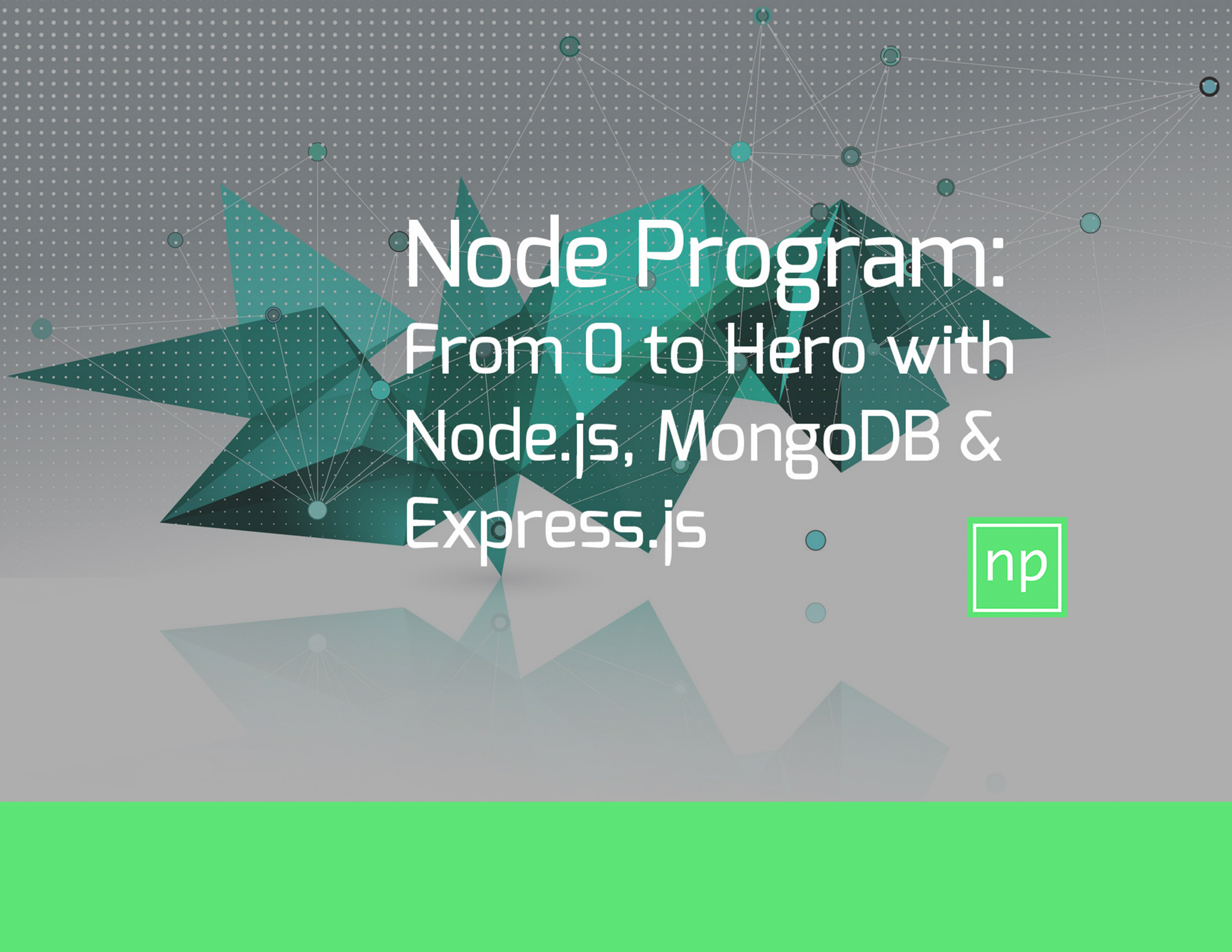

Node Program:
From 0 to Hero with
Node.js, MongoDB &
Express.js
np

Node Program Manual

From 0 to Hero with Node.js, MongoDB and Express.js

Azat Mardan

Contents

Getting Started

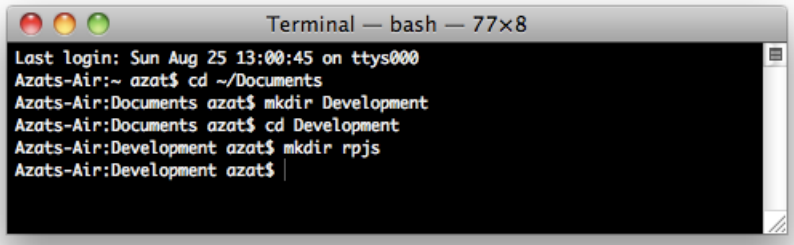
This document consists of the following sections that will help you to set up the development environment:

- Development folder
- Browsers
- IDEs and Text Editors
- Version Control Systems
- Local HTTP Servers
- Database: MongoDB
- Node.js Installation
- JS Libraries
- LESS App
- SSH Keys
- GitHub
- Windows Azure
- Heroku
- Cloud9

Development Folder

If you don't have a specific development folder for your web development projects, you could create a *Development* folder in the *Documents* folder (path will be *Documents/Development*). To work on the code example, create a *rpjs* folder inside your web development projects folder, e.g., if you create a *rpjs* folder inside of the *Development* folder, the path will be *Documents/Development/rpjs*. You could use the Finder on Mac OS X or the following terminal commands on OS X/Linux systems:

```
$ cd ~/Documents
$ mkdir Development
$ cd Development
$ mkdir rpjs
```



Initial development environment setup.

Tip

To open Mac OS Finder app in the current directory from Terminal, just type and run the `$ open .` command.

To get the list of files and folders, use this UNIX/Linix command:

```
1  $ ls
```

or to display hidden files and folders, like *.git*:

```
1  $ ls -lah
```

Another alternative to `$ ls` is `$ ls -alt`. The difference between the `-lah` and the `-alt` options is that the latter sorts items chronologically and the former alphabetically.

Note

You can use the Tab key to autocomplete names of the files and folders.

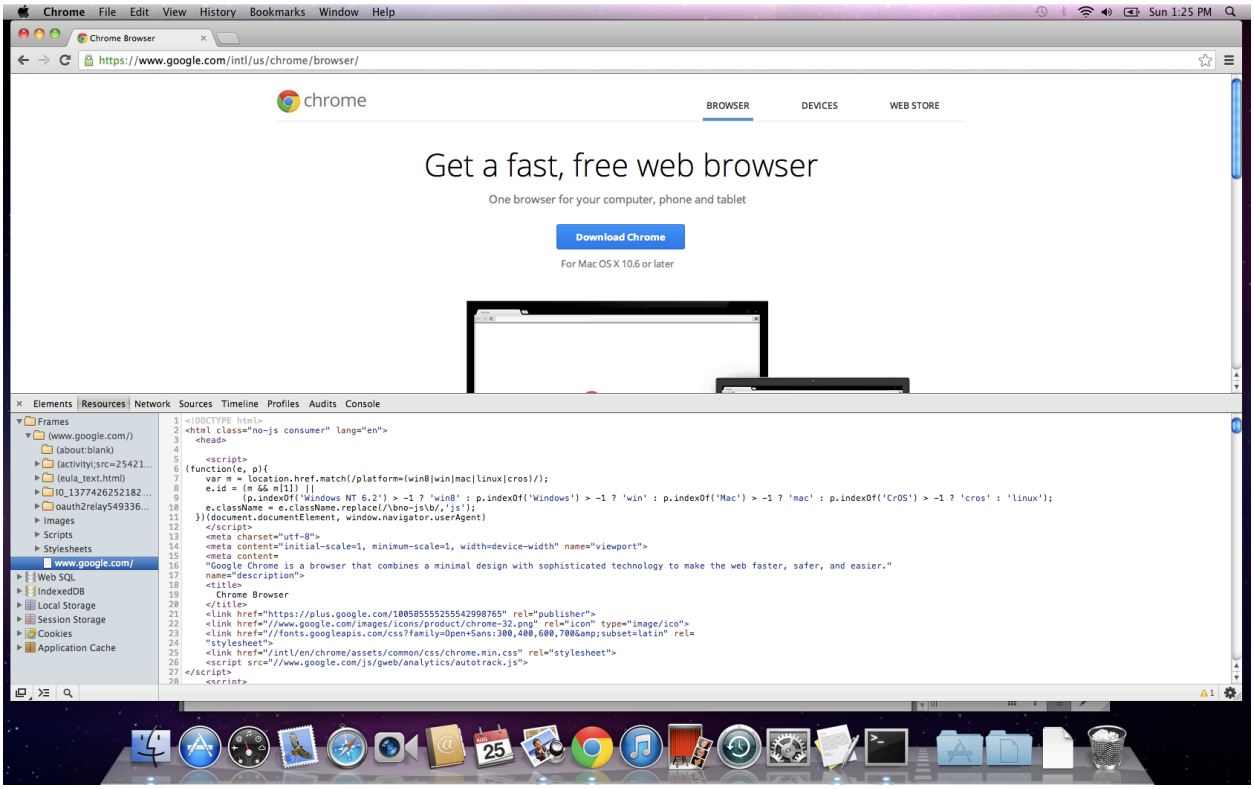
Later, you could copy examples into the *rpjs* folder as well as create apps in that folder.

Note

Another useful thing is to have the “New Terminal at Folder” option in Finder on Mac OS X. To enable it, open your “System Preferences” (you could use Command + Space, a.k.a. Spotlight, for it). Find “Keyboard” and click on it. Open “Keyboard Shortcuts” and click on “Services.” Check the “New Terminal at Folder” and “New Terminal Tab at Folder” boxes. Close the window (optional).

Browsers

We recommend downloading the latest version of the WebKit[1] or Gecko[2] browser of your choice: Chrome[3], Safari[4] or Firefox[5]. While Chrome and Safari already come with built-in Developer Tools, you'll need the Firebug[6] plug-in for Firefox.

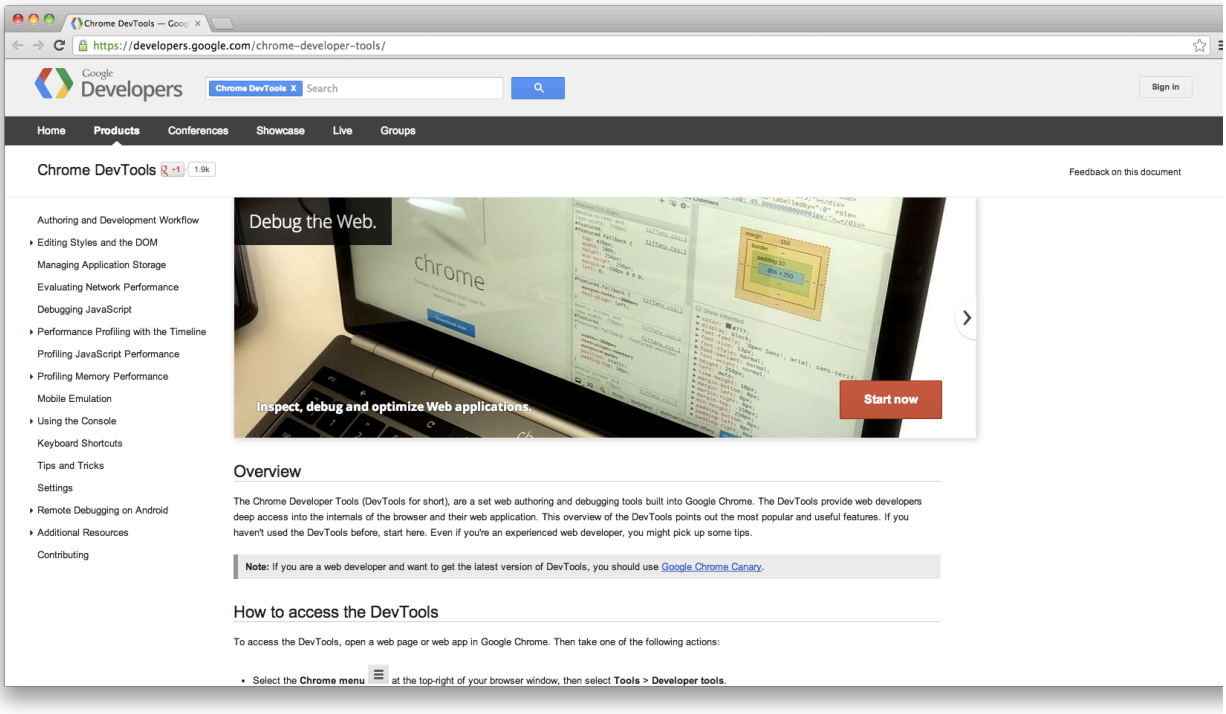


Chrome dev tools.

Firebug and Developer Tools allow developers to do many things like:

- Debug JavaScript
- Manipulate HTML and DOM elements
- Modify CSS on the fly
- Monitor HTTP requests and responses
- Run profiles and inspect heap dumps
- See loaded assets such as images, CSS and JS files

[1] http://en.wikipedia.org/wiki/WebKit
[2] http://en.wikipedia.org/wiki/Gecko_(layout_engine)
[3] http://www.google.com/chrome
[4] http://www.apple.com/safari/
[5] http://www.mozilla.org/en-US/firefox/new/
[6] http://getfirebug.com/



Google tutorials for mastering web deb tools.

Great Chrome DevTools tutorials:

- Explore and Master Chrome DevTools[7] with Code School
- Chrome DevTools videos[8]
- Chrome DevTools overview[9]

[7] http://discover-devtools.codeschool.com/
[8] https://developers.google.com/chrome-developer-tools/docs/videos
[9] https://developers.google.com/chrome-developer-tools/

Mastering chrome DevTools.

IDEs and Text Editors

One of the best things about JavaScript is that you don't need to compile the code. Because JS lives in and is run in a browser, you can do debugging right there, in a browser! Therefore, we highly recommend a lightweight text editor vs. a full-blown integrated development environment[10], or IDE, but if you are already familiar and comfortable with the IDE of your choice like Eclipse[11], NetBeans[12] or Aptana[13], feel free to stick with it.

[10] http://en.wikipedia.org/wiki/Integrated_development_environment
[11] http://www.eclipse.org/
[12] http://netbeans.org/
[13] http://aptana.com/

Sublime Text code editor home page.

Here is the list of the most popular text editors and IDEs used in web development:

- TextMate[14]: Mac OS X version only, free 30-day trial for v1.5, dubbed *The Missing Editor for Mac OS X*.
- Sublime Text[15]: Mac OS X and Windows versions are available, even better alternative to TextMate, unlimited evaluation period.
- Coda[16]: all-in-one editor with FTP browser and preview, has support for development with/on an iPad.
- Aptana Studio[17]: full-sized IDE with a built-in terminal and many other tools.
- Notepad ++[18]: free Windows-only lightweight text editor with the support of many languages.
- WebStorm IDE[19]: feature-rich IDE which allows for Node.js debugging; it's developed by JetBrains and marketed as *the smartest JavaScript IDE*.

[14] http://macromates.com/
[15] http://www.sublimetext.com/
[16] http://panic.com/coda/
[17] http://aptana.com/
[18] http://notepad-plus-plus.org/
[19] http://www.jetbrains.com/webstorm/

WebStorm IDE home page.

Version Control Systems

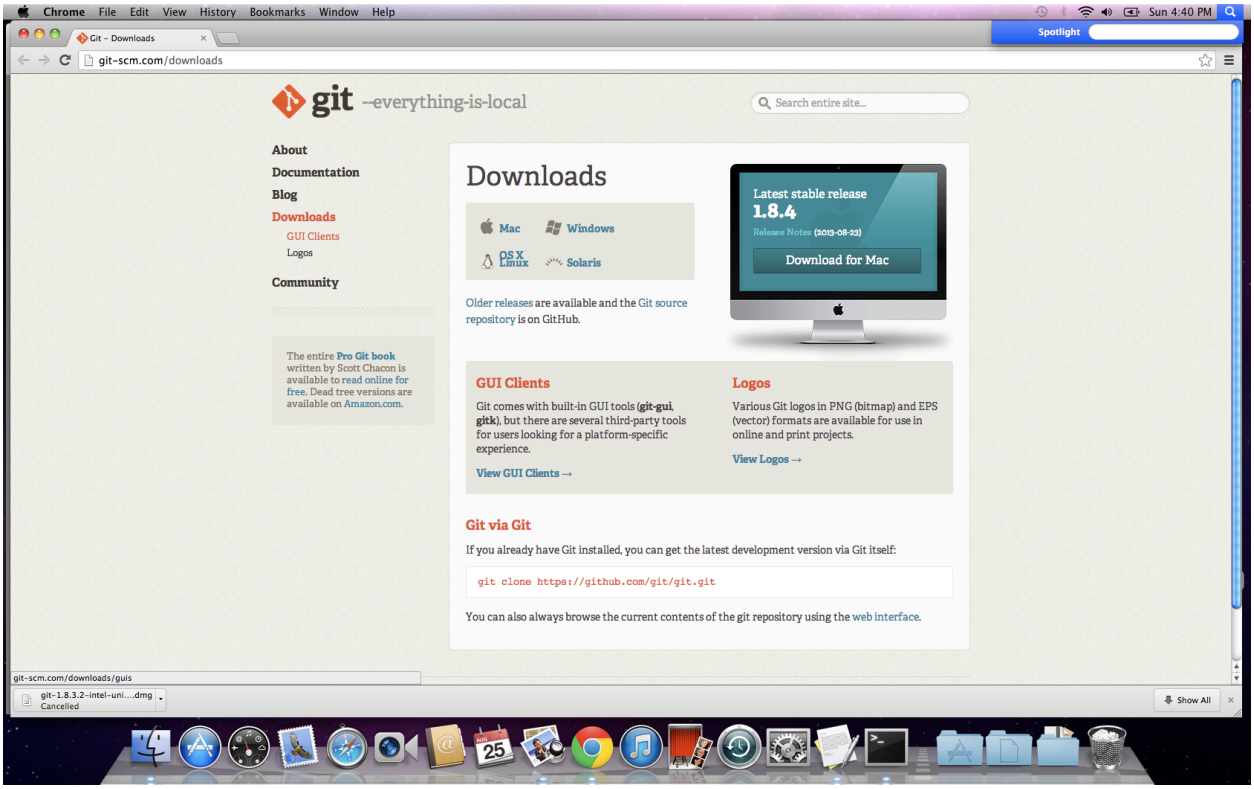
Version control system[20] is a must-have even in an only-one-developer situation. Also many cloud services, e.g., Heroku, require Git for deployment. We also highly recommend getting used to Git and Git terminal commands instead of using Git visual clients/apps with a graphical user interface: GitX[21], Gitbox[22] or GitHub for Mac[23].

Subversion is a non-distributed version control system. This article compares Git vs. Subversion[24].

Here are the steps to install and set up Git on your machine:

1. Download the latest version for your OS at http://git-scm.com/downloads.

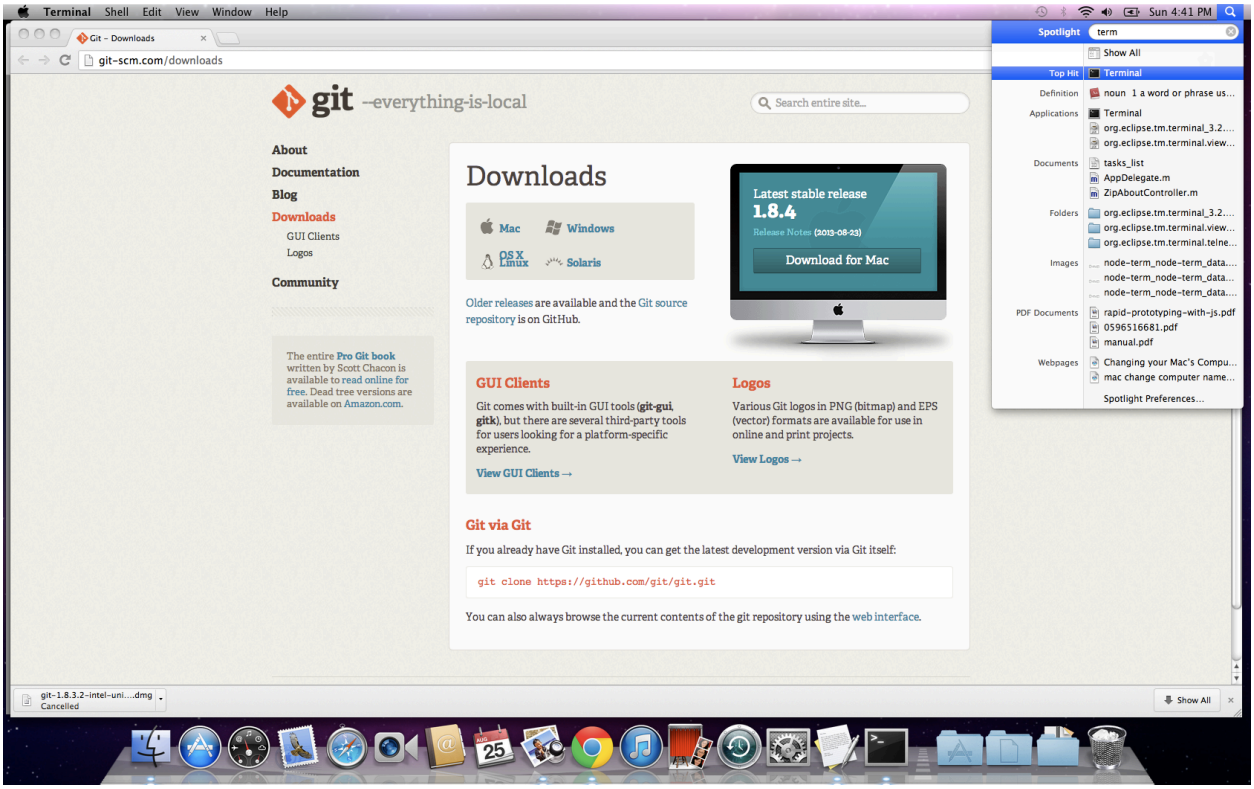
[20] http://en.wikipedia.org/wiki/Revision_control
[21] http://gitx.frim.nl/
[22] http://www.gitboxapp.com/
[23] http://mac.github.com/
[24] https://git.wiki.kernel.org/index.php/GitSvnComparison



Downloading latest release of Git.

2. Install Git from the downloaded `*.dmg` package, i.e., run `*.pkg` file and follow the wizard.
3. Find the terminal app by using Command + Space, a.k.a. Spotlight (please see the screenshot below), on OS X. For Windows you could use PuTTY[25] or Cygwin[26].

[25] http://www.chiark.greenend.org.uk/~sgtatham/putty/
[26] http://www.cygwin.com/



Using Spotlight to find and run an application.

4. In your terminal, type these commands, **substituting** "John Doe" and johndoe@example.com with your name and email:

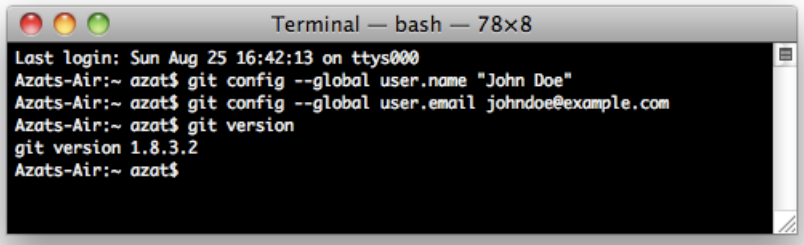
```
1  $ git config --global user.name "John Doe"
2  $ git config --global user.email johndoe@example.com
```

5. To check the installation, run command:

```
1  $ git version
```

6. You should see something like this in your terminal window (your version might vary; in our case it's 1.8.3.2):

```
1  git version 1.8.3.2
```



Configuring & Testing Git installation.

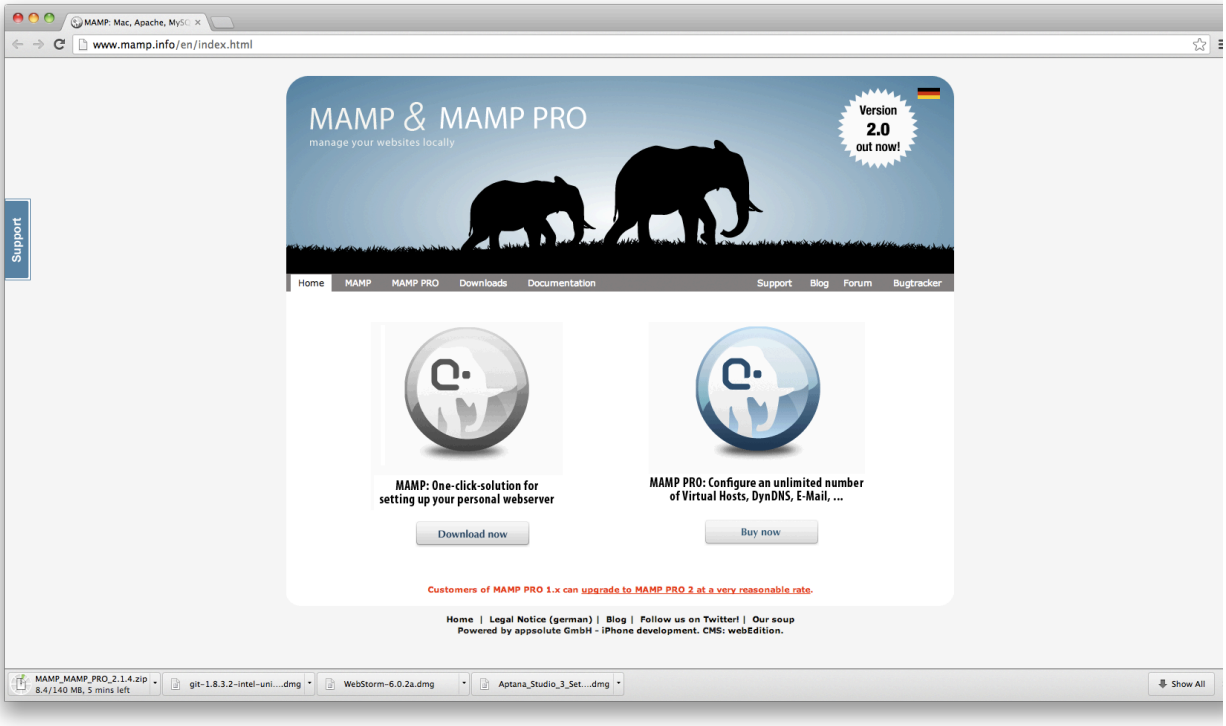
Generation of SSH keys and uploading them to SaaS/PaaS websites will be covered later.

Local HTTP Servers

While you can do most of the front-end development without a local HTTP server, it is needed for loading files with HTTP Requests/AJAX calls. Also, it's just a good practice in general to use a local HTTP server. This way, your development environment is as close to the production environment as possible. You might want to consider the following modifications of the Apache web server:

- MAMP[27]: Mac, Apache, MySQL, PHP personal web server for Mac OS X
- MAMP Stack[28]: Mac app with PHP, Apache, MySQL and phpMyAdmin stack build by BitNami (Apple app store[29])
- XAMPP[30]: Apache distribution containing MySQL, PHP and Perl for Windows, Mac, Linux and Solaris.

[27] http://www.mamp.info/en/index.html
[28] http://bitnami.com/stack/mamp
[29] https://itunes.apple.com/es/app/mamp-stack/id571310406?l=en
[30] http://www.apachefriends.org/en/xampp.html



MAMP for Mac home page.

MAMP, MAMP Stack and XAMPP have intuitive Graphical User Interfaces (GUIs) which allow you to change configurations and host file settings.

Note

Node.js as many other back-end technologies have their own servers for development.

Database: MongoDB

The following steps are better suited for Mac OS X/Linux based systems but with some modification can be used for Windows systems as well, i.e., $PATH variable - step #3. Below, we describe the MongoDB installation from the official package, because we found that this approach is more robust and leads to less conflicts. However, there are many other ways to install it on Mac[31], for example using Brew, as well as on other systems[32].

1. MongoDB can be downloaded at http://www.mongodb.org/downloads. For the latest Apple laptops, like MacBook Air, select OS X 64-bit version. The owners of older Macs should browse the link http://dl.mongodb.org/dl/osx/i386.

[31] http://docs.mongodb.org/manual/tutorial/install-mongodb-on-os-x/
[32] http://docs.mongodb.org/manual/installation/

Tip

To figure out the architecture type of your processor, type the `$ uname -p` in the command line.

1. Unpack the package into your web development folder (~/*Documents/Development* or any other). If you want, you could install MongoDB into */usr/local/mongodb* folder.
2. **Optional**: If you would like to access MongoDB commands from anywhere on your system, you need to add your **mongodb** path to the $PATH variable. For Mac OS X the open system *paths* file with:

```
1  sudo vi /etc/paths
```

   or, if you prefer TextMate:

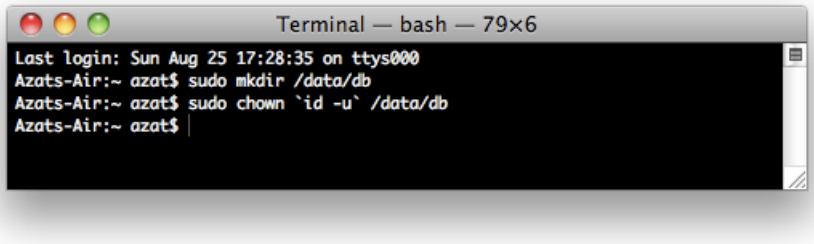
```
1  mate /etc/paths
```

   And add this line to the */etc/paths* file:

```
1  /usr/local/mongodb/bin
```

3. Create a data folder; by default, MongoDB uses */data/db*. Please note that this might be different in a new versions of MongoDB. To create it, type and execute the following commands in the terminal:

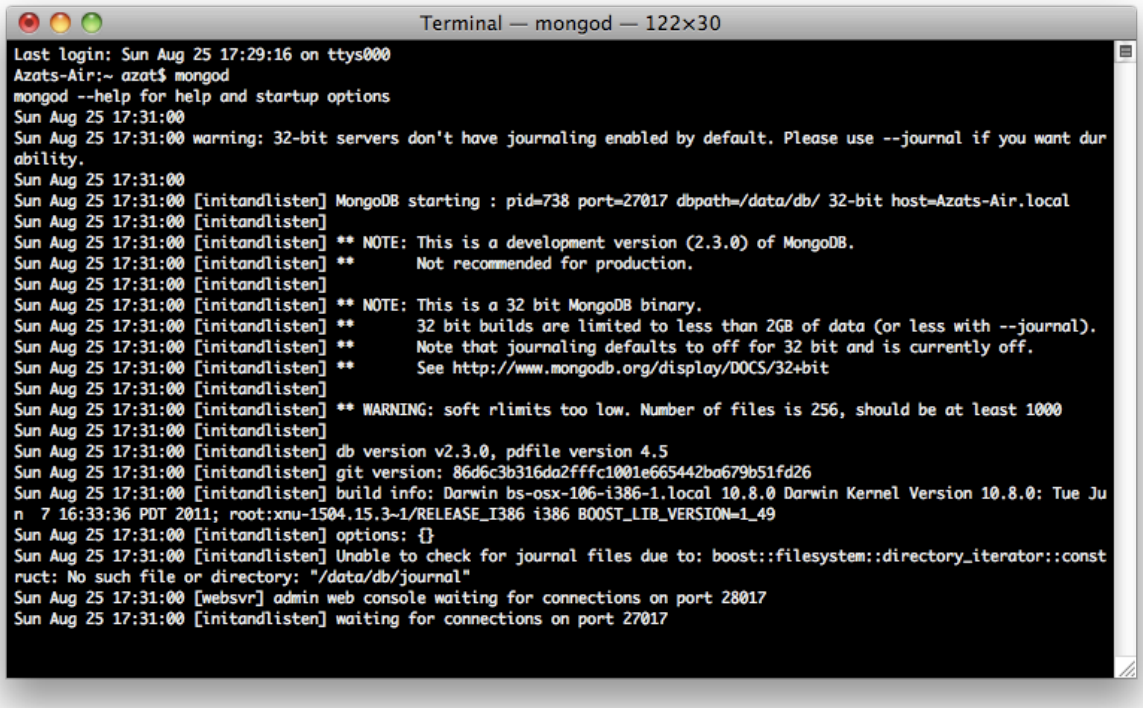
```
1  $ sudo mkdir -p /data/db
2  $ sudo chown `id -u` /data/db
```



Initial setup for MongoDB: create the data directory.

   If you prefer to use path other than */data/db* you could specify it using the –dbpath option to **mongod** (main MongoDB service).
4. Go to the folder where you unpacked MongoDB. That location should have a *bin* folder in it. From there, type the following command in your terminal:

```
1   $ ./bin/mongod
```



Starting-up the MongoDB server.

5. If you see something like

```
1   MongoDB starting: pid =7218 port=27017...
```

   it means that the MongoDB database server is running. By default, it's listening at http://localhost:27017. If you go to your browser and type http://localhost:28017 you should be able to see the version number, logs and other useful information. In this case MondoDB server is using **two** different ports (27017 and 28017): one is primary (native) for the communications with apps and the other is web based GUI for monitoring/statistics. In our Node.js code we'll be using only 27017.

Note

Don't forget to restart the Terminal window after adding a new path to the $PATH variable.

Now, to take it even further, we can test to determine if we have access to the MongoDB console/shell, which will act as a client to this server. This means that we'll have to keep the terminal window with the server open and running.

1. Open another terminal window at the same folder and execute:

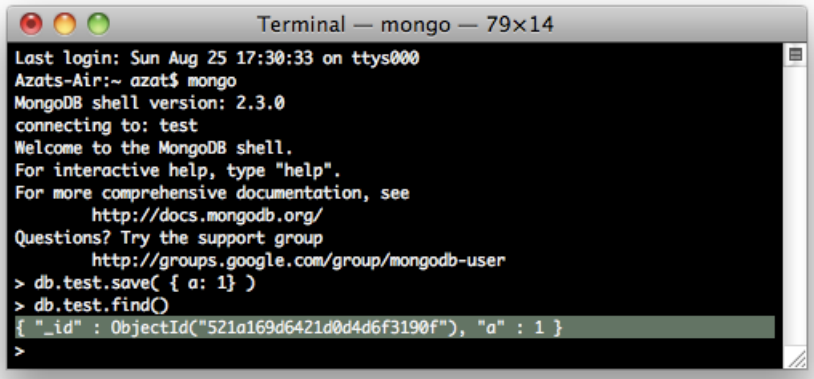
```
$ ./bin/mongo
```

You should be able to see something like “MongoDB shell version 2.0.6 ...”

2. Then type and execute:

```
> db.test.save( { a: 1 } )
> db.test.find()
```

If you see that your record is being saved, then everything went well:



Running MongoDB client and storing sample data.

Commands *find* and *save* do exactly what you might think they do. ;-)

Detailed instructions are also available at MongoDB.org: Install MongoDB on OS X[33]. For Windows, users there is a good walk-through article: Installing MongoDB[34].

Note

MAMP and XAMPP applications come with MySQL — open-source traditional SQL database, and phpMyAdmin — web interface for MySQL database.

[33] http://docs.mongodb.org/manual/tutorial/install-mongodb-on-os-x/
[34] http://www.tuanleaded.com/blog/2011/10/installing-mongodb/

Note

On Mac OS X (and most Unix systems), to close the process use `control + c`. If you use `control + z` it will put the process to sleep (or detach the terminal window); in this case, you might end up with the lock on data files and will have to use the kill command or Activity Monitor, and manually delete the locked file in the data folder. In vanilla Mac Terminal `command + .` is an alternative to `control + c`.

Other Components

Node.js Installation

Node.js is available at http://nodejs.org/#download (please see the screenshot below). The installation is trivial, i.e., download the archive, run the `*.pkg` package installer. To check the installation of Node.js you could type and execute:

```
1 $ node -v
```

It should show something similar to this (we use v.0.8.1, but your version might vary):

```
1 v0.8.1
```

Node.js package already includes Node Package Manager[35] (NPM). We'll use NPM extensively to install Node.js modules.

[35] https://npmjs.org

Node.js home page.

JS Libraries

Front-end JavaScript libraries are downloaded and unpacked from their respective websites. Those files are usually put in Development folder (e.g., `~/Documents/Development`) for future use. Oftentimes, there is a choice between minified production version (more on that in AMD and Require.js section of the *Intro to Backbone.js* chapter) and extensively rich in comments development one.

Another approach is to hot-link these scripts from CDNs such as Google Hosted Libraries[36], CDNJS[37], Microsoft Ajax Content Delivery Network[38] and others. By doing so the apps will be faster for some users, but won't work locally at all without the Internet.

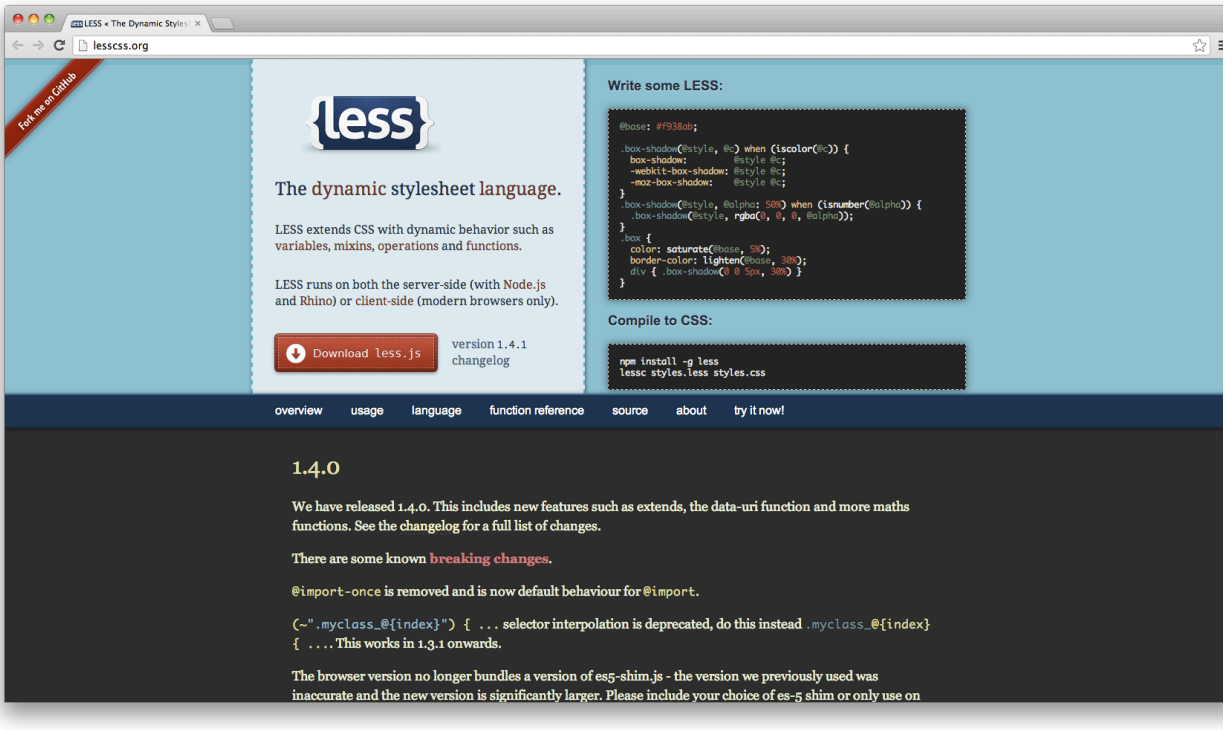
- LESS as a front-end interpreter is available at lesscss.org[39] — you could unpack it into your development folder (*~/Documents/Development*) or any other.
- Twitter Bootstrap is a CSS/LESS framework. It's available at twitter.github.com/bootstrap[40].
- jQuery is available at jquery.com[41].
- Backbone.js is available at backbonejs.org[42].

[36] https://developers.google.com/speed/libraries/devguide
[37] http://cdnjs.com/
[38] http://www.asp.net/ajaxlibrary/cdn.ashx
[39] http://lesscss.org/
[40] http://twitter.github.com/bootstrap/
[41] http://jquery.com
[42] http://backbonejs.org

- Underscore.js is available at underscorejs.org[43].
- Require.js is available at requirejs.org[44].

LESS App

The LESS App is a Mac OS X application for "on-the-fly" compilation of LESS to CSS. It's available at incident57.com/less[45].



LESS App for Mac home page.

Cloud Setup

SSH Keys

SSH keys provide a secure connection without the need to enter username and password every time. For GitHub repositories, the latter approach is used with HTTPS URLs, e.g., `https://github.com/azat-co/rpjs.git` and the former with SSH URLs, e.g., `git@github.com:azat-co/rpjs.git`.

To generate SSH keys for GitHub on Mac OS X/Unix machines do the following:

1. Check for existing SSH keys

[43] http://underscorejs.org
[44] http://requirejs.org
[45] http://incident57.com/less/

```
$ cd ~/.ssh
$ ls -lah
```

2. If you see some files like **id_rsa** (please refer to the screenshot below for an example), you could delete them or backup into a separate folder by using following commands:

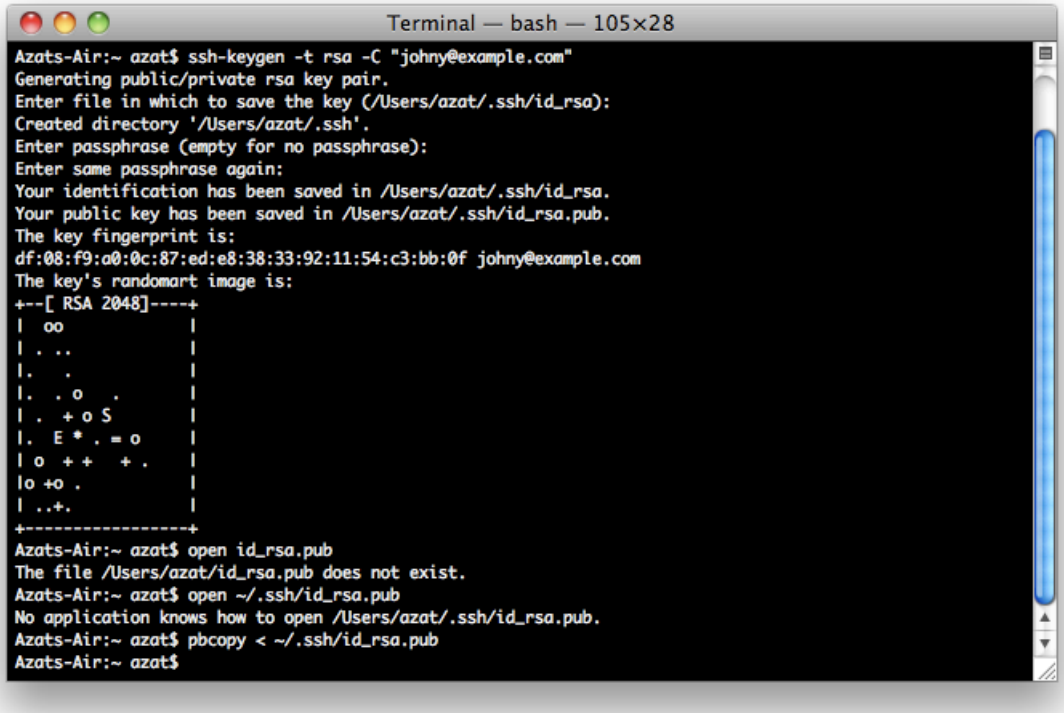
```
$ mkdir key_backup
$ cp id_rsa* key_backup
$ rm id_rsa*
```

3. Now we can generate a new SSH key pair using the **ssh-keygen** command, assuming we are in ~/.ssh folder:

```
$ ssh-keygen -t rsa -C "your_email@youremail.com"
```

4. Answer the questions; it is better to keep the default name: **id_rsa**. Then copy the content of the **id_rsa.pub** file to your clipboard:

```
$ pbcopy < ~/.ssh/id_rsa.pub
```



Generating RSA key for SSH and copying public key to clipboard.

5. Or alternatively, open **id_rsa.pub** file in the default editor:

```
$ open id_rsa.pub
```

6. Or in TextMate:

```
$ mate id_rsa.pub
```

GitHub

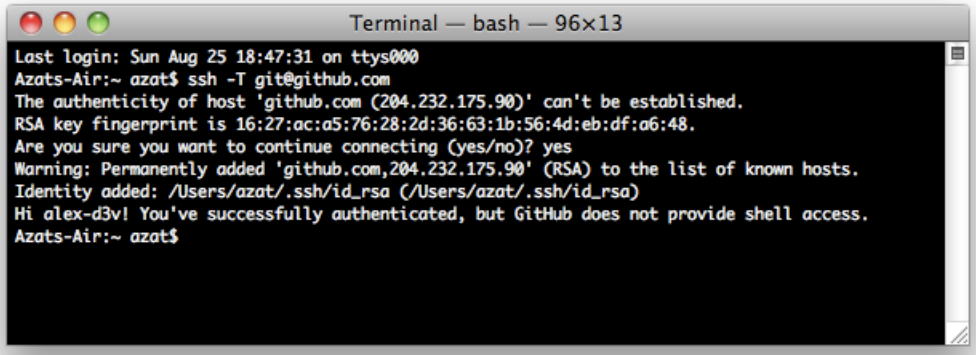
1. After you have copied the public key, go to github.com[46], log in, go to your account settings, select "SSH key" and add the new SSH key. Assign a name, e.g., the name of your computer, and paste the value of your **public** key.
2. To check if you have an SSH connection to GitHub, type and execute the following command in your terminal:

```
$ ssh -T git@github.com
```

   If you see something like:

```
Hi your-GitHub-username! You've successfully authenticated,
but GitHub does not provide shell access.
```

   then everything is set up.
3. While the first time connecting to GitHub, you can receive "authenticity of host ... can't be established" warning. Please don't be confused with such a message — just proceed by answering 'yes' as shown on the screenshot below.



Testing SSH connection to GitHub for the very first time.

   If for some reason you have a different message, please repeat steps 3-4 from the previous section on *SSH Keys* and/or re-upload the content of your `*.pub` file to GitHub.

[46] http://github.com

Warning

Keep your **id_rsa** file private and don't share it with anybody!

More instructions are available at GitHub: Generating SSH Keys[47].

Windows users might find useful the SSH key generator feature in [PuTTY].

Windows Azure

Here are the steps to set up a Windows Azure account:

1. You'll need to sign up for Windows Azure Web Site and Virtual Machine previews. Currently they have a 90-day free trial https://www.windowsazure.com/en-us/.
2. Enable Git Deployment and create a username and password. Then upload SSH public key to Windows Azure.
3. Install Node.js SDK, which is available at https://www.windowsazure.com/en-us/develop/nodejs/.
4. To check your installation type:

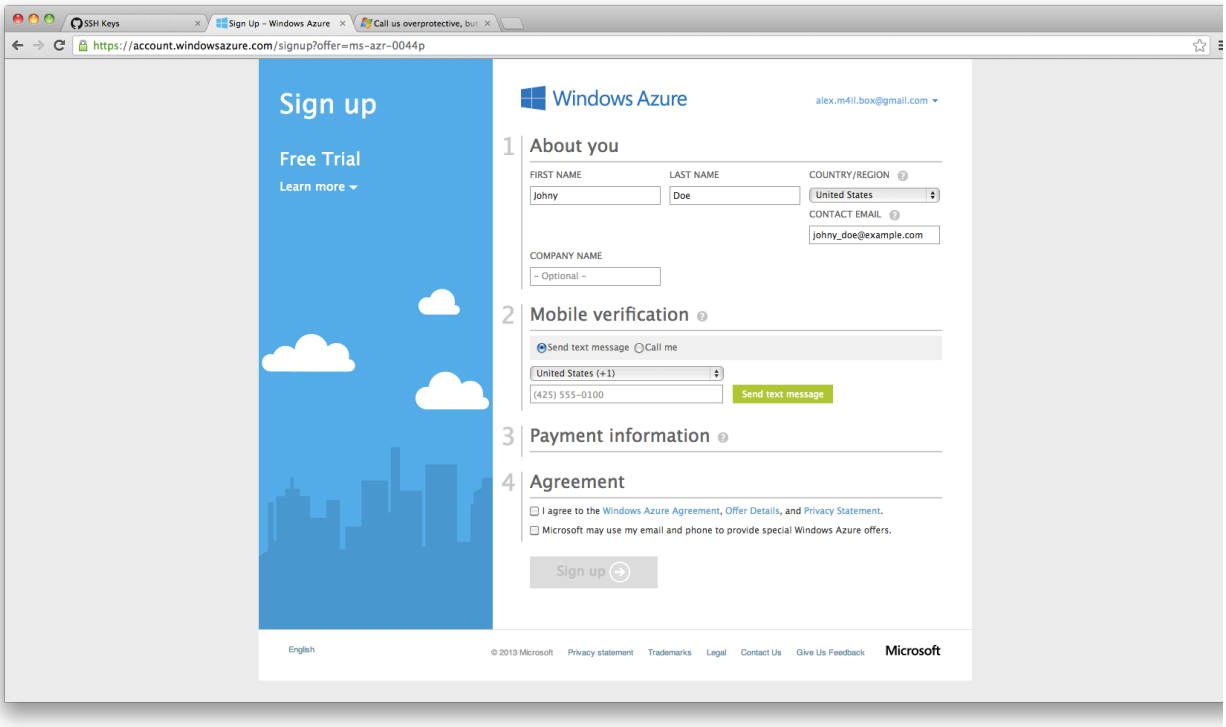
```
$ azure -v
```

   You should be able to see something like:

```
Windows Azure: Microsoft's Cloud Platform... Tool Version 0.6.0
```

5. Log in to Windows Azure Portal at https://windows.azure.com/.

[47] https://help.github.com/articles/generating-ssh-keys



Registering on Windows Azure.

6. Select "New," then select "Web Site," "Quick Create." Type the name which will serve as the URL for your website, and click "OK."
7. Go to this newly created Web Site's Dashboard and select "Set up Git publishing." Come up with a username and password. This combination can be used to deploy to any web site in your subscription, meaning that you do not need to set credentials for every web site you create. Click "OK."
8. On the follow-up screen, it should show you the Git URL to push to, something like

```
1  https://azatazure@azat.scm.azurewebsites.net/azat.git
```

   and instructions on how to proceed with deployment. We'll cover them later.
9. **Advanced user option**: follow this tutorial to create a virtual machine and install MongoDB on it: Install MongoDB on a virtual machine running CentOS Linux in Windows Azure[48].

Heroku

Heroku is a polyglot agile application deployment http://www.heroku.com/ platform. Heroku works similarly to Windows Azure in the sense that you can use Git to deploy applications. There is no need to install Virtual Machine for MongoDB because Heroku has MongoHQ add-on[49].

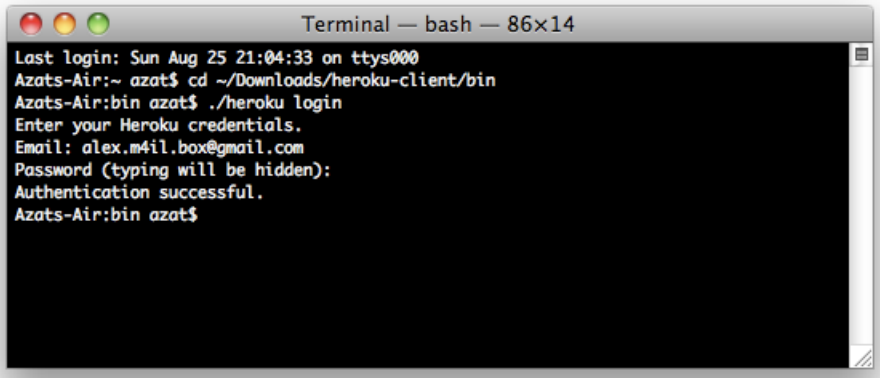
[48] https://www.windowsazure.com/en-us/manage/linux/common-tasks/mongodb-on-a-linux-vm/
[49] https://addons.heroku.com/mongohq

To set up Heroku, follow these steps:

1. Sign up at http://heroku.com. Currently they have a free account; to use it, select all options as minimum (0) and database as shared.
2. Download Heroku Toolbelt at https://toolbelt.heroku.com. Toolbelt is a package of tools, i.e., libraries which consists of Heroku, Git, and Foreman[50]. For users of older Macs get this client[51] directly. If you utilize another OS, browse Heroku Client GitHub[52].
3. After the installation is done, you should have access to the **heroku** command. To check it and log in to Heroku, type:

```
$ heroku login
```

   It will ask you for Heroku credentials (username and password), and if you've already created the SSH key, it will automatically upload it to the Heroku website:



The response to the successful `$ heroku login` command.

4. If everything went well, to create a Heroku application inside of your specific project folder, you should be able to run:

```
$ heroku create
```

More detailed step-by-step instructions are available at Heroku: Quickstart[53] and Heroku: Node.js[54].

[50] https://github.com/ddollar/foreman
[51] http://assets.heroku.com/heroku-client/heroku-client.tgz
[52] https://github.com/heroku/heroku
[53] https://devcenter.heroku.com/articles/quickstart
[54] https://devcenter.heroku.com/articles/nodejs

Cloud9

Cloud9 is an in-browser IDE with which, by using your GitHub or BitBucket account, you can browse your repositories, edit them and deploy to Windows Azure or other services. No installations are needed; everything works in the browser, pretty much like Google Docs.

“Hello World” in Node.js

This section contains

- “Hello World” application in Node.js
- List of some of Node.j most important core modules
- NPM workflow
- Detailed commands for deployment of Node.js apps to Heroku and Microsoft Windows Azure

To check if you have Node.js installed on your computer, type and execute this command in your terminal:

```
1 $ node -v
```

As of this writing, the latest version is 0.8.1. If you don’t have Node.js installed, or if your version is behind, you can download the latest version at nodejs.org/#download[55].

As usual, you could copy example code from rpjs/hello[56] or write your own program from scratch. If you wish to do the latter, create a folder **hello** for your “Hello World” Node.js application. Then create file a **server.js** and line by line type the code below.

This will load the core **http** module for the server (more on the modules later):

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

We’ll need a port number for our Node.js server. To get it from the environment, or assign 1337 if the environment is not set, use:

```
1 var port = process.env.PORT || 1337;
```

This will create a server and a call-back function will contain the response handler code:

```
1 var server = http.createServer(function (req, res) {
```

To set the right header and status code, use:

[55] http://nodejs.org/#download

[56] https://github.com/azat-co/rpjs/tree/master/hello

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

To output "Hello World" with the line end symbol, use:

```
  res.end('Hello World\n');
});
```

To set a port and display the address of the server and the port number, use:

```
server.listen(port, function() {
  console.log('Server is running at %s:%s ',
    server.address().address, server.address().port);
});
```

From the folder in which you have *server.js*, launch in your terminal the following command:

```
$ node server.js
```

Open localhost:1337[57] or 127.0.0.1:1337[58] or any other address you see in the terminal as a result of `console.log()` function, and you should see "Hello World" in a browser. To shut down the server, press Control + C.

Note

The name of the main file could be different from server.js, e.g., index.js or app.js. In case you need to launch the **app.js** file, just use `$ node app.js`.

Node.js Core Modules

Unlike other programming technologies, Node.js doesn't come with a heavy standard library. The core modules of node.js are a bare minimum and the rest can be cherry-picked via the Node Package Manager (NPM) registry. The main core modules, classes, methods and events include:

- http[59]
- util[60]

[57] http://localhost:1337/
[58] http://127.0.0.1:1337/
[59] http://nodejs.org/api/http.html#http_http
[60] http://nodejs.org/api/util.html

- querystring[61]
- url[62]
- fs[63]

http[64]

This is the main module responsible for Node.js HTTP server. Here are the main methods:

- `http.createServer()`: returns a new web server object
- `http.listen()`: begins accepting connections on the specified port and hostname
- `http.createClient()`: node app can be a client and make requests to other servers
- `http.ServerRequest()`: incoming requests are passed to request handlers
  - **data**: emitted when a piece of the message body is received
  - **end**: emitted exactly once for each request
  - `request.method()`: the request method as a string
  - `request.url()`: request URL string
- `http.ServerResponse()`: this object is created internally by an HTTP server — not by the user, and used as an output of request handlers
  - `response.writeHead()`: sends a response header to the request
  - `response.write()`: sends a response body * `response.end()`: sends and ends a response body

util[65]

This module provides utilities for debugging. Some of the methods include:

- `util.inspect()`: Return a string representation of an object, which is useful for debugging

querystring[66]

This module provides utilities for dealing with query strings. Some of the methods include:

- `querystring.stringify()`: Serialize an object to a query string
- `querystring.parse()`: Deserialize a query string to an object

url[67]

This module has utilities for URL resolution and parsing. Some of the methods include:

[61] http://nodejs.org/api/querystring.html
[62] http://nodejs.org/api/url.html
[63] http://nodejs.org/api/fs.html
[64] http://nodejs.org/api/http.html#http_http
[65] http://nodejs.org/api/util.html
[66] http://nodejs.org/api/querystring.html
[67] http://nodejs.org/api/url.html

- `parse()`: Take a URL string, and return an object

fs[68]

fs handles file system operations such as reading and writing to/from files. There are synchronous and asynchronous methods in the library. Some of the methods include:

- `fs.readFile()`: reads file asynchronously
- `fs.writeFile()`: writes data to file asynchronously

There is no need to install or download core modules. To include them in your application, all you need is to follow the syntax:

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

The lists of non-core modules can be found at:

- npmjs.org[69]: Node Package Manager registry
- GitHub hosted list[70]: list of Node.js modules maintainded by Joyent
- nodetoolbox.com[71]: registry based on stats
- Nipster[72]: NPM search tool for Node.js
- Node Tracking[73]: registry based on GitHub stats

If you would like to know how to code your own modules, take a look at the article Your first Node.js module[74].

Node Package Manager

Node Package Manager, or NPM, manages dependencies and installs modules for you. Node.js installation comes with NPM, whose website is npmjs.org[75].

package.json contains meta information about our Node.js application such as a version number, author name and, most importantly, what dependencies we use in the application. All of that information is in the JSON formatted object, which is read by NPM.

If you would like to install packages and dependencies specified in **package.json**, type:

[68] http://nodejs.org/api/fs.html
[69] https://npmjs.org
[70] https://github.com/joyent/node/wiki/Modules
[71] http://nodetoolbox.com/
[72] http://eirikb.github.com/nipster/
[73] http://nodejsmodules.org
[74] http://cnnr.me/blog/2012/05/27/your-first-node-dot-js-module/
[75] http://npmjs.org/

```
$ npm install
```

A typical **package.json** file might look like this:

```
{
   "name": "Blerg",
   "description": "Blerg blerg blerg.",
   "version": "0.0.1",
   "author": {
      "name" : "John Doe",
      "email" : "john.doe@gmail.com"
   },
   "repository": {
       "type": "git",
       "url": "http://github.com/johndoe/blerg.git"
   },
   "engines": [
       "node >= 0.6.2"
   ],
   "license" : "MIT",
   "dependencies": {
       "express": ">= 2.5.6",
       "mustache": "0.4.0",
       "commander": "0.5.2"
   },
   "bin" : {
       "blerg" : "./cli.js"
   }
}
```

To update a package to its current latest version, or the latest version that is allowable by the version specification defined in **package.json**, use:

```
$ npm update name-of-the-package
```

Or for single module installation:

```
$ npm install name-of-the-package
```

The only module used in the examples — and which does not belong to the core Node.js package — is **mongodb**. We'll install it later in the book.

Heroku will need *package.json* to run NPM on the server.

For more information on NPM, take a look at the article Tour of NPM[76].

Deploying "Hello World" to PaaS

For Heroku and Windows Azure deployment, we'll need a Git repository. To create it from the root of your project, type the following command in your terminal:

```
$ git init
```

Git will create a hidden .**git** folder. Now we can add files and make the first commit:

```
$ git add .
$ git commit -am "first commit"
```

Tip

To view hidden files on the Mac OS X Finder app, execute this command in a terminal window: `defaults write com.apple.finder AppleShowAllFiles -bool true`. To change the flag back to hidden: `defaults write com.apple.finder AppleShowAllFiles -bool false`.

Deploying to Windows Azure

In order to deploy our "Hello World" application to Windows Azure, we must add Git **remote**. You could copy the URL from Windows Azure Portal, under Web Site, and use it with this command:

```
$ git remote add azure yourURL
```

Now we should be able to make a push with this command:

```
$ git push azure master
```

If everything went okay, you should see success logs in the terminal and "Hello World" in the browser of your Windows Azure Web Site URL.

To push changes, just execute:

[76] http://tobyho.com/2012/02/09/tour-of-npm/

```
$ git add .
$ git commit -m "changing to hello azure"
$ git push azure master
```

A more meticulous guide can be found in the tutorial Build and deploy a Node.js web site to Windows Azure[77].

Deploying to Heroku

For Heroku deployment, we need to create 2 extra files: **Procfile** and **package.json**. You could get the source code from rpjs/hello[78] or write your own one.

The structure of the "Hello World" application looks like this:

```
/hello
  -package.json
  -Procfile
  -server.js
```

Procfile is a mechanism for declaring what commands are run by your application's dynos on the Heroku platform. Basically, it tells Heroku what processes to run. Procfile has only one line in this case:

```
web: node server.js
```

For this example, we keep **package.json** simple:

```
{
  "name": "node-example",
  "version": "0.0.1",
  "dependencies": {
  },
  "engines": {
    "node": ">=0.6.x"
  }
}
```

After we have all of the files in the project folder, we can use Git to deploy the application. The commands are pretty much the same as with Windows Azure except that we need to add Git remote, and create Cedar stack with:

[77] http://www.windowsazure.com/en-us/develop/nodejs/tutorials/create-a-website-(mac)/

[78] https://github.com/azat-co/rpjs/tree/master/hello

```
$ heroku create
```

After it's done, we push and update with:

```
$ git push heroku master
$ git add .
$ git commit -am "changes"
$ git push heroku master
```

If everything went okay, you should see success logs in the terminal and "Hello World" in the browser of your Heroku app URL.

Chat REST API Server (Memory Store)

This section contains:

- Chat API server with memory store
- Example of a test-driven development practice.

The first version of the Chat back-end application will store messages only in run-time memory storage for the sake of KISS[79]. That means that each time we start/reset the server, the data will be lost.

We'll start with a simple test case first to illustrate the Test-Driven Development approach. The full code is available under the rpjs/test[80] folder.

Test Case for Chat

We should have two methods:

1. Get all of the messages as an array of JSON objects for the GET `/message` endpoint using the `getMessages()` method
2. Add a new message with properties **name** and **message** for POST `/messages` route via the `addMessage()` function

We'll start by creating an empty **mb-server.js** file. After it's there, let's switch to tests and create the **test.js** file with the following content:

[79] http://en.wikipedia.org/wiki/KISS_principle

[80] https://github.com/azat-co/rpjs/tree/master/test

```
var http = require('http');
var assert = require('assert');
var querystring = require('querystring');
var util = require('util');

var messageBoard = require('./mb-server');

assert.deepEqual('[{"name":"John","message":"hi"}]',
  messageBoard.getMessages());
assert.deepEqual ('{"name":"Jake","message":"gogo"}',
  messageBoard.addMessage ("name=Jake&message=gogo"));
assert.deepEqual('[{"name":"John","message":"hi"},{"name":"Jake","message":"gogo\
"}]',
  messageBoard.getMessages("name=Jake&message=gogo"));
```

Please keep in mind that, this is a very simplified comparison of strings and not JavaScript objects. So every space, quote and case matters. You could make the comparison "smarter" by parsing a string into a JSON object with:

```
JSON.parse(str);
```

For testing our assumptions, we use core the Node.js module assert[81]. It provides a bunch of useful methods like `equal()`, `deepEqual()`, etc.

More advanced libraries include alternative interfaces with TDD and/or BDD approaches:

- Should[82]
- Expect[83]

For more Test-Driven Development and cutting-edge automated testing, you could use the following libraries and modules:

- Mocha[84]
- NodeUnit[85]
- Jasmine[86]
- Vows[87]

[81] http://nodejs.org/api/assert.html
[82] https://github.com/visionmedia/should.js/
[83] https://github.com/LearnBoost/expect.js/
[84] http://visionmedia.github.com/mocha/
[85] https://github.com/caolan/nodeunit
[86] http://pivotal.github.com/jasmine/
[87] http://vowsjs.org/

- Chai[88]

You could copy the "Hello World" script into the **mb-server.js** file for now or even keep it empty. If we run **test.js** by the terminal command:

```
$ node test.js
```

We should see an error. Probably something like this one:

```
TypeError: Object #<Object> has no method 'getMessages'
```

That's totally fine, because we haven't written `getMessages()` method yet. So let's do it and make our application more useful by adding two new methods: to get the list of the messages for Chat and to add a new message to the collection.

mb-server.js file with global `exports` object:

```
exports.getMessages = function() {
  return JSON.stringify(messages);
};
exports.addMessage = function (data){
  messages.push(querystring.parse(data));
  return JSON.stringify(querystring.parse(data));
};
```

So far, nothing fancy, right? To store the list of messages, we'll use an array:

```
var messages=[];
//this array will hold our messages
messages.push({
  "name":"John",
  "message":"hi"
});
//sample message to test list method
```

Tip

Generally, fixtures like dummy data belong to the test/spec files and not to the main application codebase.

[88] http://chaijs.com/

Our server code will look slightly more interesting. For getting the list of messages, according to REST methodology, we need to make a GET request. For creating/adding a new message, it should be a POST request. So in our createServer object, we should add `req.method()` and `req.url()` to check for an HTTP request type and a URL path.

Let's load the http module:

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

We'll need some of the handy functions from the util and querysting modules (to serialize and deserialize objects and query strings):

```
var util = require('util');
var querystring = require('querystring');
```

To create a server and expose it to outside modules (i.e., test.js):

```
exports.server=http.createServer(function (req, res) {
```

Inside of the request handler callback, we should check if the request method is POST and the URL is `messages/create.json`:

```
  if (req.method == "POST" &&
    req.url == "/messages/create.json") {
```

If the condition above is true, we add a message to the array. However, **data** must be converted to a string type (with encoding UTF-8) prior to the adding, because it is a type of Buffer:

```
    var message = '';
    req.on('data', function(data, message){
      console.log(data.toString('utf-8'));
      message = exports.addMessage(data.toString('utf-8'));
```

These logs will help us to monitor the server activity in the terminal:

```
    })
    console.log(util.inspect(message, true, null));
    console.log(util.inspect(messages, true, null));
```

The output should be in a text format with a status of 200 (okay):

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

We output a message with a newly created object ID:

```
    res.end(message);
  }
```

If the method is GET and the URL is `/messages/list.json` output a list of messages:

```
  if (req.method == "GET" &&
    req.url == "/messages/list.json") {
```

Fetch a list of messages:

```
    var body = exports.getMessages();
```

The response body will hold our output:

```
    res.writeHead(200, {
      'Content-Length': body.length,
      'Content-Type': 'text/plain'
    });
    res.end(body);
  }
  else {
```

This sets the right header and status code:

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

In case it's neither of the two endpoints above, we output a string with a line-end symbol:

```
    res.end('Hello World\n');
  };
  console.log(req.method);


}).listen(1337, "127.0.0.1");
```

Now, we should set a port and IP address of the server:

```
console.log('Server running at http://127.0.0.1:1337/');
```

We expose methods for the unit testing in **test.js** (exports keyword), and this function returns an array of messages as a string/text:

```
exports.getMessages = function() {
  return JSON.stringify(messages);
};
```

`addMessage()` converts a string into a JavaScript object with the parse/deserializer method from querystring:

```
exports.addMessage = function (data){
  messages.push(querystring.parse(data));
```

Also returning a new message in a JSON-as-a-string format:

```
  return JSON.stringify(querystring.parse(data));
};
```

Here is the full code of **mb-server.js**. It's also available at rpjs/test[89]:

[89] https://github.com/azat-co/rpjs/tree/master/test

```
var http = require('http');
//loads http module
var util = require('util');
//useful functions
var querystring = require('querystring');
//loads querystring module,
//we'll need it to serialize and
//deserialize objects and query strings

var messages=[];
//this array will hold our messages
messages.push({
  "name": "John",
  "message": "hi"
});
//sample message to test list method

exports.server=http.createServer(function (req, res) {
//creates server
  if (req.method == "POST" &&
    req.url == "/messages/create.json") {
    //if method is POST and
    //URL is messages/ add message to the array
    var message = '';
    req.on('data', function(data, message){
      console.log(data.toString('utf-8'));
      message = exports.addMessage(data.toString('utf-8'));
      //data is type of Buffer and
      //must be converted to string
      //with encoding UTF-8 first
      //adds message to the array
    })
    console.log(util.inspect(message, true, null));
    console.log(util.inspect(messages, true, null));
    //debugging output into the terminal
    res.writeHead(200, {
      'Content-Type': 'text/plain'
     });
    //sets the right header and status code
    res.end(message);
    //out put message, should add object id
  }
```

```
43    if (req.method == "GET" &&
44      req.url == "/messages/list.json") {
45    //if method is GET and
46    //URL is /messages output list of messages
47      var body = exports.getMessages();
48      //body will hold our output
49      res.writeHead(200, {
50        'Content-Length': body.length,
51        'Content-Type': 'text/plain'
52      });
53      res.end(body);
54    }
55    else {
56      res.writeHead(200, {
57        'Content-Type': 'text/plain'
58      });
59      //sets the right header and status code
60      res.end('Hello World\n');
61    };
62    console.log(req.method);
63    //outputs string with line end symbol
64  }).listen(1337, "127.0.0.1");
65  //sets port and IP address of the server
66  console.log('Server running at http://127.0.0.1:1337/');
67
68  exports.getMessages = function() {
69    return JSON.stringify(messages);
70    //output array of messages as a string/text
71  };
72  exports.addMessage = function (data){
73    messages.push(querystring.parse(data));
74    //to convert string into
75    //JavaScript object we use parse/deserializer
76    return JSON.stringify(querystring.parse(data));
77    //output new message in JSON as a string
78  };
```

To check it, go to localhost:1337/messages/list.json[90]. You should see an example message. Alternatively, you could use the terminal command:

[90] http://localhost:1337/messages/list.json

```
$ curl http://127.0.0.1:1337/messages/list.json
```

To make the POST request by using a command line interface:

```
curl -d "name=BOB&message=test" http://127.0.0.1:1337/messages/create.json
```

And you should get the output in a server terminal window and a new message "test" when you refresh localhost:1337/messages/list.json[91]. Needless to say, all three tests should pass.

Your application might grow bigger with more methods, URL paths to parse and conditions. That is where frameworks come in handy. They provide helpers to process requests and other nice things like static file support, sessions, etc. In this example, we intentionally didn't use any frameworks like Express (http://expressjs.com/) or Restify (http://mcavage.github.com/node-restify/). Other notable Node.js frameworks:

- Derby[92]: MVC framework making it easy to write real-time, collaborative applications that run in both Node.js and browsers
- Express.js[93]: the most robust, tested and used Node.js framework
- Restify[94]: lightweight framework for REST API servers
- Sails.js[95]: MVC Node.js framework
- hapi[96]: Node.js framework built on top of Express.js
- Connect[97]: a middleware framework for node, shipping with over 18 bundled middlewares and a rich selection of third-party middleware
- GeddyJS[98]: a simple, structured MVC web framework for Node
- CompoundJS[99] (ex-RailswayJS): Node.JS MVC framework based on ExpressJS
- Tower.js[100]: a full stack web framework for Node.js and the browser
- Meteor[101]: open-source platform for building top-quality web apps in a fraction of the time

Ways to improve the application:

- Improve existing test cases by adding object comparison instead of a string one
- Move the seed data to **test.js** from **mb-server.js**

[91] http://localhost:1337/messages/list.json
[92] http://derbyjs.com/
[93] http://expressjs.com
[94] http://mcavage.github.com/node-restify/
[95] http://sailsjs.org/
[96] http://spumko.github.io/
[97] http://www.senchalabs.org/connect/
[98] http://geddyjs.org
[99] http://compoundjs.com/
[100] http://towerjs.org
[101] http://meteor.com

- Add test cases to support your front-end, e.g., up-vote, user log in
- Add methods to support your front-end, e.g., up-vote, user log in
- Generate unique IDs for each message and store them in a Hash instead of an Array
- Install Mocha and re-factor test.js so it uses this library

So far we've been storing our messages in the application memory, so each time the application is restarted, we lose it. To fix it, we need to add a persistence, and one of the ways is to use a database like MongoDB.

MongoDB

This section covers:

- MongoDB Shell
- MongoDB Native Driver
- MongoDB on Heroku: MongoHQ
- BSON

MongoDB Shell

If you haven't done so already, please install the latest version of MongoDB from mongodb.org/downloads[102]. For more instructions, please refer to the *Setup, Database: MongoDB* section.

Now from the folder where you unpacked the archive, launch the **mongod** service with:

```
1 $ ./bin/mongod
```

You should be able to see information in your terminal and in the browser at localhost:28017[103].

For the MongoDB shell, or **mongo**, launch in a new terminal window (**important!**), and at the same folder this command:

```
1 $ ./bin/mongo
```

You should see something like this, depending on your version of the MongoDB shell:

[102] http://www.mongodb.org/downloads
[103] http://localhost:28017

```
MongoDB shell version: 2.0.6
connecting to: test
```

To test the database, use the JavaScript-like interface and commands **save** and **find**:

```
> db.test.save( { a: 1 } )
> db.test.find()
```

More detailed step-by-step instructions are available at Setup, Database: MongoDB.

Some other useful MongoDB shell commands:

```
> help
> show dbs
> use board
> show collections
> db.messages.remove();
> var a = db.messages.findOne();
> printjson(a);
> a.message = "hi";
> db.messages.save(a);
> db.messages.find({});
> db.messages.update({name: "John"},{$set: {message: "bye"}});
> db.messages.find({name: "John"});
> db.messages.remove({name: "John"});
```

A full overview of the MongoDB interactive shell is available at mongodb.org: Overview - The MongoDB Interactive Shell[104].

MongoDB Native Driver

We'll use Node.js Native Driver for MongoDB (https://github.com/christkv/node-mongodb-native) to access MongoDB from Node.js applications. Full documentation is also available at http://mongodb.github.com/node-mongodb-native/api-generated/db.html.

To install MongoDB Native driver for Node.js, use:

```
$ npm install mongodb
```

More details are at http://www.mongodb.org/display/DOCS/node.JS.

Don't forget to include the dependency in the **package.json** file as well:

[104] http://www.mongodb.org/display/DOCS/Overview+-+The+MongoDB+Interactive+Shell

```
{
  "name": "node-example",
  "version": "0.0.1",
  "dependencies": {
        "mongodb":"",
        ...
  },
  "engines": {
    "node": ">=0.6.x"
  }
}
```

Alternatively, for your own development you could use other mappers, which are available as an extension of the Native Driver:

- Mongoskin[105]: the future layer for node-mongodb-native
- Mongoose[106]: asynchronous JavaScript driver with optional support for modeling
- Mongolia[107]: lightweight MongoDB ORM/driver wrapper
- Monk[108]: Monk is a tiny layer that provides simple yet substantial usability improvements for MongoDB usage within Node.js

This small example will test if we can connect to local MongoDB instance from a Node.js script.

After we installed the library, we can include the **mongodb** library in our **db.js** file:

```
var util = require('util');
var mongodb = require ('mongodb');
```

This is one of the ways to establish connection to the MongoDB server in which the db variable will hold reference to the database at a specified host and port:

[105] https://github.com/guileen/node-mongoskin
[106] http://mongoosejs.com/
[107] https://github.com/masylum/mongolia
[108] https://github.com/LearnBoost/monk

```
var Db = mongodb.Db;
var Connection = mongodb.Connection;
var Server = mongodb.Server;
var host = '127.0.0.1';
var port = 27017;

var db=new Db ('test', new Server(host,port, {}));
```

To actually open a connection:

```
db.open(function(e,c){
        //do something with the database here
  // console.log (util.inspect(db));
  console.log(db._state);
  db.close();
});
```

This code snippet is available under the rpjs/db/db.js[109] folder. If we run it, it should output "**connected**" in the terminal. When you're in doubt and need to check the properties of an object, there is a useful method in the **util** module:

```
console.log(util.inspect(db));
```

MongoDB on Heroku: MongoHQ

After you made your application that displays 'connected' work locally, it's time to slightly modify it and deploy to the platform as a service, i.e., Heroku.

We recommend using the MongoHQ add-on[110], which is a part of MongoHQ[111] technology. It provides a browser-based GUI to look up and manipulate the data and collections. More information is available at https://devcenter.heroku.com/articles/mongohq.

Note

You might have to provide your credit card information to use MongoHQ even if you select the free version. You should not be charged, though.

In order to connect to the database server, there is a database connection URL (a.k.a. MongoHQ URL/URI), which is a way to transfer all of the necessary information to make a connection to the database in one string.

The database connection string MONGOHQ_URL has the following format:

[109] https://github.com/azat-co/rpjs/blob/master/db/db.js
[110] https://addons.heroku.com/mongohq
[111] https://www.mongohq.com/home

```
mongodb://user:pass@server.mongohq.com/db_name
```

You could either copy the MongoHQ URL string from the Heroku website (and hard-code it) or get the string from the Node.js **process.env** object:

```
process.env.MONGOHQ_URL;
```

or

```
var connectionUri = url.parse(process.env.MONGOHQ_URL);
```

Tip

The global object process gives access to environment variables via process.env. Those variables conventionally used to pass database host names and ports, passwords, API keys, port numbers, and other system information that shouldn't be hard-coded into the main logic.

To make our code work both locally and on Heroku, we can use the logical OR operator `||` and assign a local host and port if environment variables are undefined:

```
var port = process.env.PORT || 1337;
var dbConnUrl = process.env.MONGOHQ_URL ||
  'mongodb://@127.0.0.1:27017';
```

Here is out updated cross-environment ready **db.js** file:

```
var url = require('url')
var util = require('util');
var mongodb = require ('mongodb');
var Db = mongodb.Db;
var Connection = mongodb.Connection;
var Server = mongodb.Server;

var dbConnUrl = process.env.MONGOHQ_URL ||
'mongodb://127.0.0.1:27017';
var host = url.parse(dbConnUrl).hostname;
var port = new Number(url.parse(dbConnUrl).port);

var db=new Db ('test', new Server(host,port, {}));
```

```
14 db.open(function(e,c){
15   // console.log (util.inspect(db));
16   console.log(db._state);
17   db.close();
18 });
```

Following the modification of **db.js** by addition of **MONGOHQ_URL**, we can now initialize Git repository, create a Heroku app, add the MongoHQ add-on to it and deploy the app with Git.

Utilize the same steps as in the previous examples to create a new git repository:

```
1 git init
2 git add .
3 git commit -am 'initial commit'
```

Create the Cedar stack Heroku app:

```
1 $ heroku create
```

If everything went well you should be able to see a message that tell you the new Heroku app name (and URL) along with a message that remote was added. Having remote in your local git is crucial, you can always check a list of remote by:

```
1 git remote show
```

To install free MongoHQ on the existing Heroku app (add-ons work on a per app basis), use:

```
1 $ heroku addons:add mongohq:sandbox
```

Or log on to addons.heroku.com/mongohq[112] with your Heroku credentials and choose MongoHQ Free for that particular Heroku app, if you know the name of that app.

If you get **db.js** and modified **db.js** files working, let's enhance by adding a HTTP server, so the 'connected' message will be displayed in the browser instead of the terminal window. To do so, we'll wrap the server object instanciation in a database connection callback:

[112] https://addons.heroku.com/mongohq

```
...
db.open(function(e,c){
        // console.log (util.inspect(db));
        var server = http.createServer(function (req, res) {
    //creates server
          res.writeHead(200, {'Content-Type': 'text/plain'});
    //sets the right header and status code
          res.end(db._state);
    //outputs string with line end symbol
        });
        server.listen(port, function() {
                console.log('Server is running at %s:%s '
    , server.address().address
    , server.address().port);
    //sets port and IP address of the server
        });
        db.close();
});
...
```

The final deployment-ready file **app.js** from rpjs/db[113]:

```
/*
Rapid Prototyping with JS is a JavaScript
and Node.js book that will teach you how to build mobile
and web apps fast. — Read more at
http://rapidprototypingwithjs.com.
*/
var util = require('util');
var url = require('url');
var http = require('http');
var mongodb = require('mongodb');
var Db = mongodb.Db;
var Connection = mongodb.Connection;
var Server = mongodb.Server;
var port = process.env.PORT || 1337;
var dbConnUrl = process.env.MONGOHQ_URL ||
  'mongodb://@127.0.0.1:27017';
var dbHost = url.parse(dbConnUrl).hostname;
var dbPort = new Number(url.parse(dbConnUrl).port);
console.log(dbHost + dbPort)
```

[113] https://github.com/azat-co/rpjs/blob/master/db

```
20 var db = new Db('test', new Server(dbHost, dbPort, {}));
21 db.open(function(e, c) {
22   // console.log (util.inspect(db));
23   // creates server
24   var server = http.createServer(function(req, res) {
25     //sets the right header and status code
26     res.writeHead(200, {
27       'Content-Type': 'text/plain'
28     });
29     //outputs string with line end symbol
30     res.end(db._state);
31   });
32   //sets port and IP address of the server
33   server.listen(port, function() {
34     console.log(
35       'Server is running at %s:%s ',
36     server.address().address,
37     server.address().port);
38   });
39   db.close();
40 });
```

After the deployment you should be able to open the URL provided by Heroku and see the message 'connected'.

Here is the manual on how to use MongoDB from Node.js code: mongodb.github.com/node-mongodb-native/api-articles/nodekoarticle1.html[114].

Another approach is to use the MongoHQ Module, which is available at github.com/MongoHQ/mongohq-nodejs[115].

This example illustrates a different use of the **mongodb** library by outputting collections and a document count. The full source code from rpjs/db/collections.js[116]:

[114] http://mongodb.github.com/node-mongodb-native/api-articles/nodekoarticle1.html
[115] https://github.com/MongoHQ/mongohq-nodejs
[116] https://github.com/azat-co/rpjs/blob/master/db/collections.js

```
var mongodb = require('mongodb');
var url = require('url');
var log = console.log;
var dbUri = process.env.MONGOHQ_URL || 'mongodb://localhost:27017/test';

var connectionUri = url.parse(dbUri);
var dbName = connectionUri.pathname.replace(/^\//, '');

mongodb.Db.connect(dbUri, function(error, client) {
  if (error) throw error;

  client.collectionNames(function(error, names){
    if(error) throw error;

    //output all collection names
    log("Collections");
    log("===========");
    var lastCollection = null;
    names.forEach(function(colData){
      var colName = colData.name.replace(dbName + ".", '')
      log(colName);
      lastCollection = colName;
    });
    if (!lastCollection) return;
    var collection = new mongodb.Collection(client, lastCollection);
    log("\nDocuments in " + lastCollection);
    var documents = collection.find({}, {limit:5});

    //output a count of all documents found
    documents.count(function(error, count){
      log("  " + count + " documents(s) found");
      log("====================");

      // output the first 5 documents
      documents.toArray(function(error, docs) {
        if(error) throw error;

        docs.forEach(function(doc){
          log(doc);
        });

        // close the connection
```

```
43          client.close();
44        });
45      });
46    });
47  });
```

We've used a shortcut for `console.log()` with `var log = console.log;` and `return` as a control flow via `if (!lastCollection) return;`.

BSON

Binary JSON, or BSON, it is a special data type which MongoDB utilizes. It is like to JSON in notation, but has support for additional more sophisticated data types.

Warning

A word of caution about BSON: ObjectId in MongoDB is an equivalent to ObjectID in MongoDB Native Node.js Driver, i.e., make sure to use the proper case. Otherwise you'll get an error. More on the types: ObjectId in MongoDB[117] vs Data Types in MongoDB Native Node.js Drier[118]. Example of Node.js code with `mongodb.ObjectID()`: `collection.findOne({_id: new ObjectID(idString)}, console.log) // ok.` On the other hand, in the MongoDB shell, we employ: `db.messages.findOne({_id:ObjectId(idStr)});`.

[117] http://www.mongodb.org/display/DOCS/Object+IDs

[118] https://github.com/mongodb/node-mongodb-native/#data-types

Chat REST API Server (DB Store)

We should have everything set up for writing the Node.js application which will work both locally and on Heroku. The source code is available under rpjs/mongo[119]. The structure of the application is simple:

```
/mongo
  -web.js
  -Procfile
  -package.json
```

This is what *web.js* looks like; first we include our libraries:

```
var http = require('http');
var util = require('util');
var querystring = require('querystring');
var mongo = require('mongodb');
```

Then put out a magic string to connect to MongoDB:

```
var host = process.env.MONGOHQ_URL || "mongodb://@127.0.0.1:27017/twitter-clone";
//MONGOHQ_URL=mongodb://user:pass@server.mongohq.com/db_name
```

Note

The URI/URL format contains the optional database name in which our collection will be stored. Feel free to change it to something else, for example 'rpjs' or 'test'.

We put all the logic inside of an open connection in the form of a callback function:

[119] https://github.com/azat-co/rpjs/tree/master/mongo

```
mongo.Db.connect(host, function(error, client) {
  if (error) throw error;
  var collection = new mongo.Collection(
    client,
    'messages');
  var app = http.createServer(function(request, response) {
    if (request.method === "GET" &&
      request.url === "/messages/list.json") {
      collection.
        find().
        toArray(function(error,results) {
        response.writeHead(200,{
          'Content-Type':'text/plain'
        });
        console.dir(results);
        response.end(JSON.stringify(results));
      });
    };
    if (request.method === "POST" &&
      request.url === "/messages/create.json") {
      request.on('data', function(data) {
        collection.insert(
          querystring.parse(data.toString('utf-8')),
          {safe:true},
          function(error, obj) {
            if (error) throw error;
            response.end(JSON.stringify(obj));
          }
        )
      })
    };
  });
  var port = process.env.PORT || 5000;
  app.listen(port);
})
```

Note

We don't have to use additional words after the collection/entity name, i.e., instead of `/messages/list.json` and `/messages/create.json` it's perfectly fine to have just `/messages` for all the HTTP methods such as GET, POST, PUT, DELETE. If you change them in your application code make sure to use the updated CURL commands and front-end code.

To test via CURL terminal commands run:

```
curl http://localhost:5000/messages/list.json
```

Or open your browser at the http://locahost:5000/messages/list.json location.

It should give you an empty array: `[]` which is fine. Then POST a new message:

```
curl  -d "username=BOB&message=test" http://localhost:5000/messages/create.json
```

Now we must see a response containing an ObjectID of a newly created element, for example: `[{"username":"BOB","message":"test","_id":"51edcad45862430000000001"}]`. Your ObjectId might vary.

If everything works as it should locally, try to deploy it to Heroku.

To test the application on Heroku, you could use the same CURL[120] commands, substituting `http://localhost/` or "http://127.0.0.1" with your unique Heroku app's host/URL:

```
$ curl http://your-app-name.herokuapp.com/messages/list.json
$ curl -d "username=BOB&message=test"
  http://your-app-name.herokuapp.com/messages/create.json
```

It's also nice to double check the database either via Mongo shell: `$ mongo` terminal command and then `use twitter-clone` and `db.messages.find()`; or via MongoHub[121], mongoui[122], mongo-express[123] or in case of MongoHQ through its web interface accessible at heroku.com website.

If you would like to use another domain name instead of http://your-app-name.herokuapp.com, you'll need to do two things:

1. Tell Heroku your domain name:

```
$ heroku domains:add www.your-domain-name.com
```

2. Add the CNAME DNS record in your DNS manager to point to http://your-app-name.herokuapp.com.

More information on custom domains can be found at devcenter.heroku.com/articles/custom-domains[124].

[120] http://curl.haxx.se/docs/manpage.html
[121] https://github.com/bububa/MongoHub-Mac
[122] https://github.com/azat-co/mongoui
[123] https://github.com/andzdroid/mongo-express
[124] https://devcenter.heroku.com/articles/custom-domains

Tip

For more productive and efficient development we should automate as much as possible, i.e., use tests instead of CURL commands. There is an article on the Mocha library in the BONUS chapter which, along with the superagent or request libraries, is a timesaver for such tasks.

Express.js 4, Node.js and MongoDB REST API Tutorial

Here's a brand new, revisited tutorial for **Express.js 4**, **Node.js and MongoDB** (Mongoskin) free-JSON RESTful API server and testing it with Mocha and Superagent,.

The code for this new tutorial is available at github.com/azat-co/rest-api-express[125] (`master` branch). The old tutorial's code for Express 3.x, is still working and in the `express3` branch.

The Express.js 4 and MongoDB REST API tutorial consists of these parts:

1. Node.js and MongoDB REST API Overview
2. REST API Tests with Mocha and Superagent
3. NPM-ing Node.js Server Dependencies
4. Express.js 4.x Middleware Caveat
5. Express.js and MongoDB (Mongoskin) Implementation
6. Running The Express.js 4 App and Testing MongoDB Data with Mocha
7. Conclusion and Further Express.js and Node.js Reading

Instead of TL;DR:

If you're only interested in a working code from the repository and **know what to do**, here are brief instructions on how to download and run the REST API server:

```
$ git clone git@github.com:azat-co/rest-api-express.git
$ npm install
$ node express.js
```

Start MongoDB with `$ mongod`. Then, in a new terminal window run the Mocha tests:

```
$ mocha express.test.js
```

Or, if you don't have mocha installed globally:

[125] https://github.com/azat-co/rest-api-express

```
$ ./node_modules/mocha/bin/mocha express.test.js
```

Node.js and MongoDB REST API Overview

This Node.js, Express.js and MongoDB (Mongoskin) tutorial will walk you through writing the test using the Mocha[126] and Super Agent[127] libraries. This is needed for a test-driven development building of a Node.js[128] free JSON REST API server.

The server application itself will utilize Express.js[129] 4.x framework and Mongoskin[130] library for MongoDB[131]. In this REST API server, we'll perform **create**, **read**, **update and delete** (CRUD) operations and harness Express.js middleware[132] concept with `app.param()` and `app.use()` methods.

First of all, make sure you have MongoDB installed. You can follow the steps on the official website[133].

We'll be using the following versions of libraries:

- `express`: ~4.1.1
- `body-parser`: ~1.0.2
- `mongoskin`: ~1.4.1
- `expect.js`: ~0.3.1
- `mocha`: ~1.18.2
- `superagent`: ~0.17.0

If you try to attempt to use later or older versions the code might not work. :-(

REST API Tests with Mocha and Superagent

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[134] 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[135] and MongoDB installed, let's create a *new* folder (or if you wrote the tests use that folder):

[126] http://visionmedia.github.io/mocha/
[127] http://visionmedia.github.io/superagent/
[128] http://nodejs.org
[129] http://expressjs.com/
[130] https://github.com/kissjs/node-mongoskin
[131] http://www.mongodb.org/
[132] http://expressjs.com/api.html#middleware
[133] http://www.mongodb.org/downloads
[134] http://visionmedia.github.io/mocha/
[135] http://npmjs.org

```
$ mkdir rest-api
$ cd rest-api
```

We'll use Mocha[136], Expect.js[137] and Super Agent[138] libraries. To install them, run these commands from the project folder:

```
$ npm install mocha@1.18.2 --save-dev
$ npm install expect.js@0.3.1 --save-dev
$ npm install superagent@0.17.0 --save-dev
```

Note: You can also install Mocha globally with the `-g` flag.

Now, let's create the **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. To keep this tutorial focused on the REST API with Express.js 4 and MongoDB, and not on Mocha, we won't go into the details of test suits. Feel free to copy and paste the code!

Here is the full source code for the **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){
```

[136] http://visionmedia.github.io/mocha/
[137] https://github.com/LearnBoost/expect.js/
[138] http://visionmedia.github.io/superagent/

```
13          // console.log(res.body)
14          expect(e).to.eql(null)
15          expect(res.body.length).to.eql(1)
16          expect(res.body[0]._id.length).to.eql(24)
17          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
42          done()
43        })
44    })
45
46    it('updates an object', function(done){
47      superagent.put('http://localhost:3000/collections/test/'+id)
48        .send({name: 'Peter'
49          , email: 'peter@yahoo.com'})
50        .end(function(e, res){
51          // console.log(res.body)
52          expect(e).to.eql(null)
53          expect(typeof res.body).to.eql('object')
54          expect(res.body.msg).to.eql('success')
```

```
55          done()
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
71    it('removes an object', function(done){
72      superagent.del('http://localhost:3000/collections/test/'+id)
73        .end(function(e, res){
74          // console.log(res.body)
75          expect(e).to.eql(null)
76          expect(typeof res.body).to.eql('object')
77          expect(res.body.msg).to.eql('success')
78          done()
79        })
80    })
81  })
```

To run the tests, we can use the `$ mocha express.test.js` command (if you have Mocha globally) or `$ ./node_modules/mocha/bin/mocha express.test.js`.

NPM-ing Node.js Server Dependencies

In this tutorial, we'll utilize Mongoskin[139], a MongoDB library which is a better alternative to the plain, good old native MongoDB driver for Node.js[140]. In addition, Mongoskin is more lightweight than Mongoose and schema-less. For more insight, please check out Mongoskin comparison blurb[141].

[139] https://github.com/kissjs/node-mongoskin
[140] https://github.com/mongodb/node-mongodb-native
[141] https://github.com/kissjs/node-mongoskin#comparation

Express.js[142] is a wrapper for the core Node.js HTTP module[143] objects. The Express.js framework is built on top of Connect[144] middleware and provides 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:

```
$ npm install express@4.1.1 --save
$ npm install mongoskin@1.4.1 --save
```

Express.js 4.x Middleware Caveat

Sadly, just NPM-ing `express` is not enough anymore for building minimal REST API servers with Express.js, because in version 4.x **the middlewares are not bundled** with the framework! Developers have to install separate modules, except for `express.static`, which was left in the Express.js 4.x. So to parse incoming information, we add `body-parser`:

```
$ npm install mongoskin@1.0.2 --save
```

Express.js 4 and MongoDB (Mongoskin) Implementation

First thing's first, so let's define our dependencies:

```
var express = require('express')
  , mongoskin = require('mongoskin')
  , bodyParser = require('body-parser')
```

After the version 3.x (this of course includes v4), Express.js streamlines the instantiation of its app instance, 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:

[142] http://expressjs.com/
[143] http://nodejs.org/api/http.html
[144] https://github.com/senchalabs/connect

```
app.use(bodyParser())
```

Middleware (in this[145] and other forms[146]) is a powerful and convenient pattern in Express.js and Connect[147] 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[148] makes it possible to connect to the MongoDB database in one effortless line of code:

```
var db = mongoskin.db('mongodb://@localhost:27017/test', {safe:true})
```

Note: If you wish to connect to a remote database, e.g., MongoHQ[149] 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][?optio`

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 sting `collectionName` prefixed with a colon (you'll see it later in the routes). Then, we save that collection as a property (`collection` but could be anything) of the request object (widespread `req`), which will be available in the next request handlers:

```
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) {
  res.send('please select a collection, e.g., /collections/messages')
})
```

Now the real work begins, here is how we retrieve a list of any items (first parameter is an empty object `{}` which means *any*). The results will be capped at a limit of 10 and sorted by `_id` (second parameter). The `find()` method returns a cursor, so we call `toArray()` to get the JavaScript/Node.js array:

[145] http://expressjs.com/api.html#app.use
[146] http://expressjs.com/api.html#middleware
[147] https://github.com/senchalabs/connect
[148] https://github.com/kissjs/node-mongoskin
[149] https://www.mongohq.com/home

```
app.get('/collections/:collectionName', function(req, res, next) {
  req.collection.find({} ,{limit:10, sort: [['_id',-1]]}).toArray(function(e, re\
sults){
    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. This method often called free JSON REST API because server and then the database accept any data structure. Parse.com and other Back-end as a Service providers pioneered the free JSON approach. In our Express.js app, we use `req.body` for this:

```
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 like `findById` and `findOne` are faster than `find()`, but they use a different interface (they return the 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.findById(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 us 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.updateById(req.params.id, {$set:req.body}, {safe:true, multi:fa\
lse}, function(e, result){
    if (e) return next(e)
    res.send((result===1)?{msg:'success'}:{msg:'error'})
  })
})
```

Finally, the DELETE HTTP method is processed by `app.del()`. In the request handler, we utilize `removeById()` which does exactly what it sounds like it should do, and takes an ID and a callback. Then, we output a custom JSON message `success` on the deletion:

```
app.del('/collections/:collectionName/:id', function(req, res, next) {
  req.collection.remove({_id: req.collection.id(req.params.id)}, function(e, res\
ult){
    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 the **express.js** file (also check with the GitHub which is sometimes more up-to-date and workable):

```
var express = require('express')
  , mongoskin = require('mongoskin')
  , bodyParser = require('body-parser')

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

var db = mongoskin.db('mongodb://@localhost:27017/test', {safe:true})

app.param('collectionName', function(req, res, next, collectionName){
  req.collection = db.collection(collectionName)
  return next()
})
```

```
14
15  app.get('/', function(req, res, next) {
16    res.send('please select a collection, e.g., /collections/messages')
17  })
18
19  app.get('/collections/:collectionName', function(req, res, next) {
20    req.collection.find({} ,{limit:10, sort: [['_id',-1]]}).toArray(function(e, re\
21  sults){
22      if (e) return next(e)
23      res.send(results)
24    })
25  })
26
27  app.post('/collections/:collectionName', function(req, res, next) {
28    req.collection.insert(req.body, {}, function(e, results){
29      if (e) return next(e)
30      res.send(results)
31    })
32  })
33
34  app.get('/collections/:collectionName/:id', function(req, res, next) {
35    req.collection.findById(req.params.id, function(e, result){
36      if (e) return next(e)
37      res.send(result)
38    })
39  })
40
41  app.put('/collections/:collectionName/:id', function(req, res, next) {
42    req.collection.updateById(req.params.id, {$set:req.body}, {safe:true, multi:fa\
43  lse}, function(e, result){
44      if (e) return next(e)
45      res.send((result===1)?{msg:'success'}:{msg:'error'})
46    })
47  })
48
49  app.del('/collections/:collectionName/:id', function(req, res, next) {
50    req.collection.removeById(req.params.id, function(e, result){
51      if (e) return next(e)
52      res.send((result===1)?{msg:'success'}:{msg:'error'})
53    })
54  })
55
```

```
56 app.listen(3000)
```

Save the code and exit your editor, because we're done with our small Express.js REST API server.

Running The Express.js 4 App and Testing MongoDB Data with Mocha

Now, assuming you have MongoDB installed and running (`$ mongod`), we should be able to run this in your terminal (separate window from `mongod`):

```
1 $ node express.js
```

And in a different window (without closing the first one):

```
1 $ mocha express.test.js
```

Or, if you don't have mocha installed globally:

```
1 $ ./node_modules/mocha/bin/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:

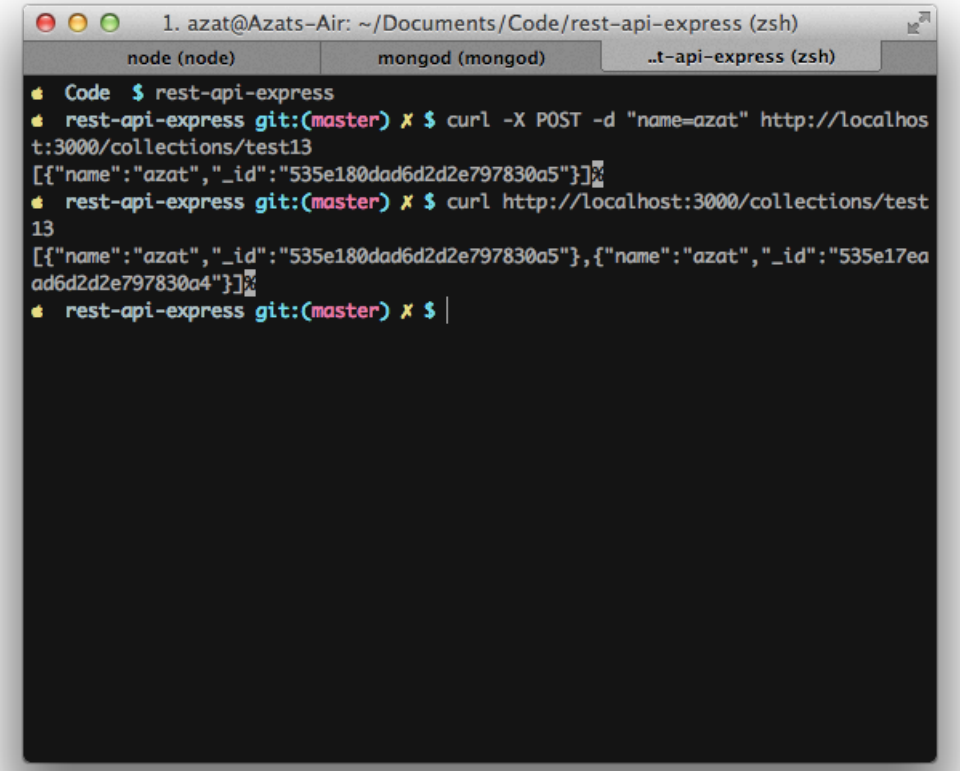
```
1 $ curl -X POST -d "name=azat" http://localhost:3000/collections/test13
```

And the result should look something like this:

```
1 {"name":"azat","_id":"535e180dad6d2d2e797830a5"}]%
```

We can easily check this object either by using our REST API server:

```
1 $ curl http://localhost:3000/collections/test13
```



Using CURL with Express 4 and MongoDB REST API

GET requests also work in the browser. For example, open this link while your local server is running on port 3000 http://localhost:3000/collections/test13.

Or if we don't trust server results (why wouldn't we? but let's pretend we don't), we can open MongoDB shell (`$ mongo`) and type:

```
1  > db.test13.find()
```

Note: If you changed the database name and it's not `test`, then precede the above command with `> use your_database_name`.

In this tutorial, our tests are longer than the app code itself. For some it might be tempting to abandon the test-driven development, but believe me **the good habits of TDD will save you hours and hours** during any serious development when the complexity of the applications you work on is big.

Conclusion and Further Express.js and Node.js Reading

The Express.js 4 and MongoDB/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.

Note

The full source code for `express.text.js`, `express.js` and `package.json` is available at github.com/azat-co/rest-api-express[150].

Note: In this example I'm using semi-colon less style. Semi-colons in JavaScript are absolutely optional[151] except in two cases: in the for loop and before expression/statement that starts with parenthesis (e.g., Immediately-Invoked Function Expression[152]).

[150] https://github.com/azat-co/rest-api-express

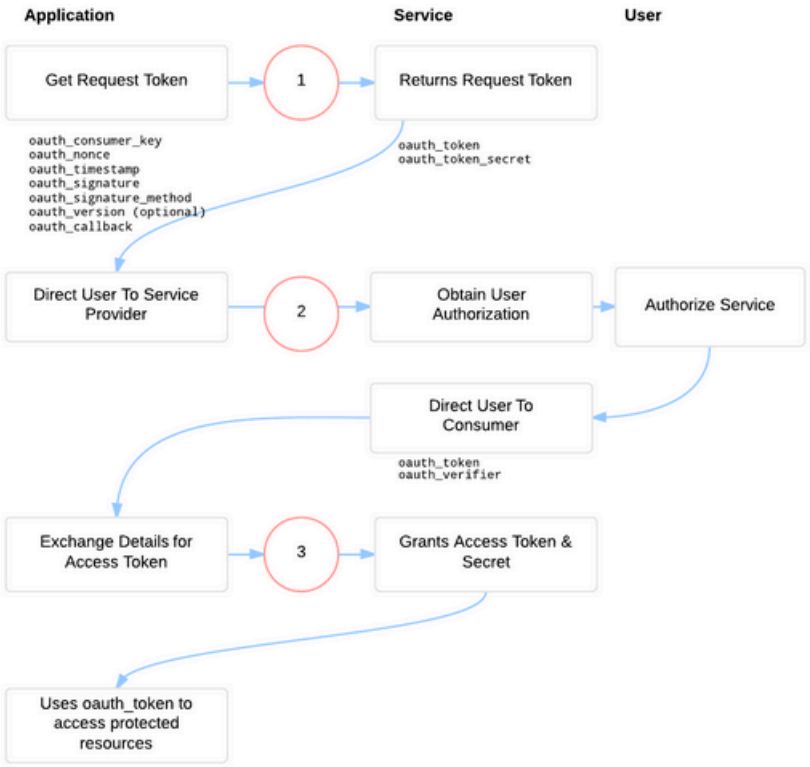
[151] http://blog.izs.me/post/2353458699/an-open-letter-to-javascript-leaders-regarding

[152] http://en.wikipedia.org/wiki/Immediately-invoked_function_expression

OAuth 1.0 Sign in with Everyauth

While it is possible to implement the OAuth 1.0 *Sign in*, or three-legged OAuth 1.0, with just the `oauth` module, that's not what most developers do. They tend to favor modules like Everyauth[153] or Passport[154], because these modules have strategies (support) for many service providers (Twitter, Google, Facebook, etc.). With these strategies, it's very easy to add several Sign ins at once.

We'll write a small Express.js app that has a *Sign in* link that allows to authenticate users. As a result, the app will be able to output Twitter user information after the authentication, and get access to protected Twitter resources on behalf of this particular user. The *Sign in*, or three-legged OAuth 1.0 flow, is illustrated below.



Three-legged OAuth 1.0 flow

The `package.json` for this project looks like this:

[153] http://everyauth.com/

[154] http://passportjs.org/

```
{
  "name": "everyauth-twitter",
  "version": "0.0.6",
  "private": true,
  "scripts": {
    "start": "node app.js",
    "test": "mocha test"
  },
  "dependencies": {
    "express": "4.1.2",
    "jade": "1.3.1",
    "stylus": "0.44.0",
    "mongoskin": "1.4.1",
    "cookie-parser": "1.0.1",
    "body-parser": "1.0.2",
    "method-override": "1.0.0",
    "serve-favicon": "2.0.0",
    "express-session": "1.0.4",
    "morgan": "1.0.1",
    "errorhandler": "1.0.1",
    "everyauth": "~0.4.5"
  },
  "devDependencies": {
  }
}
```

To install the dependencies, simply run:

```
$ npm install
```

Let's start with the `app.js` file, in which we get the key and secret from environment variables:

```
var twitterKey = process.env.TWITTER_CONSUMER_KEY
var twitterSecret = process.env.TWITTER_CONSUMER_SECRET
```

Next, we import the modules:

```
var express = require('express'),
  routes = require('./routes'),
  http = require('http'),
  path = require('path'),
  mongoskin = require('mongoskin'),
```

We configure the database:

```
  dbUrl = process.env.MONGOHQ_URL
    ||'mongodb://@localhost:27017/blog',
  db = mongoskin.db(
    dbUrl,
    {safe: true}
  ),
  collections = {
    users: db.collection('users')
  }
```

and import `everyauth`:

```
  everyauth = require('everyauth');
```

Now, import the middleware:

```
// Express.js Middleware
var session = require('express-session'),
  logger = require('morgan'),
  errorHandler = require('errorhandler'),
  cookieParser = require('cookie-parser'),
  bodyParser = require('body-parser'),
  methodOverride = require('method-override');
```

Set `everyauth` in debug mode:

```
everyauth.debug = true;
```

Start the chained configuration for our Twitter *Sign in*:

```
everyauth.twitter
  .consumerKey(twitterKey)
  .consumerSecret(twitterSecret)
```

Now we can configure what follows once we get the user information back from Twitter, i.e., when the user has been authenticated successfully. In this example, we store the date in the session (`session.user`), but we can also save it to the database at the same time:

```
  .findOrCreateUser(function (session,
    accessToken,
    accessTokenSecret,
    twitterUserMetadata) {
      var promise = this.Promise();
      process.nextTick(function(){
        var username = twitterUserMetadata.screen_name;
        console.log('username is '+ username);
        session.user = twitterUserMetadata;
        session.protected = true;
        // Find or save to the database
        // with collections.users
        promise.fulfill(twitterUserMetadata);
      })
    return promise;
  })
```

The last chained method tells everyauth to redirect the user to the `/protected` page. This page can be something that you show only to logged-in users:

```
  .redirectPath('/protected');
```

Everyauth automatically adds the logout route to the Express.js app, but it doesn't know that we added some extra info to the session. So we need it to destroy the custom properties `session.user` and `session.protected`; otherwise the session will be kept alive:

```
everyauth.everymodule.handleLogout(routes.user.logout);
```

Because we arenâ€™t saving the user to the database in this example, we can just return the user right back in this `findUserById` setup:

```
everyauth.everymodule.findUserById(
  function (user, callback) {
   callback(user);
  }
);
```

Instantiate the Express.js object and assign the title:

```
var app = express();
app.locals.appTitle = 'everyauth-twitter';
```

Define the middleware to check for the `users` collection:

```
app.use(function(req, res, next) {
  if (! collections.users)
    return next(new Error('No collections.'));
  req.collections = collections;
  return next();
});
```

Next, there are standard Express.js configuration statements:

```
app.set('port', process.env.PORT || 3000);
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');
```

and some Express.js middleware configurations for logging requests and parsing payloads (request bodies):

```
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded());
```

Important! It's necessary to have the following middleware defined in this order:

```
app.use(cookieParser('3CCC4ACD-6ED1-4844-9217-82131BDCB239'));
app.use(session({
  secret: '2C44774A-D649-4D44-9535-46E296EF984F'
}))
app.use(everyauth.middleware());
```

The rest of the Express.js middleware:

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

Authentication middleware, in which we expose the `protected` flag to Jade templates, is as follows:

```
app.use(function(req, res, next) {
  if (req.session && req.session.protected)
    res.locals.protected = true;
  next();
});
```

Authorization middleware will protect unauthorized access to the protected pages:

```
var authorize = function(req, res, next) {
  if (req.session && req.session.auth)
    return next();
  else
    return res.send(401);
};
```

The default Express.js error-handling middleware is as follows:

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

We have only two pages and routes:

```
app.get('/', routes.index);
app.get('/protected', authorize, routes.admin);
```

Catch other routes with 404 status and start up the server:

```
app.all('*', function(req, res) {
  res.send(404);
})

var server = http.createServer(app);
var boot = function () {
  server.listen(app.get('port'), function(){
    console.info(
      'Express server listening on port '
       + app.get('port'));
  });
}
var shutdown = function() {
  server.close();
}
if (require.main === module) {
  boot();
} else {
  console.info('Running app as a module')
  exports.boot = boot;
  exports.shutdown = shutdown;
  exports.port = app.get('port');
}
```

For more information on Express.js, check out my up-coming *Pro Express.js (Apress, 2014)*[155] and *Express.js Guide: The Comprehensive Book on Express.js (LeanPub, 2013)*[156]

The full source code of the `everyauth/app.js` file is:

```
var twitterKey = process.env.TWITTER_CONSUMER_KEY
var twitterSecret = process.env.TWITTER_CONSUMER_SECRET

var express = require('express'),
  routes = require('./routes'),
  http = require('http'),
  path = require('path'),
  mongoskin = require('mongoskin'),
  dbUrl = process.env.MONGOHQ_URL
    ||'mongodb://@localhost:27017/blog',
```

[155] http://proexpressjs.com

[156] http://expressjsguide.com

```
11    db = mongoskin.db(
12      dbUrl,
13      {safe: true}
14    ),
15    collections = {
16      users: db.collection('users')
17    }
18    everyauth = require('everyauth');
19
20  // Express.js Middleware
21  var session = require('express-session'),
22    logger = require('morgan'),
23    errorHandler = require('errorhandler'),
24    cookieParser = require('cookie-parser'),
25    bodyParser = require('body-parser'),
26    methodOverride = require('method-override');
27
28  everyauth.debug = true;
29  everyauth.twitter
30    .consumerKey(twitterKey)
31    .consumerSecret(twitterSecret)
32    .findOrCreateUser(function (session,
33      accessToken,
34      accessTokenSecret,
35      twitterUserMetadata) {
36        var promise = this.Promise();
37        process.nextTick(function(){
38          var username = twitterUserMetadata.screen_name;
39          console.log('username is '+ username);
40          session.user = twitterUserMetadata;
41          session.protected = true;
42          // Find or save to the database
43          // with collections.users
44          promise.fulfill(twitterUserMetadata);
45        })
46      return promise;
47    })
48    .redirectPath('/protected');
49
50  // We need it to destroy custom properties session.user
51  // and session.protected, because otherwise
52  // the session will be kept alive
```

```
everyauth.everymodule.handleLogout(routes.user.logout);

everyauth.everymodule.findUserById(
  function (user, callback) {
   callback(user);
  }
);

var app = express();
app.locals.appTitle = 'everyauth-twitter';

app.use(function(req, res, next) {
  if (! collections.users)
    return next(new Error('No collections.'));
  req.collections = collections;
  return next();
});


// Express.js configurations
app.set('port', process.env.PORT || 3000);
app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

// Express.js middleware configuration
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded());
app.use(cookieParser('3CCC4ACD-6ED1-4844-9217-82131BDCB239'));
app.use(session({
  secret: '2C44774A-D649-4D44-9535-46E296EF984F'
}))
app.use(everyauth.middleware());
app.use(methodOverride());
app.use(require('stylus').middleware(__dirname + '/public'));
app.use(express.static(path.join(__dirname, 'public')));

// Authentication middleware
app.use(function(req, res, next) {
  if (req.session && req.session.protected)
    res.locals.protected = true;
  next();
```

```
});

// Authorization
var authorize = function(req, res, next) {
  if (req.session && req.session.auth)
    return next();
  else
    return res.send(401);
};

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

// Pages and routes
app.get('/', routes.index);
app.get('/protected', authorize, routes.admin);


app.all('*', function(req, res) {
  res.send(404);
})

var server = http.createServer(app);
var boot = function() {
  server.listen(app.get('port'), function(){
    console.info(
      'Express server listening on port '
       + app.get('port'));
  });
}
var shutdown = function() {
  server.close();
}
if (require.main === module) {
  boot();
} else {
  console.info('Running app as a module')
  exports.boot = boot;
  exports.shutdown = shutdown;
  exports.port = app.get('port');
}
```

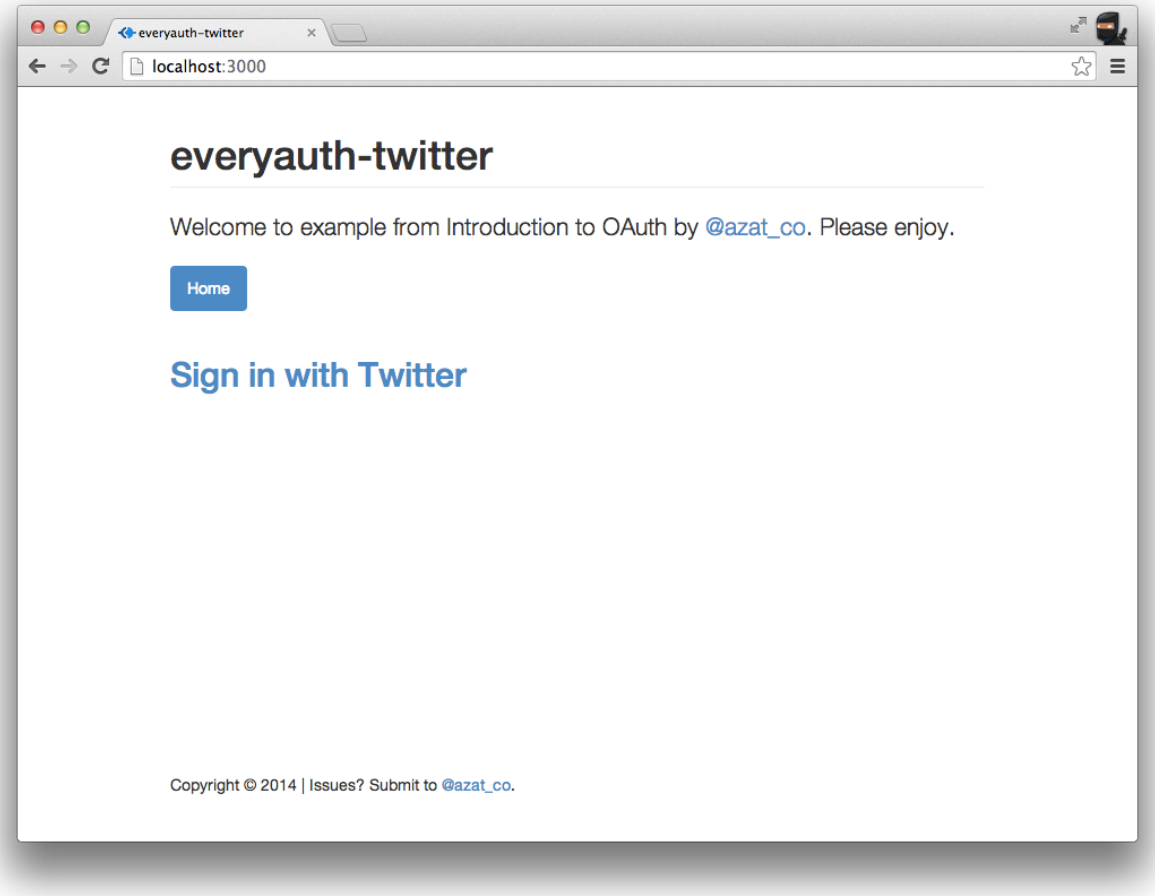
The `routes/index.js` file renders Jade templates for the corresponding pages:

```
exports.user = require('./user');

exports.index = function(req, res, next){
  res.render('index');
};

exports.admin = function(req, res, next) {
  console.log('user info: ', req.session.user)
  res.render('protected');
}
```

The `protected.jade` file outputs the `everyauth.twitter` value:

```
extends layout
block page
  - var menu = 'protected'
block content
  div.admin
    p This is your Twitter user information:
    textarea(cols="120", rows="40")= JSON.stringify(everyauth.twitter, 0, 2)
```

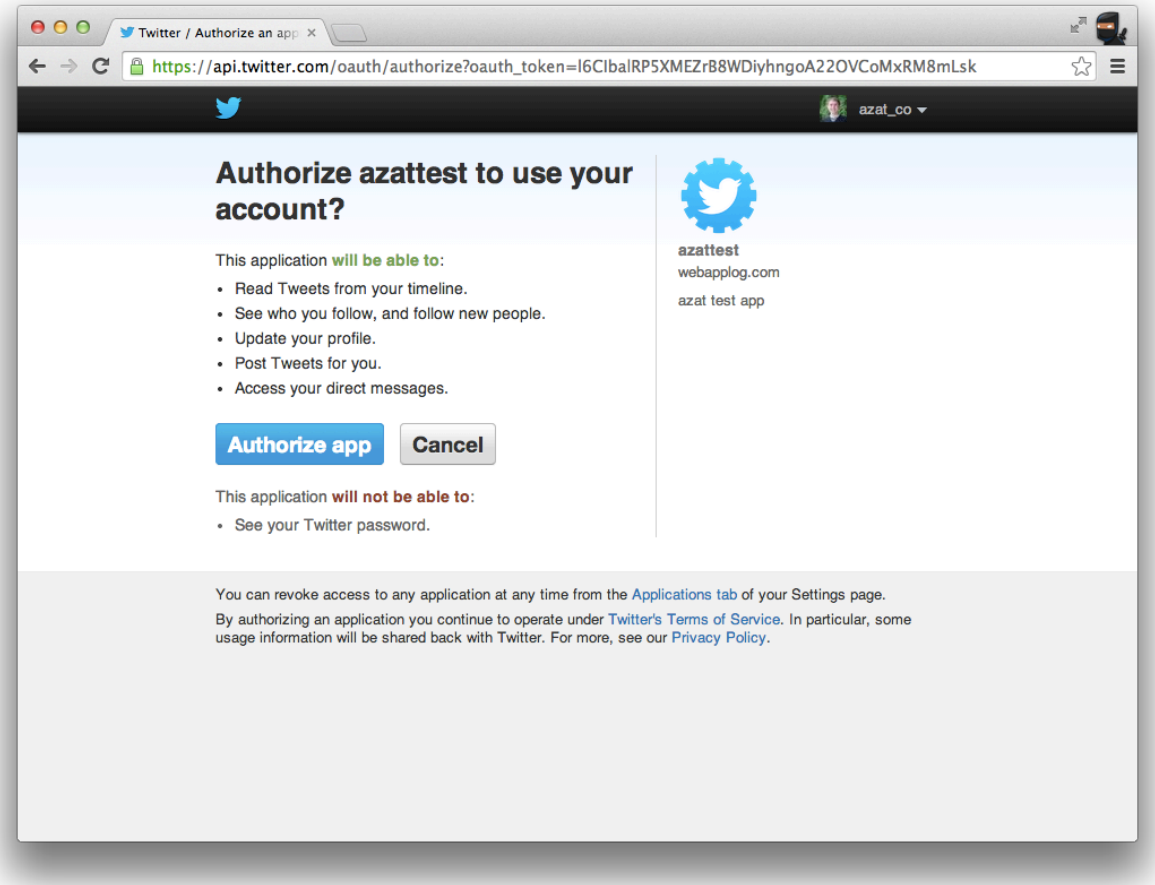
The next step is to start MongoDB and paste your own Twitter consumer key and secret. Then, if we run `$ node app.js` and open a browser at http://localhost:3000, we should see the home page as shown below.



Home page with the *Sign in* link

Notice that *Sign in* is just a regular anchor link to the `/auth/twitter` route. It can be a button or an image. If we didn't use `everyauth`, we'd have to implement the `/auth/twitter` and `/auth/twitter/callback` route ourselves.

After you click on the *Sign in* link, you'll be redirected to Twitter and prompted to authorize your application as shown in the figure below.

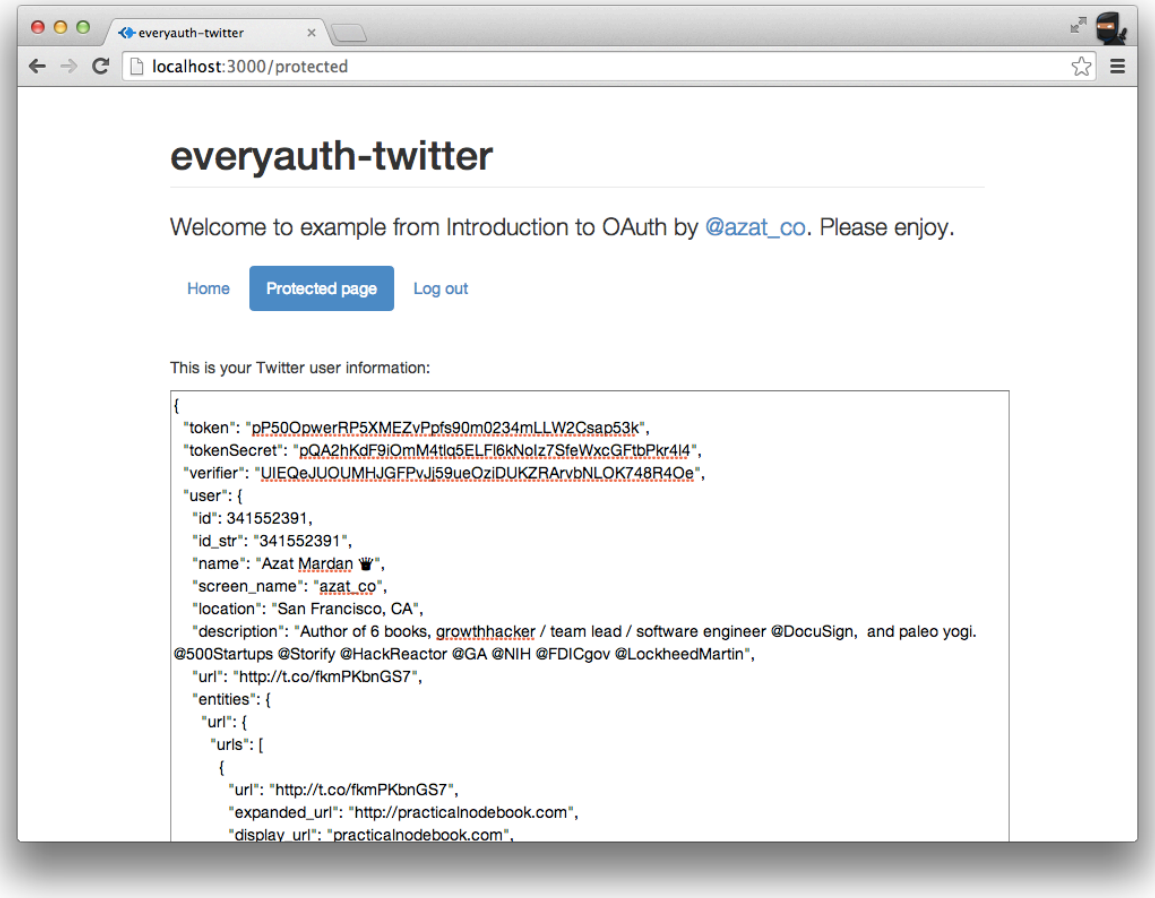


Authorize your app to sign in with Twitter

Twitter will then redirect you back to the localhost app with the authenticated response. If you open the Developer Tools in Chrome (or a similar console in Safari or Firefox), you can see for yourself the redirect request in the Network tab. The redirect is to:

```
1 http://localhost:3000/auth/twitter/callback
```

This request will have `oauth_token` and `oauth_verifier` parameters in the query string. And you'll see the protected page with your Twitter information. You can see an example of this using my data is shown in the figure below.



.

Now that we have the user's token and secret, we can store it in our own database to avoid asking for permissions each time. With this token and secret, all we need to do is perform one-legged OAuth 1.0 requests, as described in the first chapter.

Deploying to Amazon Web Services

Cloud is eating the world of computing. There are private and public clouds. Amazon Web Services (AWS), probably the most popular choice among the public cloud offerings, falls under the IaaS category. The advantages of using an IaaS such as AWS over PaaS-like Heroku are as follows:

1. It's more configurable (any services, packages, or operation systems).
2. It's more controllable. There are no restrictions or limitations.
3. It's cheaper to maintain. PaaS can quickly cost a fortune for high-performance resources.

In this tutorial, we use 64-bit Amazon Linux AMI (http://aws.amazon.com/amazon-linux-ami) with CentOS. It might be easier to use the Extra Packages for Enterprise Linux (EPEL) package manager to install Node.js and NPM. If you don't have EPEL, skip to the manual C++ build instructions.

Assuming you have your Elastic Compute Cloud (EC2) instance up and running, make an SSH connection into it and see if you have yum with EPEL (https://fedoraproject.org/wiki/EPEL). To do so, just see if this command says epel:

```
$ yum repolist
```

If there are no mentions of epel, run:

```
$ rpm -Uvh http://download-i2.fedoraproject.org/pub/epel/6/i386/epel-release-6-8\
.noarch.rpm
```

Then, to install both Node.js and NPM, simply run this command:

```
sudo yum install nodejs npm --enablerepo=epel
```

This might take a while. Answer with y as the process goes. In the end, you should see something like this (your results may vary): Installed:

```
  nodejs.i686 0:0.10.26-1.el6      npm.noarch 0:1.3.6-4[…]


Dependency Installed:

...
Dependency Updated:
...
Complete!
```

You probably know this, but just in case, to check installations, type the following:

```
$ node –V
$ npm –v
```

For more information on using yum, see Managing Software with yum (https://www.centos.org/docs/5/html/yum) and Tips on securing your EC2 instance (http://aws.amazon.com/articles/1233).

So, if the previous EPEL option didn't work for you, follow these build steps. On your EC2 instance, install all system updates with yum:

```
$ sudo yum update
```

Then, install the C++ compiler (again with yum):

```
$ sudo yum install gcc-c++ make
```

Do the same with openssl:

```
$ sudo yum install openssl-devel
```

Then install Git, which is needed for delivering source files to the remote machine. When Git is unavailable, rsync (http://ss64.com/bash/rsync.html) can be used:

```
$ sudo yum install git
```

Last, clone the Node repository straight from GitHub:

```
$ git clone git://github.com/joyent/node.git
```

and build Node.js:

```
$ cd node
$ git checkout v0.10.12
$ ./configure
$ make
$ sudo make install
```

Note

For a different version of Node.js, you can list them all with `$ git tag -l` and check out the one you need.

To install NPM, run:

```
$ git clone https://github.com/isaacs/npm.git
$ cd npm
$ sudo make install
```

Relax and enjoy the build. The next step is to configure AWS ports / firewall settings. Here's a short example of server.js, which outputs "Hello readers" and looks like this:

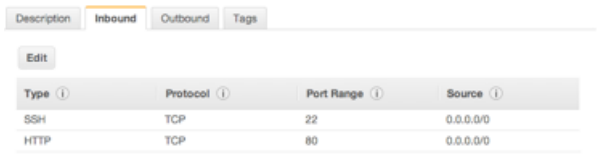
```
var http = require('http')
http.createServer(function(req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  console.log ('responding');
  res.end('Hello readers!'
    + 'If you see this, then your Node.js server '
    + 'is running on AWS EC2!');
}).listen(80);
console.log ('server is up');
```

On the EC2 instance, either configure the firewall to redirect connections (e.g., port to Node.js 3000, but this is too advanced for our example) or disable the firewall (okay for our quick demonstration and development purposes):

```
$ service iptables save
$ service iptables stop
$ chkconfig iptables off
```

In the AWS console, find your EC2 instance and apply a proper rule to allow for in "bound traffic, as shown in below. For example,

```
Type: HTTP
```

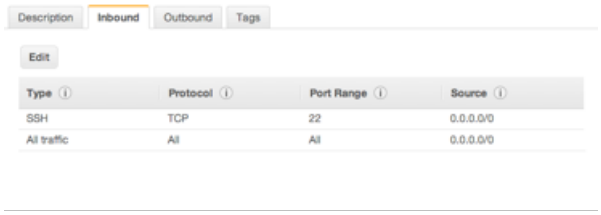


Allowing inbound HTTP traffic on port 80

The other fields fill automatically:

```
Protocol: TCP
Port Range: 80
Source: 0.0.0.0/0
```

Or we can just allow all traffic (again, for development purposes only), as shown in below.



Allowing inbound HTTP traffic on port 80

Now, while the Node.js app is running, executing `$ netstat -apn | grep 80`, the remote machine should show the process. For example,

```
tcp    0      0 0.0.0.0:80      0.0.0.0:*      LISTEN      1064/node
```

And from your local machine, i.e., your development computer, you can either use the public IP or the public DNS (the Domain Name System) domain, which is found and copied from the AWS console under that instance's description. For example,

```
$ curl XXX.XXX.XXX.XXX –v
```

Or, just open the browser using the public DNS. For the proper iptables setup, please consult experienced development operations engineers and manuals, because this is an important security aspect and it is out of the scope of this book. However, here are some commands to redirect traffic to, say, port 3001:

```
$ sudo iptables -A PREROUTING -t nat -i eth0 -p tcp --dport 80 -j REDIRECT --to-\
port 8080
```

```
$ sudo iptables -t nat -A INPUT -p tcp --dport 80 -j REDIRECT --to-ports 3001
```

```
$ sudo iptables -t nat -A OUTPUT -p tcp --dport 80 -j REDIRECT --to-ports 3001
```

You can also use commands such as the following:

```
$ service iptables save
$ service iptables start
$ service iptables restart
$ chkconfig iptables on
```

It's worth mentioning that AWS supports many other operating systems via its AWS Marketplace (https://aws.amazon.com/marketplace). Although AWS EC2 is a very popular and affordable choice, some companies opt for special Node.js tools available in the SmartOS (http://smartos.org), e.g., DTrace (http://dtrace.org/blogs), built on top of Solaris by Joyent (http://www.joyent.com), the company that maintains Node.js.