://facebook.com/someuser',
43        linkedinUrl: 'http://linkedin.com/someuser',
44        githubUrl: 'http://github.com/someuser'
45      }).end(function(res){
46        assert.equal(res.status,200);
47        userId = res.body._id;
48        done();
49      });
50
51    });
52    test('logout admin',function(done){
53      user1.post(port+'/api/logout').end(function(res){
54          assert.equal(res.status,200);
55        done();
56      });
57    });
58    test('get profile again after logging out',function(done){
59      user1.get(port+'/api/profile').end(function(res){
60          assert.equal(res.status,500);
61        done();
62      });
63    });
64    test('log in as user3 - unapproved', function(done){
65      user1.post(port+'/api/login').send({email:'3@3.com',password:'3'}).end(functi\
66  on(res){
67          assert.equal(res.status,200);
68        done();
69      });
70    });
71    test('get user application', function(done){
72      user1.get(port+'/api/application/').end(function(res){
73        // console.log(res.body)
74        assert.equal(res.status, 200);
75        done();
76      });
77    });
78    test('update user application', function(done){
79      user1.put(port+'/api/application/').send({
```

```
80        firstName: 'boo'}).end(function(res){
81        // console.log(res.body)
82        assert.equal(res.status, 200);
83        done();
84      });
85    });
86    test('get user application', function(done){
87      user1.get(port+'/api/application/').end(function(res){
88        // console.log(res.body)
89        assert.equal(res.status, 200);
90        done();
91      });
92    });
93    test('check for posts - fail (unapproved?)', function(done){
94      user1.get(port+'/api/posts/').end(function(res){
95        // console.log(res.body)
96        assert.equal(res.status, 500);
97
98        done();
99      });
100   });
101   test('logout user',function(done){
102     user1.post(port+'/api/logout').end(function(res){
103         assert.equal(res.status,200);
104       done();
105     });
106   });
107   test('log in as admin', function(done){
108     user1.post(port+'/api/login').send({email:'1@1.com',password:'1'}).end(functi\
109 on(res){
110         assert.equal(res.status,200);
111       done();
112     });
113   });
114   test('delete user3', function(done){
115     user1.del(port+'/api/users/'+userId).end(function(res){
116       assert.equal(res.status, 200);
117       done();
118     });
119   });
120   test('logout admin',function(done){
121     user1.post(port+'/api/logout').end(function(res){
```

```
122         assert.equal(res.status,200);
123       done();
124     });
125   });
126   test('log in as user - should fail', function(done){
127     user1.post(port+'/api/login').send({email:'3@3.com',password:'3'}).end(functi\
128 on(res){
129       // console.log(res.body)
130         assert.equal(res.status,500);
131 
132       done();
133     });
134   });
135   test('check for posts - must fail', function(done){
136     user1.get(port+'/api/posts/').end(function(res){
137       // console.log(res.body)
138       assert.equal(res.status, 500);
139 
140       done();
141     });
142   });
143   suiteTeardown(function(done){
144     done();
145   });
146 
147 });
```

The full content for `hackhall/tests/posts.js`:

```
1  var app = require('../server').app,
2    assert = require('assert'),
3    request = require('superagent');
4 
5  app.listen(app.get('port'), function() {
6    console.log('Express server listening on port ' + app.get('port'));
7  });
8 
9  var user1 = request.agent();
10 var port = 'http://localhost:' + app.get('port');
11 var postId;
12 
13 suite('POSTS API', function() {
```

```
14    suiteSetup(function(done) {
15      done();
16    });
17    test('log in', function(done) {
18      user1.post(port + '/api/login').send({
19        email: '1@1.com',
20        password: '1'
21      }).end(function(res) {
22        assert.equal(res.status, 200);
23        done();
24      });
25    });
26    test('add post', function(done) {
27      user1.post(port + '/api/posts').send({
28        title: 'Yo Test Title',
29        text: 'Yo Test text',
30        url: ''
31      }).end(function(res) {
32        assert.equal(res.status, 200);
33        postId = res.body._id;
34        done();
35      });
36
37    });
38    test('get profile', function(done) {
39      user1.get(port + '/api/profile').end(function(res) {
40        assert.equal(res.status, 200);
41        done();
42      });
43    });
44    test('post get', function(done) {
45      // console.log('000'+postId);
46      user1.get(port + '/api/posts/' + postId).end(function(res) {
47        assert.equal(res.status, 200);
48        assert.equal(res.body._id, postId);
49        done();
50      });
51    });
52    test('delete post', function(done) {
53      user1.del(port + '/api/posts/' + postId).end(function(res) {
54        assert.equal(res.status, 200);
55        done();
```

```
    });
  });
  test('check for deleted post', function(done) {
    user1.get(port + '/api/posts/' + postId).end(function(res) {
      // console.log(res.body)
      assert.equal(res.status, 500);

      done();
    });
  });
  suiteTeardown(function(done) {
    done();
  });

});
```

The full content for `hackhall/tests/users.js`:

```
var app = require('../server').app,
  assert = require('assert'),
  request = require('superagent');
// http = require('support/http');

var user1 = request.agent();
var port = 'http://localhost:' + app.get('port');


app.listen(app.get('port'), function() {
  console.log('Express server listening on port ' + app.get('port'));
});

suite('Test root', function() {
  setup(function(done) {
    console.log('setup');

    done();
  });

  test('check /', function(done) {
    request.get('http://localhost:3000').end(function(res) {
      assert.equal(res.status, 200);
      done();
```

```
25      });
26    });
27    test('check /api/profile', function(done) {
28      request.get('http://localhost:' + app.get('port') + '/api/profile').end(funct\
29  ion(res) {
30        assert.equal(res.status, 500);
31        done();
32      });
33    });
34    test('check /api/users', function(done) {
35      user1.get('http://localhost:' + app.get('port') + '/api/users').end(function(\
36  res) {
37        assert.equal(res.status, 500);
38        // console.log(res.text.length);
39        done();
40      });
41      // done();
42    });
43    test('check /api/posts', function(done) {
44      user1.get('http://localhost:' + app.get('port') + '/api/posts').end(function(\
45  res) {
46        assert.equal(res.status, 500);
47        // console.log(res.text.length);
48        done();
49      });
50      // done();
51    });
52    teardown(function(done) {
53      console.log('teardown');
54      done();
55    });
56
57  });
58
59  suite('Test log in', function() {
60    setup(function(done) {
61      console.log('setup');
62
63      done();
64    });
65    test('login', function(done) {
66      user1.post('http://localhost:3000/api/login').send({
```

```
67        email: '1@1.com',
68        password: '1'
69      }).end(function(res) {
70        assert.equal(res.status, 200);
71        done();
72      });
73
74    });
75    test('check /api/profile', function(done) {
76      user1.get('http://localhost:' + app.get('port') + '/api/profile').end(functio\
77  n(res) {
78        assert.equal(res.status, 200);
79        // console.log(res.text.length);
80        done();
81      });
82      // done();
83    });
84    test('check /api/users', function(done) {
85      user1.get('http://localhost:' + app.get('port') + '/api/users').end(function(\
86  res) {
87        assert.equal(res.status, 200);
88        // console.log(res.text);
89        done();
90      });
91      // done();
92    });
93    test('check /api/posts', function(done) {
94      user1.get('http://localhost:' + app.get('port') + '/api/posts').end(function(\
95  res) {
96        assert.equal(res.status, 200);
97        // console.log(res.text.length);
98        done();
99      });
100     // done();
101   });
102
103   teardown(function(done) {
104     console.log('teardown');
105     done();
106   });
107
108 });
```

```
109 suite('User control', function() {
110   var user2 = {
111     firstName: 'Bob',
112     lastName: 'Dilan',
113     displayName: 'Bob Dilan',
114     email: '2@2.com'
115   };
116   suiteSetup(function(done) {
117     user1.post('http://localhost:3000/api/login').send({
118       email: '1@1.com',
119       password: '1'
120     }).end(function(res) {
121       assert.equal(res.status, 200);
122       // done();
123     });
124     user1.get('http://localhost:' + app.get('port') + '/api/profile').end(functio\
125 n(res) {
126       assert.equal(res.status, 200);
127       // console.log(res.text.length);
128       // done();
129     });
130 
131     done();
132   })
133 
134   test('new user POST /api/users', function(done) {
135     user1.post(port + '/api/users')
136       .send(user2)
137       .end(function(res) {
138         assert.equal(res.status, 200);
139         // console.log(res.text.length);
140         user2 = res.body;
141         // console.log(user2)
142         done();
143       })
144   });
145   test('get user list and check for new user GET /api/users', function(done) {
146     user1.get('http://localhost:' + app.get('port') + '/api/users').end(function(\
147 res) {
148       assert.equal(res.status, 200);
149       // console.log(res.body)
150       var user3 = res.body.filter(function(el, i, list) {
```

```
151         return (el._id == user2._id);
152       });
153       assert(user3.length === 1);
154       // assert(res.body.indexOf(user2)>-1);
155       // console.log(res.body.length)
156       done();
157     })
158   });
159   test('Approve User: PUT /api/users/' + user2._id, function(done) {
160     assert(user2._id != '');
161     user1.put(port + '/api/users/' + user2._id)
162       .send({
163         approved: true
164       })
165       .end(function(res) {
166         assert.equal(res.status, 200);
167         // console.log(res.text.length);
168         assert(res.body.approved);
169         user1.get(port + '/api/users/' + user2._id).end(function(res) {
170           assert(res.status, 200);
171           assert(res.body.approved);
172           done();
173         })
174 
175       })
176   });
177   test('Banned User: PUT /api/users/' + user2._id, function(done) {
178     assert(user2._id != '');
179     user1.put(port + '/api/users/' + user2._id)
180       .send({
181         banned: true
182       })
183       .end(function(res) {
184         assert.equal(res.status, 200);
185         // console.log(res.text.length);
186         assert(res.body.banned);
187         user1.get(port + '/api/users/' + user2._id).end(function(res) {
188           assert(res.status, 200);
189           assert(res.body.banned);
190           done();
191         })
192 
```

```
193       })
194   });
195   test('Promote User: PUT /api/users/' + user2._id, function(done) {
196     assert(user2._id != '');
197     user1.put(port + '/api/users/' + user2._id)
198       .send({
199         admin: true
200       })
201       .end(function(res) {
202         assert.equal(res.status, 200);
203         // console.log(res.text.length);
204         assert(res.body.admin);
205         user1.get(port + '/api/users/' + user2._id).end(function(res) {
206           assert(res.status, 200);
207           assert(res.body.admin);
208           done();
209         })
210
211       })
212   });
213   test('Delete User: DELETE /api/users/:id', function(done) {
214     assert(user2._id != '');
215     user1.del(port + '/api/users/' + user2._id)
216       .end(function(res) {
217         assert.equal(res.status, 200);
218         // console.log('id:' + user2._id)
219         user1.get(port + '/api/users').end(function(res) {
220           assert.equal(res.status, 200);
221           var user3 = res.body.filter(function(el, i, list) {
222             return (el._id === user2._id);
223           });
224           // console.log('***');
225           // console.warn(user3);
226           assert(user3.length === 0);
227           done();
228         });
229       });
230
231
232   });
233 });
234 // app.close();
```

```
235 // console.log(app)
```

Warning

Please don't store plain passwords/keys in the database. Any serious production app should at least salt the passwords[11] before storing them.

36.8 Conclusion

HackHall is still in development, but there are important real production applications components, such as REST API architecture, OAuth, Mongoose and its models, MVC structure of Express.js apps, access to environment variables, etc.

[11] https://crackstation.net/hashing-security.htm

ExpressWorks

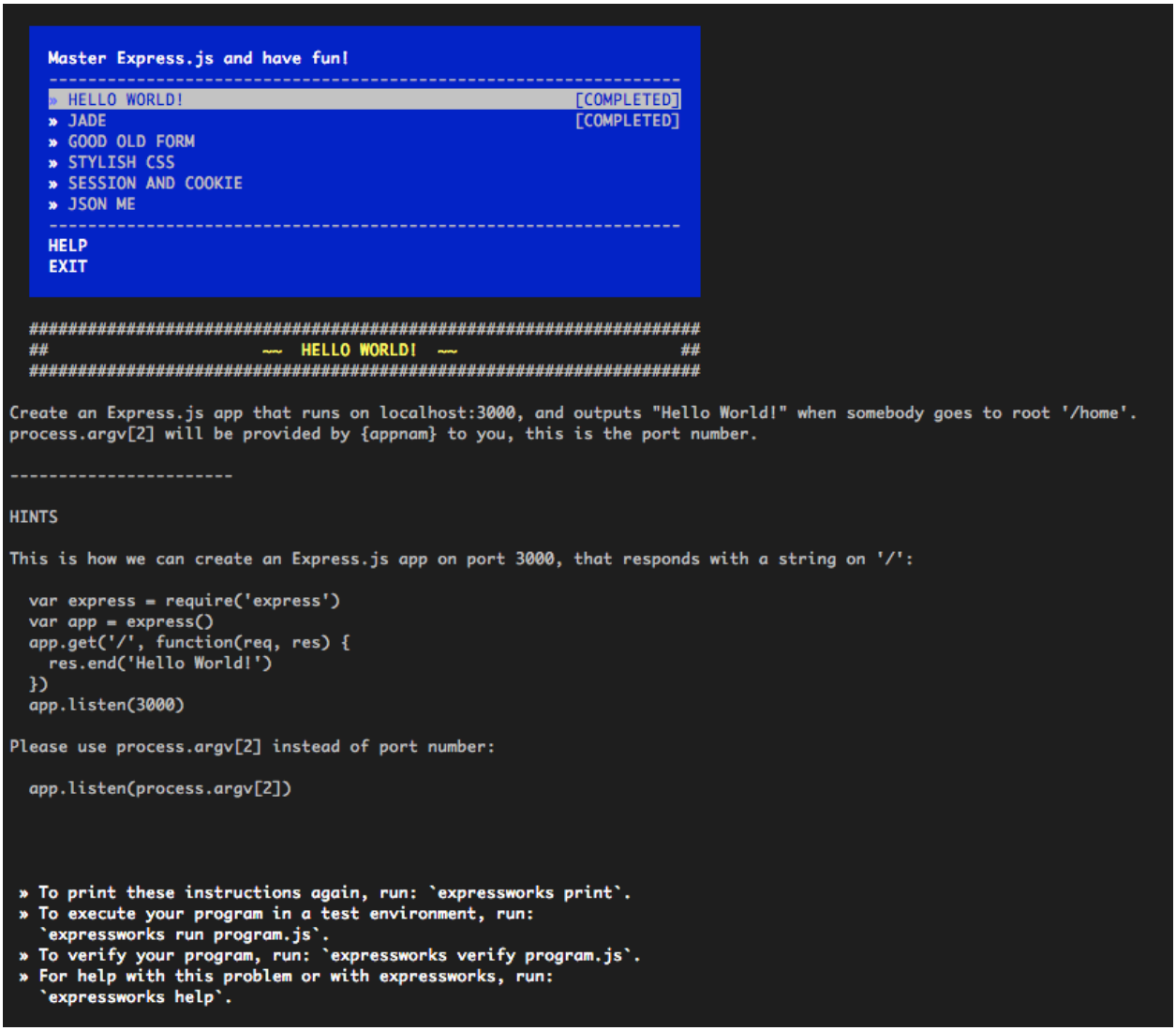
> ExpressWorks is an automated workshop that will walk you through building of Express.js servers, processing of GET, POST and PUT requests, extracting of query string, payload and URL parameters.

What is ExpressWorks?

ExpressWorks is the Express.js workshop based on workshopper[12] and inspired by stream-adventure[13] by @substack[14] and @maxogden[15].

---

[12] https://github.com/rvagg/workshopper
[13] https://github.com/substack/stream-adventure
[14] https://twitter.com/substack
[15] https://twitter.com/maxogden



Hello World Express.js app

Installation (recommended)

Recommended global installation:

```
$ npm install -g expressworks
$ expressworks
```

If you see errors, try:

```
$ sudo npm install -g expressworks
```

Local Installation (advanced)

Run&install locally:

```
$ npm install expressworks
$ cd expressworks
$ npm install
$ node expressworks
```

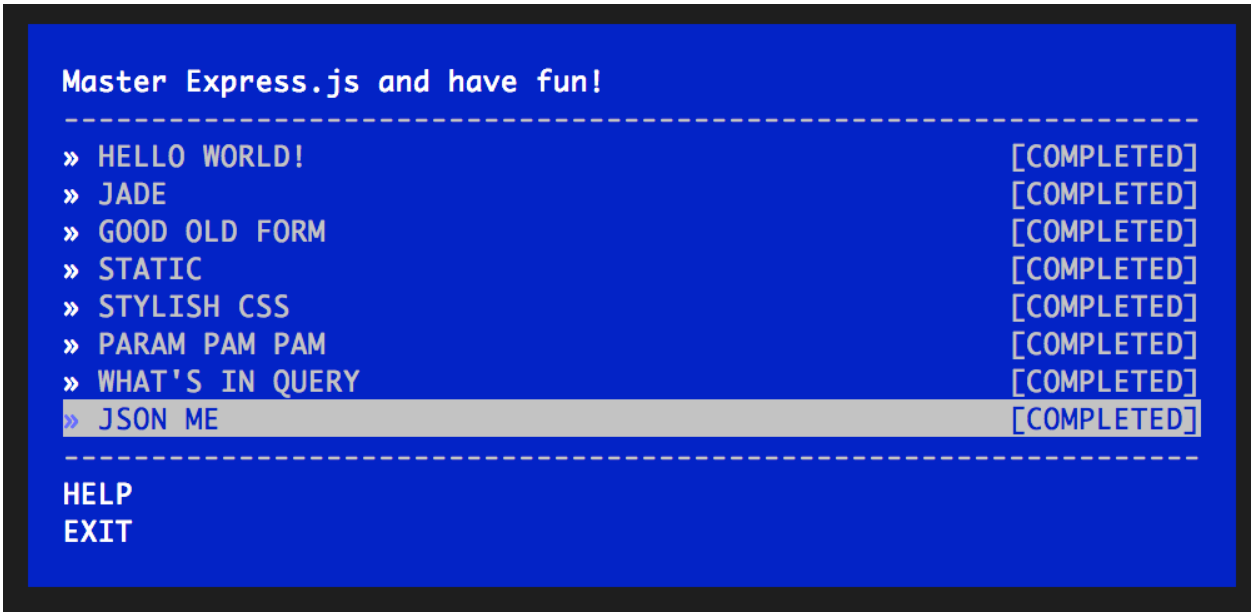
Usage

ExpressWorks understands these commands:

```
Usage

  expressworks
    Show a menu to interactively select a workshop.
  expressworks list
    Show a newline-separated list of all the workshops.
  expressworks select NAME
    Select a workshop.
  expressworks current
    Show the currently selected workshop.
  expressworks run program.js
    Run your program against the selected input.
  expressworks verify program.js
    Verify your program against the expected output.
```

Reset

If you want to reset the list of completed tasks, clean the `~/.config/expressworks/completed.json` file.



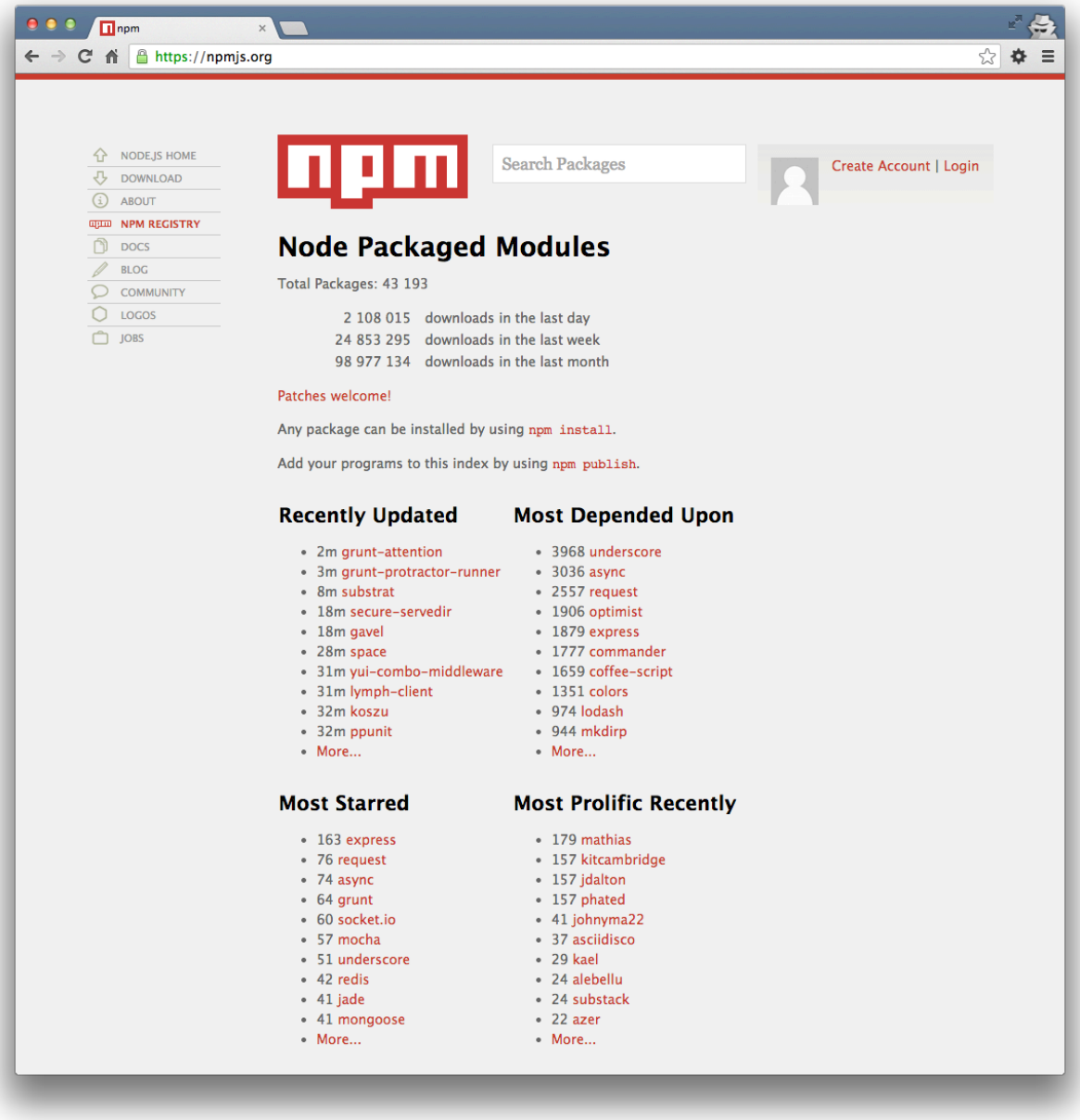
Hello World Express.js app

Related Reading and Resources

Other Node.js Frameworks

The Express.js framework is no doubt the most mature, popular, robust, tested and used project for Node.js web services. As of this writing, Express.js is also the most starred NPM repository with 2x more stars that request and async libraries. There are plenty of real production apps that rely on Express 2.x and 3.x including Storify[16] (acquired by LiveFyre[17]), DocuSign[18], new MySpace[19], LearnBoost[20], Geekli.st[21], Klout[22], Prismatic[23], Segment.io[24] and more[25].

[16] http://storify.com
[17] http://livefyre.com
[18] http://docusign.com
[19] http://new.myspace.com
[20] https://www.learnboost.com/
[21] http://geekli.st/
[22] http://klout.com/
[23] http://getprismatic.com/
[24] http://segment.io/
[25] http://expressjs.com/applications.html



Express.js is also the most starred NPM repository.

Nevertheless, there are plenty of alternatives for which we created our analog of todomvc.com[26] collection: nodeframework.com[27]. By the way, some of the more comprehensive frameworks depend on Express.js (i.e., SailsJS[28]). So it's great that our readers know Express.js by now!

[26] http://todomvc.com

[27] http://nodeframework.com

[28] https://github.com/balderdashy/sails/blob/v0.9.0/package.json#L32

Node.js Books

For more core Node.js overview and/or other components of the Node.js stack, such as databases and websockets, please consider these resources:

- Pro Node.js[29] (Colin J. Ihrig) — ISBN: 978-1-4302-5860-5, comprehensive low-level book on Node.js sans any non-core modules
- Node.js in Action[30] (TJ Holowaychuk et al) — ISBN: 9781617290572 not published yet (est. end of 2013), the book about Express.js from the creators of the framework
- Learning Node[31] (Shelley Powers) — ISBN: 978-1-4493-2307-3 book covers Express, MongoDB, Mongoose and Socket.IO
- Node Cookbook[32] (David Mark Clements) — ISBN: 978-1-84951-718-8 book covers databases and websockets
- Node: Up and Running[33] (Tom Hughes-Croucher et al) brief overview of Node.js
- Smashing Node.js[34] (Guillermo Rauch) covers Express.js, Jade, Stylus from the creator of Mongoose ORM for MongoDB

JavaScript Classics

For deeper understanding of the most misunderstood and the most popular programming language, please make sure to read these proven classics:

- Eloquent JavaScript[35]: programming fundamentals in the JavaScript coating
- JavaScript: The Good Parts[36]: tricky part of the language

[29] http://www.apress.com/9781430258605
[30] http://www.manning.com/cantelon/
[31] http://shop.oreilly.com/product/0636920024606.do
[32] http://my.safaribooksonline.com/9781849517188
[33] http://shop.oreilly.com/product/0636920015956
[34] http://www.amazon.com/Smashing-Node-js-JavaScript-Everywhere-Magazine/dp/1119962595
[35] http://eloquentjavascript.net/
[36] http://www.amazon.com/dp/0596517742