Agile Web Development with Angular.js

Learn how to develop an AngularJS app from scratch to deploy to production, using Gulp to automate and enhance your workflow.

Carlos Azaustre WEB DEVELOPER

Table of Contents

Agile web development with AngularJS

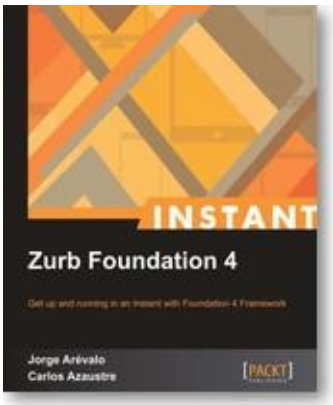
- 1st Edition (Spanish): **August 2014**
- 2nd Edition (Spanish): **January 2015**
- English Edition: **March 2015**

Published by carlosazaustre.es Books

Biography

> **Carlos Azaustre** (Madrid, 1984) Web developer, focused on the *front end*, lover of JavaScript. Various years of experience at private companies, *startups* and as a *freelancer*. Currently working as CTO of the *startup* Chefly
>
> BSc in Telematics Engineering from Charles III University of Madrid and studies for the MSc in Web Technologies from the University of Castile-La Mancha (Spain). Outside of formal education, he loves self-learning using the internet. You can find his articles and tutorials on his blog carlosazaustre.es

Other recently published books



Instant Zurb Foundation 4

Get up and running in an instant with Zurb Foundation 4 Framework

- **ISBN**: 9781782164029 (September 2013)
- **Pages**: 56
- **Language**: English
- **Authors**: Jorge Arévalo y Carlos Azaustre

- **Buy on Amazon**

2. Basic Concepts

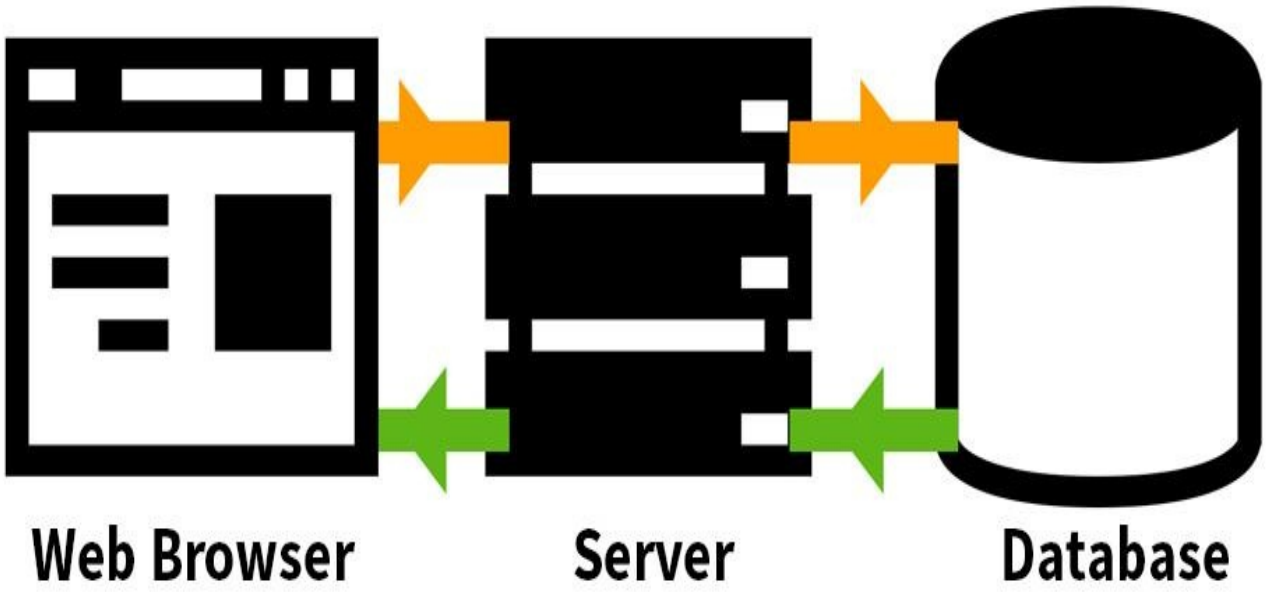
> **Note**
> This book is intended for those who have basic knowledge of programming, web development, JavaScript and are familiar with the Angular.js framework even at a very basic level.

> **Note**
> This book follows the development of an example of a simple web application (blog), which consumes data from an external API that we will not develop, therefore it is beyond the scope of this book. Angular.js is much more than this; this book merely provides the basis for the implementation of scalable and maintainable web applications and the use of task managers like Gulp.js to be more agile.

2.1 Structure of a Modern Web Application

A web application, currently, is usually composed of three main parts:

- The public or client part, also known as the *front end*
- The server part, known as the *back end*
- The data storage, or database



The database is responsible for storing all of our application's information. Users, data related to our application, etc… This database communicates with the *back end*, which is responsible for monitoring security, data processing, authorization, etc… Finally the *front end* is the part that runs on the end user's browser, and is responsible for displaying information in an appealing manner and communicating with the *back end* to create and display data. In a modern web application, communication is achieved asynchronously with JavaScript (AJAX) using the document format `JSON` to send and receive data from the *back end* through a REST API.

In summary, to create a complete web application we need:

- A database so that the information can be stored persistently.
- A *back end* that is responsible for security, authorizations and data processing through a REST API.
- And a *front end* that lays out the data, presents it and communicates with the API using AJAX and `JSON`. In this example we are going to deal with the *front end*.

2.2 Technologies

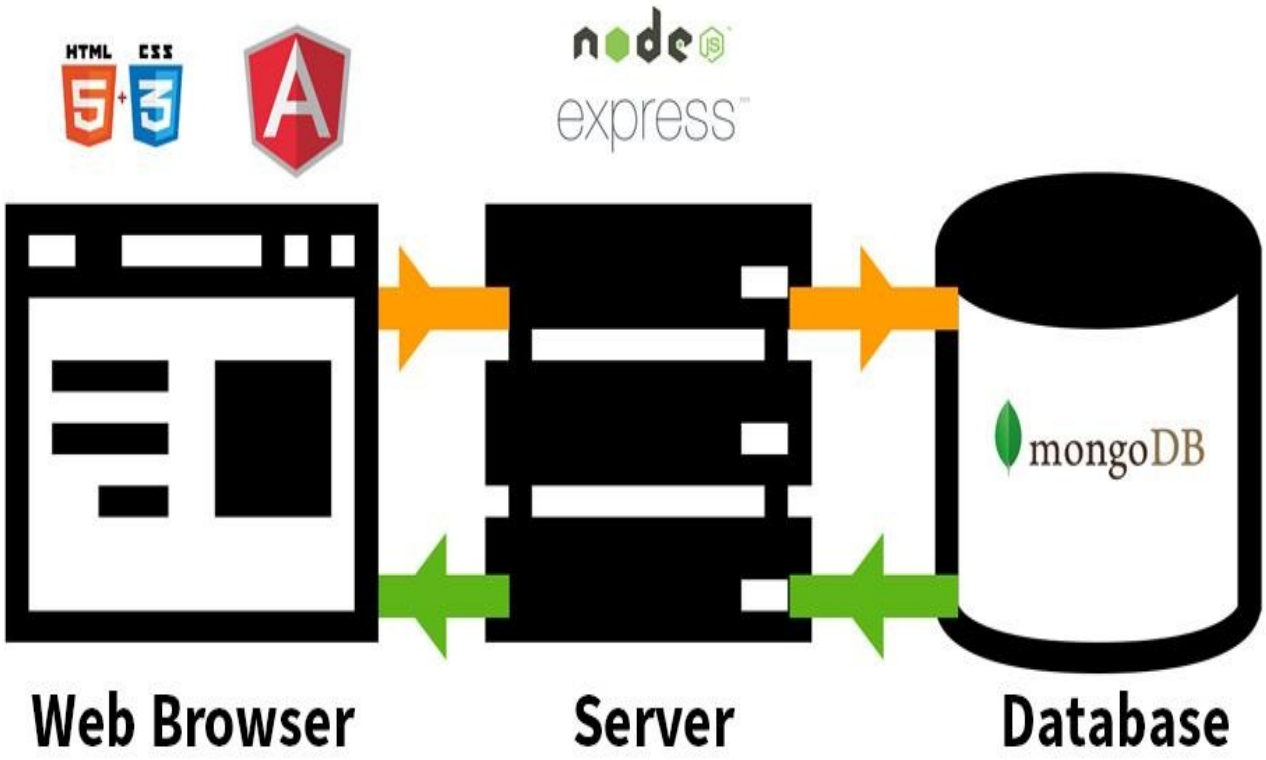
Throughout this example we will program what is known as a *single page application*, a web application of a single page that does not reload with each server request we make,

since it will communicate asynchronously thanks to JavaScript.

JavaScript was the language used on the web to add effects and animations to pages. But this language has evolved significantly in the present day, to the point of being taken to the server with *Node.js*. Node.js is a programming environment driven by events, meaning that the user defines the program's flow through the user interface, in this case the web page. Following a non-blocking input and output model, this allows us to do asynchronous programming, similar to AJAX in the client JavaScript. At present there are numerous frameworks that make programming easier on the server-side. The most known and widespread is *Express*, although others exist, depending on your needs, such as: *Meteor*, *Restify*, *Sails*, *Koa*, *Hapi*,…

JavaScript has evolved to the point that it is also found in databases, as in the case of *MongoDB*, a non-relational database (*NoSQL*) whose entries are stored as documents in the `BSON` format: *Binary JSON* that allows for the reading and writing of data to be done very quickly and in an atomic manner, as long as we organize our data model following a non-relational structure. It is a different way of thinking from the classic SQL.

Of course JavaScript has also evolved on the *front end*, where it has always resided, though its implementation has improved thanks to the emergence of MVC frameworks that allow us to modularize code and avoid the famous *spaghetti code* that is generated when we are not careful. Within this family of frameworks we can find *Backbone.js*, *Ember.js* and the one that we will use which is *Angular.js*, a project developed at Google in 2009 that is now achieving widespread popularity.



By combining these technologies in all of the parts that form a web application, the same programming language (JavaScript) is used throughout the entire technology stack from beginning to end, which is known as JavaScript *end-to-end* or also ***MEAN** Stack*, which refers to its use of (**M**)ongoDB, (**E**)xpressJS, (**A**)ngularJS and (**N**)odeJS.

3. Configuration of the Work Environment

Our work environment must have the appropriate tools that allow us to be agile and check errors without wasting time on repetitive tasks.

We need a modern web browser, which supports new standards and includes a console and development tools. Today, one of those that provides the best results, in terms of development, is the Google browser, Chrome.

We also need a text editor. There are developers who prefer an IDE like Eclipse, WebStorm, NetBeans, etc… I personally find myself to be most comfortable with minimalist editors like Sublime Text or the recent Atom.io.

Although we are *front ends* and the terminal seems more related to the *back end* or server management, we need to use it too. We need one that is simple to use and efficient. If you have a MacOS X system, I recommend iTerm2.

> **Note** The next steps are for installing the programs on a MacOS X system. For other operating systems consult the documentation available on their respective websites.

3.1 Installing Google Chrome

Chrome Browser

https://www.google.com/chrome/browser/

chrome

DOWNLOAD SET UP CHROMEBOOKS CHROMECAST

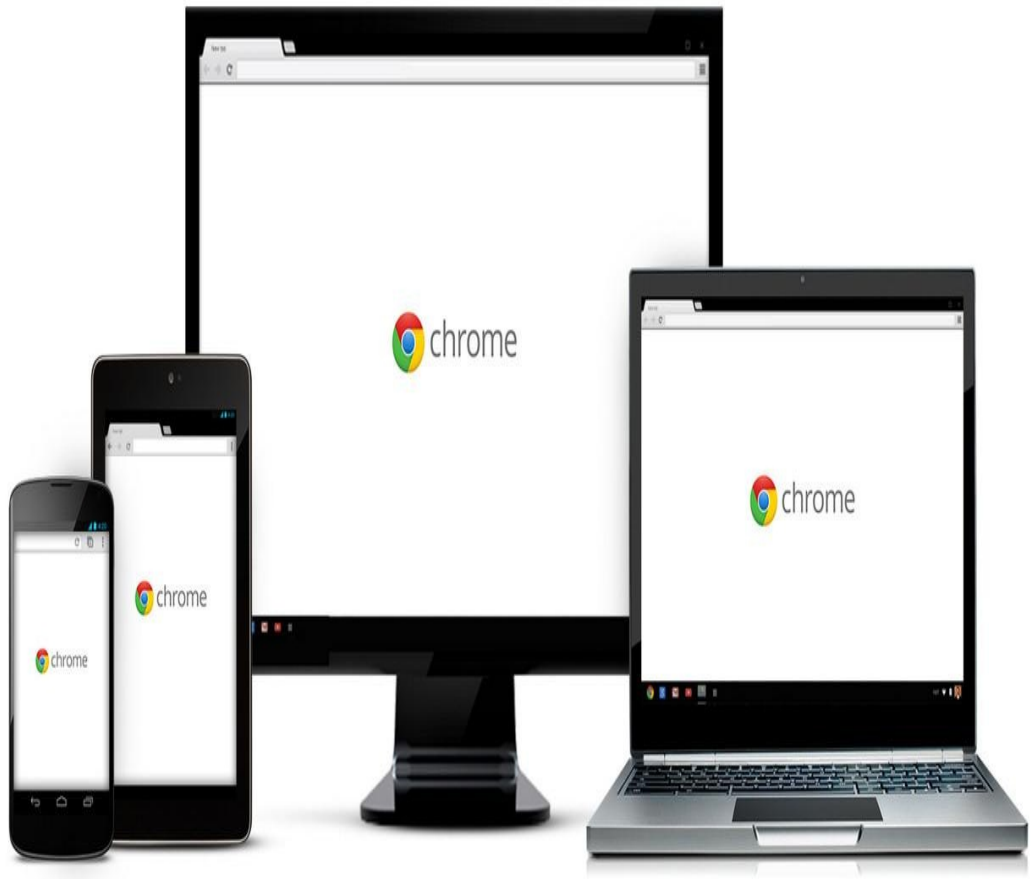
Get a fast, free web browser

One browser for your computer, phone and tablet

Download Chrome

For Mac OS X 10.6 or later

You can also download Chrome for **Windows** or **Linux**.



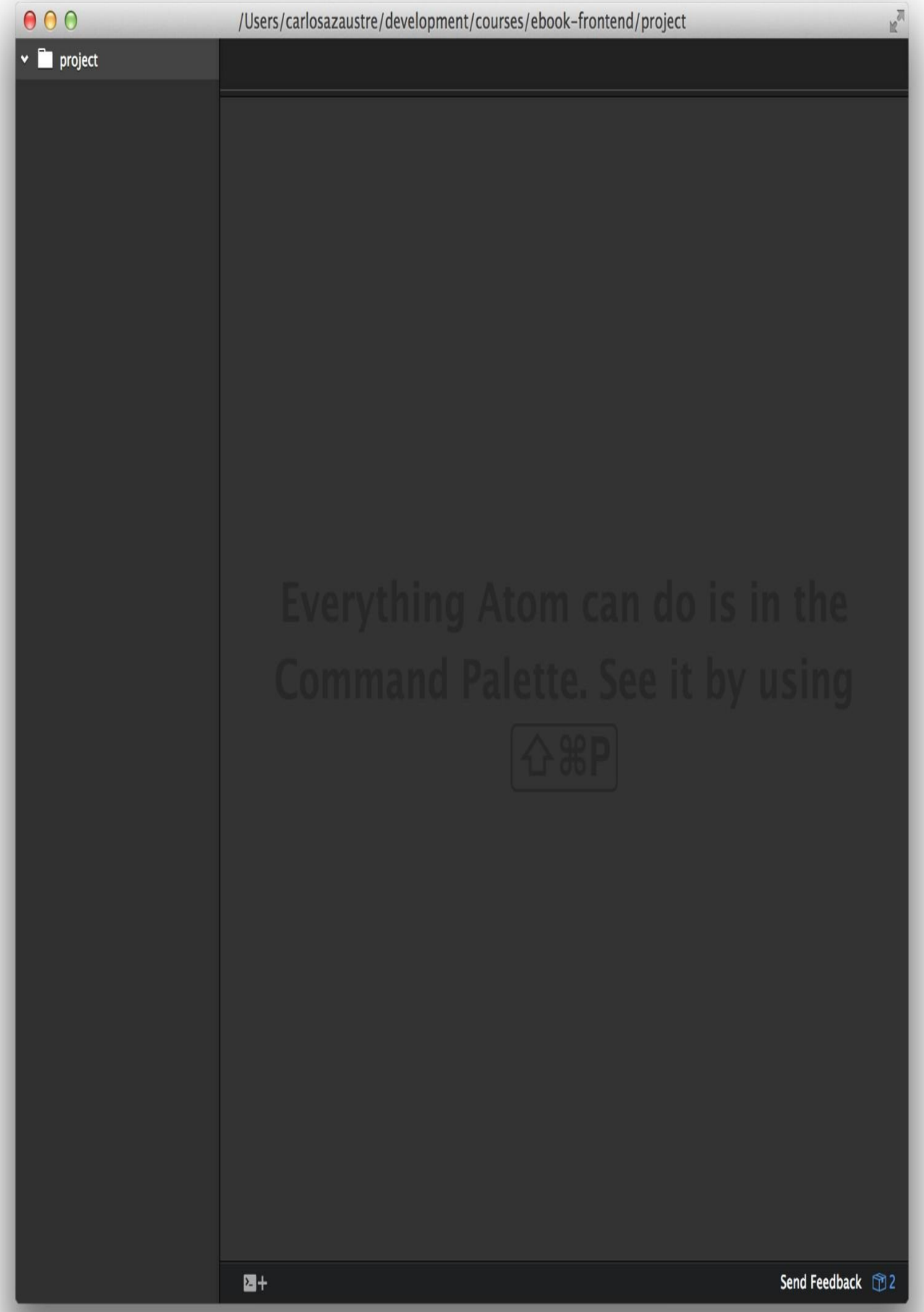
javascript:void(0)

Go to `http://google.com/chrome`. Download the file `.dmg` and open it once it has downloaded. Drag the **Google Chrome** application to the applications folder.

To access the development tools. Go to Chrome's `View` menu, `Development / Developer tools`. There you will find the JavaScript console, access to scripts, network tools, etc…

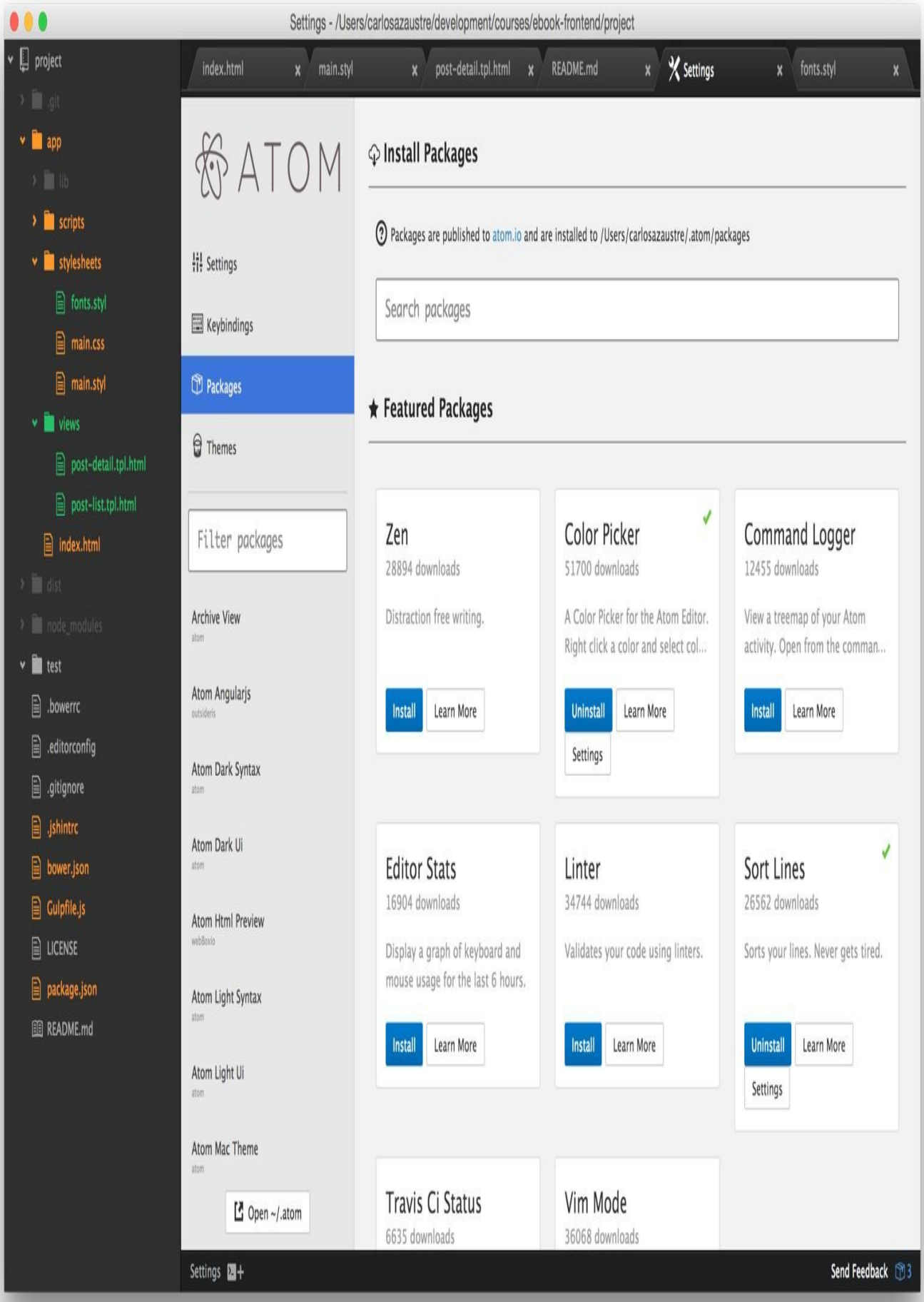
In the Chrome Store there is an extension for Angular, called Batarang. You can download it from here. This extension allows you to display the `scopes` and other special features that are useful when debugging.

3.2 Installing and Configuring Sublime Text or Atom


/Users/carlosazaustre/development/courses/ebook-frontend/project
project
Everything Atom can do is in the
Command Palette. See it by using
⇧⌘P
Send Feedback
2

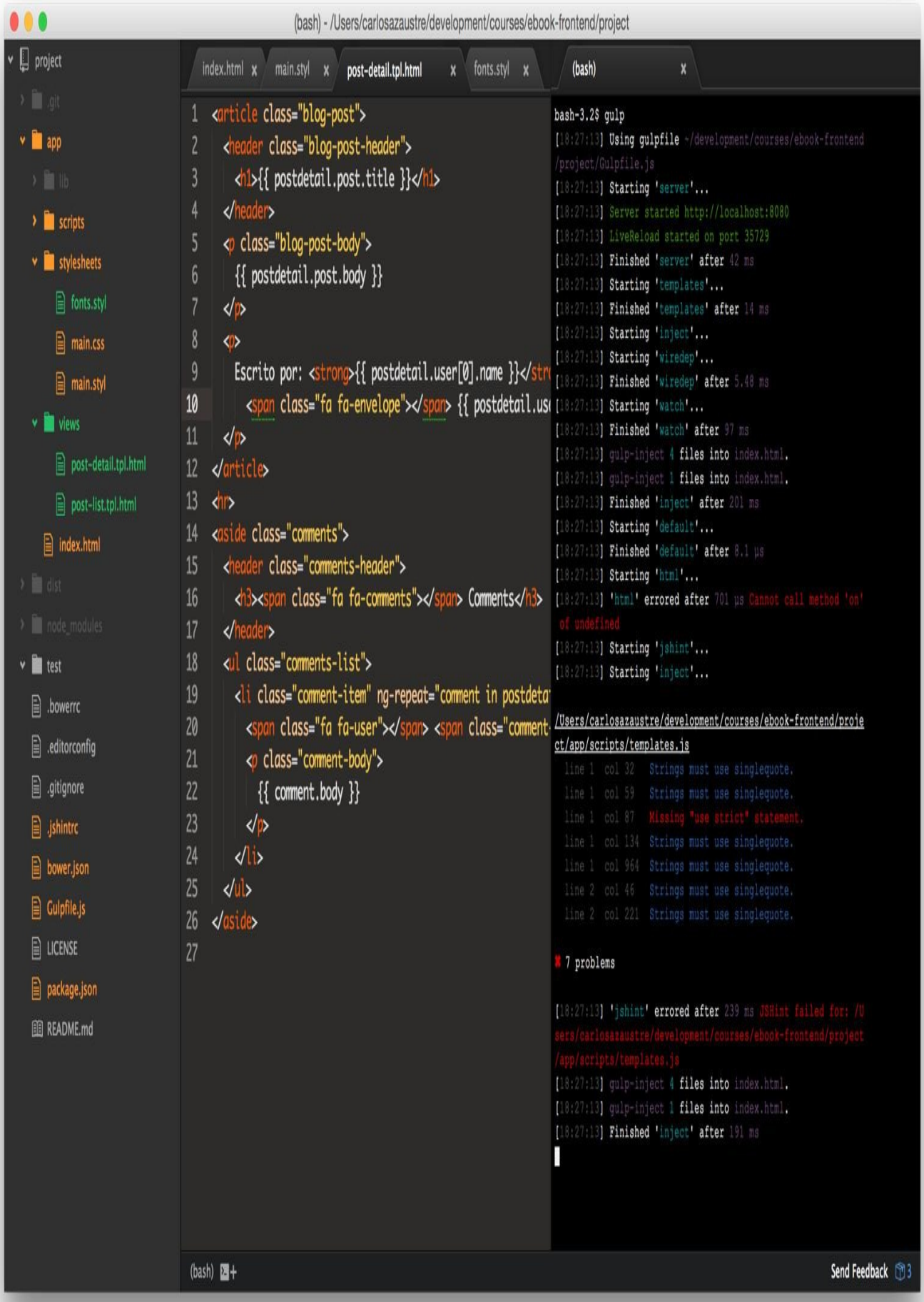
You can download SublimeText from its website. Using it is free, but purchasing your license (to avoid the pop-ups that appear occasionally) costs about $70. Another very similar editor is Atom created by the developers at GitHub which is what I currently use. If you want to try more *hipster* options there's the beta of Chrome Dev Editor.

Atom has many plugins, developed by the Atom team and the community. You can install them from the menu `Atom > Preferences`.


Settings - /Users/carlosazaustre/development/courses/ebook-frontend/project
project
.git
app
lib
scripts
stylesheets
fonts.styl
main.css
main.styl
views
post-detail.tpl.html
post-list.tpl.html
index.html
dist
node_modules
test
.bowerrc
.editorconfig
.gitignore
.jshintrc
bower.json
Gulpfile.js
LICENSE
package.json
README.md
index.html
main.styl
post-detail.tpl.html
README.md
Settings
fonts.styl
ATOM
Settings
Keybindings
Packages
Themes
Filter packages
Archive View
atom
Atom Angularjs
outsideris
Atom Dark Syntax
atom
Atom Dark Ui
atom
Atom Html Preview
webBoxio
Atom Light Syntax
atom
Atom Light Ui
atom
Atom Mac Theme
atom
Open ~/.atom
Install Packages
Packages are published to atom.io and are installed to /Users/carlosazaustre/.atom/packages
Search packages
Featured Packages
Zen
28894 downloads
Distraction free writing.
Install
Learn More
Color Picker
51700 downloads
A Color Picker for the Atom Editor. Right click a color and select col...
Uninstall
Learn More
Settings
Command Logger
12455 downloads
View a treemap of your Atom activity. Open from the comman...
Install
Learn More
Editor Stats
16904 downloads
Display a graph of keyboard and mouse usage for the last 6 hours.
Install
Learn More
Linter
34744 downloads
Validates your code using linters.
Install
Learn More
Sort Lines
26562 downloads
Sorts your lines. Never gets tired.
Uninstall
Learn More
Settings
Travis Ci Status
6635 downloads
Vim Mode
36068 downloads
Settings
Send Feedback
3

.

Among the most curious and not very well-known is the terminal plugin, to have a command terminal in a tab of the editor.


(bash) - /Users/carlosazaustre/development/courses/ebook-frontend/project
project
.git
app
lib
scripts
stylesheets
fonts.styl
main.css
main.styl
views
post-detail.tpl.html
post-list.tpl.html
index.html
dist
node_modules
test
.bowerrc
.editorconfig
.gitignore
.jshintrc
bower.json
Gulpfile.js
LICENSE
package.json
README.md
index.html
main.styl
post-detail.tpl.html
fonts.styl
(bash)
<article class="blog-post">
<header class="blog-post-header">
<h1>{{ postdetail.post.title }}</h1>
</header>

{{ postdetail.post.body }}

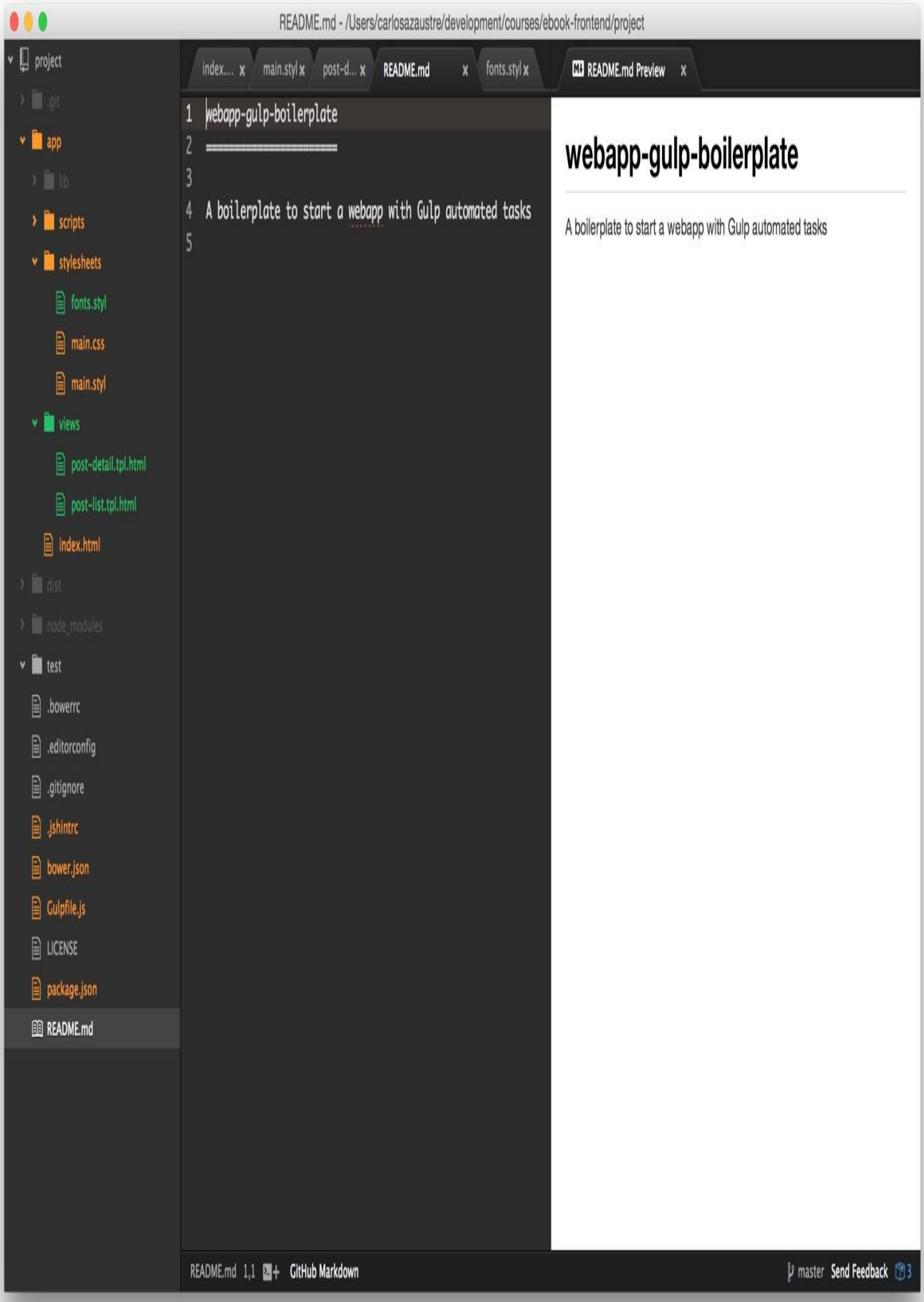
Escrito por: {{ postdetail.user[0].name }}</str
 {{ postdetail.us

</article>
<hr>
<aside class="comments">
<header class="comments-header">
<h3> Comments</h3>
</header>
<ul class="comments-list">
<li class="comment-item" ng-repeat="comment in postdeta
 <span class="comment

{{ comment.body }}

</aside>
bash-3.2$ gulp
[18:27:13] Using gulpfile ~/development/courses/ebook-frontend
/project/Gulpfile.js
[18:27:13] Starting 'server'...
[18:27:13] Server started http://localhost:8080
[18:27:13] LiveReload started on port 35729
[18:27:13] Finished 'server' after 42 ms
[18:27:13] Starting 'templates'...
[18:27:13] Finished 'templates' after 14 ms
[18:27:13] Starting 'inject'...
[18:27:13] Starting 'wiredep'...
[18:27:13] Finished 'wiredep' after 5.48 ms
[18:27:13] Starting 'watch'...
[18:27:13] Finished 'watch' after 97 ms
[18:27:13] gulp-inject 4 files into index.html.
[18:27:13] gulp-inject 1 files into index.html.
[18:27:13] Finished 'inject' after 201 ms
[18:27:13] Starting 'default'...
[18:27:13] Finished 'default' after 8.1 µs
[18:27:13] Starting 'html'...
[18:27:13] 'html' errored after 701 µs Cannot call method 'on'
of undefined
[18:27:13] Starting 'jshint'...
[18:27:13] Starting 'inject'...
/Users/carlosazaustre/development/courses/ebook-frontend/proje
ct/app/scripts/templates.js
line 1 col 32 Strings must use singlequote.
line 1 col 59 Strings must use singlequote.
line 1 col 87 Missing "use strict" statement.
line 1 col 134 Strings must use singlequote.
line 1 col 964 Strings must use singlequote.
line 2 col 46 Strings must use singlequote.
line 2 col 221 Strings must use singlequote.
✖ 7 problems
[18:27:13] 'jshint' errored after 239 ms JSHint failed for: /U
sers/carlosazaustre/development/courses/ebook-frontend/project
/app/scripts/templates.js
[18:27:13] gulp-inject 4 files into index.html.
[18:27:13] gulp-inject 1 files into index.html.
[18:27:13] Finished 'inject' after 191 ms
(bash)
Send Feedback

Or a *Markdown* file viewer to see what you’re writing displayed in real-time.


README.md - /Users/carlosazaustre/development/courses/ebook-frontend/project
project
.git
app
lib
scripts
stylesheets
fonts.styl
main.css
main.styl
views
post-detail.tpl.html
post-list.tpl.html
index.html
dist
node_modules
test
.bowerrc
.editorconfig
.gitignore
.jshintrc
bower.json
Gulpfile.js
LICENSE
package.json
README.md
index.... x
main.styl x
post-d... x
README.md x
fonts.styl x
README.md Preview x
1 webapp-gulp-boilerplate
2 ==========
3
4 A boilerplate to start a webapp with Gulp automated tasks
5
webapp-gulp-boilerplate
A boilerplate to start a webapp with Gulp automated tasks
README.md 1,1 GitHub Markdown
master Send Feedback 3

3.3 Installing and Configuring iTerm2


1. /Users/carlosazaustre/development? (bash)
carlosazaustre at MacBookPro in ~/development/courses/ebook-frontend/project
$

Download iTerm2 for Mac. Once it is installed, start the application. To get a `prompt` like the one in the image, in which you can see your username, your computer, the current directory, and even the repository branch you're in, you can download the following scripts that will configure the terminal like I have mine, by writing the following:

```
$ cd ~
$ curl -O https://raw.githubusercontent.com/carlosazaustre/mac-dev-setup/master/.bash_profile
$ curl -O https://raw.githubusercontent.com/carlosazaustre/mac-dev-setup/master/.bash_prompt
$ curl -O https://raw.githubusercontent.com/carlosazaustre/mac-dev-setup/master/.aliases
$ curl -O https://raw.githubusercontent.com/carlosazaustre/mac-dev-setup/master/.gitconfig
```

Restart iTerm and it will already be configured for you.

3.4 Installing Git

Git is very important for a developer; it is a way to save versions and changes that we make in our developments, to collaborate on other free software projects with the community and gradually create a CV.

You can install Git on your system using `homebrew` on Mac. To install this package manager on Mac, you need to run the following scripts in your terminal:

```
# Installing homebrew for Mac. The missing package manager for OS X
$ ruby -e "$(curl -fsSL https://raw.github.com/Homebrew/homebrew/go/install)"
$ brew update
$ export PATH="/usr/local/bin:$PATH"
```

Then we can install Git with:

```
$ brew install git
```

When you're done, you can check the version with:

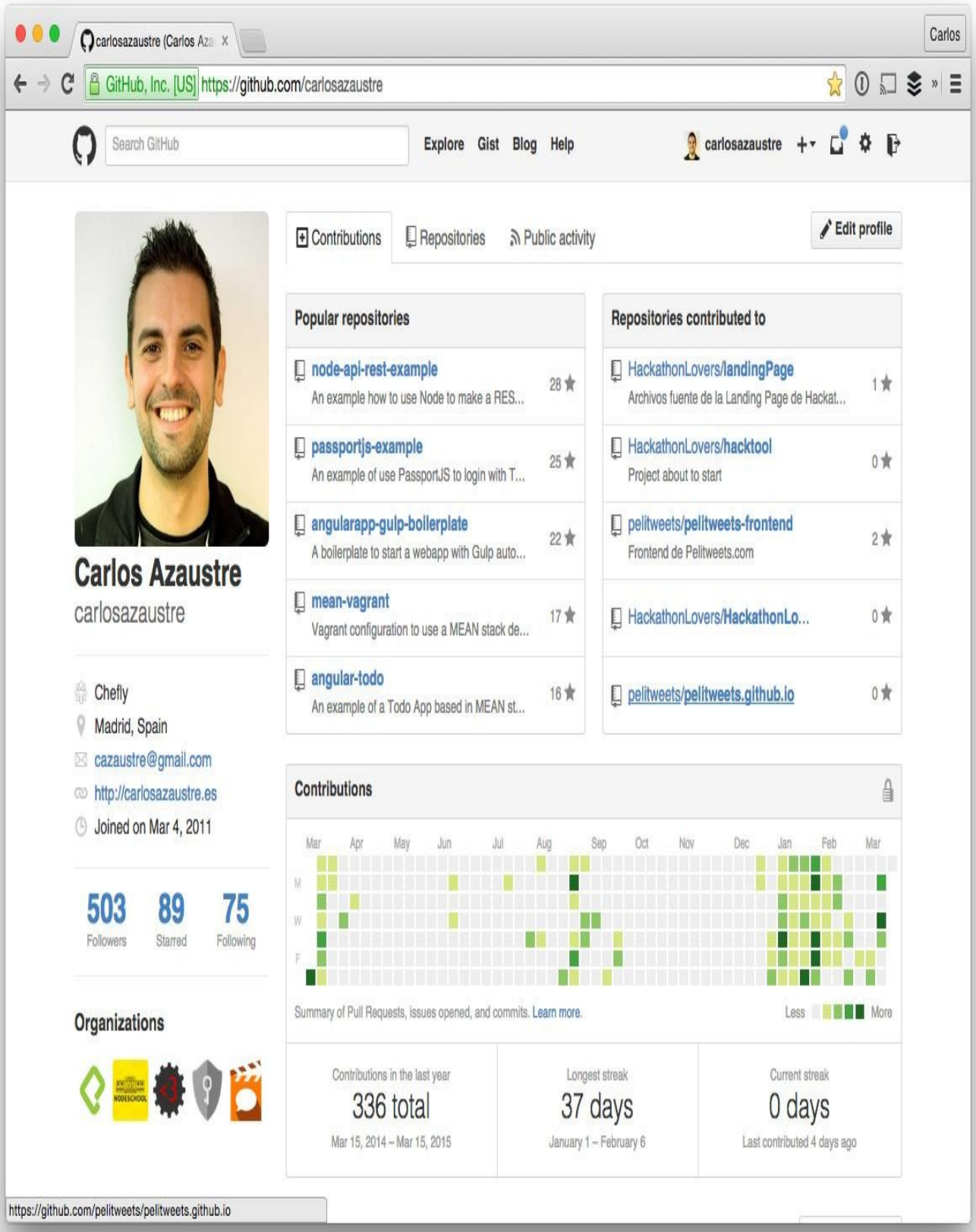
```
$ git –version
git version 1.8.4
```

We proceed to configure Git with our username:

```
$ git config –global user.name "Carlos Azaustre"
$ git config –global user.email "cazaustre@gmail.com"
```

Every time we `push` to a remote repository, it will ask for the name and password of our account that we will now create.

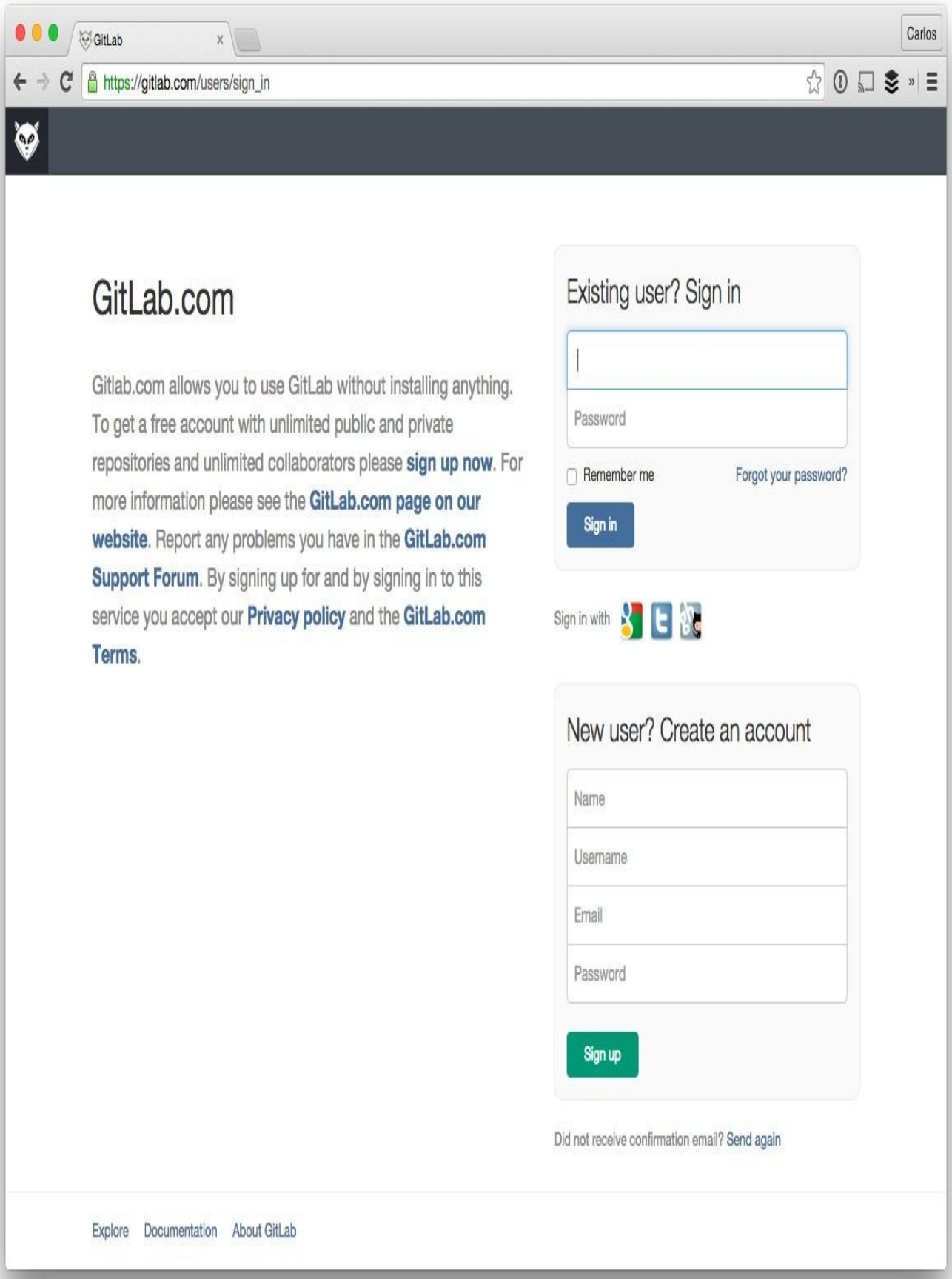
3.4.1 Registration on GitHub



In addition to being a version control system (or repository) on the cloud, GitHub is also a social network for developers, where communities are formed around *Open Source* projects. If you are a developer and you don't have a GitHub account, I'm sorry to say that you don't exist. Create your GitHub account today and start to publish your open projects. Their website is github.com

3.4.2 Registration on GitLab

If, however, you want to use the cloud for your private repositories, you have the option of GitLab, which gives you unlimited repositories and also, due to being *Open Source*, the option of installing it on your own server and therefore having more control over it.



3.5 File and Directory Structure

To begin developing our *front end* project, we are going to structure the files in the following manner:

```
project/
  ├── app/
  ├── dist/
  ├── package.json
  ├── bower.json
  ├── README.md
  ├── .gitignore
  ├── .editorconfig
  └── .jshintrc
```

Later we will be adding more files and subdirectories, but to start this will be our structure.

- `project` is the root folder of our web project and we have 2 subdirectories below it.
- `app` contains all of the complete source code of our web application, without minification or optimization, only the development code. And in `dist` we have the entire project minified and optimized in order to be deployed in production.
- `package.json` contains our project information as well as the name and versions of the dependencies that we will use for development.
- `bower.json` is similar to `package.json` for managing the dependencies that we will use in the *front end*, such as Angular libraries, CSS frameworks, etc…
- `README.md` is the file that we use to provide information about the application that we are developing and as project documentation.
- `.gitignore` tells Git which files we don't want to be uploaded to the repository (for example, passwords, configuration files, etc.).
- `.editorconfig` allows us to configure our text editor so that all of the developers on the same project, who use it, have the same spacing, tabbing, etc. in the code.
- `.jshintrc` is a JSON file that allows us to check errors in the code as well as in the syntax, use of variables, or style in our JavaScript code as we save the files. Combined with a task manager, like Gulp, which we will use later, it allows the agile development of our projects.

.editorconfig

This is the content that we will use, which you can adapt to your preferences:

```
root = true

[*]
indent_style = space
indent_size = 2
end_of_line = lf
charset = utf-8
trim_trailing_whitespace = true
insert_final_newline = true

[*.md]
trim_trailing_whitespace = false
indent_style = tab
```

In this file we are indicating that we want the indent to be created with spaces instead of tabs. The size of the tab is 2 spaces. The end of the line should be marked as `LF`. The character encoding to use is `UTF-8`. And for `.md` files such as `README.md` the indentation is with tabs.

You can consult the rest of the properties to configure in the documentation on their website: http://editorconfig.org/ and also download the plugin for your favorite text editor.

.jshintrc

This is the content of our JSHint file. You can see properties to add on the web page with their documentation: http://www.jshint.com/docs/options/:

```
{
  "node": true,
  "browser": true,
  "esnext": true,
  "bitwise": true,
  "camelcase": true,
  "curly": true,
  "eqeqeq": true,
  "immed": true,
  "indent": 4,
  "latedef": true,
  "newcap": true,
  "noarg": true,
  "quotmark": "single",
  "undef": true,
  "unused": true,
  "strict": true,
  "trailing": true,
  "smarttabs": true,
  "jquery": true,
  "globals": {
    "angular": false
  }
}
```

package.json

This is a `JSON` file which is where we include all of the information concerning our project. You can add numerous fields. You can see which fields to add in the documentation on the following web page: https://www.npmjs.org/doc/files/package.json.html. For this book's example, this is the content that we have:

```
{
  "name": "angularapp-gulp-boilerplate",
  "version": "0.0.1",
  "description": "Ejemplo de desarrollo de aplicación web con AngularJS",
  "bugs": {
```

```
    "url": "https://github.com/carlosazaustre/angularapp-gulp-boilerplate/issues",
    "email": "cazaustre@gmail.com"
  },
  "license": "MIT",
  "author": "Carlos Azaustre <cazaustre@gmail.com> - http://carlosazaustre.es",
  "repository": {
    "type": "git",
    "url": "https://github.com/carlosazaustre/angularapp-gulp-boilerplate"
  },
  "dependencies": {},
  "devDependencies": {}
}
```

We will fill in the `devDependencies` object each time we install a new package via `npm` from the terminal.

bower.json

As we said earlier, it is similar to `package.json` but for the dependencies and libraries we will use for the "front end". The content of this file will be the following:

```
{
  "name": "angularapp-gulp-boilerplate",
  "version": "0.0.1",
  "description": "Example of a web application with JavaScript",
  "dependencies": {}
}
```

We will fill in the `dependencies` object each time we install a new dependency via `bower` from the terminal.

3.6 Automating our Workflow

First of all, we must install Node.js. Although this is a *front end* project and we are not going to develop the *back end* (we will use a *fake back end*), we need Node.js in order to install Bower, Gulp and run the tasks that we specify in `Gulpfile.js`.

To install Node you can go to their website [http://nodejs.org/downloads] or if you have a Mac, you can do it from the terminal with `homebrew` like we did with `git`:

```
# Install NodeJS from Homebrew
$ brew install node
$ node -v
v0.10.26
$ npm -v
1.4.3
```

With Node.js installed, we proceed to install the following dependencies, globally (using the flag `-g`) that we will use throughout our project. After being globally installed, we can run the commands from any directory and in any other project:

```
$ npm install -g gulp
$ npm install -g bower
```

Gulp is a task launcher that runs under Node.js, allowing us to automate tasks that we do frequently. Until now the best known manager was *Grunt* but the simplicity and speed that *Gulp* offers has made its use spread to many projects. *Gulp* has a large community and plugins for anything that comes to mind. We proceed then to install the dependencies and

plugins that we are going to use locally in our project. In this case we must be in the root directory of our application and write the following:

```
$ npm install –save-dev gulp
$ npm install –save-dev gulp-connect
$ npm install –save-dev connect-history-api-fallback
$ npm install –save-dev gulp-jshint
$ npm install –save-dev gulp-useref
$ npm install –save-dev gulp-if
$ npm install –save-dev gulp-uglify
$ npm install –save-dev gulp-minify-css
$ npm install –save-dev gulp-stylus
$ npm install –save-dev nib
```

These dependencies will automate the JavaScript code correction, the minification of the CSS, the creation of a web development server in order to see the changes that we make in the code in real-time in the browser, etc… We will see each package in detail while we move forward.

If we look closely, our `package.json` has changed, and the `devDependencies` object now looks like this:

```
“devDependencies”: {
    “gulp”: “^3.8.6”,
    “gulp-connect”: “^2.0.6”,
    “connect-history-api-fallback”: “0.0.4”,
    “gulp-jshint”: “^1.8.0”,
    “gulp-useref”: “^0.6.0”,
    “gulp-if”: “^1.2.4”,
    “gulp-uglify”: “^0.3.1”,
    “gulp-minify-css”: “^0.3.7”,
    “gulp-stylus”: “^1.3.0”,
    “nib”: “^1.0.3”
  }
```

Dependencies along with their version number have been added to the file automatically, thanks to the flag `–save` and with `-dev` in the object in question.

We are going to create a `Gulpfile.js` file in the root directory of our project, where we are going to specify some tasks to begin to streamline our life:

Gulpfile.js

```
var gulp    = require(‘gulp’),
    connect = require(‘gulp-connect’),
    historyApiFallback = require(‘connect-history-api-fallback’);

// Development web server
gulp.task(‘server’, function() {
  connect.server({
    root: ‘./app’,
    hostname: ‘0.0.0.0’,
    port: 8080,
    livereload: true,
    middleware: function(connect, opt) {
      return [ historyApiFallback ];
    }
  });
});
```

This task takes the content from the directory `app` and shows it as if it were a web server, at the address `http://localhost:8080`, `http://127.0.0.1:8080` or `http://0.0.0.0:8080`. Entering the

hostname `0.0.0.0` gets us the `livereload`, which is what allows us to see the changes we make in real-time, visible from any device connected to the same network.

Imagine that in our local network, our computer has the private IP `192.168.1.20`. If we access the address `http://192.168.1.20:8080` from a mobile phone, tablet or other computer connected to the same network as our development computer, we will see our web application and the changes we make will be shown automatically. Black magic.

To do this we must continue configuring the file `Gulpfile.js`, we are going to add that the changes we make in a Stylus `.styl` file, are shown as CSS in the browser. **Stylus** is a CSS preprocessor, which allows you to write the design in a language and then change it to CSS. In chapter 4 we will look at this topic in more detail. For now, to see a quick example, we add the following tasks:

```
var stylus = require('gulp-stylus'),
    nib    = require('nib');

// Preprocess Stylus files to CSS and reload the changes
gulp.task('css', function() {
  gulp.src('./app/stylesheets/main.styl')
    .pipe(stylus({ use: nib() }))
    .pipe(gulp.dest('./app/stylesheets'))
    .pipe(connect.reload());
});

// Reload the browser on HTML changes
gulp.task('html', function() {
  gulp.src('./app/**/*.html')
    .pipe(connect.reload());
});

// Watch code changes and to run related tasks
gulp.task('watch', function() {
  gulp.watch(['./app/**/*.html'], ['html']);
  gulp.watch(['./app/stylesheets/**/*.styl'], ['css']);
});

gulp.task('default', ['server', 'watch']);
```

The task `css` takes the file `app/sytlesheets/main.styl` and preprocesses its contents to CSS in the file `app/stylesheets/main.css`, using the library `nib` that automatically adds the CSS properties for Firefox, Internet Explorer and Webkit. So we saved having to write the specific prefixes for each browser.

The task `html` simply restarts the browser using `connect.reload()` each time a change is made in an HTML file.

The task `watch` watches the changes that take place in HTML and Stylus files and launches the `html` and `css` tasks respectively.

Finally, we define a default task `default` that launches the `server` and `watch` tasks. To run the task that includes all of the tasks, in the terminal we write:

```
$ gulp
```

All of the tasks will be launched and we will be able to make changes in our code that will be shown in the browser without needing to reload manually.

Here's a quick example to put all this in place and see how Gulp and its automated tasks

works. We create the test file `index.html` in the `app` folder with the following content:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <link rel="stylesheet" href="/stylesheets/main.css">
  <title>Example</title>
</head>
<body>
  Hello World!
</body>
</html>
```

As you can see, we link to the style file `main.css` in the folder `app/stylesheets`. This file will be generated by Gulp using the `.styl` files in which we encode our design.

Therefore we create the file `main.styl` in `app/stylesheets` with the following content:

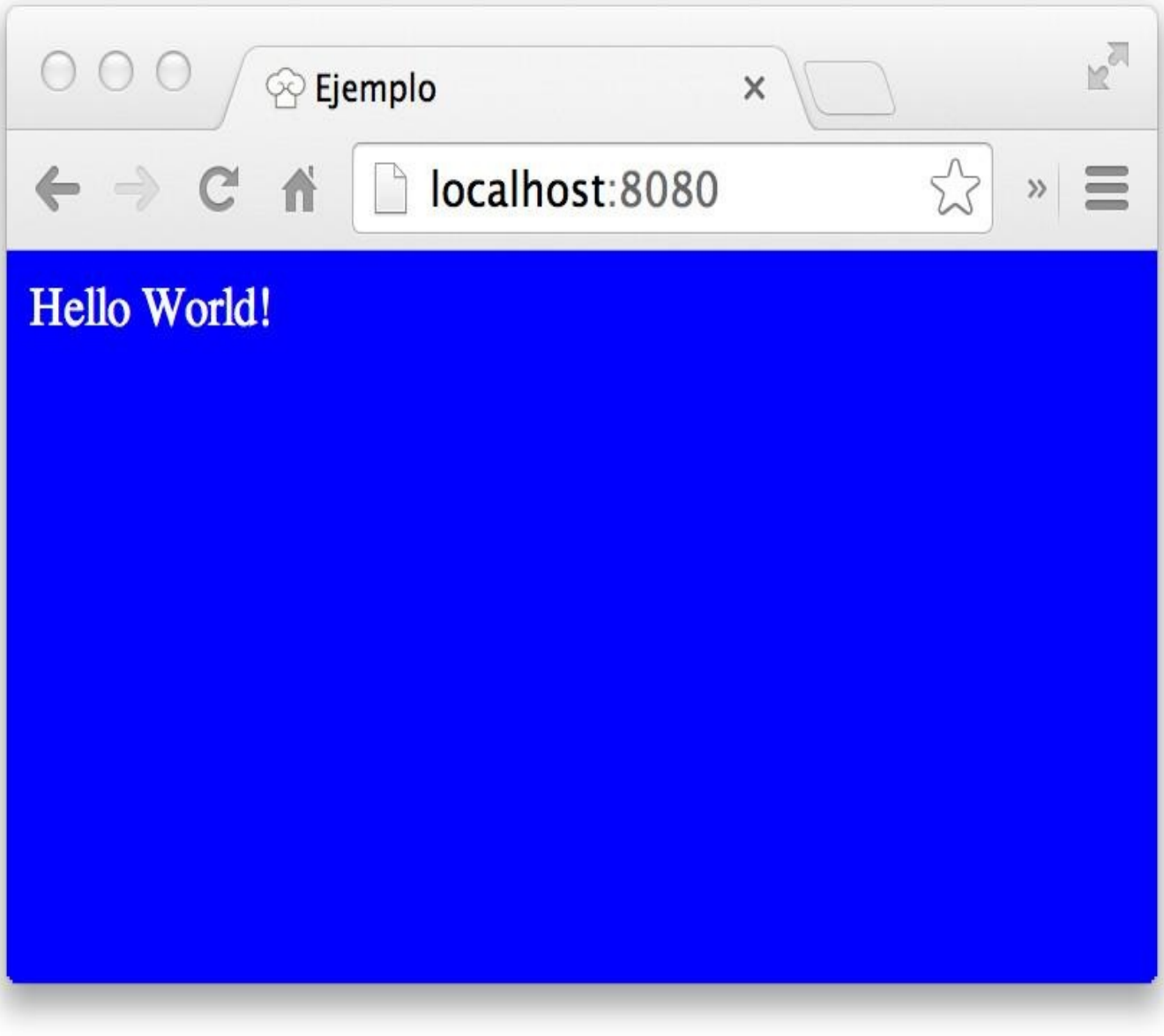
```
body
  background-color white;
  color black;
```

We save and run Gulp. If we go to `http://localhost:8080` in a browser window we can see the following:

Without stopping Gulp's process or closing the browser, we return to our `main.styl` file and change the following:

```
body
  background-color blue;
  color white;
```

We save and our browser will automatically change to this:



In this way we have configured Gulp to reload the browser automatically according to the changes we make in the HTML and the CSS, leaving us with a much more agile workflow that allows us to save a lot of development time.

Finally we will add one last task (for now), whose function is to review the JavaScript code that we write, in search of errors, style consistency, etc. This is achieved using the plugin `gulp-jshint` with the `.jshintrc` file that we had written earlier. However beforehand, we will add this new dependency, which will allow us a better reading of the errors from the terminal:

```
$ npm install –save-dev jshint-stylish
```

Then we add the following task to our `Gulpfile.js` file:

```
var jshint  = require('gulp-jshint'),
    stylish = require('jshint-stylish');

// Search errors on JS code and to show in screen
gulp.task('jshint', function() {
  return gulp.src('./app/scripts/**/*.js')
    .pipe(jshint('.jshintrc'))
```

```
    .pipe(jshint.reporter('jshint-stylish'))
    .pipe(jshint.reporter('fail'));
});
```

We also add a new subtask to the `watch` task to watch the changes we make in the JavaScript code and so launch the task we just created:

```
gulp.task('watch', function() {
  gulp.watch(['./app/**/*.html'], ['html']);
  gulp.watch(['./app/stylesheets/**/*.styl'], ['css']);
  gulp.watch(['./app/scripts/**/*.js'], ['jshint']);
});
```

Our complete `Gulpfile.js` file, for the moment, will stay like this:

```
// File: Gulpfile.js
'use strict';

var gulp    = require('gulp'),
    connect = require('gulp-connect'),
    stylus  = require('gulp-stylus'),
    nib     = require('nib'),
    jshint  = require('gulp-jshint'),
    stylish = require('jshint-stylish'),
    historyApiFallback = require('connect-history-api-fallback');

gulp.task('server', function() {
  connect.server({
    root: './app',
    hostname: '0.0.0.0',
    port: 8080,
    livereload: true,
    middleware: function(connect, opt) {
      return [ historyApiFallback ];
    }
  });
});

gulp.task('jshint', function() {
  return gulp.src('./app/scripts/**/*.js')
    .pipe(jshint('.jshintrc'))
    .pipe(jshint.reporter('jshint-stylish'))
    .pipe(jshint.reporter('fail'));
});

gulp.task('css', function() {
  gulp.src('./app/stylesheets/main.styl')
    .pipe(stylus({ use: nib() }))
    .pipe(gulp.dest('./app/stylesheets'))
    .pipe(connect.reload());
});

gulp.task('html', function() {
  gulp.src('./app/**/*.html')
    .pipe(connect.reload());
});

gulp.task('watch', function() {
  gulp.watch(['./app/**/*.html'], ['html']);
  gulp.watch(['./app/stylesheets/**/*.styl'], ['css']);
  gulp.watch(['./app/scripts/**/*.js', './Gulpfile.js'], ['jshint']);
});

gulp.task('default', ['server', 'watch']);
```

To test how JSHint works, we are going to write a small example file in JavaScript with errors, so that it launches the task. We add the file `main.js` to the directory `app/scripts`.

```
(function() {
  console.log('Hello World!');
})();
```

If we run Gulp, we will see the following in the terminal:

```
$ gulp
[12:32:29] Using gulpfile ~/development/courses/ebook-frontend/project/Gulpfile.js
[12:32:29] Starting 'server'…
[12:32:29] Server started http://localhost:8080
[12:32:29] LiveReload started on port 35729
[12:32:29] Finished 'server' after 25 ms
[12:32:29] Starting 'watch'…
[12:32:29] Finished 'watch' after 13 ms
[12:32:29] Starting 'default'…
[12:32:29] Finished 'default' after 5.52 µs
[12:32:38] Starting 'jshint'…

/Users/carlosazaustre/development/courses/ebook-frontend/project/app/scripts/main.js
  line 2  col 3  Missing "use strict" statement.

✖ 1 problem

[12:32:38] 'jshint' errored after 69 ms JSHint failed for: /Users/carlosazaustre/development/cou
frontend/project/app/scripts/main.js
```

We encountered an error, which is that we need to add the line `use strict;` to the beginning of the file. That is because we have instructed our `.jshintrc` file to verify that the element `/js` exists in all of our files or the *parser* will show an error message. This way, in our team, all of the developers will follow the same style rules.

We add this line to the file `main.js`:

```
// File: app/scripts/main.js
'use strict';

(function() {
  console.log('Hello World!');
})();
```

Immediately after saving the file, we see in the terminal that the task has been run automatically and no longer displays errors:

```
[12:37:08] Starting 'jshint'…
[12:37:08] Finished 'jshint' after 26 ms
```

Done, we have automated our workflow each time we make a change. In the next chapter we will look at how to structure JavaScript files and the use of Bower to start to add functionality to our application.

4. Anatomy of an AngularJS Application.

In this chapter we will implement our example web application with Angular.js. To do this we will need templates and libraries that allow us to do our work. Let's begin.

4.1 HTML5Boilerplate

We are going to use the HTML5 Boilerplate template with all the elements necessary to begin developing. You can download it from the project website or in their official repository, or simply copy the following file to your `app/index.html` since we have already made some changes according to our project:

```
<!doctype html>
<html lang="es-ES">
  <head>
    <meta charset="utf-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <title></title>
    <meta name="description" content="">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <!– Place favicon.ico in the root directory –>
    <link rel="stylesheet" href="/stylesheets/main.css">
  </head>
  <body>
    <!–[if lt IE 8]>
    <p class="browserupgrade">
      You are using an <strong>outdated</strong> browser. Please <a href="http://browsehappy.com
    </p>
    <![endif]–>

    <p>Hola Mundo! Esto es HTML5 Boilerplate.</p>

    <!– Google Analytics: change UA-XXXXX-X to be your site's ID. –>
    <script>
      (function(b,o,i,l,e,r){b.GoogleAnalyticsObject=l;b[l]||(b[l]=
      function(){(b[l].q=b[l].q||[]).push(arguments)});b[l].l=+new Date;
      e=o.createElement(i);r=o.getElementsByTagName(i)[0];
      e.src='//www.google-analytics.com/analytics.js';
      r.parentNode.insertBefore(e,r)}(window,document,'script','ga'));
      ga('create','UA-XXXXX-X','auto');ga('send','pageview');
    </script>
  </body>
</html>
```

This will be the only complete HTML page that we have, since the rest will be templates that will dynamically load depending on the application status or user interaction. This is what is currently known as **Single Page Applications (SPA)**.

4.2 Installing Dependencies

We are going to install the *front end* libraries that we need with the tool **Bower**. First of all, we need to create the file `.bowerrc` in the project's root directory with the following content:

```
{
  "directory": "app/lib"
}
```

This ensures that each library we install remains saved in the directory `app/lib`. If we didn't have this file, the libraries would be installed in the default directory `bower_components` in the

root.

We are going to prepare our HTML and implement a new Gulp task that will make life easier. The task is `gulp-inject`, which will take the files that we have in the style and script folders and inject them as links into the HTML.

We add the following comment tags to the HTML where the CSS links and the JavaScript scripts will go:

```
<!doctype html>
<html lang="en">
  <head>
    […]
    <!– bower:css –>
    <!– endbower –>
    <!– inject:css –>
    <!– endinject –>
  </head>
  <body>
    […]
    <!– bower:js –>
    <!– endbower –>
    <!– inject:js –>
    <!– endinject –>
  </body>
</html>
```

Lines like the comments `<!– inject:css –>` and `<!– inject:js –>` will be read by the Gulp task that we are going to write next that will inject the files. And lines like the comments `<– bower:css –>` and `<!– bower:js –>` will inject the libraries that we install with Bower. It will put them in one place or another in the HTML depending on whether they are style (css) or script (js) files. To implement the task, we must first install the required plugins:

```
$ npm install –save-dev gulp-inject
$ npm install –save-dev wiredep
```

Then we can insert the following tasks in `Gulpfile.js` and update `watch` and `default`:

```
var inject  = require('gulp-inject');
var wiredep = require('wiredep').stream;


// Inject into HTML the path of JS scripts and CSS files
gulp.task('inject', function() {
  var sources = gulp.src([ './app/scripts/**/*.js', './app/stylesheets/**/*.css' ]);
  return gulp.src('index.html', { cwd: './app' })
    .pipe(inject(sources, {
      read: false,
      ignorePath: '/app'
    }))
    .pipe(gulp.dest('./app'));
});
// Inject the path of Bower dependencies into HTML
gulp.task('wiredep', function () {
  gulp.src('./app/index.html')
    .pipe(wiredep({
      directory: './app/lib'
    }))
    .pipe(gulp.dest('./app'));
});

gulp.task('watch', function() {
  […]
  gulp.watch(['./app/stylesheets/**/*.styl'], ['css', 'inject']);
```

```
  gulp.watch(['./app/scripts/**/*.js', './Gulpfile.js'], ['jshint', 'inject']);
  gulp.watch(['./bower.json'], ['wiredep']);
});

gulp.task('default', ['server', 'inject', 'wiredep', 'watch']);
```

With Gulp running in one terminal window, we can install the dependencies with Bower from another, for example, starting with those of Angular and the Bootstrap CSS framework:

```
$ bower install –save angular
$ bower install –save bootstrap
```

If we open `index.html` we can see that Gulp has automatically injected the AngularJS library and the files `main.css` and `main.js` that we had as styles and the Bootstrap scripts have also been injected:

```
<!doctype html>
<html lang="en">
  <head>
    […]
    <!– bower:css –>
    <link rel="stylesheet" href="lib/bootstrap/dist/css/bootstrap.css" />
    <!– endbower –>
    <!– inject:css –>
    <link rel="stylesheet" href="/stylesheets/main.css">
    <!– endinject –>
  </head>
  <body>
    […]
    <!– bower:js –>
    <script src="lib/jquery/dist/jquery.js"></script>
    <script src="lib/angular/angular.js"></script>
    <script src="lib/bootstrap/dist/js/bootstrap.js"></script>
    <!– endbower –>
    <!– inject:js –>
    <script src="/scripts/main.js"></script>
    <!– endinject –>
  </body>
</html>
```

If for any reason we don't need to use one of the libraries that we have installed, we can delete them with `uninstall` as in the following example:

```
$ bower uninstall –save angular
```

4.3 Application Modules

All of the files that contain the functionality of our application are in the directory `app/scripts`.

In Angular, it is ideal to create modules for each functionality that we have in the application. If the application isn't very large, as in the case of the example in this book, we can gather related functionalities in the same file and only separate controllers, services, directives, etc… This way we have a scalable and maintainable application.

The project that we are going to complete for this example will be a blog, with posts and comments.

It is ideal for us to receive the data from a RESTful API that returns the data in JSON

format. The development of the *back end* is beyond the scope of this book, which is why we are going to use the JSONPlaceholder project, which provides a test API for testing and prototyping that returns JSONs of blog posts and comments.

The URLs of the API that we are going to use are:

URL	Method
/posts	POST
/posts	GET
/posts/:postId	GET
/comments	GET
/comments/:commentId	GET
/users	GET
/users/:userId	GET

The following examples are API Requests and responses from JSONPlaceholder

GET /posts/1:


```
{
  "userId": 1,
  "id": 1,
  "title": "sunt aut facere repellat provident occaecati excepturi optio reprehenderit",
  "body": "quia et suscipit\nsuscipit recusandae consequuntur expedita et cum\nreprehenderit mol
}
```


GET /comments/1:


```
{
  "postId": 1,
  "id": 1,
  "name": "id labore ex et quam laborum",
  "email": "Eliseo@gardner.biz",
  "body": "laudantium enim quasi est quidem magnam voluptate ipsam eos\ntempora quo necessitatib
}
```


GET /users/1:


```
{
  "id": 1,
  "name": "Leanne Graham",
  "username": "Bret",
  "email": "Sincere@april.biz",
  "address": {
    "street": "Kulas Light",
    "suite": "Apt. 556",
    "city": "Gwenborough",
    "zipcode": "92998-3874",
    "geo": {
      "lat": "-37.3159",
      "lng": "81.1496"
    }
  },
  "phone": "1-770-736-8031 x56442",
  "website": "hildegard.org",
  "company": {
    "name": "Romaguera-Crona",
    "catchPhrase": "Multi-layered client-server neural-net",
```

```
    "bs": "harness real-time e-markets"
  }
}
```

4.3.1 Architecture

The file structure that we are going to use in the directory `app` will be the following:

```
app/
 ├── lib/
 ├── scripts/
 |      ├── services.js
 |      ├── controllers.js
 |      └── app.js
 ├── stylesheets/
 ├── views/
 |       ├── post-detail.tpl.html
 |       ├── post-list.tpl.html
 |       └── post-create.tpl.html
 |
 └── index.html
```

> **Note**: The ideal, for an Angular application to be more modular, is to separate the functionalities by folders, and that each functionality has its services, controllers, routes, etc… separated from the rest. It would be something like this:

```
app/
 ├── lib/
 ├── modules/
 |      ├── users/
 |      |      ├── module.js
 |      |      ├── controllers.js
 |      |      └── services.js
 |      ├── posts/
 |      |      ├── module.js
 |      |      ├── controllers.js
 |      |      ├── services.js
 |      |      └── views/
 |      |            ├── post-detail.tpl.html
 |      |            ├── post-list.tpl.html
 |      |            └── post-create.tpl.html
 |      └── comments
 |            ├── module.js
 |            ├── controllers.js
 |            └── services.js
 ├── stylesheets/
 |
 └── index.html
```

> Including separating the styles by functionality. But this is beyond the scope of this book, since the functionality that we are going to carry out for the example is very simple and can be written with the previous architecture.

We are going to install the following libraries that we will need, which we'll do like always with Bower, with the flag `–save-dev` so that they remain saved in the `bower.json` file and Gulp can run the appropriate task.

```
$ bower install –save angular-route
$ bower install –save angular-resource
```

- `angular-route` allows us to make use of the directive `$routeProvider` in order to manage URLs from the browser and display one page or another to the user.

- `angular-resource` meanwhile, allows us to use the directive `$resource` which allows us to manage AJAX requests to REST resources in a simpler way and with cleaner syntax, instead of using the directives `$http.get` or `$http.post`.

To tell the HTML that we are using an Angular application, we have to put the attribute `ng-app` in part of our `index.html`; in this case we will put it in the `body` tag so that it includes the entire page. We will call this application "blog":

index.html

```
<body ng-app="blog">
  <div class="container-fluid">
    <div class="row">
      <aside class="col-sm-3">
          <a ng-href="/new">Write Post</a>
      </aside>
      <section class="col-sm-9" ng-view></section>
    </div>
  </div>
</body>
```

In the above example, in addition to the attribute `ng-app="blog"` we have added some layout using Bootstrap classes, such as `col-sm-3` and `col-sm-9`, which allow us to have 2 columns on the page, one of 3/12 size that can be used for information on the author of the blog, and another of 9/12 size for the content and the list of blog posts.

This last column in turn contains the attribute `ng-view` that tells the Angular `blog` application that it will load the partial views in that space, which we will manage thanks to the routing of `$routeProvider` later on.

4.3.2 Main

We begin with the first of the JS scripts that will give the application functionality, in this case the main file `app/scripts/app.js`.

In all of the JS files, in addition to using the notation `"use strict";` we are going to group them by Angular modules and in turn as *closures*. This will help us so that the variables we use in that function only remain defined within it, and so that errors don't appear when it is minified. The file would start like this:

```
(function () {
  'use strict';
  // Functionality goes here.
})();
```

Next we configure the `blog` module with the dependency `ngRoute` which we get by adding

the library `angular-route`.

```
(function () {
  ‘use strict’;

  angular.module(‘blog’, [‘ngRoute’]);

})();
```

We are going to create a configuration function to tell the application which routes to listen for in the browser and which partial views to load in each case. We do this with the directive `$routeProvider`.

```
function config ($locationProvider, $routeProvider) {
  $locationProvider.html5Mode(true);

  $routeProvider
    .when(‘/’, {
      templateUrl: ‘views/post-list.tpl.html’,
      controller: ‘PostListController’,
      controllerAs: ‘postlist’
    })
    .when(‘/post/:postId’, {
      templateUrl: ‘views/post-detail.tpl.html’,
      controller: ‘PostDetailController’,
      controllerAs: ‘postdetail’
    })
    .when(‘/new’, {
      templateUrl: ‘views/post-create.tpl.html’,
      controller: ‘PostCreateController’,
      controllerAs: ‘postcreate’
    });
}
```

The line `$locationProvider.html5Mode(true)` is important, since it allows URLs not to have the character `#` at the beginning of them, which Angular uses by default. This keeps them cleaner.

> **Important** In order for the HTML5 mode to function correctly, you must add the tag `base` in your index.html document, within the headers `<head>` like this:

```
<head>
  <base href=”/”>
  …
</head>
```

You can find more information in this link

We have added three routes, the root or main `/` route, one that details a blog post `post/:postId` and one with the form for publishing a new post. Each one of these loads a partial view that we will create in a few moments and will be stored in the `app/views` folder. Each view also has an associated controller that will manage the associated functionality.

These will be `PostListController`, to manage the list of posts and `PostDetailController` to manage a specific post and `PostCreateController`. We will declare each of them in a separate file and module, `blog.controllers`, so in order to make use of them in this file, we must include it as a dependency when declaring the module, as we did with `ngRoute`.

The attribute `controllerAs` allows us to use controller variables within the HTML template

without needing to use the directive `$scope`.

```
(function () {
  'use strict';

  angular.module('blog', ['ngRoute', 'blog.controllers']);
  [...]
})
```

To finish with this file, we just need to associate the `config` function we created with the module:

```
angular
   .module('blog')
   .config(config);
```

The finished `app/scripts/app.js` file would be as follows:

```
(function () {
  'use strict';

  angular.module('blog', ['ngRoute', 'blog.controllers']);

  function config ($locationProvider, $routeProvider) {
    $locationProvider.html5Mode(true);

    $routeProvider
      .when('/', {
        templateUrl: 'views/post-list.tpl.html',
        controller: 'PostListController',
        controllerAs: 'postlist'
      })
      .when('/post/:postId', {
        templateUrl: 'views/post-detail.tpl.html',
        controller: 'PostDetailController',
        controllerAs: 'postdetail'
      })
      .when('/new', {
        templateUrl: 'views/post-create.tpl.html',
        controller: 'PostCreateController',
        controllerAs: 'postcreate'
      });
  }

  angular
    .module('blog')
    .config(config);

})();
```

4.3.3 Services

Before implementing the controller functions, we are going to create some services with the directive `$resource` that will allow us to make AJAX calls to the API in a simpler way. We will create three factories (that's what they're called), one for the **Posts**, another for the **Comments** and another for the **Users**. Each one of these will be associated with a REST API URL.

As in the previous file, we begin by creating a closure, and the name of the module (`blog.services`) to which we include the dependency `ngResource` contained in the library `angular-resource` that allows us to use the directive `$resource`:

```
(function () {
  'use strict';

  angular.module('blog.services', ['ngResource']);

})();
```

We then create 3 functions, one for each `factory` that will point to a URL. The server base URL we will use as a constant.

```
  function Post ($resource, BaseUrl) {
    return $resource(BaseUrl + '/posts/:postId', { postId: '@_id' });
  }

  function Comment ($resource, BaseUrl) {
    return $resource(BaseUrl + '/comments/:commentId', { commentId: '@_id' });
  }

  function User ($resource, BaseUrl) {
    return $resource(BaseUrl + '/users/:userId', { userId: '@_id' });
  }
```

We associate these factories with the created module, and also create a constant `BaseUrl` that points to the URL of the API:

```
angular
    .module('blog.services')
    .constant('BaseUrl', 'http://jsonplaceholder.typicode.com')
    .factory('Post', Post)
    .factory('Comment', Comment)
    .factory('User', User);
```

The finished `app/scripts/services.js` file would be:

```
(function () {
  'use strict';

  angular.module('blog.services', ['ngResource']);

  function Post ($resource, BaseUrl) {
    return $resource(BaseUrl + '/posts/:postId',
      { postId: '@_id' });
  }

  function Comment ($resource, BaseUrl) {
    return $resource(BaseUrl + '/comments/:commentId',
      { commentId: '@_id' });
  }

  function User ($resource, BaseUrl) {
    return $resource(BaseUrl + '/users/:userId',
      { userId: '@_id' });
  }

  angular
    .module('blog.services')
    .constant('BaseUrl', 'http://jsonplaceholder.typicode.com')
    .factory('Post', Post)
    .factory('Comment', Comment)
    .factory('User', User);

})();
```

4.3.4 Controllers

Having already created the services, we can move on to implementing the partial view

controllers and therefore the application. As always, we create an Angular module, in this case `blog.controllers`, with the dependency `blog.services` and we include it within a *closure*:

```
(function () {
  'use strict';

  angular.module('blog.controllers', ['blog.services']);

})();
```

The first controller, `PostListController`, is the simplest of all, the function would be like this:

```
function PostListController (Post) {
    this.posts = Post.query();
  }
```

What we are doing here is an AJAX call to the URL `http://jsonplaceholder.typicode.com/posts` and that returns the result within the variable `posts`. This is achieved using the service `Post`.

Now we turn to the detail view controller, `PostDetailController`. If we only want to see the contents of a specific post and its comments, we use the ID of the post that we want to show, which will provide the browser route via `$routeParams` the function would be like this:

```
function PostDetailController ($routeParams, Post, Comment) {
  this.post = Post.query({ id: $routeParams.postId });
  this.comments = Comment.query({ postId: $routeParams.postId });
}
```

But, if we want to see the information of the user that wrote the post, how do we do it? The first thing you think of is something like this:

```
function PostDetailController ($routeParams, Post, Comment, User) {
  this.post = Post.query({ id: $routeParams.postId });
  this.comments = Comment.query({ postId: $routeParams.postId });
  this.user = User.query({ id: this.post.userId });
}
```

But our friend JavaScript isn't sequential, it's asynchronous, and when we run the 3rd line, `this.post` still contains nothing and shows us an error message, because at that time `this.post.userId` is `undefined`. How do we fix it? Using the directive `$promise` and in a callback function:

```
function PostDetailController ($routeParams, Post, Comment) {
  this.post = {};
  this.comments = {};
  this.user = {}

  var self = this; // To save the reference

  Post.query({ id: $routeParams.postId })
    .$promise.then(
      //Success
      function (data) {
        self.post = data[0];
        self.user = User.query({ id: self.user.userId });
      },
      //Error
      function (error) {
        console.log(error);
```

```
      }
    );

    this.comments = Comment.query({ postId: $routeParams.postId });
  }
```

This way, only when we have data relating to the post, can we access it to make more queries, as in this case, for the data of a user related to the post.

The next controller is for creating a new post:

```
function PostCreateController (Post) {
    var self = this;

    this.create = function() {
      Post.save(self.post);
    };
  }
```

The finished file would be like this:

```
(function() {

  angular
  .module('blog.controllers', ['blog.services'])
  .controller('PostListController', PostListController)
  .controller('PostCreateController', PostCreateController)
  .controller('PostDetailController', PostDetailController)

  function PostListController (Post) {
    this.posts = Post.query();
  };

  function PostCreateController (Post) {
    var self = this;

    this.create = function() {
      Post.save(self.post);
    };
  };

  function PostDetailController ($routeParams, Post, Comment) {
    this.post = {};
    this.comments = {};
    this.user = {}

    var self = this;

    Post.query({ id: $routeParams.postId })
    .$promise.then(
      //Success
      function (data) {
        self.post = data[0];
        self.user = User.query({ id: self.user.userId });
      },
      //Error
      function (error) {
        console.log(error);
      }
    );

    this.comments = Comment.query({ postId: $routeParams.postId });
  };

})();
```

Thanks to the resource `Post` we can use the method `save()` which is responsible for making a POST request to the API that we are managing. Since the API we are using is *fake* the POST will not be stored, but in a real API that would occur.

We add the created functions to the module `blog.controllers` and then we can use the variables `this.posts`, `this.post`, `this.comments` and `this.user` in our partial views, as well as collect the data that is sent in the form for creating posts. Next we will create the partial views or templates.

4.3.5 Partial views

views/post-list.tpl.html

This will be the view that shows the list of posts that the API returns to us. It is managed by the controller `PostListController`, which if we recall, contained the variable `this.posts` where they are all stored. To access that variable from the HTML we only need to indicate the alias that we gave the controller in the `config` function of `app/scripts/app.js` that was `postlist`, and then using point notation access the attribute `posts` with `postlist.posts`.

Since it is an array of elements, we can iterate over them using the Angular directive `ng-repeat` in the following manner:

```
<ul class="blog-post-list">
   <li class="blog-post-link" ng-repeat="post in postlist.posts">
     <a ng-href="/post/{{ post.id }}">{{ post.title }}</a>
   </li>
 </ul>
```

The previous code runs through the array of posts, and for each one of them, creates an `<li>` element in the HTML, with a link that points to the post `id` with `post.id` and the title of the post with `post.title`.

If we access `http://localhost:8080` in a browser we will see a list of titles, all returned via the API that provides us `http://jsonplaceholder.com/posts`

We have also added classes to the HTML tags in order to use them later in the CSS files to add style to them.

views/post-detail.tpl.html

Now we are going to create the detail view of a specific post, where we will use the variables `post`, `comments` and `user` that are used by the controller `PostDetailController` via the alias `postdetail`.

First we design the element `<article>` where we will place the contents of the post and the user who wrote it, which are stored in the variables `postdetail.post` and `postdetail.user`:

```
<article class="blog-post">
  <header class="blog-post-header">
    <h1>{{ postdetail.post.title }}</h1>
  </header>
  <p class="blog-post-body">
    {{ postdetail.post.body }}
  </p>
  <p>
    Escrito por: <strong>{{ postdetail.user[0].name }}</strong>
      <span class="fa fa-mail"></span> {{ postdetail.user[0].email }}
  </p>
</article>
```

Then we add the element `<aside>` where the list of associated comments will be, stored in

the variable `postdetail.comments`. Since it is an array of elements, we can use the directive `ng-repeat` as we did in the list of posts, to display them in the view:

```
<aside class="comments">
  <header class="comments-header">
    <h3>
      <span class="fa fa-comments"></span>
      Comments
    </h3>
  </header>
  <ul class="comments-list">
    <li class="comment-item" ng-repeat="comment in postdetail.comments">
      <span class="fa fa-user"></span>
      <span class="comment-author">{{ comment.email }}</span>
      <p class="comment-body">
        {{ comment.body }}
      </p>
    </li>
  </ul>
</aside>
```

views/post-create.tpl.html

This will be the view that we see when we click the link `Create Post`; it is simply a form with the necessary fields to enter the title of the post and its contents:

```
<section>
  <form name="createPost" role="form" ng-submit="postcreate.create()">

    <fieldset class="form-group">
      <input class="form-control input-lg"
             type="text"
             placeholder="Post title"
             ng-model="postcreate.post.title">
    </fieldset>
    <fieldset class="form-group">
      <textarea class="form-control input-lg"
                placeholder="Post text"
                ng-model="postcreate.post.text"></textarea>
    </fieldset>
    <hr>

    <button class="btn btn-primary btn-lg"
            type="submit" ng-disabled="!createPost.$valid">
      <span class="fa fa-rocket"> Publish</span>
    </button>

  </form>
</section>
```

The classes used are part of the Bootstrap CSS Framework which we use to add style to the example.

If we look closely, both the `input` and the `textarea` have the attribute `ng-model`. This will allow Angular, via the controller that we previously defined for this form, to collect this data and for us to send it to the API. They will be included within the object `this.post`.

Another important thing we have done is disable the "Publish" button until the form is completed. We achieve this with the attribute `ng-disabled="!createPost.$valid"` with `createPost` as the `name` we have given the form.

The processing of this form is done using the `this.create()` function of the `PostCreateController` that we call from the attribute `ng-submit`.

This completes the layout and functionality of this example application with AngularJS, using factories as a model to obtain the data from an external API. Now it’s time to add some drops of CSS style (using **Stylus**) to finish off our app.

5. Design with CSS Preprocessors

Although you can design a site from scratch with CSS, for this example we have started from one of the best known frameworks, Bootstrap which as its name suggests, is for starting quickly, having a grid layout (columns) and some predefined elements with a pleasant style.

Aside from using Bootstrap, you can add style to elements of your website with CSS, or use a preprocessor such as **LESS, Sass** or **Stylus** which make design easier, using variables, functions, etc. In our case we are going to use **Stylus**, which uses a syntax very similar to Python, based on tabs; it does not use keys `{…}`, or `:`, or `;`.

Stylus works with Node.js. To make it run we must globally install it on our computer with `npm`:

```
$ npm install -g stylus
```

We are going to save the `.styl` files in `app/stylesheets`, and the Gulp task that we configured in chapter 2 will compile them directly to CSS and we will see the changes reflected in the browser in real-time.

5.1 FontAwesome

Font Awesome is a CSS library of font icons, ideal for use in all kinds of resolutions, since the font files are vectors and don't *pixelate*. To use it in our application we can install it via Bower.

```
$ bower install –save fontawesome
```

And from now on when we want to use an icon in our HTML we only need to add the following class to a `<span>` element.

```
<span class="fa fa-iconName"></span>
```

For this example we have used several in the template `app/views/post-detail.tpl.html` such as:

```
<span class="fa fa-mail"></span> {{ postdetail.user[0].email }}
```

to display an icon of an envelope that represents email, or:

```
<span class="fa fa-comments"></span>
```

to display speech bubbles to represent comments, or finally:

```
<span class="fa fa-user"></span>
```

to represent a user icon.

There is a complete list of all the icons that you can use in their documentation


Font Awesome Icons
fortawesome.github.io/Font-Awesome/icons/
71 New Icons in 4.1
fa-automobile (alias)
fa-bank (alias)
fa-behance
fa-behance-square
fa-bomb
fa-building
fa-cab (alias)
fa-car
fa-child
fa-circle-o-notch
fa-circle-thin
fa-codepen
fa-cube
fa-cubes
fa-database
fa-delicious
fa-deviantart
fa-digg
fa-drupal
fa-empire
fa-envelope-square
fa-fax
fa-file-archive-o
fa-file-audio-o
fa-file-code-o
fa-file-excel-o
fa-file-image-o
fa-file-movie-o (alias)
fa-file-pdf-o
fa-file-photo-o (alias)
fa-file-picture-o (alias)
fa-file-powerpoint-o
fa-file-sound-o (alias)
fa-file-video-o
fa-file-word-o
fa-file-zip-o (alias)
fa-ge (alias)
fa-git
fa-git-square
fa-google
fa-graduation-cap
fa-hacker-news
fa-header
fa-history
fa-institution (alias)
fa-joomla
fa-jsfiddle
fa-language
fa-life-bouy (alias)
fa-life-ring
fa-life-saver (alias)
fa-mortar-board (alias)
fa-openid
fa-paper-plane
fa-paper-plane-o
fa-paragraph
fa-paw
fa-pied-piper
fa-pied-piper-alt
fa-pied-piper-square (alias)
fa-qq
fa-ra (alias)
fa-rebel
fa-recycle
fa-reddit
fa-reddit-square
fa-send (alias)
fa-send-o (alias)
fa-share-alt
fa-share-alt-square
fa-slack
fa-sliders
fa-soundcloud
fa-space-shuttle
fa-spoon
fa-spotify
fa-steam
fa-steam-square
fa-stumbleupon
fa-stumbleupon-circle
fa-support (alias)
fa-taxi
fa-tencent-weibo
fa-tree
fa-university
fa-vine
fa-wechat (alias)
fa-weixin
fa-wordpress
fa-yahoo

which is very comprehensive and easy to use.

5.2 Typographic Fonts

The browser includes a series of fonts by default, but we can use others just by including them in the CSS, such as Google Web Fonts. In my case I chose the `Raleway` typeface to use in this example.

To do this we create a `fonts.styl` file where we add the following code:

```
@font-face
  font-family 'Raleway-Light'
  font-style normal
  font-weight 300
                    src           local('Raleway           Light'),           local('Raleway-
Light'),                          url('//themes.googleusercontent.com/static/fonts/raleway/v7/-
_Ctzj9b56b8RgXW8FArib3hpw3pgy2gAi-Ip7WPMi0.woff') format('woff');

@font-face
  font-family 'Raleway'
  font-style normal
  font-weight 400
  src local('Raleway'), url('//themes.googleusercontent.com/static/fonts/raleway/v7/cIFypx4yrWPI
```

5.3 Application Styles

In order to link this file with the main file we use `@import`. We add it in `main.styl` which is the file we use to tell to Gulp to process and generate the `css` files. To use it, we assign it the to the attribute `font-family` of the `body`, like this:

```
@import 'fonts'
body
  font-family 'Raleway-Light'
  font-size 16px
```

Now our application will have a different font style. We are going to edit the style of the lists of posts a bit:

```
.blog-post-list
  list-style-type none

  .blog-post-link
    font-family 'Raleway'
    margin .5em 0 1em 0
    padding 0 0 .25em 0
    border-bottom 1px solid b_silver
```

And now the title of the post in the detail view:

```
.blog-post
  .blog-post-header
    h1
      font-family 'Raleway'
```

And the comments:

```
.comments

  .comments-list
```

```
    list-style-type none

    .comment-item
      border 1px solid b_silver
      border-radius 5px
      padding .5em
      margin .5em

      .comment-author
        font-family 'Raleway'
        font-weight bold

      .comment-body
        font-style italic
```

If we look closely, we indicated that the border color of the comment box be `b_silver`, and that color doesn't exist in standard HTML/CSS. We are going to add it as a variable in Stylus so that we can reuse it later if we want:

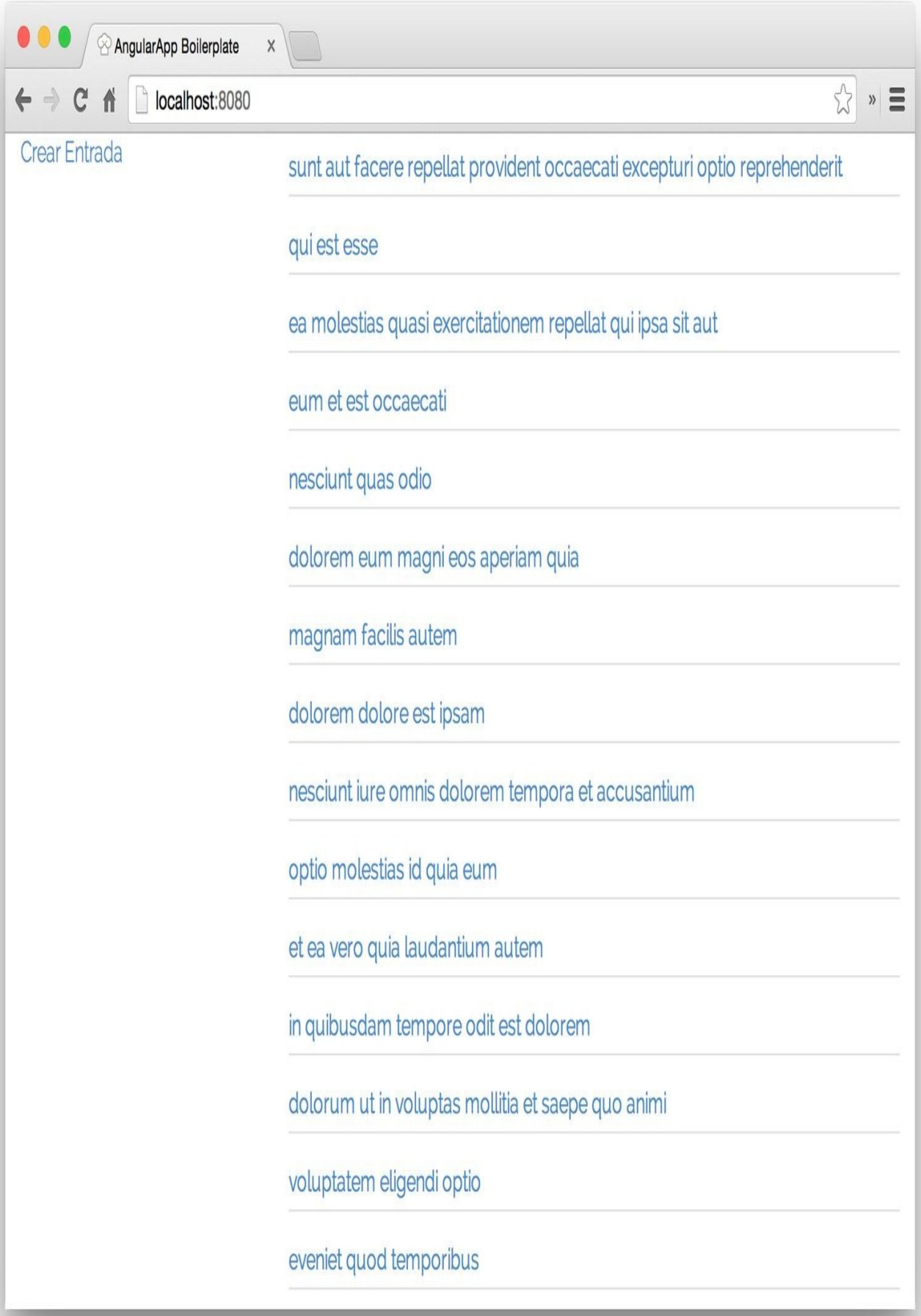
```
b_silver =  #ddd
```

The finished `main.styl` file would be like this:

```
@import 'fonts'

b_silver = #ddd

body
  font-family 'Raleway-Light'
  font-size 16px

.blog-post-list
  list-style-type none

  .blog-post-link
    font-family 'Raleway'
    margin .5em 0 1em 0
    padding 0 0 .25em 0
    border-bottom 1px solid b_silver

.blog-post

  .blog-post-header
    h1
      font-family 'Raleway'

.comments

  .comments-list
    list-style-type none

    .comment-item
      border 1px solid b_silver
      border-radius 5px
      padding .5em
      margin .5em

      .comment-author
        font-family 'Raleway'
        font-weight bold

      .comment-body
        font-style italic
```

And this is what our app would look like, if we run `Gulp` in the terminal and open the browser to the URL `http://localhost:8080`:

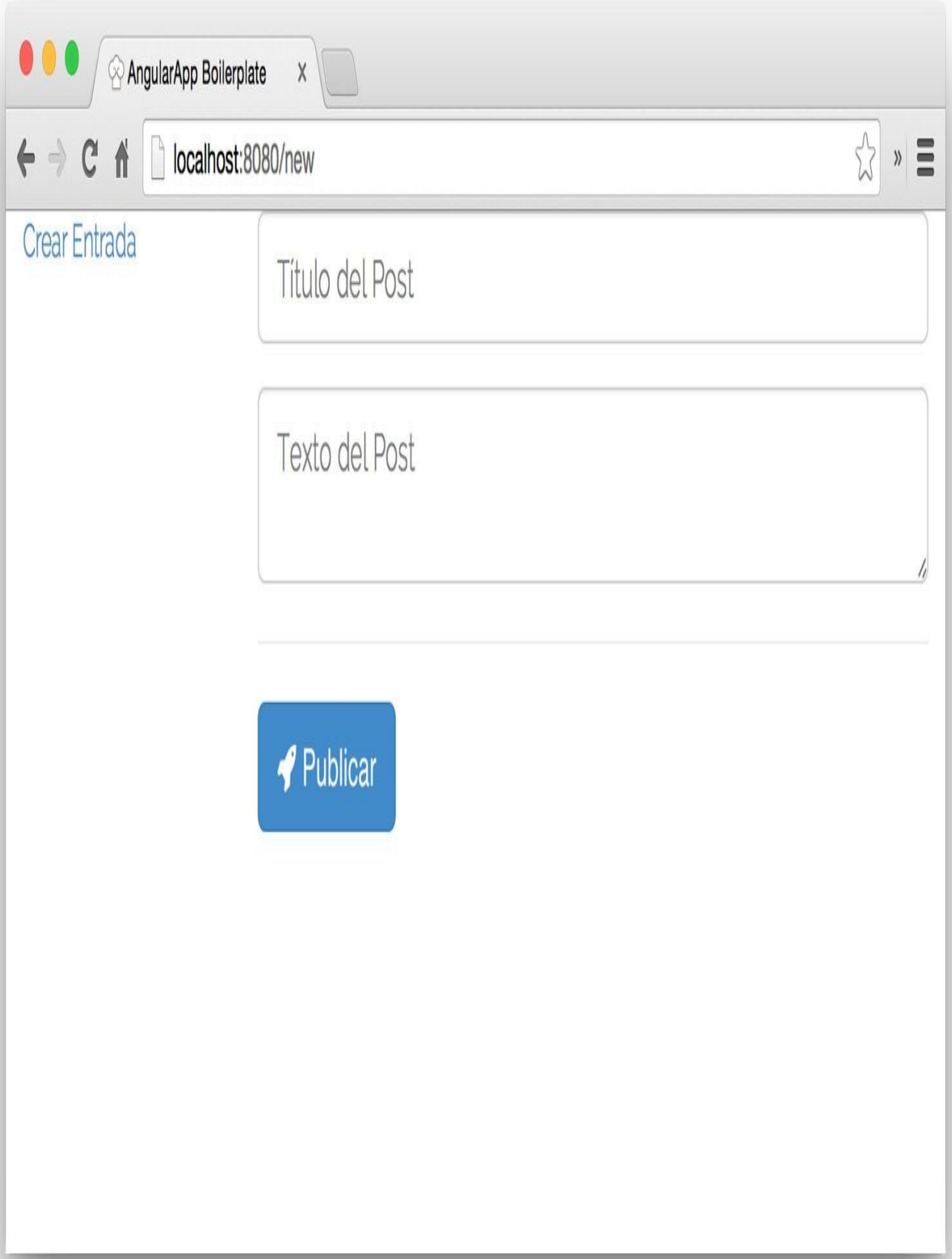
Post List


AngularApp Boilerplate
localhost:8080
Crear Entrada
sunt aut facere repellat provident occaecati excepturi optio reprehenderit
qui est esse
ea molestias quasi exercitationem repellat qui ipsa sit aut
eum et est occaecati
nesciunt quas odio
dolorem eum magni eos aperiam quia
magnam facilis autem
dolorem dolore est ipsam
nesciunt iure omnis dolorem tempora et accusantium
optio molestias id quia eum
et ea vero quia laudantium autem
in quibusdam tempore odit est dolorem
dolorum ut in voluptas mollitia et saepe quo animi
voluptatem eligendi optio
eveniet quod temporibus

Post Detail


AngularApp Boilerplate
localhost:8080/post/1
Crear Entrada
sunt aut facere repellat provident occaecati excepturi optio reprehenderit
quia et suscipit suscipit recusandae consequuntur expedita et cum reprehenderit molestiae ut ut quas totam nostrum rerum est autem sunt rem eveniet architecto
Escrito por: Leanne Graham Sincere@april.biz
Comments
Eliseo@gardner.biz
laudantium enim quasi est quidem magnam voluptate ipsam eos tempora quo necessitatibus dolor quam autem quasi reiciendis et nam sapiente accusantium
Jayne_Kuhic@sydney.com
est natus enim nihil est dolore omnis voluptatem numquam et omnis occaecati quod ullam at voluptatem error expedita pariatur nihil sint nostrum voluptatem reiciendis et
Nikita@garfield.biz
quia molestiae reprehenderit quasi aspernatur aut expedita occaecati aliquam eveniet laudantium omnis quibusdam delectus saepe quia accusamus maiores nam est cum et ducimus et vero voluptates excepturi

Create a new post


AngularApp Boilerplate
localhost:8080/new
Crear Entrada
Título del Post
Texto del Post
Publicar

6. Optimizing for Production

We already have our application running, but it has several CSS files with parts of Bootstrap that we may never use. Too many JavaScript files, which involve multiple requests to the server and make our page take longer to load:

```
[…]
<link rel="stylesheet" href="lib/bootstrap/dist/css/bootstrap.css" />
<link rel="stylesheet" href="lib/angular-motion/dist/angular-motion.min.css" />
<link rel="stylesheet" href="lib/fontawesome/css/font-awesome.css" />
[…]
<script src="lib/jquery/dist/jquery.js"></script>
<script src="lib/angular/angular.js"></script>
<script src="lib/bootstrap/dist/js/bootstrap.js"></script>
<script src="lib/angular-animate/angular-animate.js"></script>
<script src="lib/angular-strap/dist/angular-strap.min.js"></script>
<script src="lib/angular-strap/dist/angular-strap.tpl.min.js"></script>
<script src="lib/angular-route/angular-route.js"></script>
<script src="lib/angular-resource/angular-resource.js"></script>
<script src="/scripts/templates.js"></script>
<script src="/scripts/services.js"></script>
<script src="/scripts/controllers.js"></script>
<script src="/scripts/app.js"></script>
[…]
```

Ideally, our HTML would only have one CSS file to call, and just one JS file. We should concatenate them into a single file and if possible minify them (removing blank spaces and leaving all the content on one line), so that the file weighs less and loads faster.

Doing it by hand is a rather tedious and unproductive task, which is why we have task automators like Gulp to do it for us.

We are going to install some Gulp plugins that will help us with these tasks, which we do via `npm`:

```
$ npm install –save-dev gulp-minify-css
$ npm install –save-dev gulp-angular-templatecache
$ npm install –save-dev gulp-uncss
$ npm install –save-dev gulp-if
```

Next we will look at what each plugin does and which of the tasks within `Gulpfile.js` we are going to implement them in.

6.1 Template Caching

The first task that we are going to implement is the caching of HTML templates, as a module of AngularJS, thanks to the plugin `gulp-angular-templatecache`, by adding the following to our `Gulpfile.js`:

```
var templateCache = require('gulp-angular-templatecache');

gulp.task('templates', function() {
  gulp.src('./app/views/**/*.tpl.html')
    .pipe(templateCache({
      root: 'views/',
      module: 'blog.templates',
      standalone: true
    }))
    .pipe(gulp.dest('./app/scripts'));
```

```
});
```

This task creates a `templates.js` file in the directory `app/scripts/` with the content of the HTML templates cached as String to use it as a dependency in the application.

To do this we need to include the new module created in the main file, `app/scripts/app.js`, which is the one that uses the views:

```
angular.module('blog', ['ngRoute', 'blog.controllers', 'blog.templates']);
```

With this we haven't minified anything, we have just created a new JavaScript file that contains the HTML templates, so we saved more calls.

6.2 Concatenation of JS and CSS Files

Now we begin to concatenate and minify. We are going to do it directly in the HTML and with a Gulp task. We take our `index.html` and we will add the following comments: `<!– build:css css/style.min.css –>` between the links to the CSS files to concatenate and minify them, and `<!– build:js js/app.min.js –>` and `<!– build:js js/vendor.min.js –>` to do the same but with the JS files.

```
[…]
<!– build:css css/style.min.css –>
<!– bower:css –>
<link rel="stylesheet" href="lib/bootstrap/dist/css/bootstrap.css" />
<link rel="stylesheet" href="lib/angular-motion/dist/angular-motion.min.css" />
<link rel="stylesheet" href="lib/fontawesome/css/font-awesome.css" />
<!– endbower –>
<!– inject:css –>
<link rel="stylesheet" href="/stylesheets/main.css">
<!– endinject –>
<!– endbuild –>
[…]
<!– build:js js/vendor.min.js –>
<!– bower:js –>
<script src="lib/jquery/dist/jquery.js"></script>
<script src="lib/angular/angular.js"></script>
<script src="lib/bootstrap/dist/js/bootstrap.js"></script>
<script src="lib/angular-animate/angular-animate.js"></script>
<script src="lib/angular-strap/dist/angular-strap.min.js"></script>
<script src="lib/angular-strap/dist/angular-strap.tpl.min.js"></script>
<script src="lib/angular-route/angular-route.js"></script>
<script src="lib/angular-resource/angular-resource.js"></script>
<!– endbower –>
<!– endbuild –>
<!– build:js js/app.min.js –>
<!– inject:js –>
<script src="/scripts/templates.js"></script>
<script src="/scripts/services.js"></script>
<script src="/scripts/controllers.js"></script>
<script src="/scripts/app.js"></script>
<!– endinject –>
<!– endbuild –>
[…]
```

We accompany this with the following task in `Gulpfile.js`:

```
var gulpif    = require('gulp-if');
var minifyCss = require('gulp-minify-css');
var useref    = require('gulp-useref');
var uglify    = require('gulp-uglify');

gulp.task('compress', function() {
```

```
  gulp.src('./app/index.html')
    .pipe(useref.assets())
    .pipe(gulpif('*.js', uglify({ mangle: false })))
    .pipe(gulpif('*.css', minifyCss()))
    .pipe(gulp.dest('./dist'));
});
```

This task will deposit linked files from `index.html` in the new directory for production that will be `/dist`, already minified.

We need `index.html` in this directory too, but without the comments and with the links to the new minified files. This is achieved with the following task:

```
gulp.task('copy', function() {
  gulp.src('./app/index.html')
    .pipe(useref())
    .pipe(gulp.dest('./dist'));
  gulp.src('./app/lib/fontawesome/fonts/**')
    .pipe(gulp.dest('./dist/fonts'));
});
```

In addition, this task copies the font files that we use in the library `fontawesome`.

Now we include all of these tasks within a new task that we'll call `build` and that we will run whenever we want to have our code ready for production:

```
gulp.task('build', ['templates', 'compress', 'copy']);
```

In the directory `/dist` we have the following files with this structure:

```
/dist
  ├── /js
  |    ├── vendor.min.js
  |    └── app.min.js
  ├── /css
  |    └── styles.min.css
  └── index.html
```

The files within the folders `/js` and `/css` being minified files, so that they occupy as little space as possible.

6.3 Production File Server

To test that everything is fine, we are going to create a new task in Gulp that will allow us to serve the files in the directory `/dist` like we did with the development version:

```
gulp.task('server-dist', function() {
  connect.server({
    root: './dist',
    hostname: '0.0.0.0',
    port: 8080,
    livereload: true,
    middleware: function(connect, opt) {
      return [ historyApiFallback ];
    }
  });
});
```

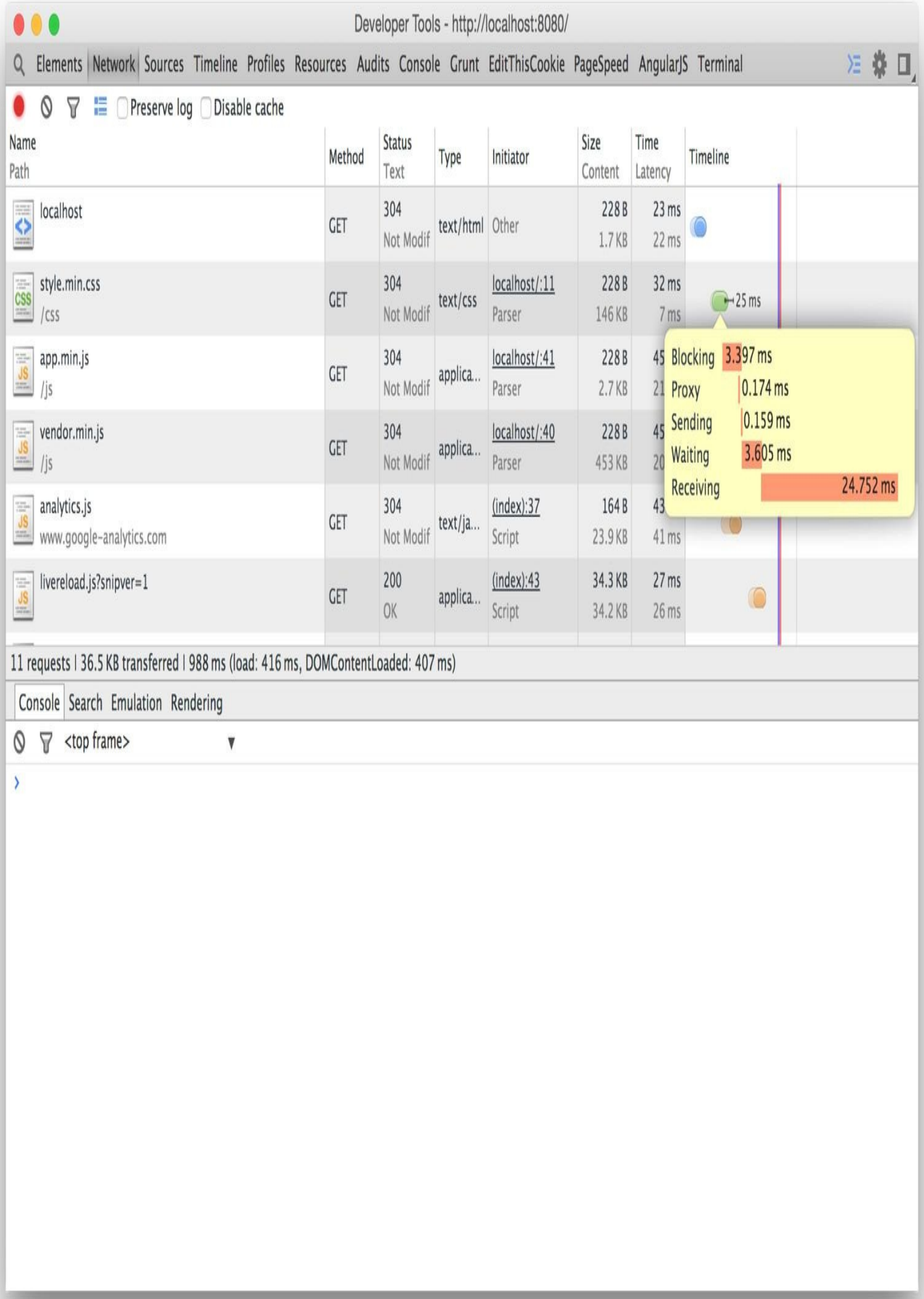
So, to test our application in production version before uploading it to a server, we must run the following in our terminal:

```
$ gulp build
$ gulp server-dist
```

And direct our browser to the URL `http://localhost:8080`. Here we see the same application running itself, but in this case `index.html` only has a link to one `.css` file and a couple of links to `.js` files, which are the library file and the application file. With them all minified, therefore reducing the number of HTTP requests and having less weight, their load time is shorter.

6.4 Reducing CSS Code

If we look closely at the browser's development tools, specifically the Network tab, we can see the requests that our application makes to the files that it links. We realize that the `styles.min.css` file occupies **146 kB**.


Developer Tools - http://localhost:8080/
Elements Network Sources Timeline Profiles Resources Audits Console Grunt EditThisCookie PageSpeed AngularJS Terminal
Preserve log Disable cache
Name Path
Method
Status Text
Type
Initiator
Size Content
Time Latency
Timeline
localhost
GET
304 Not Modif
text/html
Other
228 B 1.7 KB
23 ms 22 ms
style.min.css /css
GET
304 Not Modif
text/css
localhost/:11 Parser
228 B 146 KB
32 ms 7 ms
25 ms
app.min.js /js
GET
304 Not Modif
applica...
localhost/:41 Parser
228 B 2.7 KB
Blocking 3.397 ms
Proxy 0.174 ms
Sending 0.159 ms
Waiting 3.605 ms
Receiving 24.752 ms
vendor.min.js /js
GET
304 Not Modif
applica...
localhost/:40 Parser
228 B 453 KB
analytics.js www.google-analytics.com
GET
304 Not Modif
text/ja...
(index):37 Script
164 B 23.9 KB
41 ms
livereload.js?snipver=1
GET
200 OK
applica...
(index):43 Script
34.3 KB 34.2 KB
27 ms 26 ms
11 requests | 36.5 KB transferred | 988 ms (load: 416 ms, DOMContentLoaded: 407 ms)
Console Search Emulation Rendering
<top frame>

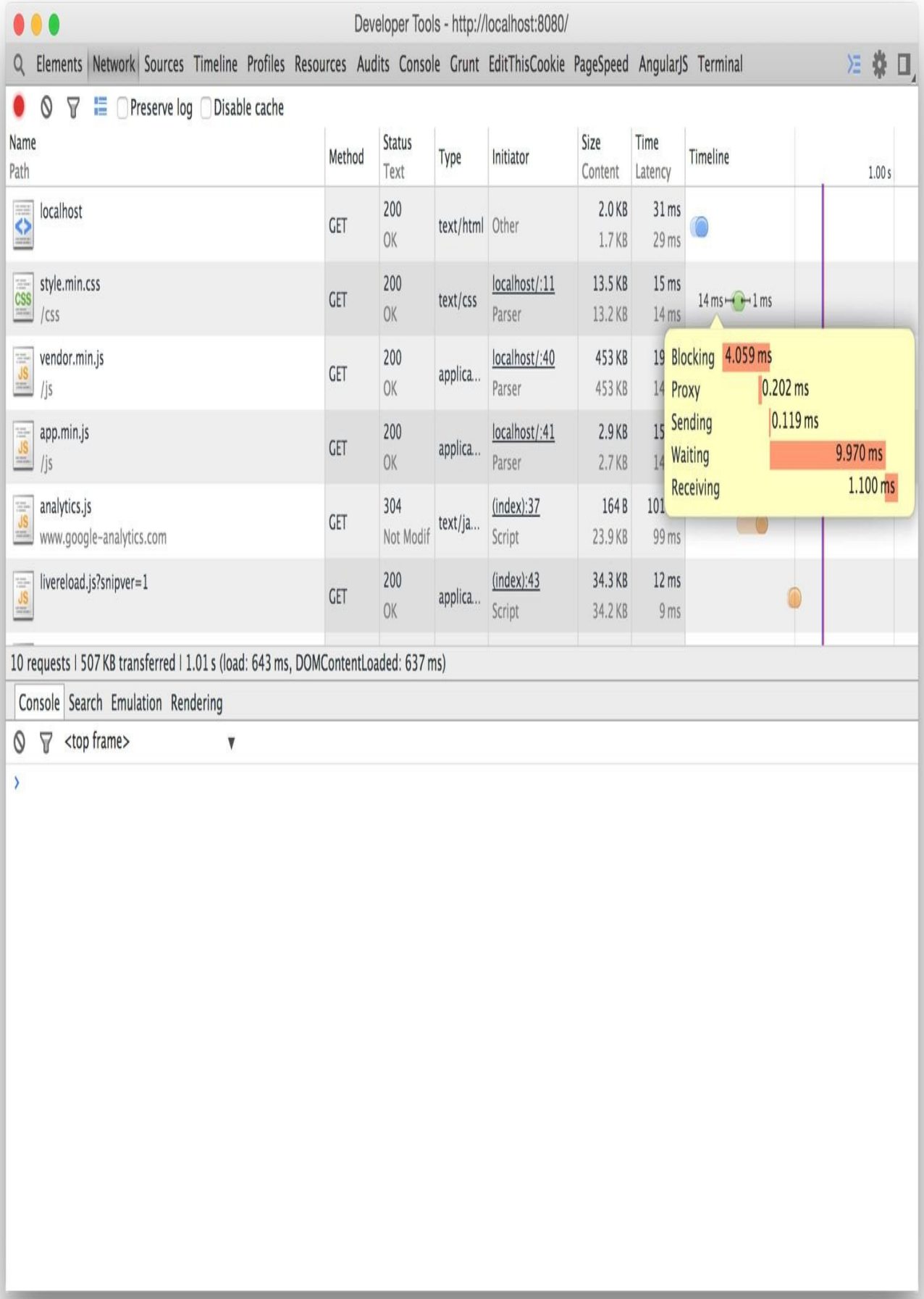
It isn’t a lot, but then we consider that this file contains all of the Bootstrap classes and all of the Font Awesome classes and we’re not using all of them. Is there a way to eliminate the classes that we don’t use? Yes there is, with a Gulp plugin called `gulp-uncss`. This allows us to indicate which CSS file we want to edit and the HTML files that it should focus on in order to remove the unused CSS, and then our very reduced CSS is ready. We install the plugin:

```
$ npm install –save-dev gulp-uncss
```

And implement the task:

```
var uncss = require('gulp-uncss');

gulp.task('uncss', function() {
  gulp.src('./dist/css/style.min.css')
    .pipe(uncss({
            html:  ['./app/index.html',  './app/views/post-detail.tpl.html',  './app/views/post-
list.tpl.html']
    }))
    .pipe(gulp.dest('./dist/css'));
});

gulp.task('build', ['templates', 'compress', 'copy', 'uncss']);
```

We run `gulp-build` and `gulp server-dist`, and when we look at the Network tab we have the following:


Developer Tools - http://localhost:8080/
Elements Network Sources Timeline Profiles Resources Audits Console Grunt EditThisCookie PageSpeed AngularJS Terminal
Preserve log Disable cache
Name Path
Method
Status Text
Type
Initiator
Size Content
Time Latency
Timeline
1.00 s
localhost
GET
200 OK
text/html
Other
2.0 KB 1.7 KB
31 ms 29 ms
style.min.css /css
GET
200 OK
text/css
localhost/:11 Parser
13.5 KB 13.2 KB
15 ms 14 ms
14 ms 1 ms
vendor.min.js /js
GET
200 OK
applica...
localhost/:40 Parser
453 KB 453 KB
Blocking 4.059 ms
Proxy 0.202 ms
Sending 0.119 ms
Waiting 9.970 ms
Receiving 1.100 ms
app.min.js /js
GET
200 OK
applica...
localhost/:41 Parser
2.9 KB 2.7 KB
analytics.js www.google-analytics.com
GET
304 Not Modif
text/ja...
(index):37 Script
164 B 23.9 KB
99 ms
livereload.js?snipver=1
GET
200 OK
applica...
(index):43 Script
34.3 KB 34.2 KB
12 ms 9 ms
10 requests | 507 KB transferred | 1.01 s (load: 643 ms, DOMContentLoaded: 637 ms)
Console Search Emulation Rendering
<top frame>

Now `styles.min.css` only occupies **13.5 kB** which allows it to load much faster.

5.5 Complete Gulpfile

Finally, this would be the complete code of the file `Gulpfile.js`:

```
// File: Gulpfile.js
'use strict';

var gulp      = require('gulp'),
    connect   = require('gulp-connect'),
    stylus    = require('gulp-stylus'),
    nib       = require('nib'),
    jshint    = require('gulp-jshint'),
    stylish   = require('jshint-stylish'),
    inject    = require('gulp-inject'),
    wiredep   = require('wiredep').stream,
    gulpif    = require('gulp-if'),
    minifyCss = require('gulp-minify-css'),
    useref    = require('gulp-useref'),
    uglify    = require('gulp-uglify'),
    uncss     = require('gulp-uncss'),
    angularFilesort = require('gulp-angular-filesort'),
    templateCache = require('gulp-angular-templatecache'),
    historyApiFallback = require('connect-history-api-fallback');

gulp.task('server', function() {
  connect.server({
    root: './app',
    hostname: '0.0.0.0',
    port: 8080,
    livereload: true,
    middleware: function(connect, opt) {
      return [ historyApiFallback ];
    }
  });
});

gulp.task('server-dist', function() {
  connect.server({
    root: './dist',
    hostname: '0.0.0.0',
    port: 8080,
    livereload: true,
    middleware: function(connect, opt) {
      return [ historyApiFallback ];
    }
  });
});

gulp.task('jshint', function() {
  return gulp.src('./app/scripts/**/*.js')
    .pipe(jshint('.jshintrc'))
    .pipe(jshint.reporter('jshint-stylish'))
    .pipe(jshint.reporter('fail'));
});

gulp.task('css', function() {
  gulp.src('./app/stylesheets/main.styl')
    .pipe(stylus({ use: nib() }))
    .pipe(gulp.dest('./app/stylesheets'))
    .pipe(connect.reload());
});

gulp.task('html', function() {
  gulp.src('./app/**/*.html')
    .pipe()
    .pipe(connect.reload());
});

gulp.task('inject', function() {
  return gulp.src('index.html', {cwd: './app'})
```

```
    .pipe(inject(
      gulp.src(['./app/scripts/**/*.js']).pipe(angularFilesort()), {
      read: false,
      ignorePath: '/app'
    }))
    .pipe(inject(
      gulp.src(['./app/stylesheets/**/*.css']), {
        read: false,
        ignorePath: '/app'
      }
    ))
    .pipe(gulp.dest('./app'));
});

gulp.task('wiredep', function () {
  gulp.src('./app/index.html')
    .pipe(wiredep({
      directory: './app/lib'
    }))
    .pipe(gulp.dest('./app'));
});

gulp.task('templates', function() {
  gulp.src('./app/views/**/*.tpl.html')
    .pipe(templateCache({
      root: 'views/',
      module: 'blog.templates',
      standalone: true
    }))
    .pipe(gulp.dest('./app/scripts'));
});

gulp.task('compress', function() {
  gulp.src('./app/index.html')
    .pipe(useref.assets())
    .pipe(gulpif('*.js', uglify({mangle: false })))
    .pipe(gulpif('*.css', minifyCss()))
    .pipe(gulp.dest('./dist'));
});

gulp.task('uncss', function() {
  gulp.src('./dist/css/style.min.css')
    .pipe(uncss({
              html:  ['./app/index.html',  './app/views/post-list.tpl.html',  './app/views/post-
detail.tpl.html']
    }))
    .pipe(gulp.dest('./dist/css'));
});

gulp.task('copy', function() {
  gulp.src('./app/index.html')
    .pipe(useref())
    .pipe(gulp.dest('./dist'));
  gulp.src('./app/lib/fontawesome/fonts/**')
    .pipe(gulp.dest('./dist/fonts'));
});

gulp.task('watch', function() {
  gulp.watch(['./app/**/*.html'], ['html', 'templates']);
  gulp.watch(['./app/stylesheets/**/*.styl'], ['css', 'inject']);
  gulp.watch(['./app/scripts/**/*.js', './Gulpfile.js'], ['jshint', 'inject']);
  gulp.watch(['./bower.json'], ['wiredep']);
});

gulp.task('default', ['server', 'templates', 'inject', 'wiredep', 'watch']);
gulp.task('build', ['templates', 'compress', 'copy', 'uncss']);
```

As you have seen throughout the development of this example web application, we have programmed in an agile way without having to constantly reload the browser in order to see the changes and forgetting to do repetitive tasks by leaving that work to *Gulp*. We have also used a CSS preprocessor that if our web project had been larger, would have helped us to maintain the design in a modular, maintainable and scalable way. Just like

Angular.js, the *Model-View-Controller* framework for the *front end* that we have used. Angular is much more and its learning curve has many ups and downs, but I think that what we've seen in this example is useful to start developing more complex applications in a scalable way.

I invite you to continue investigating and using Angular in your projects. Every day you will discover a new way to use it. That is what makes it so powerful.

I hope that this book has been useful to you and that you have learned something new. You can contact me on social networks and let me know your thoughts, any *feedback* is welcomed and appreciated:

Share your opinion on Twitter at #ebookAngular hashtag :)

- twitter.com/carlosazaustre
- github.com/carlosazaustre
- google.com/+CarlosAzaustre
- facebook.com/carlosazaustre.web
- linkedin.com/in/carlosazaustre

Until the next book!

Table of Contents