
使用React、Node.js、
MongoDB、Socket.IO
开发一个角色投票应用

前言

> 中文出处：http://idlelife.org/archives/977
> 作者： Sahat Yalkabov
> 译者：pockry
> 原文： sahatyalkabov.com

导读：在这个教程里，我们将为EVE Online游戏创建一个角色投票应用（受Facemash的启发），EVE是一个大型多人在线游戏。在本教程里你将学习到如何使用Node.js构建一个REST API、使用MongoDB保存和检索数据、使用Socket.IO跟踪在线的访问者，以及使用React + Flux和服务端渲染来构建单页面应用，最后将应用部署到云端。

概述

在这个教程里我们将为EVE Online游戏创建一个角色投票应用（受Facemash的启发），EVE是一个大型多人在线游戏。请点击播放下面美妙的音乐，来进入本教程的学习氛围当中。

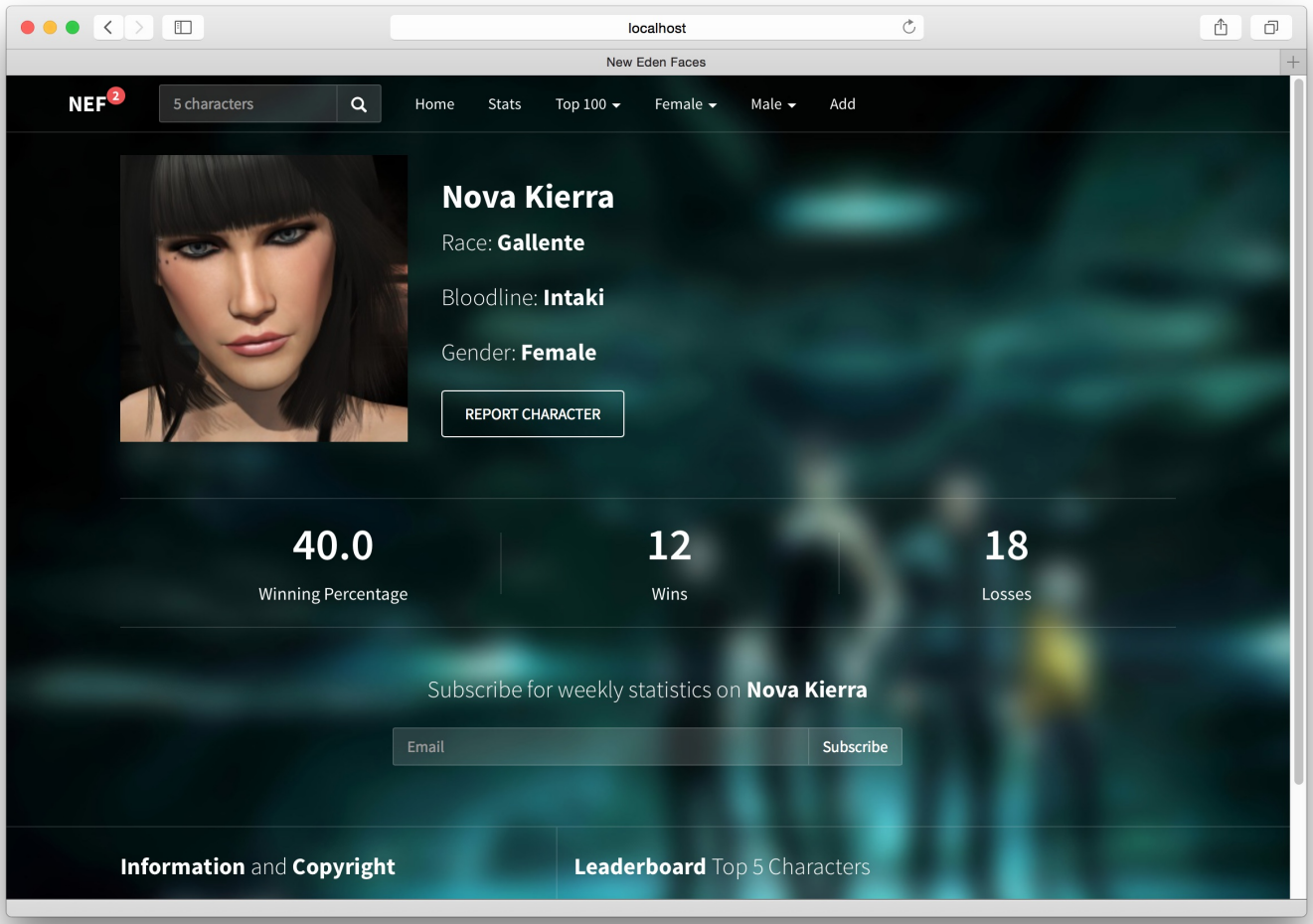
Your browser does not support the audio tag.

(EVE Online – Kronos (2014) Release Theme)

当你听着这首音乐时，想象你正身处宇宙深空的某个小行星带，一边从小行星中挖取稀有矿物，一边警戒着雷达上随时可能出现的宇宙海盗，同时，你还正在研究太空站推进系统的设计图，为你的宇宙舰队制造零件，并在完全由玩家需求和供给控制的交易系统下达买卖指令，另外还从遥远星系赶来的大型宇宙货船上卸货。在你的战舰旁边，是装有微型曲速装置的超高速截击机和武装到牙齿的战斗飞船，它们都是你的忠实护卫。在这场自由的游戏中，你可以计算如何从行星最优化的萃取矿物，也可以投身一场有数千玩家参与的大型宇宙会战。——这就是EVE Online。

在EVE Online中每个玩家都有一个3D形象以代表他们的角色，我们要开发的这个应用就是来为这些角色形象投票，以选出它们中最好看的。不管怎样，你的目标是学习Node.js和React，而不是EVE Online。但我想说的是：“在一个教程里，趣味即使不是最重要的，也至少和教程的主要目的同样重要。”我之前创建New Eden Faces的唯一目的是学习Backbone.js，我创建TV Show Tracker的目的是为了学习AngularJS。这些日子以来，貌似大家都开始用一个简单的TODO应用来当做教程项目，但对我来说，上面这些项目的任何一个，都比一个简单的TODO应用要有趣得多。

编写这些教程，我所学到的就是，无论是录屏、书籍还是视频教程，在学习一项新技术时，没有什么比创建一个让你有激情的小项目更高效的了。



项目完整源代码： https://github.com/sahat/newedenfaces-react

基于和我之前教程同样的精神（TV Show Tracker和Instagram Clone），本教程将是一篇全栈的JavaScript教程，我们将从零开始构建一个完整的应用。

> 注意：这个项目是对我之前New Eden Faces项目的重制，那是我使用Backbone.js编写的第一个单页面应用。它已经在OpenShift上基于Node.js 0.8.x稳定的运行两年多了。

对于给定主题的教程，我一般尽量少的做假设（如假设读者拥有Node.js基础之类），这也是为什么我的教程都是如此之长。不过即便如此，你至少也需要有一些客户端JavaScript框架和Node.js的经验，才能从这个教程中得到最大的收获。

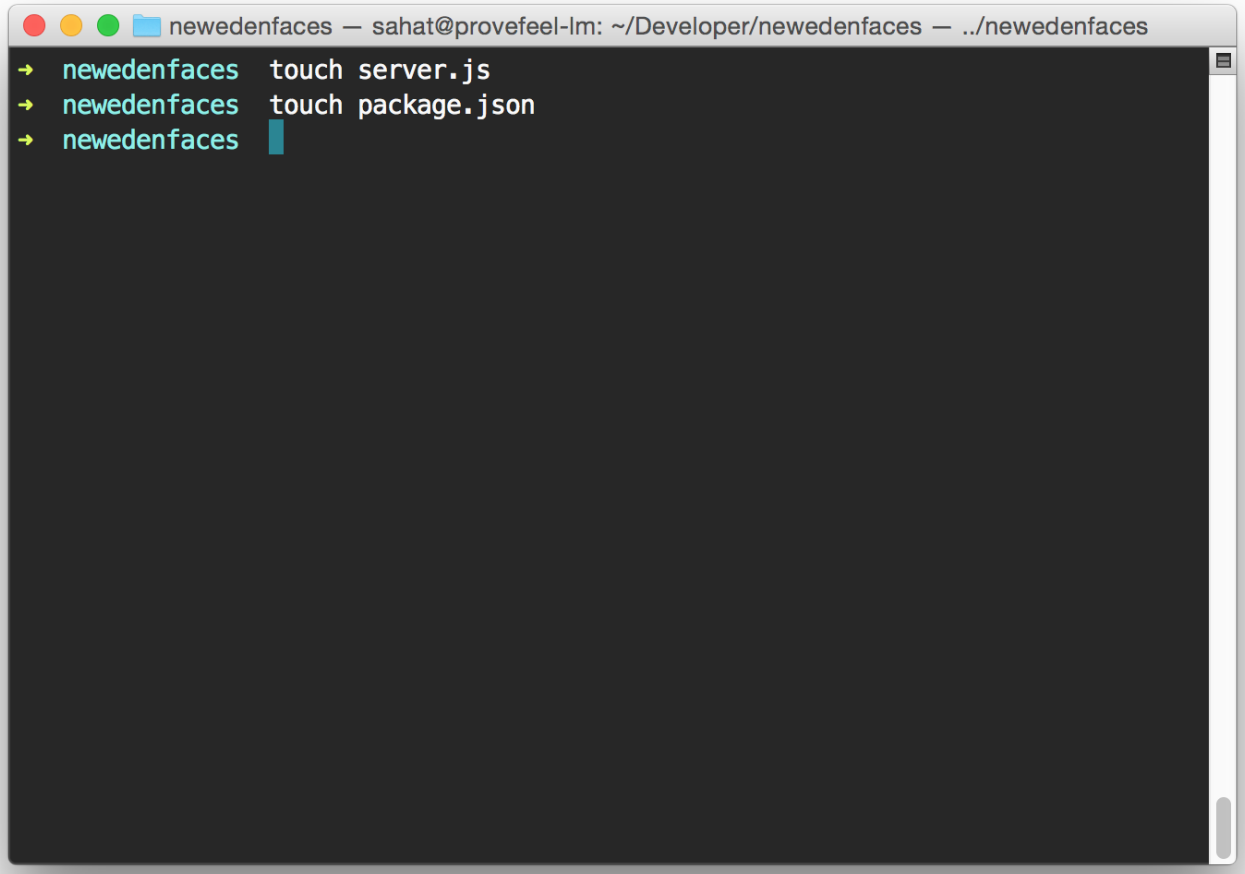
在开始之前，你需要下载并安装下列工具：

1. Node.js，或io.js
2. Bower
3. MongoDB
4. gulp
5. nodemon

第一步：新建Express项目

第一步：新建Express项目

创建一个新目录newedenfaces，进入目录并创建两个空文件 *package.json*和*server.js*：



注：我的OS X终端使用了Monokai主题和oh-my-fish框架以提供Fish shell.

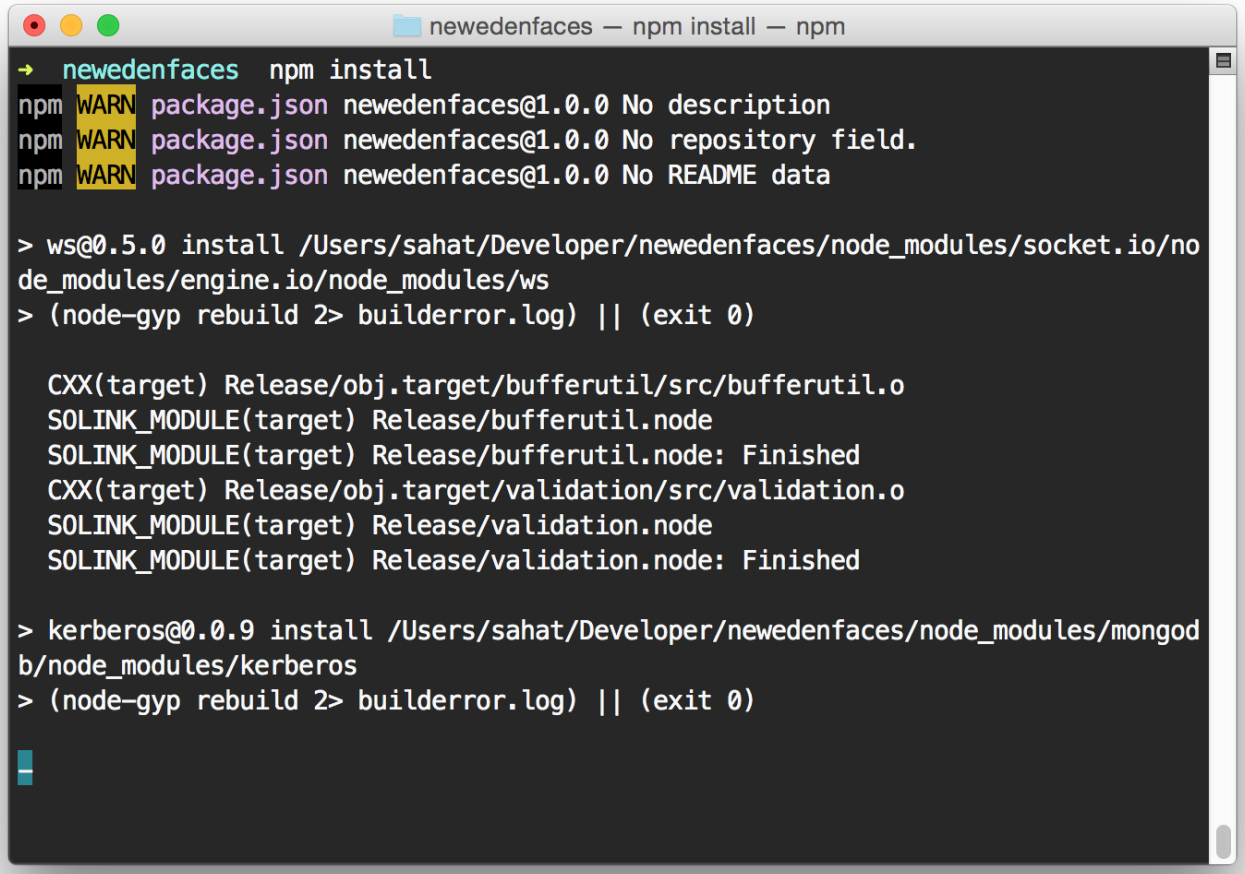
打开*package.json*并粘贴下面的代码：

```
{
  "name": "newedenfaces",
  "description": "Character voting app for EVE Online",
  "version": "1.0.0",
  "repository": {
    "type": "git",
    "url": "https://github.com/sahat/newedenfaces-react"
  },
  "main": "server.js",
  "scripts": {
    "start": "babel-node server.js",
    "watch": "nodemon --exec babel-node -- server.js"
  },
  "dependencies": {
    "alt": "^0.17.1",
    "async": "^1.4.0",
    "babel": "^5.6.23",
    "body-parser": "^1.13.2",
    "colors": "^1.1.2",
    "compression": "^1.5.1",
    "express": "^4.13.1",
    "mongoose": "^4.0.7",
    "morgan": "^1.6.1",
    "react": "^0.13.3",
    "react-router": "^0.13.3",
    "request": "^2.58.0",
    "serve-favicon": "^2.3.0",
    "socket.io": "^1.3.6",
    "swig": "^1.4.2",
    "underscore": "^1.8.3",
    "xml2js": "^0.4.9"
  },
  "devDependencies": {
    "babelify": "^6.1.3",
    "bower": "^1.4.1",
    "browserify": "^11.0.0",
    "gulp": "^3.9.0",
    "gulp-autoprefixer": "^2.3.1",
    "gulp-concat": "^2.6.0",
    "gulp-cssmin": "^0.1.7",
    "gulp-if": "^1.2.5",
    "gulp-less": "^3.0.3",
    "gulp-plumber": "^1.0.1",
    "gulp-streamify": "0.0.5",
    "gulp-uglify": "^1.2.0",
    "gulp-util": "^3.0.6",
    "vinyl-source-stream": "^1.1.0",
    "watchify": "^3.3.0"
  },
  "license": "MIT"
}
```

这是我们的应用所需要的所有依赖包，简要介绍如下：

包名称	描述
alt	React的Flux实现
async	异步流程控制
babel	ES6转换为ES5
body-parser	渲染POST请求数据
colors	美化控制台输出结果
compression	Gzip压缩
express	Node.js Web框架
mongoose	MongoDB ODM
morgan	HTTP请求日志
react	React框架
react-router	React路由库
request	HTTP请求库
serve-favicon	提供*favicon.png*
socket.io	显示有多少用户在线
swig	渲染HTML
underscore	JS辅助库
xml2js	将XML渲染为JSON

在终端输入 `npm install` 安装依赖。



> 如果你使用Windows，可以使用cmder控制台模拟器，可以达到接近OS X/Linux终端的效果。

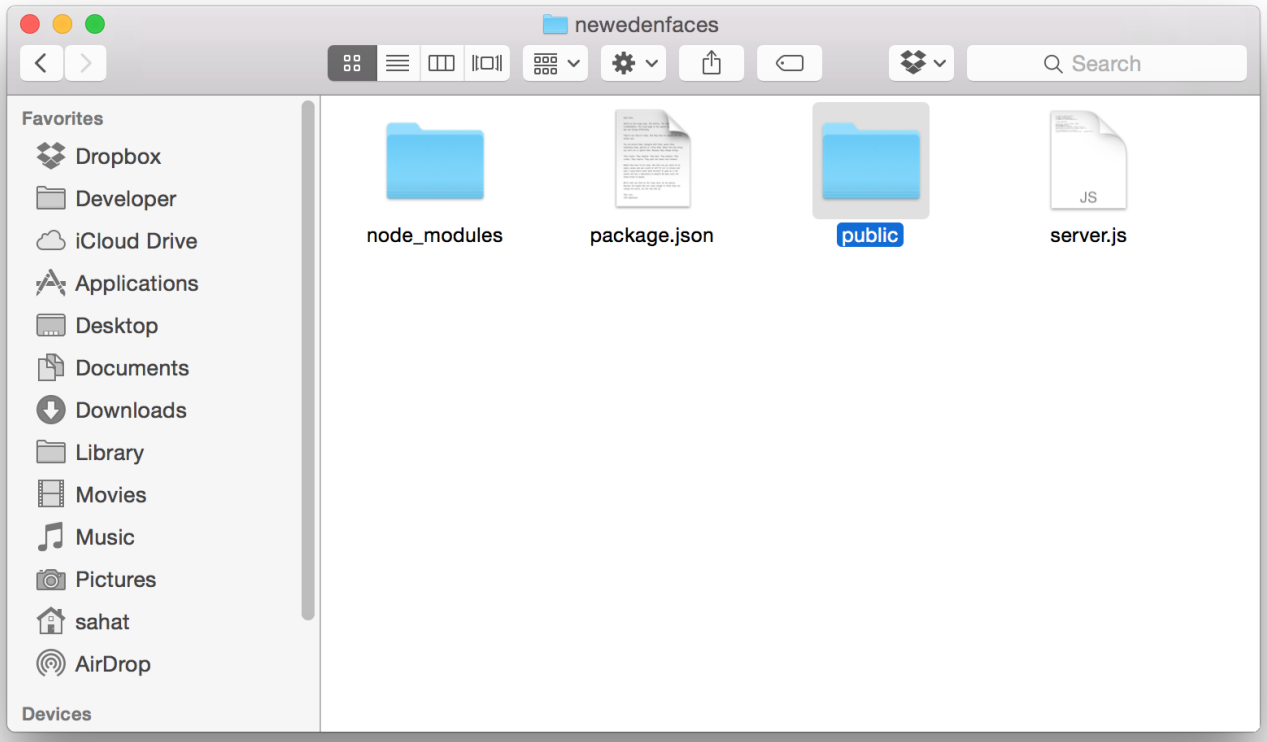
打开*server.js*粘贴下面代码，下面是最基础的Express应用，足够我们开始了。

```
var express = require('express');
var path = require('path');
var logger = require('morgan');
var bodyParser = require('body-parser');

var app = express();

app.set('port', process.env.PORT || 3000);
app.use(logger('dev'));
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(express.static(path.join(__dirname, 'public')));

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

> 注意：尽管我们将使用ES6编写React app，但在这儿我还是决定使用ES5，因为这段代码从两年前到现

> 在基本没有变过。并且，如果你第一次使用ES6，至少这个Express app对你来说还算熟悉。

然后，新建文件夹public，这里是我们放文件如图片、字体，以及压缩后的CSS和JS文件。



现在你可以在终端输入 `npm start` 启动应用，确保Express app没有问题。

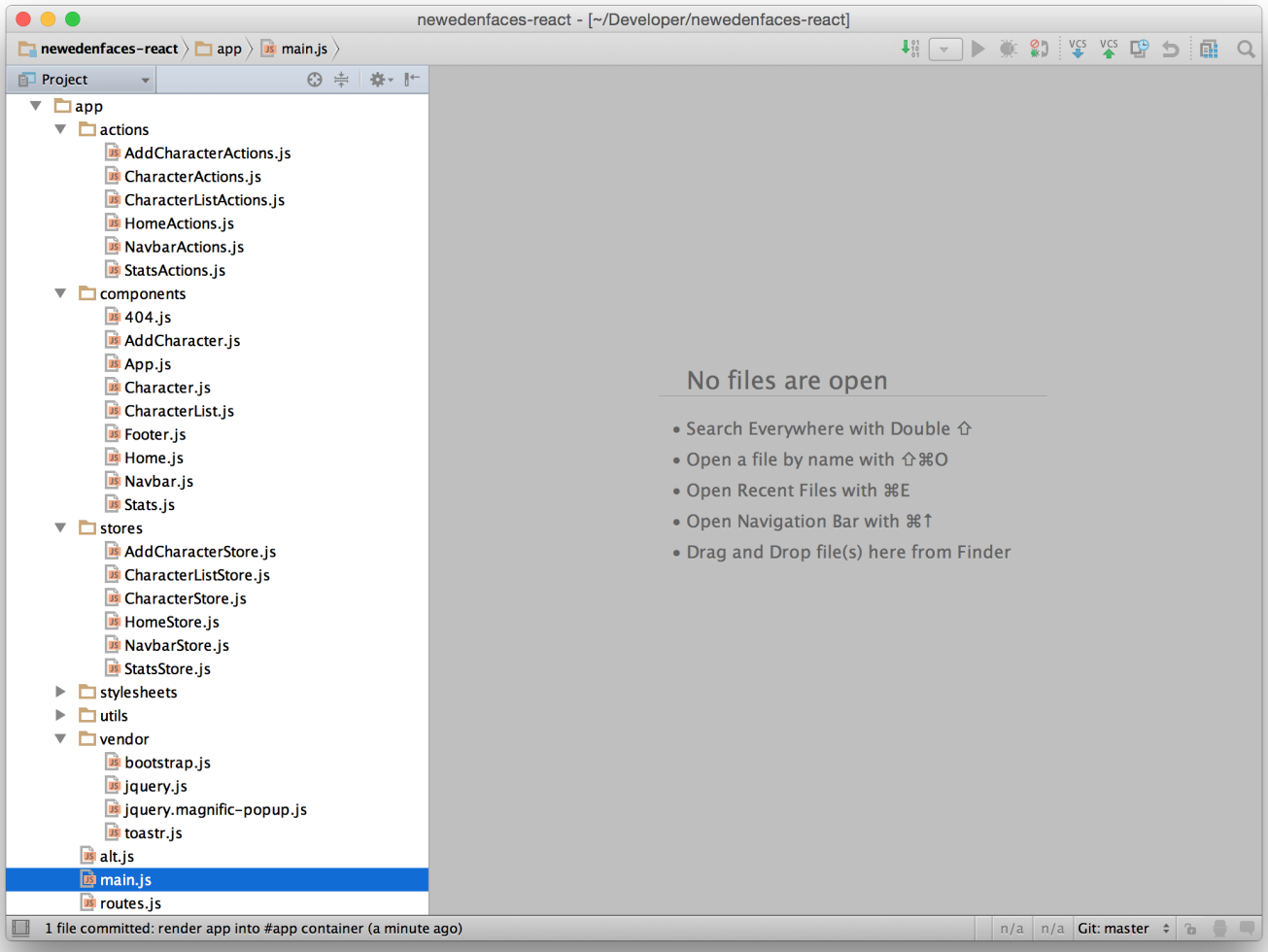
> 注意：你现在可以使用 `node server.js` 来直接启动应用，但之后我们将使用Babel来预编译文件，也就是说你需要运行 `babel-node server.js` 来启动应用。

现在你应该可以看到终端输出“Express server listening on port 3000.”

第二步：构建系统

第二步 构建系统

如果你经常参与web社区，那么你应该听说过Browserify和Webpack工具。如果不使用它们，你将面临手动的在HTML输入很多 `<script>` 标签，并且需要将JS代码放到合适的地方。



并且，我们还不能直接在浏览器使用ES6，在将代码提供给用户之前，我们需要用Babel将它们转换为ES5代码。

我们将使用Gulp和Browserify而不用Webpack。我并不认为它们两个谁优谁劣，但我想说Gulp+Browserify比等价的Webpack文件要直观多了，我还没有在任何React boilerplate项目中看到一个易于理解的*webpack.config.js*文件。

创建文件*gulpfile.js*并粘贴下面的代码：

```
var gulp = require('gulp');
var gutil = require('gulp-util');
var gulpif = require('gulp-if');
```

```
var streamify = require('gulp-streamify');
var autoprefixer = require('gulp-autoprefixer');
var cssmin = require('gulp-cssmin');
var less = require('gulp-less');
var concat = require('gulp-concat');
var plumber = require('gulp-plumber');
var source = require('vinyl-source-stream');
var babelify = require('babelify');
var browserify = require('browserify');
var watchify = require('watchify');
var uglify = require('gulp-uglify');

var production = process.env.NODE_ENV === 'production';

var dependencies = [
  'alt',
  'react',
  'react-router',
  'underscore'
];

/*
 |--------------------------------------------------------------------------
 | Combine all JS libraries into a single file for fewer HTTP requests.
 |--------------------------------------------------------------------------
 */
gulp.task('vendor', function() {
  return gulp.src([
    'bower_components/jquery/dist/jquery.js',
    'bower_components/bootstrap/dist/js/bootstrap.js',
    'bower_components/magnific-popup/dist/jquery.magnific-popup.js',
    'bower_components/toastr/toastr.js'
  ]).pipe(concat('vendor.js'))
    .pipe(gulpif(production, uglify({ mangle: false })))
    .pipe(gulp.dest('public/js'));
});

/*
 |--------------------------------------------------------------------------
 | Compile third-party dependencies separately for faster performance.
 |--------------------------------------------------------------------------
 */
gulp.task('browserify-vendor', function() {
  return browserify()
    .require(dependencies)
    .bundle()
    .pipe(source('vendor.bundle.js'))
    .pipe(gulpif(production, streamify(uglify({ mangle: false }))))
    .pipe(gulp.dest('public/js'));
});

/*
 |--------------------------------------------------------------------------
 | Compile only project files, excluding all third-party dependencies.
 |--------------------------------------------------------------------------
```

```
 |
 */
gulp.task('browserify', ['browserify-vendor'], function() {
  return browserify('app/main.js')
    .external(dependencies)
    .transform(babelify)
    .bundle()
    .pipe(source('bundle.js'))
    .pipe(gulpif(production, streamify(uglify({ mangle: false }))))
    .pipe(gulp.dest('public/js'));
});

/*
 |--------------------------------------------------------------------------
 | Same as browserify task, but will also watch for changes and re-compile.
 |--------------------------------------------------------------------------
 */
gulp.task('browserify-watch', ['browserify-vendor'], function() {
  var bundler = watchify(browserify('app/main.js', watchify.args));
  bundler.external(dependencies);
  bundler.transform(babelify);
  bundler.on('update', rebundle);
  return rebundle();

  function rebundle() {
    var start = Date.now();
    return bundler.bundle()
      .on('error', function(err) {
        gutil.log(gutil.colors.red(err.toString()));
      })
      .on('end', function() {
        gutil.log(gutil.colors.green('Finished rebundling in', (Date.now() - start) + 'ms.'));
      })
      .pipe(source('bundle.js'))
      .pipe(gulp.dest('public/js/'));
  }
});

/*
 |--------------------------------------------------------------------------
 | Compile LESS stylesheets.
 |--------------------------------------------------------------------------
 */
gulp.task('styles', function() {
  return gulp.src('app/stylesheets/main.less')
    .pipe(plumber())
    .pipe(less())
    .pipe(autoprefixer())
    .pipe(gulpif(production, cssmin()))
    .pipe(gulp.dest('public/css'));
});

gulp.task('watch', function() {
  gulp.watch('app/stylesheets/**/*.less', ['styles']);
});
```

```
gulp.task('default', ['styles', 'vendor', 'browserify-watch', 'watch']);
gulp.task('build', ['styles', 'vendor', 'browserify']);
```

> 如果你从未使用过Gulp，这里有一个很棒的教程《An Introduction to Gulp.js》

下面简单介绍一下每个Gulp任务是干嘛的。

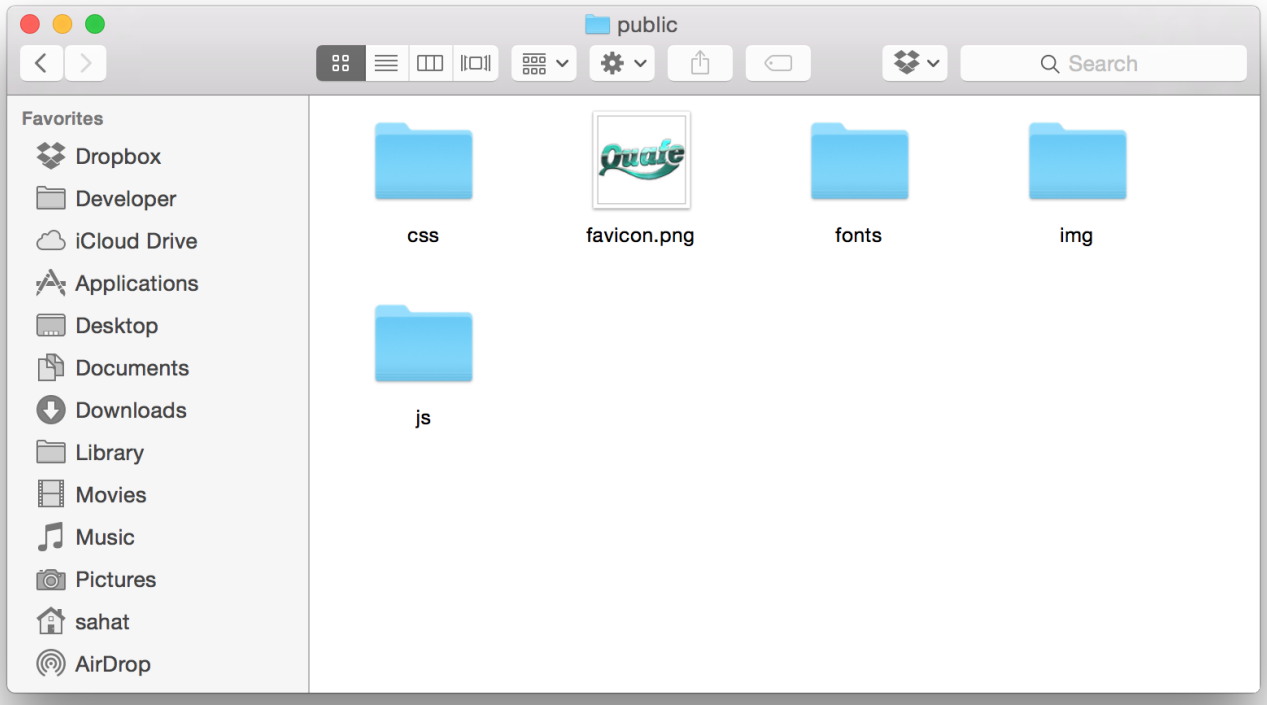
Gulp Task	Description
vendor	将所有第三方JS文件合并到一个文件
browserify-vendor	因为性能原因，我们将NPM模块和前端模块分开编译和打包，因此每次重新编译将会快个几百毫秒
browserify	仅将app文件编译并打包，不包括其它模块和库
browserify-watch	包括上面的功能，并且监听文件改变，然后重新编译打包app文件
watch	当文件改变时重新编译LESS文件
default	运行上面所有任务并开始监听文件改变
build	运行上面所有任务然后退出

下面，我们将注意力转移到项目结构上，我们将创建*gulpfile.js*需要的文件和文件夹。

第三步：项目结构

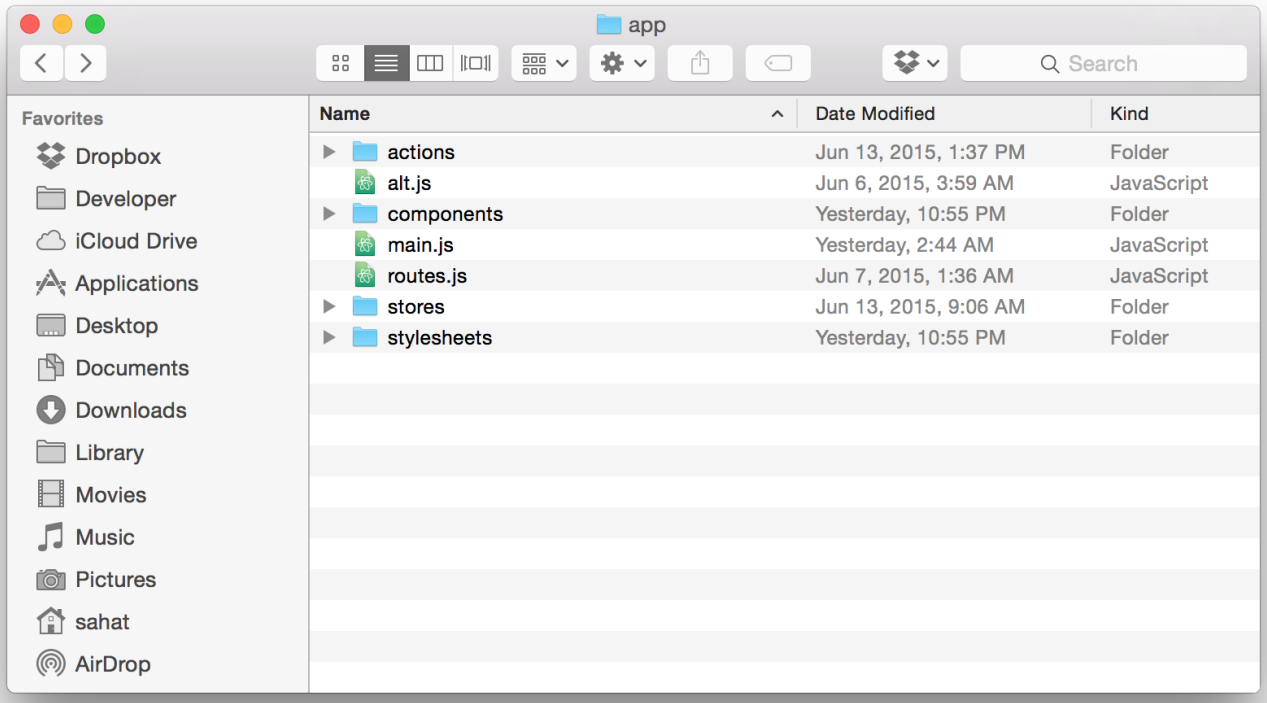
第三步：项目结构

在public目录下创建四个目录： css，js，fonts，img。然后，下载这个favicon.png，也把它放到这里。



在项目根目录，创建新目录app。

然后在app文件夹里新建四个目录：actions，components，stores，stylesheets，以及三个空文件*alt.js*，*routes.js*和*main.js*。



在stylesheets目录下新建文件*main.less*，我们将在里面填入样式。

回到根目录，创建新文件*bower.json*并粘贴下面的代码：

```
{
  "name": "newedenfaces",
  "dependencies": {
    "jquery": "^2.1.4",
    "bootstrap": "^3.3.5",
    "magnific-popup": "^1.0.0",
    "toastr": "^2.1.1"
  }
}
```

> 注意：Bower是一个让你轻松下载JavaScript库的前端包管理器，通过命令行即可下载上面的库。

在终端运行 `bower install` 然后等待包下载并安装到bower_components目录。你能在.bowerrc文件改变该路径，不过本教程我们使用默认设置。

和node_modules相似，你可能不想将bower_components提交到Git仓库，但如果你不提交的话，当你部署应用的时候如何下载这些文件？我们将在教程的部署部分来解决这个问题。

复制bower_components/bootstrap/fonts下所有的字体图标（glyphicons fonts）到public/fonts目录。

下载并解压下面的背景图片到public/img目录。

- Background Images.zip

> 有趣事实：三年前我使用Adobe Photoshop来创建高斯模糊效果，但它们今天能轻松的使用CSS滤镜实现。

打开*main.less*并粘贴下面的文件中的代码。鉴于代码的长度，我决定不将它放在文中。

- main.less

如果你以前用过Bootstrap CSS框架，那么你应该对里面的大部分代码都感到熟悉。

> 注意：我花了很长时间在这个UI上，调整fonts和颜色，添加精细的变换效果，如果你有时间的话，推荐在完成本教程之后继续探索一下样式的细节。

我不知道你是否知道最近的趋势，是将样式直接放入React组件当中，但我不太确定我是否喜欢这项实践，也许当相关的工具完善之后我会喜欢吧，但在那之前我还是会使用附加的样式表文件。不过，如果你对使用模块化的CSS感兴趣，可以看看这个css-modulesify项目。

在我们开始构建React app之前，我决定先花三个章节的时间讲讲ES6、React、Flux基础，否则要想一下子全部学会它们会让人很崩溃。对我个人来说，我曾花了不少时间理解某些用ES6编写的React + Flux示例代码上，因为我相当于同时学习一个新语法、一个新框架，以及一个全新的应用架构。

由于三者的内容众多，显然我不能在一篇文章中全讲清楚，我将会只讲那些本教程中会用到的主题。

第四步： ES6速成教程

第四步： ES6速成教程

最好的学习ES6的方法，是为每一个ES6示例提供一个等价的ES5实现。外面已经有不少介绍ES6的文章，本文将只讲其中一些。

Modules(Import)

```
// ES6
import React from 'react';
import {Route, DefaultRoute, NotFoundRoute} from 'react-router';

// ES5
var React = require('react');
var Router = require('react-router');

var Route = Router.Route;
var DefaultRoute = Router.DefaultRoute;
var NotFoundRoute = Router.NotFoundRoute;
```

使用ES6中的解构赋值（destructuring assignment），我们能导入模块的子集，这对于像*react-router*和*underscore*这样不止输出一个函数的模块尤其有用。

需要注意的是ES6 import的优先级很高，所有的依赖模块都会在模块代码执行之前加载，也就是说，你无法像在CommonJS一样有条件的加载模块。之前我尝试在一个if-else条件里import模块，结果失败了。

想了解 `import` 的更多细节，可访问它的MDN页面。

Modules(Export)

```
// ES6
function Add(x) {
  return x + x;
}

export default Add;

// ES5
function Add(x) {
  return x + x;
}

module.exports = Add;
```

想学习ES6模块的更多细节，这里有两篇文章ECMAScript 6 modules和Understanding ES6 Modules。

Classes

ES6 class只不过是现有的基于原型继承机制的一层语法糖，了解这个事实之后，`class`关键字对你来说就不再像一个其它语言的概念了。

```
// ES6
class Box {
  constructor(length, width) {
    this.length = length;
    this.width = width;
  }
  calculateArea() {
    return this.length * this.width;
  }
}

let box = new Box(2, 2);

box.calculateArea(); // 4

// ES5
function Box(length, width) {
  this.length = length;
  this.width = width;
}

Box.prototype.calculateArea = function() {
  return this.length * this.width;
}

var box = new Box(2, 2);

box.calculateArea(); // 4
```

另外，ES6中还可以用`extends`关键字来创建子类。

```
class MyComponent extends React.Component {
  // Now MyComponent class contains all React component methods
  // such as componentDidMount(), render() and etc.
}
```

了解ES6 class更多信息可查看Classes in ECMAScript 6这篇博文。

JS `var`与`let`

这两个关键字唯一的区别是，`var`的作用域在最近的函数块中，而`let`的作用域在最近的块语句中——它可以是一个函数、一个for循环，或者一个if语句块。

这里有个很好的示例，来自MDN：

```
var a = 5;
var b = 10;

if (a === 5) {
  let a = 4; // The scope is inside the if-block
  var b = 1; // The scope is inside the function

  console.log(a);  // 4
  console.log(b);  // 1
}

console.log(a); // 5
console.log(b); // 1
```

一般来说，`let`是块作用域，`var`是函数作用域。

箭头函数(=> fat arrow)

一个箭头函数表达式与函数表达式相比有更简短的语法，以及从语法上绑定了`this`值。

```
// ES6
[1, 2, 3].map(n => n * 2); // [2, 4, 6]

// ES5
[1, 2, 3].map(function(n) { return n * 2; }); // [2, 4, 6]
```

> 注意：如果参数只有一个，圆括号是可选的，到底是否强制使用取决于你，不过有些人认为去掉括号是坏的实践，有些人则无所谓。

除了更短的语法，箭头函数还有什么用途呢？

考虑下面这个示例，它来自于我将这个项目转换为使用ES6之前的代码：

```
$.ajax({ type: 'POST', url: '/api/characters', data: { name: name, gender: gender } })
  .done(function(data) {
    this.setState({ helpBlock: data.message });
  }.bind(this))
  .fail(function(jqXhr) {
    this.setState({ helpBlock: jqXhr.responseJSON.message });
  }.bind(this))
  .always(function() {
    this.setState({ name: '', gender: '' });
  }.bind(this));
```

上面的每个函数都创建了自己的`this`作用域，不绑定外层`this`的话我们是无法在示例中调用`this.setState`的，因为函数作用域的`this`一般是*undefined*。

当然，它有绕过的方法，比如将 this 赋值给一个变量，比如 var self = this ，然后在闭包里用 self.setState 代替 this.setState 即可。

而使用等价的ES6代码的话，我们没有必要如此麻烦：

```
$.ajax({ type: 'POST', url: '/api/characters', data: { name: name, gender: gender } })
  .done((data) => {
    this.setState({ helpBlock: data.message });
  })
  .fail((jqXhr) => {
    this.setState({ helpBlock: jqXhr.responseJSON.message });
  })
  .always(() => {
    this.setState({ name: '', gender: '' });
  });
```

ES6的讲解就到此为止了，下面让我们看看React，到底是什么让它如此特殊。

第五步： React速成教程

第五步： React速成教程

React是一个用于构建UI的JS库。你可以说它的竞争对手有AngularJS，Ember.js，Backbone和Polymer，尽管React专注的领域要小得多。React仅仅是MVC架构中的V，即视图层。

那么，React有什么特殊的呢？

React的组件使用特定的声明式样式书写，不像jQuery或其它传统JS库，你不与DOM直接交互。当背后的数据改变时，React接管所有的UI更新。

React还非常快，这归功于Virtual DOM和幕后的diff算法。当数据改变时，React计算所需要操作的最少的DOM，然后高效的重新渲染组件。比如，如果页面上有10000个已经渲染的元素，但只有一个元素改变，React将仅仅更新其中一个DOM，这是React为何能高效的重新渲染整个组件的原因。

React其它令人瞩目的特性包括：

- 可组合性。小的组件可以组合成大的、复杂的组件。
- 相对易于学习。需要学习的并不多，并且它不像AngularJS或Ember.js那样有庞大的文档。
- 服务端渲染。让我们能轻松的构建同型JS应用(Isomorphic JavaScript apps)。
- 最有帮助的错误和警告提示，是我从未在其它JS库中见到过的。
- 组件是自包含的。标记、行为（甚至样式）都在同一个地方，让组件非常易于重用。

我非常喜欢React v0.14 Beta 1发布中的这段话，讲了React到底是什么：

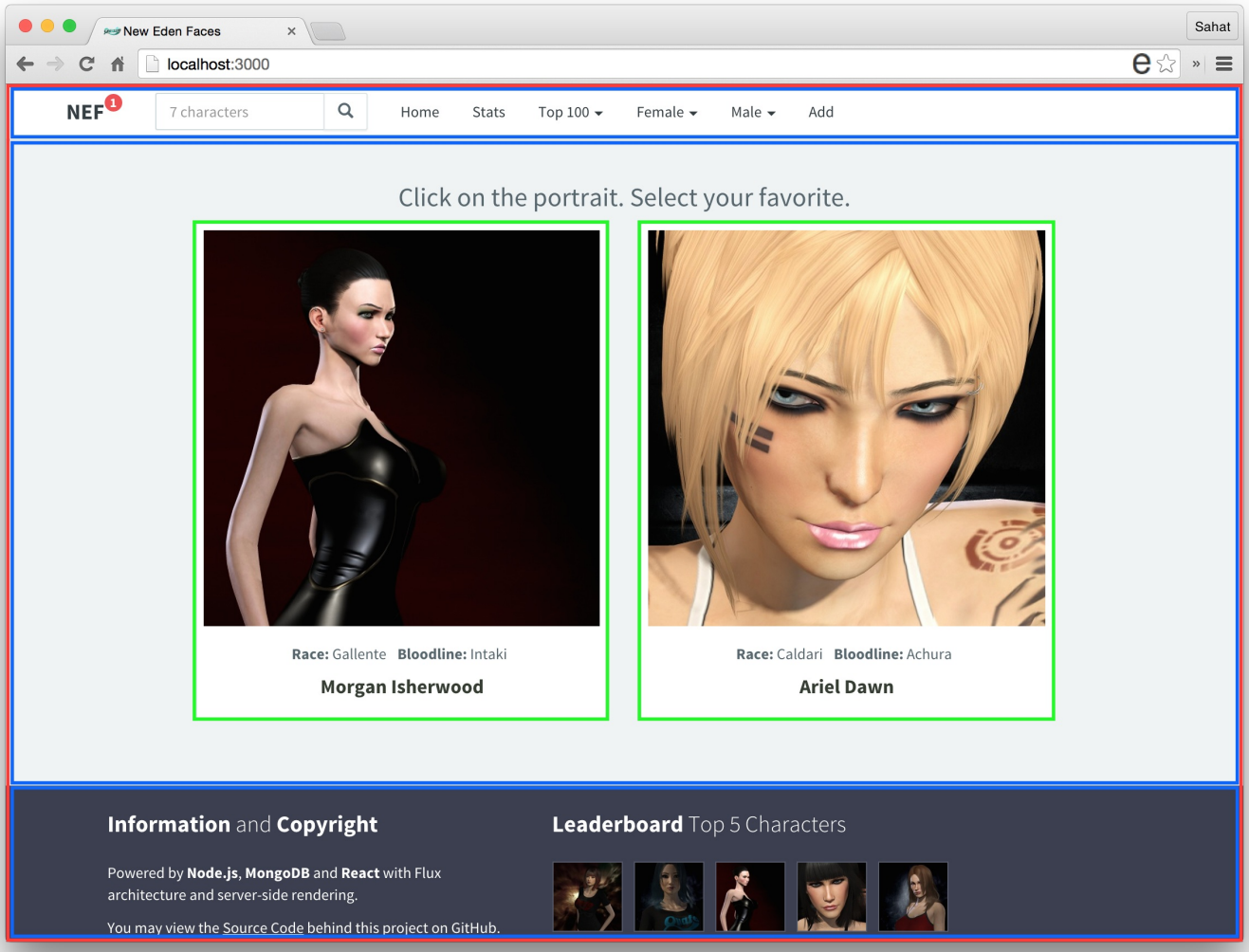
> 现在我们已经清楚，React的美妙和本质与浏览器或DOM无关，我们认为React的真正基础是关于组件和元素的质朴想法：用声明式的方式来描述任何你想渲染的东西。

在进入下一步之前，推荐你先观看这个了不起的视频React in 7 Minutes ，它的作者是John Lindquist，推荐你订阅PRO以获得更多的视频教程。

另外，也可以考虑Udemy上的这个广受好评的教程——Build Web Apps with React JS and Flux ，作者是Stephen Grider。它包含超过71个视频和10小时以上的内容，涵盖了React，Flux，React Router，Firebase，Imgur API和其它。

当学习React时，我最大的挑战是使用完全不同的思考方式去构建UI。这也是为什么你必须阅读Thinking in React（中文版）这个官方指南的原因。

和Thinking in React中的产品列表风格类似，如果我们将 *New Eden Faces* UI分开为潜在的组件，它将会是这样：



> 注意：每个组件都应该坚持单一职责原则，如果你发现你的组件做的事情太多，也许最好将它分成子组件。不过话虽如此，我首先还是编写了一个典型的的单块组件，当它能够工作后，然后将它重构为小的子组件。

在我们的项目中，顶级App组件包含Navbar，Homepage和Footer组件，Homepage组件包含两个Character组件。

所以，无论何时你想到一个UI设计，从将它分解为不同的组件开始，并且永远考虑你的数据如何在父-子、子-父以及同胞组件间传递，否则你会遇到这样的时刻：“WTF，这个到底在React里怎么实现的？这个用jQuery实现起来简单多了……”

所以，下次你决定用React构建一个新app时，在写代码之前，先画一个这样的层次大纲图。它帮你视觉化多个组件间的关系，并可以照着它来构建组件。

React中所有的组件都有 `render()` 方法，它总是返回一个单一的子元素。因此，下面的返回语句是错误的，因为它返回了3个子元素。

```
render() {
  // Invalid JSX,
  return (
    <li>Achura</li>
    <li>Civire</li>
    <li>Deteis</li>
  );
}
```

上面的HTML标记叫做JSX。它的语法和HTML仅有些微的不同，比如用 className 代替 class ，在我们开始开发应用的时候你将会学到它的更多内容。

当我第一眼看到这样的语法，我的第一反应就是拒绝，在JavaScript中我习惯返回布尔值、数字、字符串、对象以及函数，但绝不是这种东西。但是，JSX不过是一个语法糖。使用一个 <ul> 标签包裹上面的返回内容后，下面是不使用JSX时的模样：

```
render() {
  return React.createElement('ul', null,
    React.createElement('li', null, 'Achura'),
    React.createElement('li', null, 'Civire'),
    React.createElement('li', null, 'Deteis')
  );
}
```

我相信你会同意JSX远比普通的JavaScript的可读性更好，另外，Babel对JSX有内建支持，所以我们无需做任何额外的事情即可解析它。如果你用过AngularJS中的指令（directive）那么你将会欣赏React的做法，这样你就不必同时处理两个文件——*directive.js*（负责逻辑）和*template.html*（负责展现），你可以在同一个文件里同时处理逻辑和展现了。

React中的 componentDidMount 方法和jQuery中的 $(document).ready 非常相似，这个方法仅在组件第一次渲染后运行一次（只在客户端运行），这里经常用于初始化第三方库和jQuery插件，或者连接到Socket.IO。

在 render 方法中，你将经常使用三元运算符：当数据为空时隐藏元素、根据条件注入CSS类名、根据组件的状态切换元素的展示等等。

比如下面的例子展示如果根据props值作为条件将CSS类名设为text-danger或text-success：

```
render() {
  let delta = this.props.delta ? (
    <strong className={this.props.delta > 0 ? 'text-success' : 'text-danger'}>
      {this.props.delta}
    </strong>
  ) : null;

  return (
    <div className='card'>
      {delta}
      {this.props.title}
    </div>
  );
}
```

这里我们仅仅浅尝辄止了React的内容，但这应该已经足以展示React的一般概念和它的优点了。

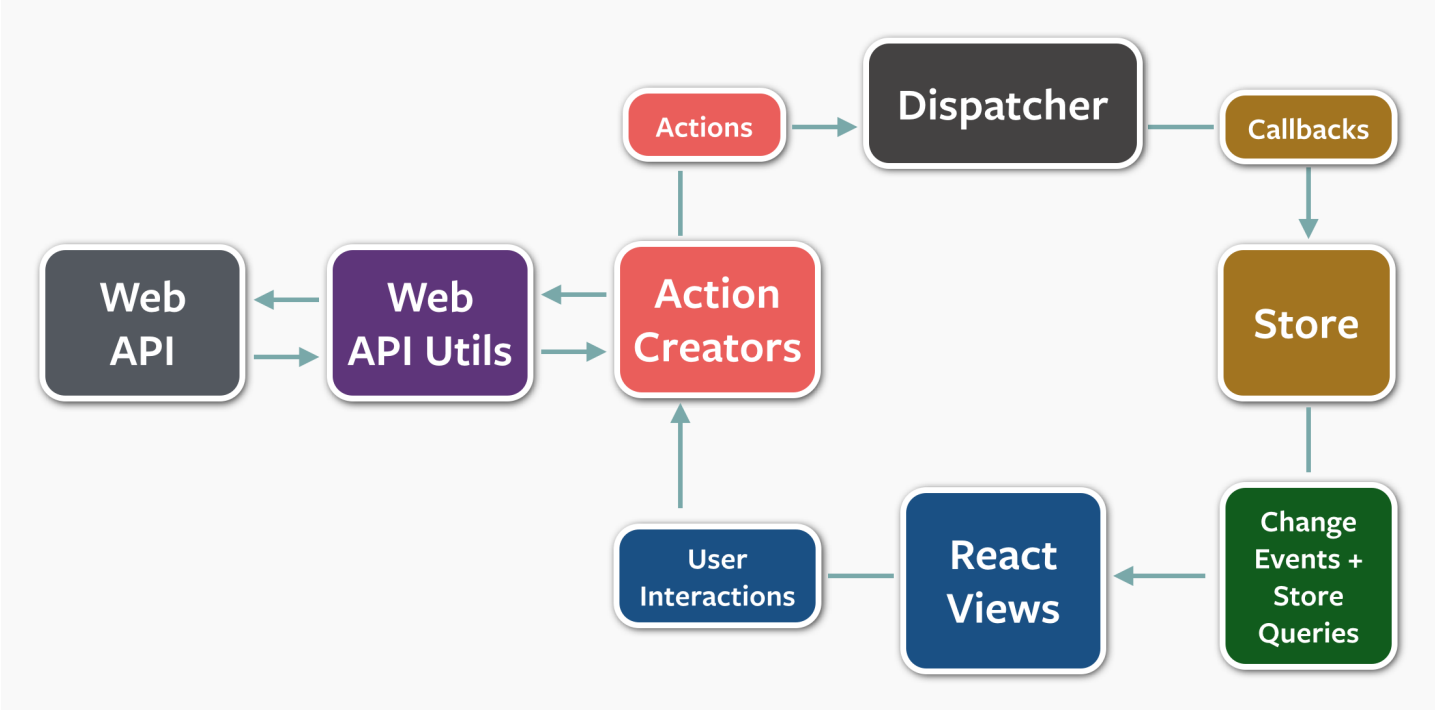
React本身是非常简单并且容易掌握的，但是，当我们谈起Flux架构时，可能会有些麻烦。

第六步：Flux架构速成教程

第六步：Flux架构速成教程

Flux是Facebook为可扩展的客户端web应用开发的应用架构。它利用单向数据流补全了React组件的一些不足。Flux更应该看做一种模式而非框架，不过，这里我们将使用一个叫做Alt的Flux实现，来减少我们写脚手架代码的时间。

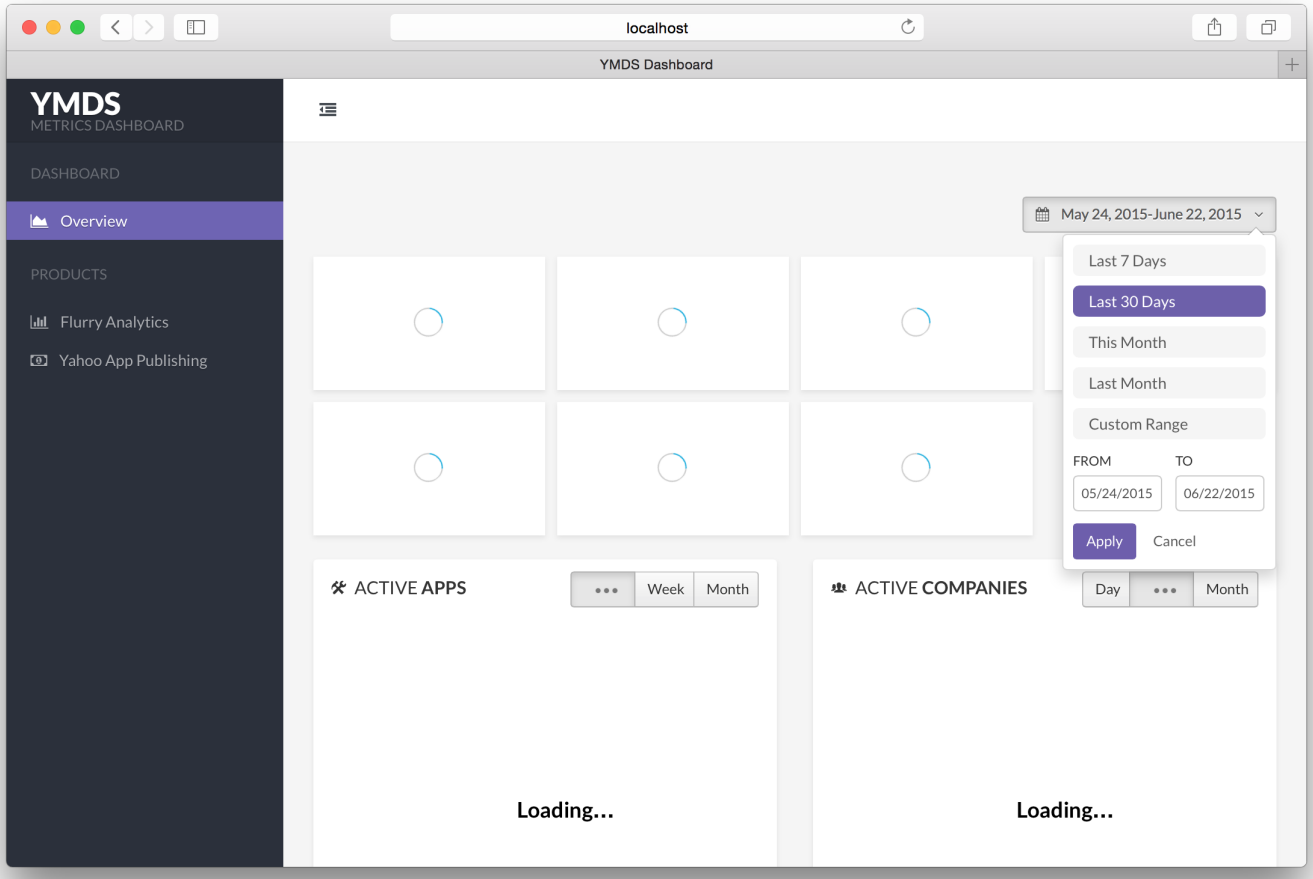
以前你看过这个图解吗？你能理解它吗？我就不能理解，无论我看它多少遍。



现在我对Flux比较了解了，我只能说，真是服了他们（Flux作者），能将简单的架构用如此复杂的方式展现出来。不过需要说明的是，它们的新Flux图解比以前好多了。

> 有趣事实：当我刚开始写这个教程时，我决定不在这个项目中使用Flux。我实在掌握不了这个东西，还是让别人去教它吧。不过谢天谢地，在Yahoo我能在上班时间把玩不同的技术并试验它们，所以花点功夫还是学会了。老实说，不用Flux我们也能构建这个app，并且写的代码还少些，因为这个项目并没有什么复杂的内嵌组件。但我相信，做一个全栈的React app，包括服务端渲染和Flux架构，看着不同的部分是如何组合到一起的，这本身有它的价值。

与其重复Flux那抽象的官方教程，不如让我们来看一个真实的用例，来展示Flux是如何工作的：



- 在 componentDidMount 中，三个action被触发：

```
OverviewActions.getSummary();
OverviewActions.getApps();
OverviewActions.getCompanies();
```

- 每一个action都创建了一个AJAX请求向服务器获取数据。
- 获取到数据后，每一个action触发另一个“success”的action，并且将数据传递给它：

```
getSummary() {
  request
    .get('/api/overview/summary')
    .end((err, res) => {
      this.actions.getSummarySuccess(res.body);
    });
}
```

- 同时，Overview的store（我们存储Overview组件状态的地方）监听所有“success”的action。当 getSummarySuccess 被触发后，Overview的store中的 onGetSummarySuccess 方法被调用，store被更新：

```
class OverviewStore {

  constructor() {
    this.bindActions(OverviewActions);
    this.summary = {};
    this.apps = [];
    this.companies = [];
  }

  onGetSummarySuccess(data) {
    this.summary = data;
  }

  onGetAppsSuccess(data) {
    this.apps = data;
  }

  onGetCompaniesSuccess(data) {
    this.companies = data;
  }
}
```

- 一旦store更新，Overview组件将会知道，因为它订阅了Overview store，当store更新/改变后，组件将会安装store中的值更新自身状态。

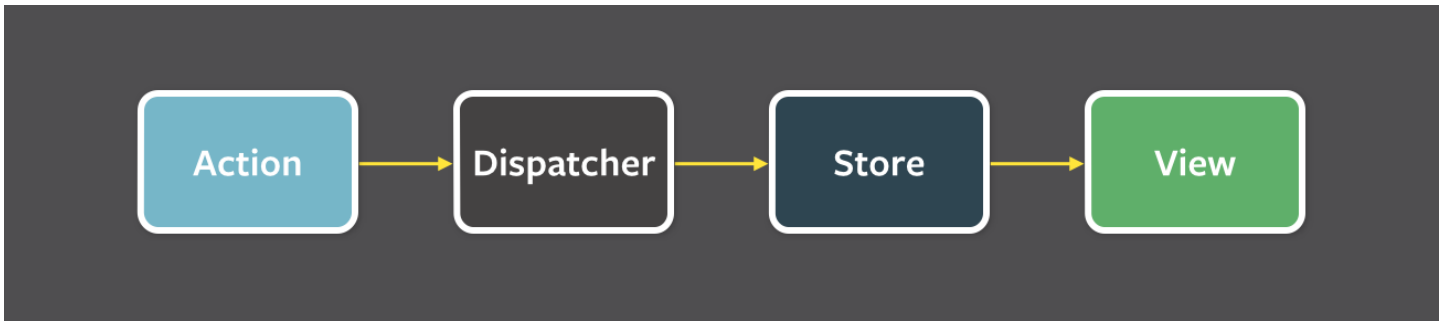
```
class Overview extends React.Component {

  constructor(props) {
    super(props);
    this.state = OverviewStore.getState();
    this.onChange = this.onChange.bind(this);
  }

  componentDidMount() {
    OverviewStore.listen(this.onChange);
  }

  onChange() {
    this.setState(OverviewStore.getState())
  }

  ...
}
```

- 此时Overview组件已经根据新数据更新完成了。
- 在上面的截图上，当从下拉菜单选择不同的日期范围，将会重复整个流程。

> 注意：Action如何命名并无规定，你可自由按照自己的习惯命名，只要它是描述性并且有意义的。

让我们暂时先忽略*Dispatcher*一会，从上面的描述你看到了一条单向的数据流吗？如果没有也没什么大不

了的，当我们开始构建应用的时候你自然就明白了。



Flux概要

Flux事实上不过是pub/sub架构的一个时髦说法，比如，应用的数据总是单向的，并被一路上的订阅者们接收。

在写这篇教程的时候，外面已经有超过一打的Flux实现，在这些实现当中，我只用过RefluxJS和Alt，在这两者之间，我个人比较喜欢Alt，因为它的简洁，以及作者goatslacker的热心、支持服务端渲染、非常棒的文档，以及积极的维护。

我强烈建议你去读一下Alt的Getting Started，不超过10分钟你就能基本入门了。

如果你对于该选择哪个Flux库感到迷茫，可以考虑一下Hacker News上一个叫glenjamin的家伙的评论，他花了大量时间试图弄清到底该用哪个Flux库：

> 令人郁闷的事实是：它们（Flux库）都很好。你几乎不可能因为一个Flux库而让你的应用出现问题。即使某个Flux库停止维护了也不打紧，你要知道，大多数正常的Flux实现都非常小（大约100行代码），出不了什么致命问题，即使出了我想你也能搞定。总之，redux很灵巧，但不要在试图获得完美的Flux库上浪费时间，瞅着哪个还算顺眼就拿来用，赶紧将关注点转回到你的应用上去。

现在，我们已经过了一遍ES6、React、Flux的一些基础，现在该将注意力集中到我们的应用上来了。

第七步：React路由（客户端）

第七步：React路由（客户端）

在app/components目录下新建文件*App.js*，粘贴下面的代码：

```
import React from 'react';
import {RouteHandler} from 'react-router';

class App extends React.Component {
  render() {
    return (
      <div>
        <RouteHandler />
      </div>
    );
  }
}

export default App;
```

`RouteHandler`是渲染当前子路由处理器的组件，它将根据URL渲染这些组件中的一个：Home、Top100、Profile，或Add Character。

> 注意：它和AngularJS中的`<div ng-view></div>`挺相似，会将当前路由中已渲染的模板包含进主布局中。

然后，打开app目录下的*routes.js*，粘贴下面的代码：

```
import React from 'react';
import {Route} from 'react-router';
import App from './components/App';
import Home from './components/Home';

export default (
  <Route handler={App}>
    <Route path='/' handler={Home} />
  </Route>
);
```

之所以将路由像这样嵌套，是因为我们将在`RouteHandler`的前后添加Navbar和Footer组件。不像其它组件，路由改变的时候，Navbar和Footer组件会保持不变。

最后，我们需要添加一个URL监听程序，当URL改变时渲染应用。打开App目录下的*main.js*并添加下列代码：

```
import React from 'react';
import Router from 'react-router';
import routes from './routes';

Router.run(routes, Router.HistoryLocation, function(Handler) {
  React.render(<Handler />, document.getElementById('app'));
});
```

> 注意：*main.js*是我们的React应用的入口点，当Browserify将整个依赖树串起来并生成最终的bundle.js时会用到，这里我们填入初始化的内容后我们基本不用再动它了。

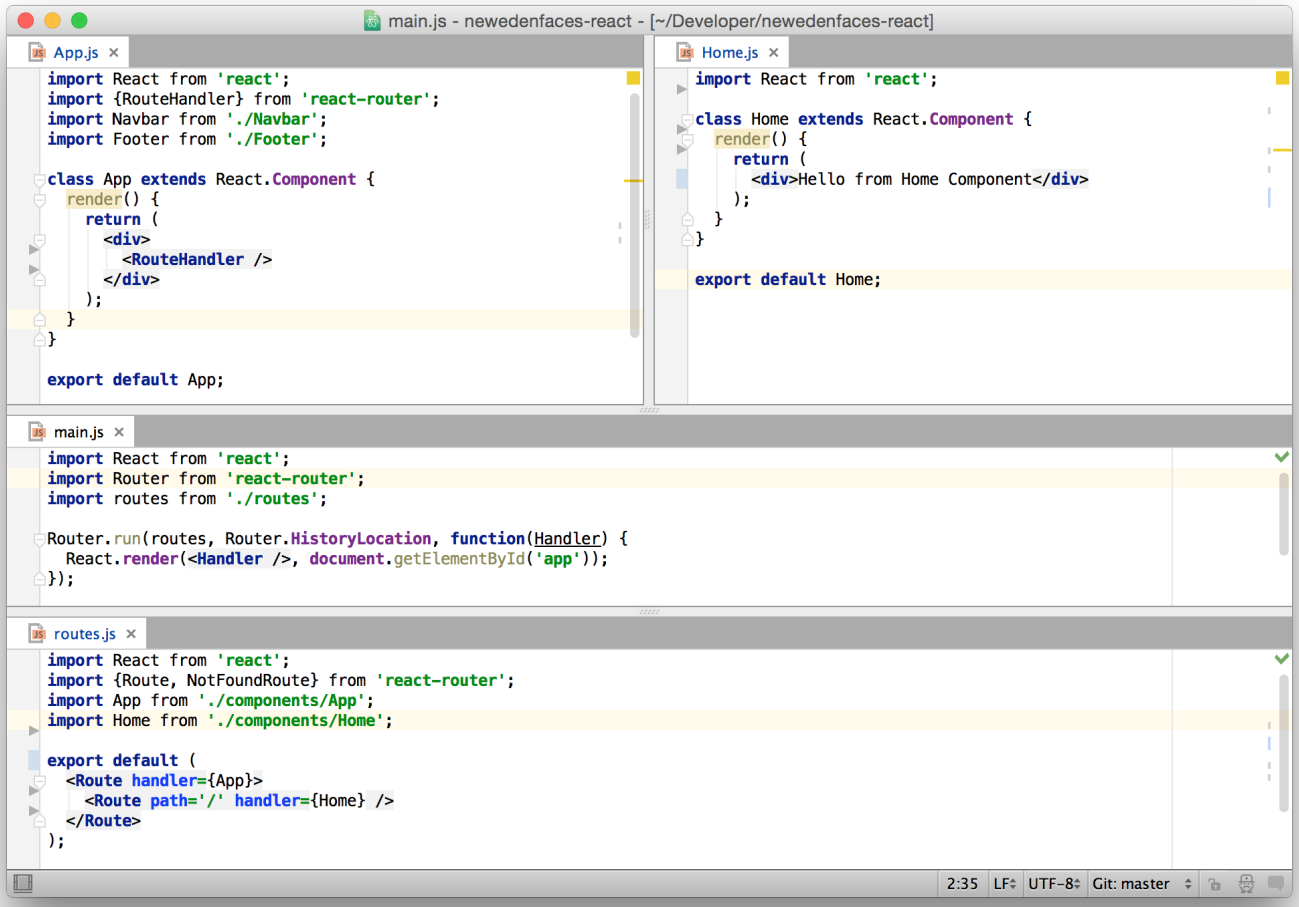
React Router引导route.js中的路由，将它们和URL匹配，然后执行相应的callback处理器，在这里即意味着渲染一个React组件到 `<div id="app"></div>` 。它是怎么知道要渲染哪个组件呢？举例来说，如果我们在 `/` URL路径，那么 `<Handler />` 将渲染Home组件，因为我们之前已经在route.js指定这个组件了。后面我们将添加更多的路由。

另外注意，为了让URL好看点，我们使用了 `HistoryLocation` 来启用HMTL History API。比如它的URL看起来会是 `http://localhost:3000/add` 而不是 `http://localhost:3000/#add` ，因为我们构建的是一个同型React应用（在客户端和服务端都能渲染），所以我们不需要用一些非正统的方式在服务器上重定向以启用这项特性，它直接就能用。

接下来让我们创建这一节最后一个React组件。在app/components目录新建文件*Home.js*，并添上内容：

```
import React from 'react';

class Home extends React.Component {
  render() {
    return (
      <div className='alert alert-info'>
        Hello from Home Component
      </div>
    );
  }
}

export default Home;
```

下面应该是我们在目前所创建的所有内容。现在是你检查代码的好时候了。



哦，还有一个，打开app目录下的alt.js并粘贴下面的代码，我将会在第9步真正用到它的时候再解释这些代码的目的。

```
import Alt from 'alt';

export default new Alt();
```

现在我们只需要在后端设置一些东西，就终于能将我们的应用运行起来了。

第八步：React路由（服务端）

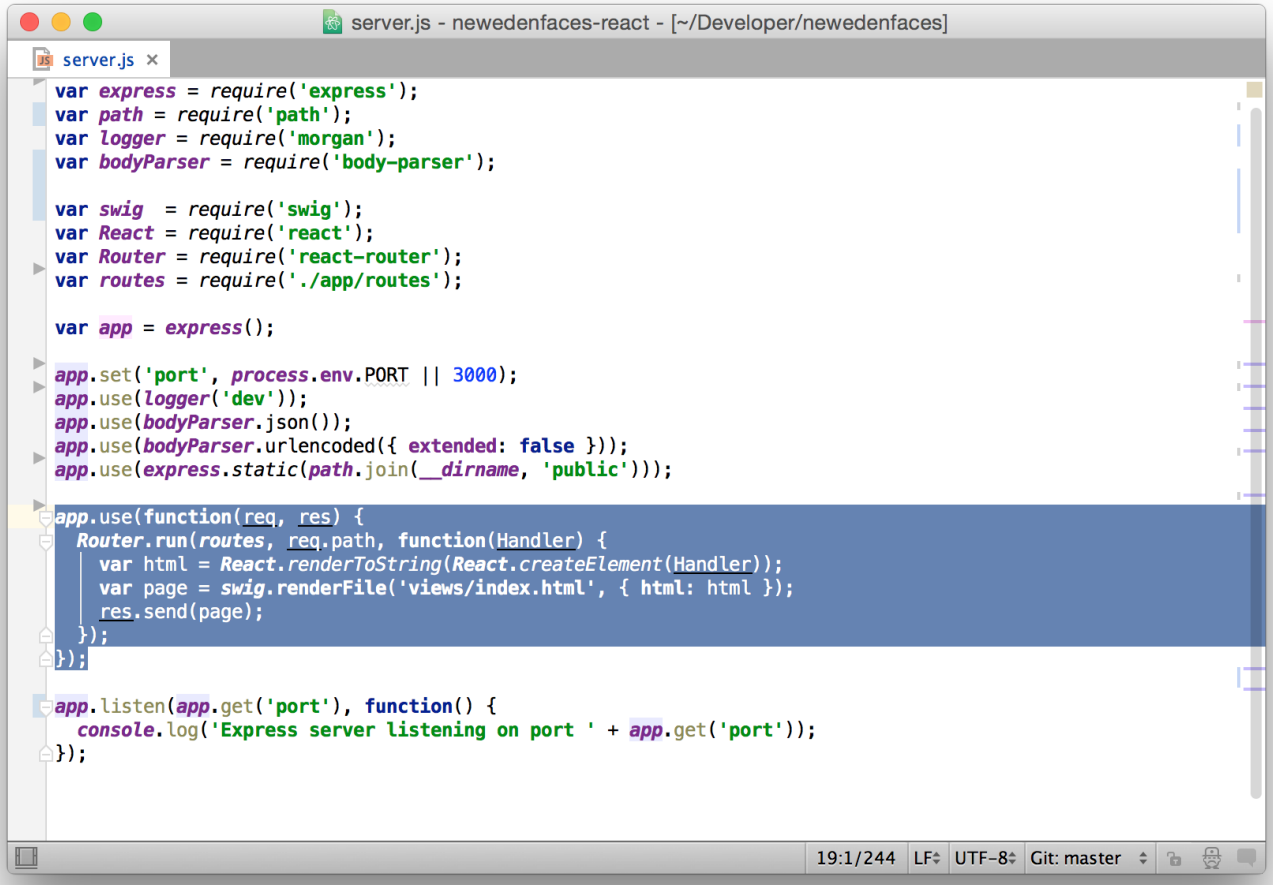
第八步：React路由（服务端）

打开*server.js*并将下面的代码粘贴到文件最前面，我们需要导入这些模块：

```
var swig  = require('swig');
var React = require('react');
var Router = require('react-router');
var routes = require('./app/routes');
```

然后，将下面的中间件也加入到*server.js*中去，放在现有的Express中间件之后。

```
app.use(function(req, res) {
  Router.run(routes, req.path, function(Handler) {
    var html = React.renderToString(React.createElement(Handler));
    var page = swig.renderFile('views/index.html', { html: html });
    res.send(page);
  });
});
```



这个中间件在每次请求时都会执行。这里server.js中的 `Router.run` 和main.js中 `Router.run` 的主要区别是应用是如何渲染的。

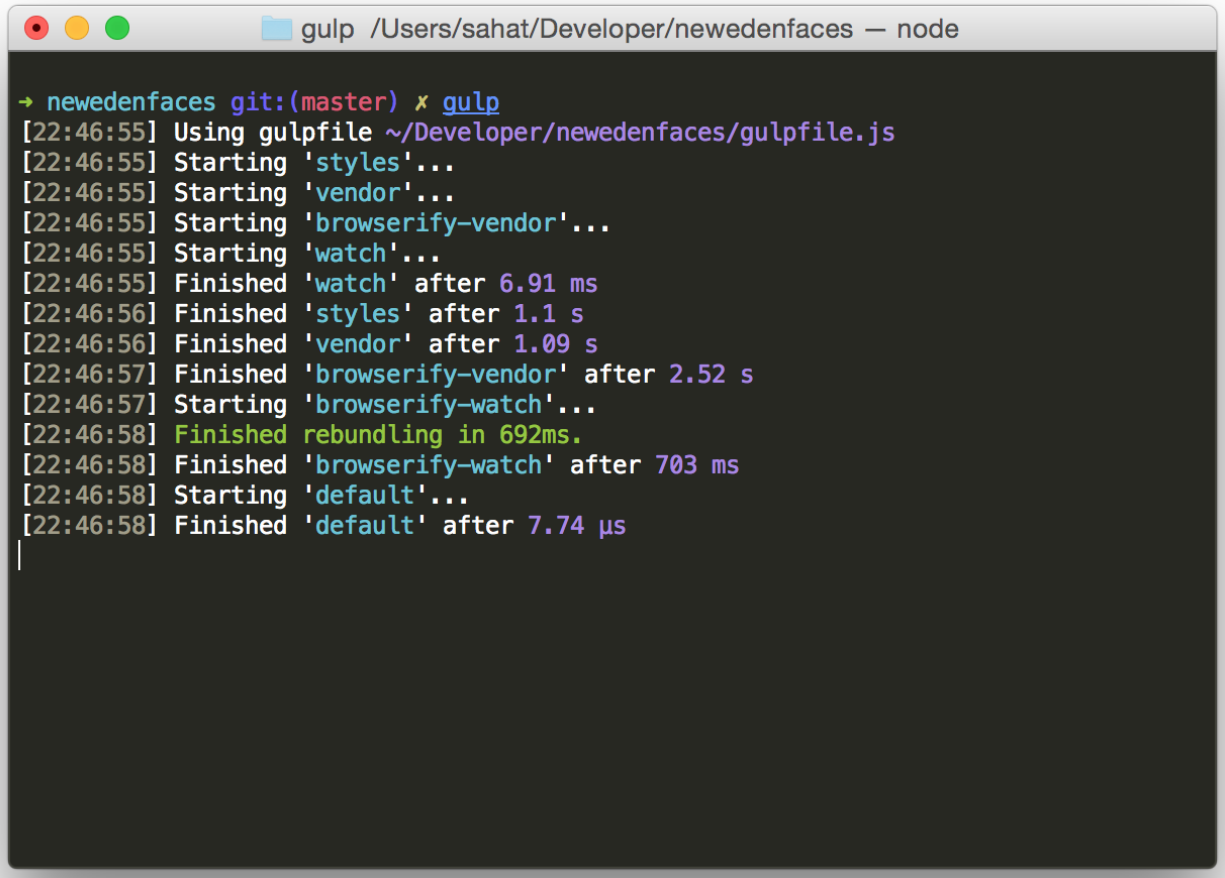
在客户端，渲染完成的HTML标记将被插入到 `<div id="app"></div>` ，在服务器端，渲染完成的HTML标记被发到index.html模板，然后被Swig模板引擎插入到 `<div id="app">{{ html|safe }}</div>` 中，我选择Swig是因为我想尝试下Jade和Handlerbars之外的选择。

但我们真的需要一个独立的模板吗？为什么不直接将内容渲染到App组件呢？是的，你可以这么做，只要你能接受违反W3C规范的HMTL标记，以及不能在组件中直接包含内嵌的script标签，比如Google Analytics。不过即便这么说，好像现在不规范的HMTL标记也不再和SEO相关了，也有一些绕过的办法来包含内嵌script标签，所以要怎么做看你咯，不过为了这个教程的目的，让我们还是使用Swig模板引擎吧。

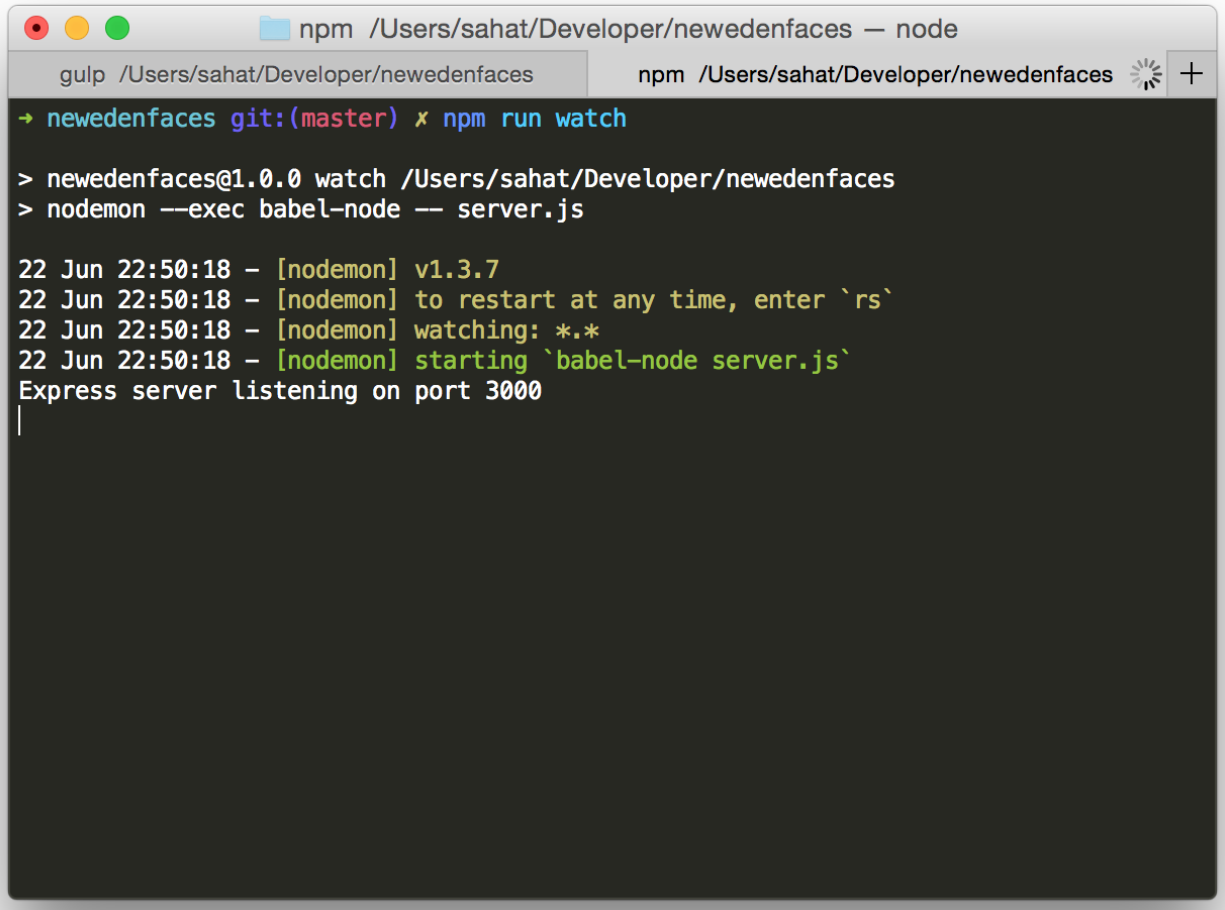
在项目根目录新建目录views，进入目录并新建文件*index.html*：

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8"/>
  <meta http-equiv="X-UA-Compatible" content="IE=edge"/>
  <meta name="viewport" content="width=device-width, initial-scale=1"/>
  <title>New Eden Faces</title>
  <link rel="stylesheet" href="https://fonts.googleapis.com/css?family=Source+Sans+Pro:300,400,600
,700,900"/>
  <link rel="stylesheet" href="/css/main.css"/>
</head>
<body>
<div id="app">{{ html|safe }}</div>
<script src="/js/vendor.js"></script>
<script src="/js/vendor.bundle.js"></script>
<script src="/js/bundle.js"></script>
</body>
</html>
```

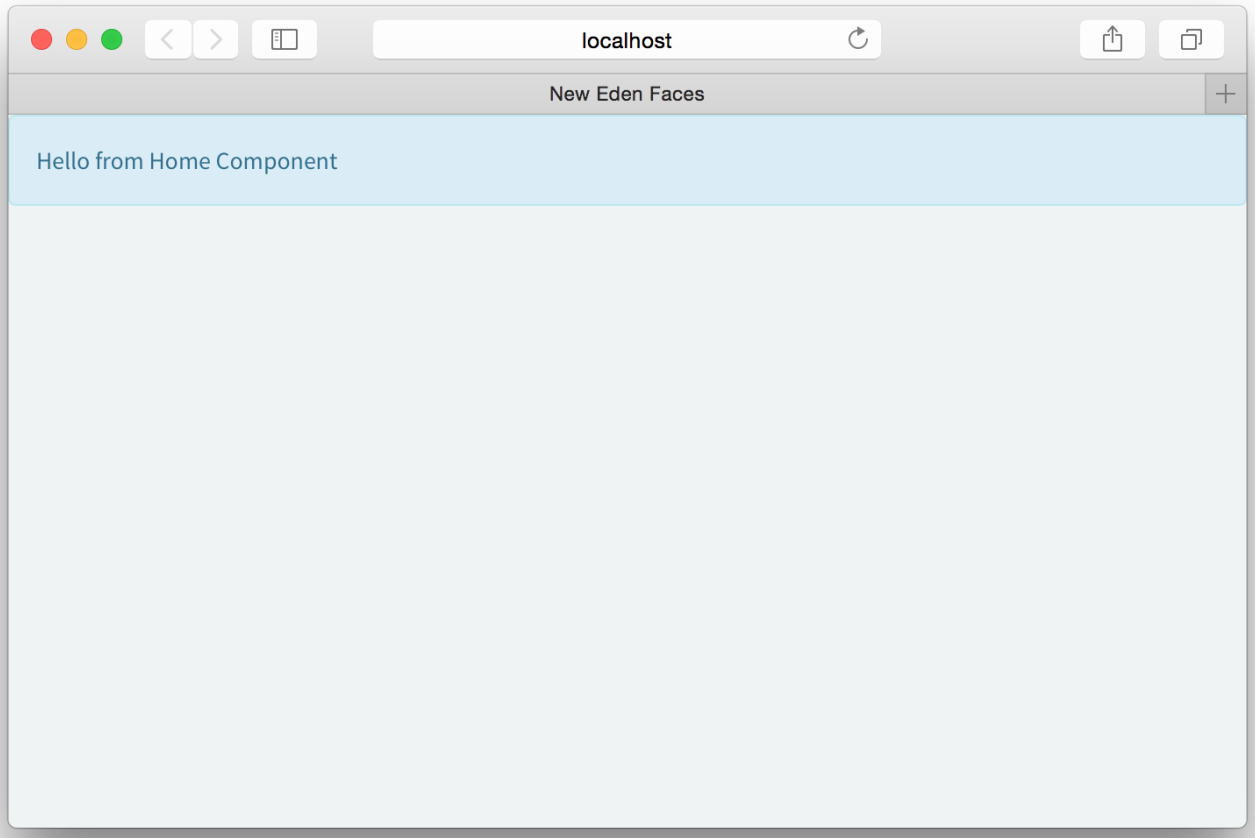
打开两个终端界面并进入根目录，在其中一个运行 `gulp` ，连接依赖文件、编译LESS样式并监视你的文件变化：



在另一个界面运行 `npm run watch` 来启动Node.js服务器并在文件变动时自动重启服务器：



在浏览器打开http://localhost:3000，现在你应该能看到React应用成功渲染了。



我们坚持到现在，做了大量的工作，结果就给我们显示了一个提示信息！不过还好，最艰难的部分已经结束了，从现在开始我们可以轻松一点，专注到创建React组件并实现REST API端点。

上面的两个命令 `gulp` 和 `npm run watch` 将为我们完成脏活累活，我们不用在添加React组件后的重新编译和Express服务器重启上费心啦。

第九步：Footer和Navbar组件

第九步：Footer和Navbar组件

Navbar和Footer都是相对简单的组件。Footer组件获取并展示Top5人物角色，Navbar组件获取并展示所有角色数量，然后还初始化一个Socket.IO事件监听器，用以跟踪在线访客的数量。

> 注意：这一节会比别的小节要稍长些，因为我会在这里谈到一些新概念，而其它小节将基于它们进行开发。

Footer组件

在components目录下新建文件*Footer.js*：

```
import React from 'react';
import {Link} from 'react-router';
import FooterStore from '../stores/FooterStore'
import FooterActions from '../actions/FooterActions';

class Footer extends React.Component {
  constructor(props) {
    super(props);
    this.state = FooterStore.getState();
    this.onChange = this.onChange.bind(this);
  }

  componentDidMount() {
    FooterStore.listen(this.onChange);
    FooterActions.getTopCharacters();
  }

  componentWillUnmount() {
    FooterStore.unlisten(this.onChange);
  }

  onChange(state) {
    this.setState(state);
  }

  render() {
    let leaderboardCharacters = this.state.characters.map((character) => {
      return (
        <li key={character.characterId}>
          <Link to={'/characters/' + character.characterId}>
            <img className='thumb-md' src={'http://image.eveonline.com/Character/' + character.chara
cterId + '_128.jpg'} />
          </Link>
        </li>
```

```
      )
    });

    return (
      <footer>
        <div className='container'>
          <div className='row'>
            <div className='col-sm-5'>
              <h3 className='lead'><strong>Information</strong> and <strong>Copyright</strong></
h3>
              <p>Powered by <strong>Node.js</strong>, <strong>MongoDB</strong> and <strong>Re
act</strong> with Flux architecture and server-side rendering.</p>
              <p>You may view the <a href='https://github.com/sahat/newedenfaces-react'>Source Code
</a> behind this project on GitHub.</p>
              <p>© 2015 Sahat Yalkabov.</p>
            </div>
            <div className='col-sm-7 hidden-xs'>
              <h3 className='lead'><strong>Leaderboard</strong> Top 5 Characters</h3>
              <ul className='list-inline'>
                {leaderboardCharacters}
              </ul>
            </div>
          </div>
        </div>
      </footer>
    );
  }
}

export default Footer;
```

为防止你还未熟悉ES6语法而晕头转向，在这里我将最后一次展示这段代码用ES5是如何写的，另外你也可以参看Using Alt with ES5 指南来了解创建action和store时语法的不同。

```
var React = require('react');
var Link = require('react-router').Link;
var FooterStore = require('../stores/FooterStore');
var FooterActions = require('../actions/FooterActions');

var Footer = React.createClass({
  getInitialState: function() {
    return FooterStore.getState();
  }

  componentDidMount: function() {
    FooterStore.listen(this.onChange);
    FooterActions.getTopCharacters();
  }

  componentWillUnmount: function() {
    FooterStore.unlisten(this.onChange);
  }
```

```
  onChange: function(state) {
    this.setState(state);
  }

  render() {
    var leaderboardCharacters = this.state.characters.map(function(character) {
      return (
        <li key={character.characterId}>
          <Link to={'/characters/' + character.characterId}>
            <img className='thumb-md' src={'http://image.eveonline.com/Character/' + character.chara
cterId + '_128.jpg'} />
          </Link>
        </li>
      );
    });

    return (
      <footer>
        <div className='container'>
          <div className='row'>
            <div className='col-sm-5'>
              <h3 className='lead'><strong>Information</strong> and <strong>Copyright</strong></
h3>
              <p>Powered by <strong>Node.js</strong>, <strong>MongoDB</strong> and <strong>Re
act</strong> with Flux architecture and server-side rendering.</p>
              <p>You may view the <a href='https://github.com/sahat/newedenfaces-react'>Source Code
</a> behind this project on GitHub.</p>
              <p>© 2015 Sahat Yalkabov.</p>
            </div>
            <div className='col-sm-7 hidden-xs'>
              <h3 className='lead'><strong>Leaderboard</strong> Top 5 Characters</h3>
              <ul className='list-inline'>
                {leaderboardCharacters}
              </ul>
            </div>
          </div>
        </div>
      </footer>
    );
  }
});

module.exports = Footer;
```

如果你还记得Flux架构那一节的内容，这些代码看上去应该挺熟悉。当组件加载后，将初始组件状态设置为FooterStore中的值，然后初始化store监听器。同样，当组件被卸载（比如导航至另一页面），store监听器也被移除。当store更新，`onChange`函数被调用，然后反过来又更新Footer的状态。

如果你之前用过React，在这里你需要注意的是，当使用ES6 class创建React组件，组件方法不再自动绑定`this`。也就是说，当你调用组件内部方法时，你需要手动绑定`this`，在之前，`React.createClass()`会帮我们自动绑定：

> 自动绑定：当在JavaScript中创建回调时，你经常需要手动绑定方法到它的实例以保证this的值正确，使用React，所有方法都自动绑定到组件实例。

以上出自于官方文档。不过在ES6中我们要这么做：

```
this.onChange = this.onChange.bind(this);
```

下面是关于这个问题更详细的例子：

```
class App extends React.Component {

  constructor(props) {
    super(props);
    this.state = AppStore.getState();
    this.onChange = this.onChange; // Need to add `.bind(this)`.
  }

  onChange(state) {
    // Object `this` will be undefined without binding it explicitly.
    this.setState(state);
  }

  render() {
    return null;
  }
}
```

现在你需要了解JavaScript中的 map() 方法，即使你之前用过，也还是可能搞不清楚它在JSX中是怎么用的（React官方教程并没有很好的解释它）。

它基本上是一个for-each循环，和Jade和Handlebars中的类似，但在这里你可以将结果分配给一个变量，然后你就可以在JSX里使用它了，就和用其它变量一样。它在React中很常见，你会经常用到。

> 注意：当渲染动态子组件时，如上面的 leaderboardCharacters ，React会要求你使用 key 属性来指定每一个子组件。

Link 组件当指定合适的*href*属性时会渲染一个链接标签，它还知道链接的目标是否可用，从而给链接加上 active 的类。如果你使用React Router，你需要使用Link模块在应用内部进行导航。

Actions

下面，我们将为Footer组件创建action和store，在app/actions目录新建*FooterActions.js*并添加：

```
import alt from '../alt';

class FooterActions {
  constructor() {
    this.generateActions(
      'getTopCharactersSuccess',
      'getTopCharactersFail'
    );
  }

  getTopCharacters() {
    $.ajax({ url: '/api/characters/top' })
      .done((data) => {
        this.actions.getTopCharactersSuccess(data)
      })
      .fail((jqXhr) => {
        this.actions.getTopCharactersFail(jqXhr)
      });
  }
}

export default alt.createActions(FooterActions);
```

首先，注意我们从第七步创建的alt.js中导入了一个Alt的实例，而不是从我们安装的Alt模块中。它是一个Alt的实例，实现了Flux dispatcher并提供创建Alt action和store的方法。你可以把它想象为我们的store和action之间的胶水。

这里我们有3个action，一个使用ajax获取数据，另外两个用来通知store获取数据是成功还是失败。在这个例子里，知道 `getTopCharacters` 何时被触发并没有什么用，我们真正想知道的是action执行成功（更新store然后重新渲染组件）还是失败（显示一个错误通知）。

Action可以很复杂，也可以很简单。有些action我们不关心它们做了什么，我们只关心它们是否被触发，比如这里的 `ajaxInProgress` 和 `ajaxComplete` 被用来通知store，AJAX请求是正在进行还是已经完成。

> 注意：Alt的action能通过 `generateActions` 方法创建，只要它们直接通向dispatch。具体可参看官方文档。

下面的两种创建action方式是等价的，可依据你的喜好进行选择：

```
  getTopCharactersSuccess(payload) {
    this.dispatch(payload);
  }

  getTopCharactersFail(payload) {
    this.dispatch(payload);
  }

  // Equivalent to this...
  this.generateActions(
    'getTopCharactersSuccess',
    'getTopCharactersFail'
  );
```

最后，我们通过 `alt.createActions` 将FooterActions封装并暴露出来，然后我们可以在Footer组件里导入并使用它。

Store

下面，在app/stores目录下新建文件*FooterStore.js*：

```
import alt from '../alt';
import FooterActions from '../actions/FooterActions';

class FooterStore {
  constructor() {
    this.bindActions(FooterActions);
    this.characters = [];
  }

  onGetTopCharactersSuccess(data) {
    this.characters = data.slice(0, 5);
  }

  onGetTopCharactersFail(jqXhr) {
    // Handle multiple response formats, fallback to HTTP status code number.
    toastr.error(jqXhr.responseJSON && jqXhr.responseJSON.message || jqXhr.responseText || jqXhr.sta
tusText);
  }
}

export default alt.createStore(FooterStore);
```

在store中创建的变量，比如 `this` 所赋值的变量，都将成为状态的一部分。当Footer组件初始化并调用 `FooterStore.getState()` ，它会获取在构造函数中指定的当前状态（在一开始只是一个空数组，而遍历空数组会返回另一个空数组，所以在Footer组件第一次加载时并没有渲染任何内容）。

`bindActions` 用于将action绑定到store中定义的相应处理函数。比如，一个命名为 `foo` 的action会匹配store中叫做 `onFoo` 或者 `foo` 的处理函数，不过需要注意它不会同时匹配两者。因此我们在

FooterActions.js中定义的action `getTopCharactersSuccess` 和 `getTopCharactersFail` 会匹配到这里的处理函数 `onGetTopCharactersSuccess` 和 `onGetTopCharactersFail` 。

> 注意：如需更精细的控制store监听的action以及它们绑定的处理函数，可参看文档中的 `bindListeners` 方法。

在 `onGetTopCharactersSuccess` 处理函数中我们更新了store的数据，现在它包含Top 5角色，并且我们在Footer组件中初始化了store监听器，当FooterStore更新后组件会自动的重新渲染。

我们会使用Toastr库来处理通知。也许你会问为什么不使用纯React通知组件呢？也许你以前看到过为React设计的通知组件，但我个人认为这是少数不太适合用React的地方（还有一个是tooltips）。我认为要从应用的任何地方显示一个通知，使用命令方式远比声明式要简单，我以前曾经构建过使用React和Flux的通知组件，但老实说，用来它处理显隐状态、动画以及z-index位置等，非常痛苦。

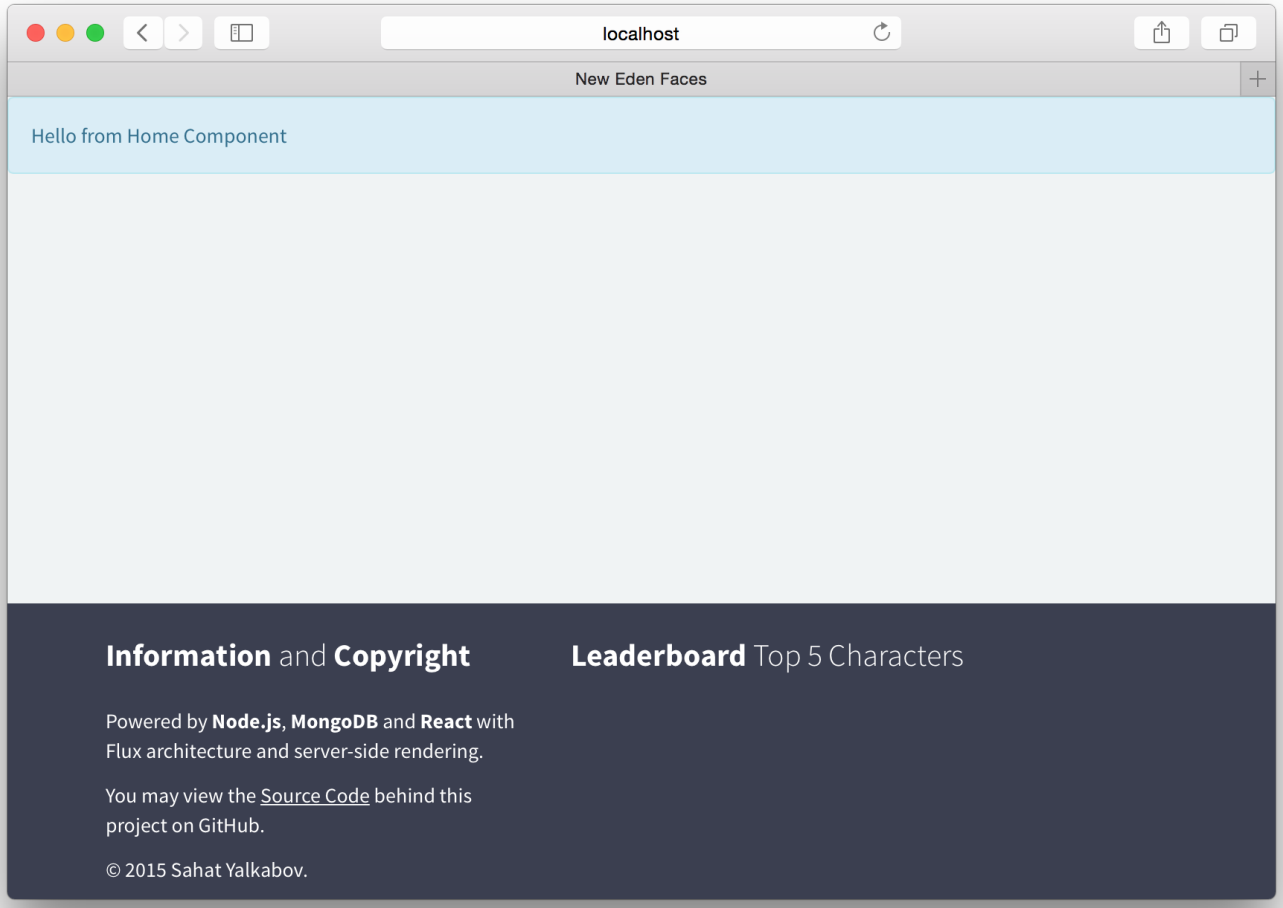
打开app/components下的*App.js*并导入Footer组件：

```
import Footer from './Footer';
```

然后将 `<Footer />` 添加到 `<RouterHandler / >` 组件后面：

```
<div>
  <RouteHandler />
  <Footer />
</div>
```

刷新浏览器你应该看到新的底部：



我们稍后会实现Express API以及添加人物角色数据库，不过现在让我们还是继续构建Navbar组件。因为之前已经讲过了alt action和store，这里将会尽量简略的说明Navbar组件如何构建。

Navbar组件

在app/components目录新建文件*Navbar.js*：

```
import React from 'react';
import {Link} from 'react-router';
import NavbarStore from '../stores/NavbarStore';
import NavbarActions from '../actions/NavbarActions';

class Navbar extends React.Component {
  constructor(props) {
    super(props);
    this.state = NavbarStore.getState();
    this.onChange = this.onChange.bind(this);
  }

  componentDidMount() {
    NavbarStore.listen(this.onChange);
    NavbarActions.getCharacterCount();

    let socket = io.connect();
```

```
    socket.on('onlineUsers', (data) => {
      NavbarActions.updateOnlineUsers(data);
    });

    $(document).ajaxStart(() => {
      NavbarActions.updateAjaxAnimation('fadeIn');
    });

    $(document).ajaxComplete(() => {
      setTimeout(() => {
        NavbarActions.updateAjaxAnimation('fadeOut');
      }, 750);
    });
  }

  componentWillUnmount() {
    NavbarStore.unlisten(this.onChange);
  }

  onChange(state) {
    this.setState(state);
  }

  handleSubmit(event) {
    event.preventDefault();

    let searchQuery = this.state.searchQuery.trim();

    if (searchQuery) {
      NavbarActions.findCharacter({
        searchQuery: searchQuery,
        searchForm: this.refs.searchForm.getDOMNode(),
        router: this.context.router
      });
    }
  }

  render() {
    return (
      <nav className='navbar navbar-default navbar-static-top'>
        <div className='navbar-header'>
          <button type='button' className='navbar-toggle collapsed' data-toggle='collapse' data-targe
t='#navbar'>
            <span className='sr-only'>Toggle navigation</span>
            <span className='icon-bar'></span>
            <span className='icon-bar'></span>
            <span className='icon-bar'></span>
          </button>
          <Link to='/' className='navbar-brand'>
            <span ref='triangles' className={'triangles animated ' + this.state.ajaxAnimationClass}>
              <div className='tri invert'></div>
              <div className='tri invert'></div>
              <div className='tri'></div>
              <div className='tri invert'></div>
```

```
            <div className='tri invert'></div>
            <div className='tri'></div>
            <div className='tri invert'></div>
            <div className='tri'></div>
            <div className='tri invert'></div>
          </span>
          NEF
          <span className='badge badge-up badge-danger'>{this.state.onlineUsers}</span>
        </Link>
      </div>
      <div id='navbar' className='navbar-collapse collapse'>
        <form ref='searchForm' className='navbar-form navbar-left animated' onSubmit={this.handle
Submit.bind(this)}>
          <div className='input-group'>
            <input type='text' className='form-control' placeholder={this.state.totalCharacters + ' char
acters'} value={this.state.searchQuery} onChange={NavbarActions.updateSearchQuery} />
            <span className='input-group-btn'>
              <button className='btn btn-default' onClick={this.handleSubmit.bind(this)}><span classN
ame='glyphicon glyphicon-search'></span></button>
            </span>
          </div>
        </form>
        <ul className='nav navbar-nav'>
          <li><Link to='/'>Home</Link></li>
          <li><Link to='/stats'>Stats</Link></li>
          <li className='dropdown'>
            <a href='#' className='dropdown-toggle' data-toggle='dropdown'>Top 100 <span classNa
me='caret'></span></a>
            <ul className='dropdown-menu'>
              <li><Link to='/top'>Top Overall</Link></li>
              <li className='dropdown-submenu'>
                <Link to='/top/caldari'>Caldari</Link>
                <ul className='dropdown-menu'>
                  <li><Link to='/top/caldari/achura'>Achura</Link></li>
                  <li><Link to='/top/caldari/civire'>Civire</Link></li>
                  <li><Link to='/top/caldari/deteis'>Deteis</Link></li>
                </ul>
              </li>
              <li className='dropdown-submenu'>
                <Link to='/top/gallente'>Gallente</Link>
                <ul className='dropdown-menu'>
                  <li><Link to='/top/gallente/gallente'>Gallente</Link></li>
                  <li><Link to='/top/gallente/intaki'>Intaki</Link></li>
                  <li><Link to='/top/gallente/jin-mei'>Jin-Mei</Link></li>
                </ul>
              </li>
              <li className='dropdown-submenu'>
                <Link to='/top/minmatar'>Minmatar</Link>
                <ul className='dropdown-menu'>
                  <li><Link to='/top/minmatar/brutor'>Brutor</Link></li>
                  <li><Link to='/top/minmatar/sebiestor'>Sebiestor</Link></li>
                  <li><Link to='/top/minmatar/vherokior'>Vherokior</Link></li>
                </ul>
              </li>
              <li className='dropdown-submenu'>
```

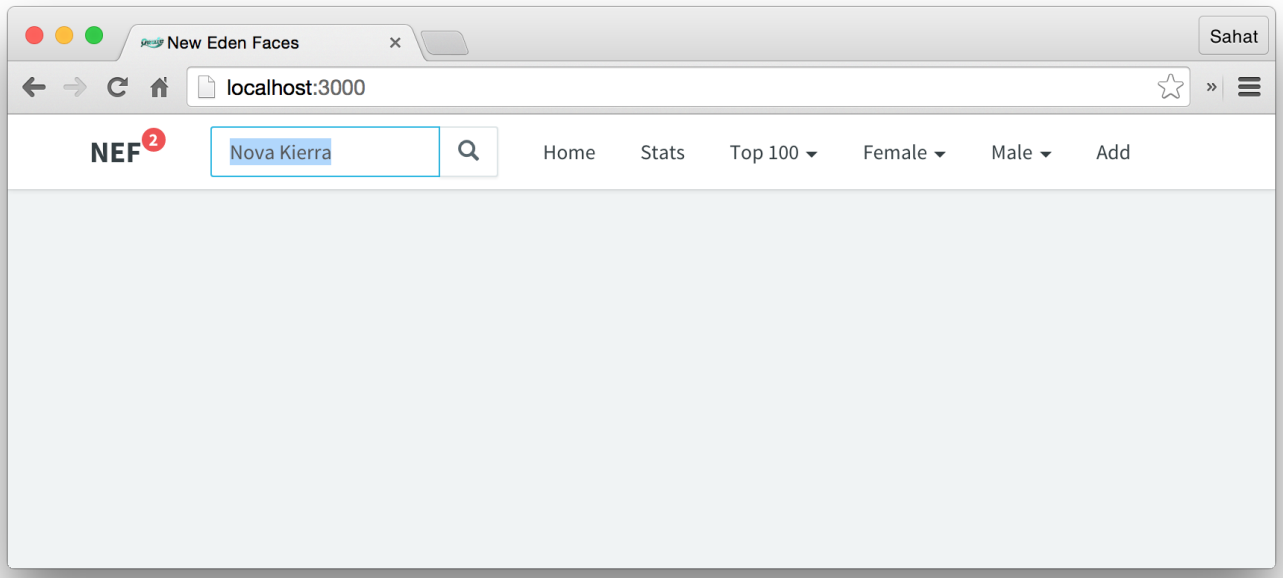
```
            <li className='dropdown-submenu'>
              <Link to='/top/amarr'>Amarr</Link>
              <ul className='dropdown-menu'>
                <li><Link to='/top/amarr/amarr'>Amarr</Link></li>
                <li><Link to='/top/amarr/ni-kunni'>Ni-Kunni</Link></li>
                <li><Link to='/top/amarr/khanid'>Khanid</Link></li>
              </ul>
            </li>
            <li className='divider'></li>
            <li><Link to='/shame'>Hall of Shame</Link></li>
          </ul>
        </li>
        <li className='dropdown'>
          <a href='#' className='dropdown-toggle' data-toggle='dropdown'>Female <span className='caret'></span></a>
          <ul className='dropdown-menu'>
            <li><Link to='/female'>All</Link></li>
            <li className='dropdown-submenu'>
              <Link to='/female/caldari'>Caldari</Link>
              <ul className='dropdown-menu'>
                <li><Link to='/female/caldari/achura'>Achura</Link></li>
                <li><Link to='/female/caldari/civire/'>Civire</Link></li>
                <li><Link to='/female/caldari/deteis'>Deteis</Link></li>
              </ul>
            </li>
            <li className='dropdown-submenu'>
              <Link to='/female/gallente'>Gallente</Link>
              <ul className='dropdown-menu'>
                <li><Link to='/female/gallente/gallente'>Gallente</Link></li>
                <li><Link to='/female/gallente/intaki'>Intaki</Link></li>
                <li><Link to='/female/gallente/jin-mei'>Jin-Mei</Link></li>
              </ul>
            </li>
            <li className='dropdown-submenu'>
              <Link to='/female/minmatar'>Minmatar</Link>
              <ul className='dropdown-menu'>
                <li><Link to='/female/minmatar/brutor'>Brutor</Link></li>
                <li><Link to='/female/minmatar/sebiestor'>Sebiestor</Link></li>
                <li><Link to='/female/minmatar/vherokior'>Vherokior</Link></li>
              </ul>
            </li>
            <li className='dropdown-submenu'>
              <Link to='/female/amarr'>Amarr</Link>
              <ul className='dropdown-menu'>
                <li><Link to='/female/amarr/amarr'>Amarr</Link></li>
                <li><Link to='/female/amarr/ni-kunni'>Ni-Kunni</Link></li>
                <li><Link to='/female/amarr/khanid'>Khanid</Link></li>
              </ul>
            </li>
          </ul>
        </li>
        <li className='dropdown'>
          <a href='#' className='dropdown-toggle' data-toggle='dropdown'>Male <span className='caret'></span></a>
          <ul className='dropdown-menu'>
```

```
              <li><Link to='/male'>All</Link></li>
              <li className='dropdown-submenu'>
                <Link to='/male/caldari'>Caldari</Link>
                <ul className='dropdown-menu'>
                  <li><Link to='/male/caldari/achura'>Achura</Link></li>
                  <li><Link to='/male/caldari/civire'>Civire</Link></li>
                  <li><Link to='/male/caldari/deteis'>Deteis</Link></li>
                </ul>
              </li>
              <li className='dropdown-submenu'>
                <Link to='/male/gallente'>Gallente</Link>
                <ul className='dropdown-menu'>
                  <li><Link to='/male/gallente/gallente'>Gallente</Link></li>
                  <li><Link to='/male/gallente/intaki'>Intaki</Link></li>
                  <li><Link to='/male/gallente/jin-mei'>Jin-Mei</Link></li>
                </ul>
              </li>
              <li className='dropdown-submenu'>
                <Link to='/male/minmatar'>Minmatar</Link>
                <ul className='dropdown-menu'>
                  <li><Link to='/male/minmatar/brutor'>Brutor</Link></li>
                  <li><Link to='/male/minmatar/sebiestor'>Sebiestor</Link></li>
                  <li><Link to='/male/minmatar/vherokior'>Vherokior</Link></li>
                </ul>
              </li>
              <li className='dropdown-submenu'>
                <Link to='/male/amarr'>Amarr</Link>
                <ul className='dropdown-menu'>
                  <li><Link to='/male/amarr/amarr'>Amarr</Link></li>
                  <li><Link to='/male/amarr/ni-kunni'>Ni-Kunni</Link></li>
                  <li><Link to='/male/amarr/khanid'>Khanid</Link></li>
                </ul>
              </li>
            </ul>
          </li>
          <li><Link to='/add'>Add</Link></li>
        </ul>
      </div>
    </nav>
  );
 }
}

Navbar.contextTypes = {
 router: React.PropTypes.func.isRequired
};

export default Navbar;
```

必须承认，这里使用循环的话可以少写一些代码，但现在这样对我来说更直观。

你可能立刻注意到的一个东西是class变量 `contextTypes` 。我们需要它来引用router的实例，从而让我们能访问当前路径、请求参数、路由参数以及到其它路由的变换。我们不在Navbar组件里直接使用它，而是

将它作为一个参数传递给Navbar action，以使它能导航到特定character资料页面。



`componentDidMount` 是我们发起与Socket.IO的连接，并初始化 `ajaxStart` 和 `ajaxComplete` 时间监听器地方，我们会在AJAX请求时在NEF logo旁边显示加载指示。

`handleSubmit` 是用来处理表单提交的程序，在按下Enter键或点击Search图标时执行。它会做一些输入清理和验证工作，然后触发 `findCharacter` action。另外我们还传递了搜索区域的DOM节点给action，以便当搜索结果为0时加载一个震动动画。

Actions

在app/actions目录下新建文件*NavbarActions.js*：

```
import alt from '../alt';
import {assign} from 'underscore';

class NavbarActions {
  constructor() {
    this.generateActions(
      'updateOnlineUsers',
      'updateAjaxAnimation',
      'updateSearchQuery',
      'getCharacterCountSuccess',
      'getCharacterCountFail',
      'findCharacterSuccess',
      'findCharacterFail'
    );
  }

  findCharacter(payload) {
    $.ajax({
      url: '/api/characters/search',
      data: { name: payload.searchQuery }
    })
      .done((data) => {
        assign(payload, data);
        this.actions.findCharacterSuccess(payload);
      })
      .fail(() => {
        this.actions.findCharacterFail(payload);
      });
  }

  getCharacterCount() {
    $.ajax({ url: '/api/characters/count' })
      .done((data) => {
        this.actions.getCharacterCountSuccess(data)
      })
      .fail((jqXhr) => {
        this.actions.getCharacterCountFail(jqXhr)
      });
  }
}

export default alt.createActions(NavbarActions);
```

我想大多数action的命名应该能够自我解释，不过为了更清楚的理解，在下面简单的描述一下它们是干什么的：

Action	Description
`updateOnlineUsers`	当Socket.IO事件更新时设置在线用户数
`updateAjaxAnimation`	添加”fadeIn”或”fadeOut”类到加载指示器

Action	Description
`updateSearchQuery`	当使用键盘时设置搜索请求
`getCharacterCount`	从服务器获取总角色数
`getCharacterCountSuccess`	返回角色总数
`getCharacterCountFail`	返回jQuery jqXhr对象
`findCharacter`	根据名称查找角色

Store

在app/stores目录下创建*NavbarStore.js*：

```
import alt from '../alt';
import NavbarActions from '../actions/NavbarActions';

class NavbarStore {
  constructor() {
    this.bindActions(NavbarActions);
    this.totalCharacters = 0;
    this.onlineUsers = 0;
    this.searchQuery = '';
    this.ajaxAnimationClass = '';
  }

  onFindCharacterSuccess(payload) {
    payload.router.transitionTo('/characters/' + payload.characterId);
  }

  onFindCharacterFail(payload) {
    payload.searchForm.classList.add('shake');
    setTimeout(() => {
      payload.searchForm.classList.remove('shake');
    }, 1000);
  }

  onUpdateOnlineUsers(data) {
    this.onlineUsers = data.onlineUsers;
  }

  onUpdateAjaxAnimation(className) {
    this.ajaxAnimationClass = className; //fadein or fadeout
  }

  onUpdateSearchQuery(event) {
    this.searchQuery = event.target.value;
  }

  onGetCharacterCountSuccess(data) {
    this.totalCharacters = data.count;
  }

  onGetCharacterCountFail(jqXhr) {
    toastr.error(jqXhr.responseJSON.message);
  }
}

export default alt.createStore(NavbarStore);
```

回忆一下我们在Navbar组件中的代码：

```
<input type='text' className='form-control' placeholder={this.state.totalCharacters + ' characters'} v
alue={this.state.searchQuery} onChange={NavbarActions.updateSearchQuery} />
```

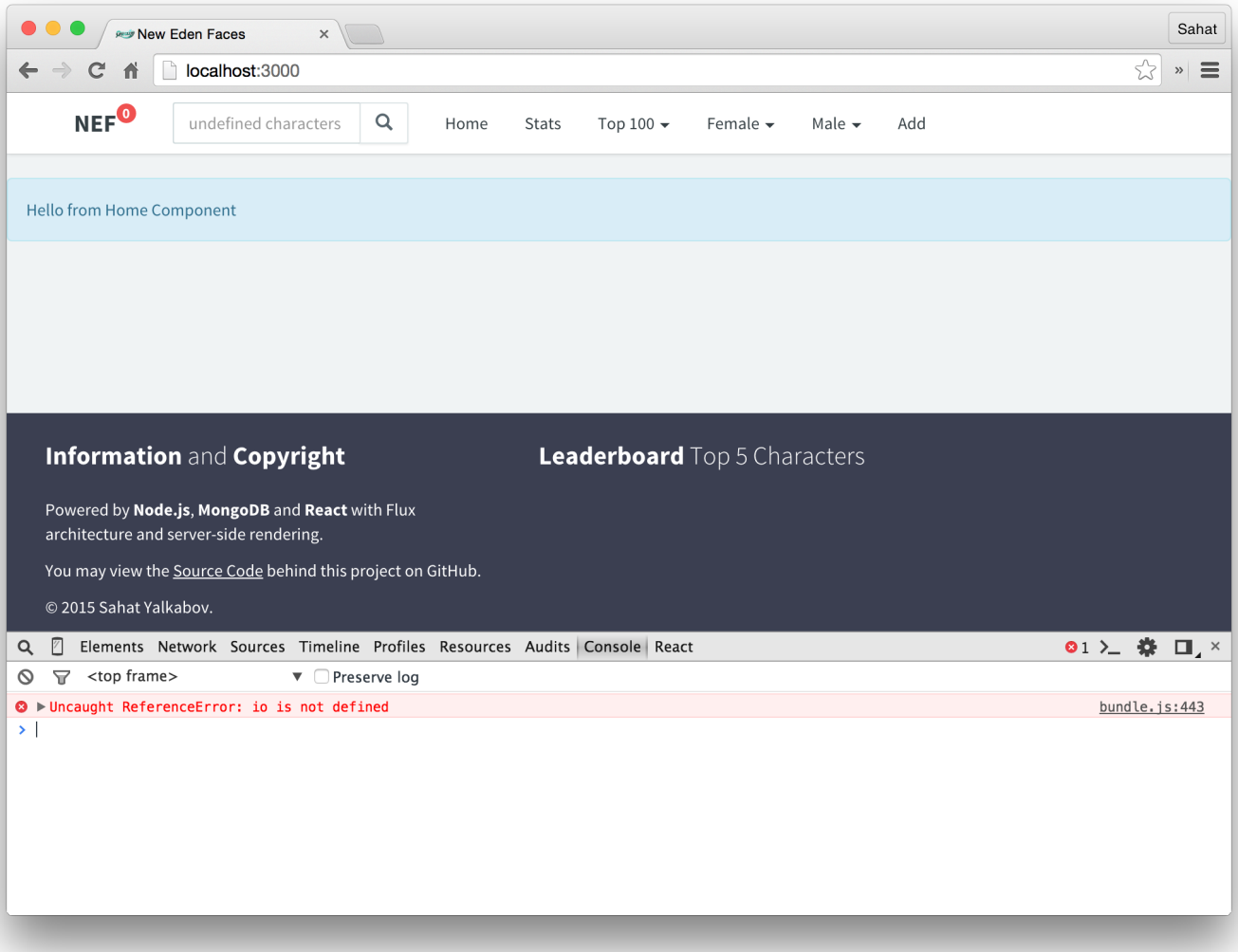
因为 onChange 方法返回一个event对象，所以这里我们在 onUpdateSearchQuery 使用 event.target.value 来获取输入框的值。

再次打开App.js并导入Navbar组件：

```
import Navbar from './Navbar';
```

然后在 <RouterHandler /> 添加 <Navbar /> 组件：

```
<div>
  <Navbar />
  <RouteHandler />
  <Footer />
</div>
```



由于我们还没有设置服务器端的Socke.IO，也没有实现任何API，所以现在你应该看不到在线访问人数或总的character数。

第十步：Socke.IO – 实时用户数

第十步：Socke.IO – 实时用户数

本节我们将聚焦在服务器端的Socket.IO。

打开server.js并找到下面的代码：

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

用下面的代码替换上面的：

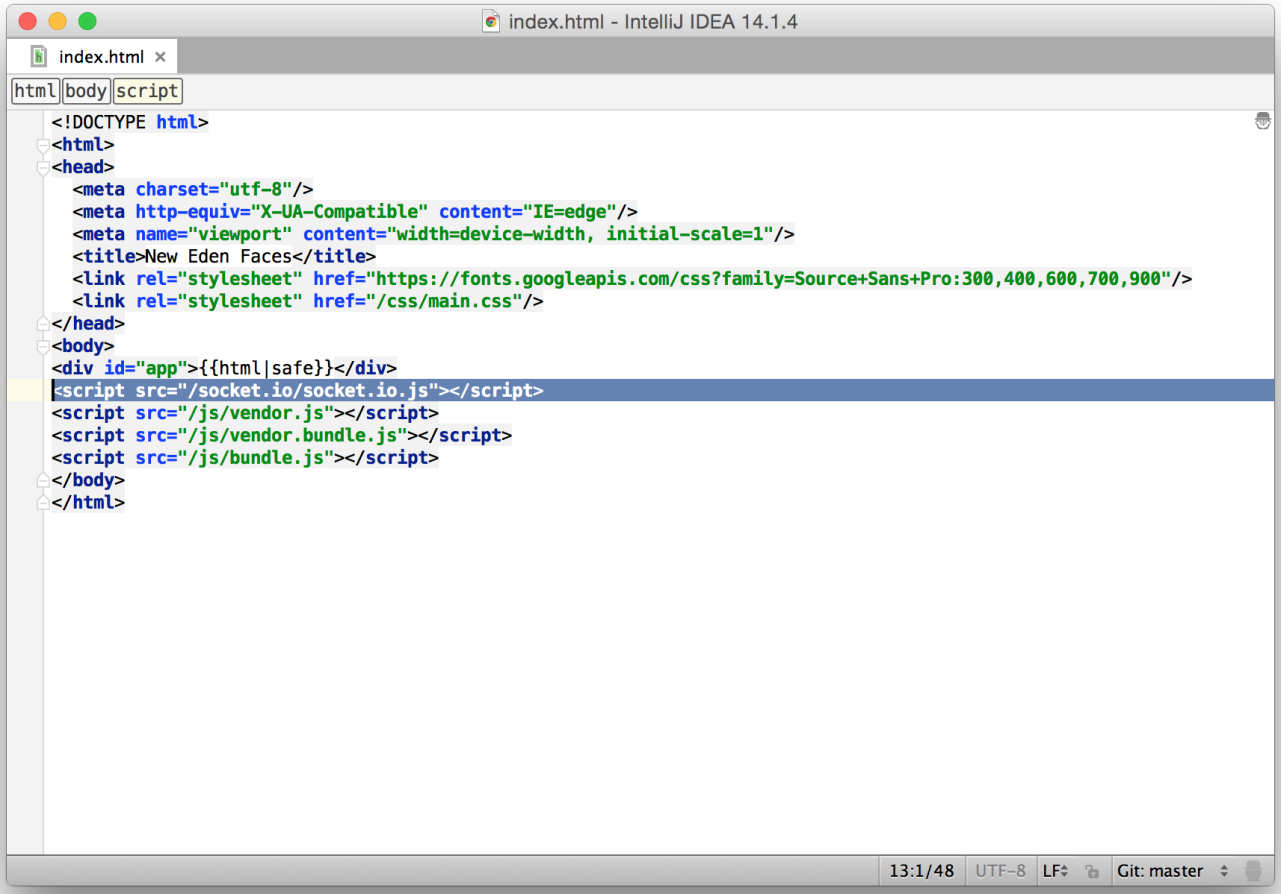
```
/**
 * Socket.io stuff.
 */
var server = require('http').createServer(app);
var io = require('socket.io')(server);
var onlineUsers = 0;

io.sockets.on('connection', function(socket) {
  onlineUsers++;

  io.sockets.emit('onlineUsers', { onlineUsers: onlineUsers });

  socket.on('disconnect', function() {
    onlineUsers--;
    io.sockets.emit('onlineUsers', { onlineUsers: onlineUsers });
  });
});

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

概括的来说，当发起一个WebSocket连接，它增加 onlineUsers 数量（一个全局变量）并发布一个广播——“嘿，我现在有这么多在线访问者啦！”当某人关闭浏览器离开， onlineUsers 数量减少并再次发布广播“嘿，有人刚刚离开了，我现在有这么多在线访问者了。”

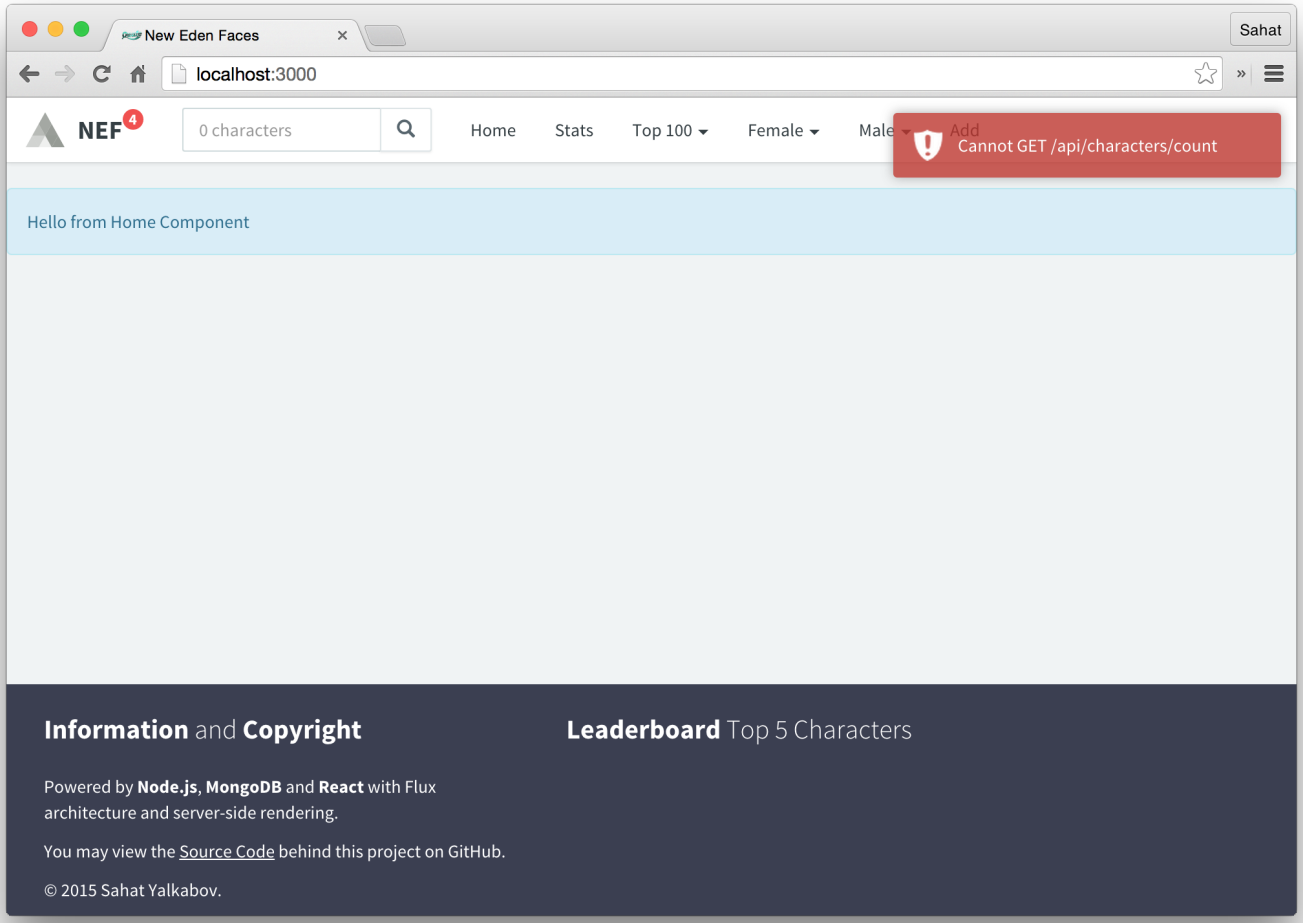
> 注意：如果你从来没用过Socket.IO，那么这个聊天室应用教程非常适合你。

打开views目录下的index.html并添加下面的代码到其它script标签下面：

```
<script src="/socket.io/socket.io.js"></script>
```



刷新浏览器，然后在不同的标签页打开http://localhost:3000以模拟不同的用户连接。现在你应该能在logo的圆点上看到访问者总数了。



到目前为止，我们既没有完成前端，也没有能用的API端点。我们可以在教程前半部分专注在前端，然后在后半部分专注于后端，或者反过来。但就我个人来说，我从来没像这样构建过任何App。在开发过程中，我一般在前端和后端之间切换着来做。

第十一步：添加Character的组件

第十一步：添加Character的组件

这个组件包含一个简单的表单。成功或失败的消息会显示在输入框下的 help-block 里。

组件

在app/components目录新建文件*AddCharacter.js* :

```
import React from 'react';
import AddCharacterStore from '../stores/AddCharacterStore';
import AddCharacterActions from '../actions/AddCharacterActions';

class AddCharacter extends React.Component {
  constructor(props) {
    super(props);
    this.state = AddCharacterStore.getState();
    this.onChange = this.onChange.bind(this);
  }

  componentDidMount() {
    AddCharacterStore.listen(this.onChange);
  }

  componentWillUnmount() {
    AddCharacterStore.unlisten(this.onChange);
  }

  onChange(state) {
    this.setState(state);
  }

  handleSubmit(event) {
    event.preventDefault();

    var name = this.state.name.trim();
    var gender = this.state.gender;

    if (!name) {
      AddCharacterActions.invalidName();
      this.refs.nameTextField.getDOMNode().focus();
    }

    if (!gender) {
      AddCharacterActions.invalidGender();
    }

    if (name && gender) {
      AddCharacterActions.addCharacter(name, gender);
```

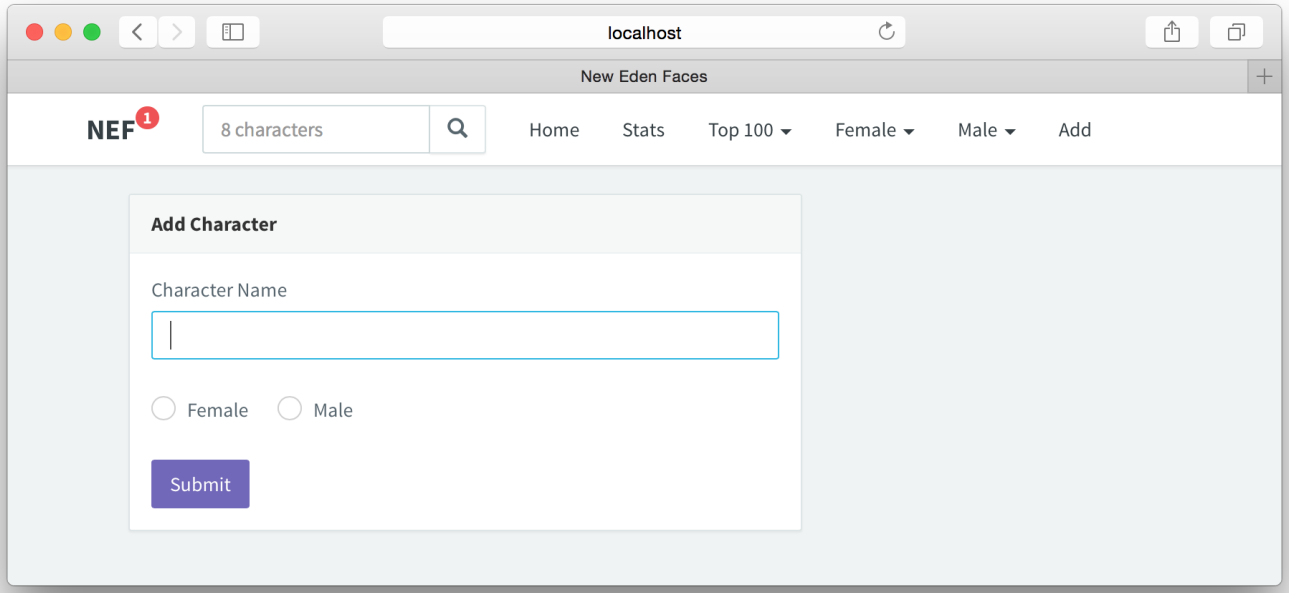
```
    }
  }

  render() {
    return (
      <div className='container'>
        <div className='row flipInX animated'>
          <div className='col-sm-8'>
            <div className='panel panel-default'>
              <div className='panel-heading'>Add Character</div>
              <div className='panel-body'>
                <form onSubmit={this.handleSubmit.bind(this)}>
                  <div className={'form-group ' + this.state.nameValidationState}>
                    <label className='control-label'>Character Name</label>
                    <input type='text' className='form-control' ref='nameTextField' value={this.state.name}
                           onChange={AddCharacterActions.updateName} autoFocus/>
                    <span className='help-block'>{this.state.helpBlock}</span>
                  </div>
                  <div className={'form-group ' + this.state.genderValidationState}>
                    <div className='radio radio-inline'>
                      <input type='radio' name='gender' id='female' value='Female' checked={this.state.gen
der === 'Female'}
                             onChange={AddCharacterActions.updateGender}/>
                      <label htmlFor='female'>Female</label>
                    </div>
                    <div className='radio radio-inline'>
                      <input type='radio' name='gender' id='male' value='Male' checked={this.state.gender
=== 'Male'}
                             onChange={AddCharacterActions.updateGender}/>
                      <label htmlFor='male'>Male</label>
                    </div>
                  </div>
                  <button type='submit' className='btn btn-primary'>Submit</button>
                </form>
              </div>
            </div>
          </div>
        </div>
      </div>
    );
  }
}

export default AddCharacter;
```

现在你可以看到这些组件的一些共同点：

1. 设置组件的初始状态为store中的值。
2. 在 componentDidMount 中添加store监听者，在 componentWillUnmount 中移除。
3. 添加 onChange 方法，无论何时当store改变后更新组件状态。

handleSubmit 方法的作用和你想的一样——处理添加新角色的表单提交。当它为真时我们能在

`addCharacter` action里完成表单验证，不过这样做的话，需要我们将输入区的DOM节点传到action，因为当`nameTextField`无效时，需要focus在输入框，这样用户可以直接输入而无需点击一下输入框。



Actions

在app/actions目录新建*AddCharacterActions.js*：

```
import alt from '../alt';

class AddCharacterActions {
  constructor() {
    this.generateActions(
      'addCharacterSuccess',
      'addCharacterFail',
      'updateName',
      'updateGender',
      'invalidName',
      'invalidGender'
    );
  }

  addCharacter(name, gender) {
    $.ajax({
      type: 'POST',
      url: '/api/characters',
      data: { name: name, gender: gender }
    })
      .done((data) => {
        this.actions.addCharacterSuccess(data.message);
      })
      .fail((jqXhr) => {
        this.actions.addCharacterFail(jqXhr.responseJSON.message);
      });
  }
}

export default alt.createActions(AddCharacterActions);
```

当角色被成功加入数据库后触发`addCharacterSuccess`，当失败时触发`addCharacterFail`，失败的原因可能是无效的名字，或角色已经在数据库中存在了。当角色的Name字段和Gender单选框改变时由`onChange`触发`updateName`和`updateGender`，同样的，当输入的名字无效或没有选择性别时触发`invalidName`和`invalidGender`。

Store

在app/stores目录新建*AddCharacterStore.js*：

```
import alt from '../alt';
import AddCharacterActions from '../actions/AddCharacterActions';

class AddCharacterStore {
  constructor() {
    this.bindActions(AddCharacterActions);
    this.name = '';
    this.gender = '';
    this.helpBlock = '';
    this.nameValidationState = '';
    this.genderValidationState = '';
  }

  onAddCharacterSuccess(successMessage) {
    this.nameValidationState = 'has-success';
    this.helpBlock = successMessage;
  }

  onAddCharacterFail(errorMessage) {
    this.nameValidationState = 'has-error';
    this.helpBlock = errorMessage;
  }

  onUpdateName(event) {
    this.name = event.target.value;
    this.nameValidationState = '';
    this.helpBlock = '';
  }

  onUpdateGender(event) {
    this.gender = event.target.value;
    this.genderValidationState = '';
  }

  onInvalidName() {
    this.nameValidationState = 'has-error';
    this.helpBlock = 'Please enter a character name.';
  }

  onInvalidGender() {
    this.genderValidationState = 'has-error';
  }
}

export default alt.createStore(AddCharacterStore);
```

`nameValidationState` 和 `genderValidationState` 指向Bootstrap提供的代表验证状态的表单控件。

`helpBlock` 是在输入框下显示的状态信息，如“Character has been added successfully”。

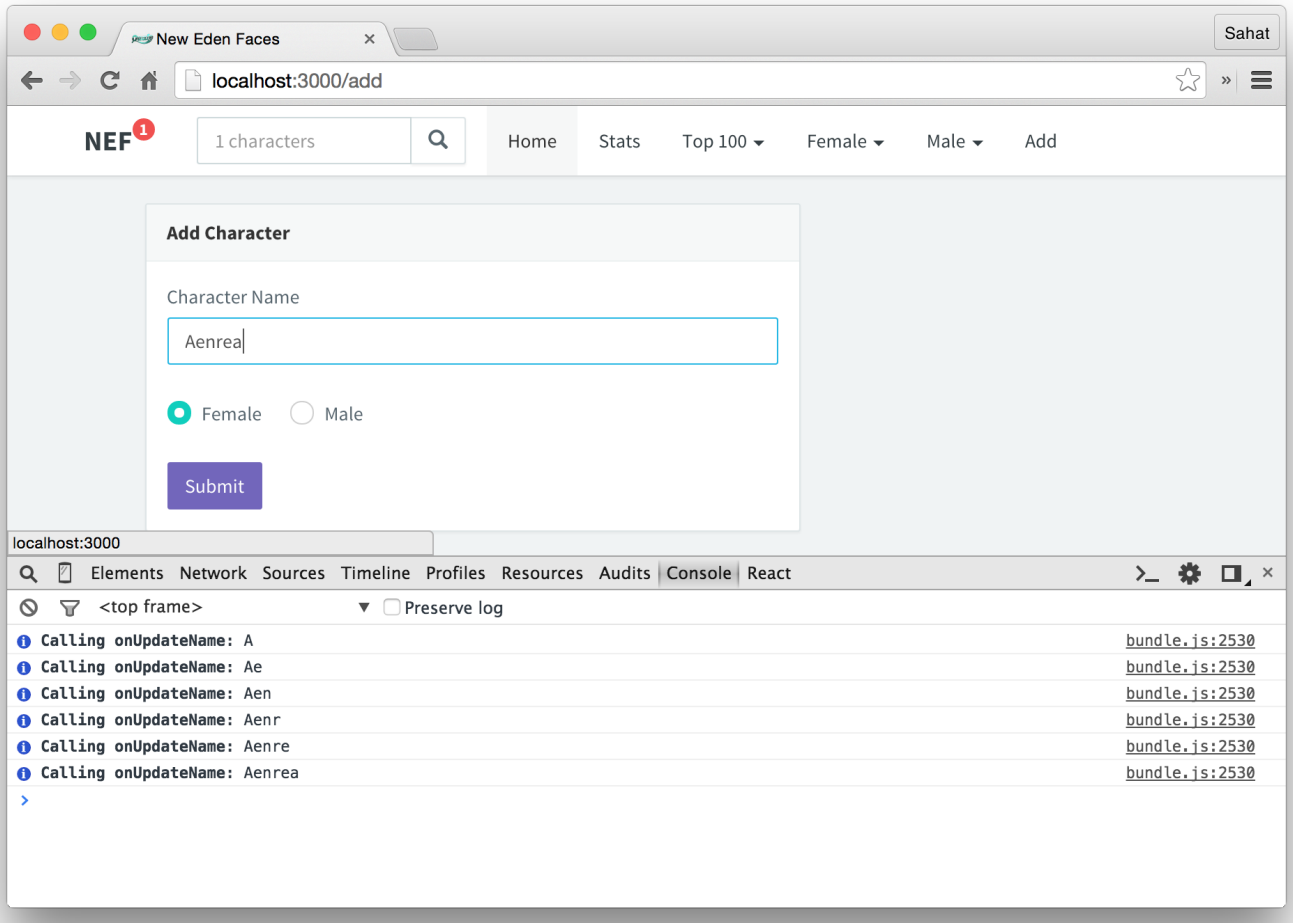
`onInvalidName` 方法当Character Name字段为空时触发。如果name在EVE中不存在，将由 `onAddCharacterFail` 输出另一个错误信息。

最后，打开routes.js并添加新的路由 `/add` ，以及 `AddCharacter` 组件方法：

```
import React from 'react';
import {Route} from 'react-router';
import App from './components/App';
import Home from './components/Home';
import AddCharacter from './components/AddCharacter';

export default (
  <Route handler={App}>
    <Route path='/' handler={Home} />
    <Route path='/add' handler={AddCharacter} />
  </Route>
);
```

这里简单总结了从你输入角色名称开始的整个流程：

1. 触发 `updateName` action，传递event对象。
2. 调用 `onUpdateName` store处理程序。
3. 使用新的名称更新状态。



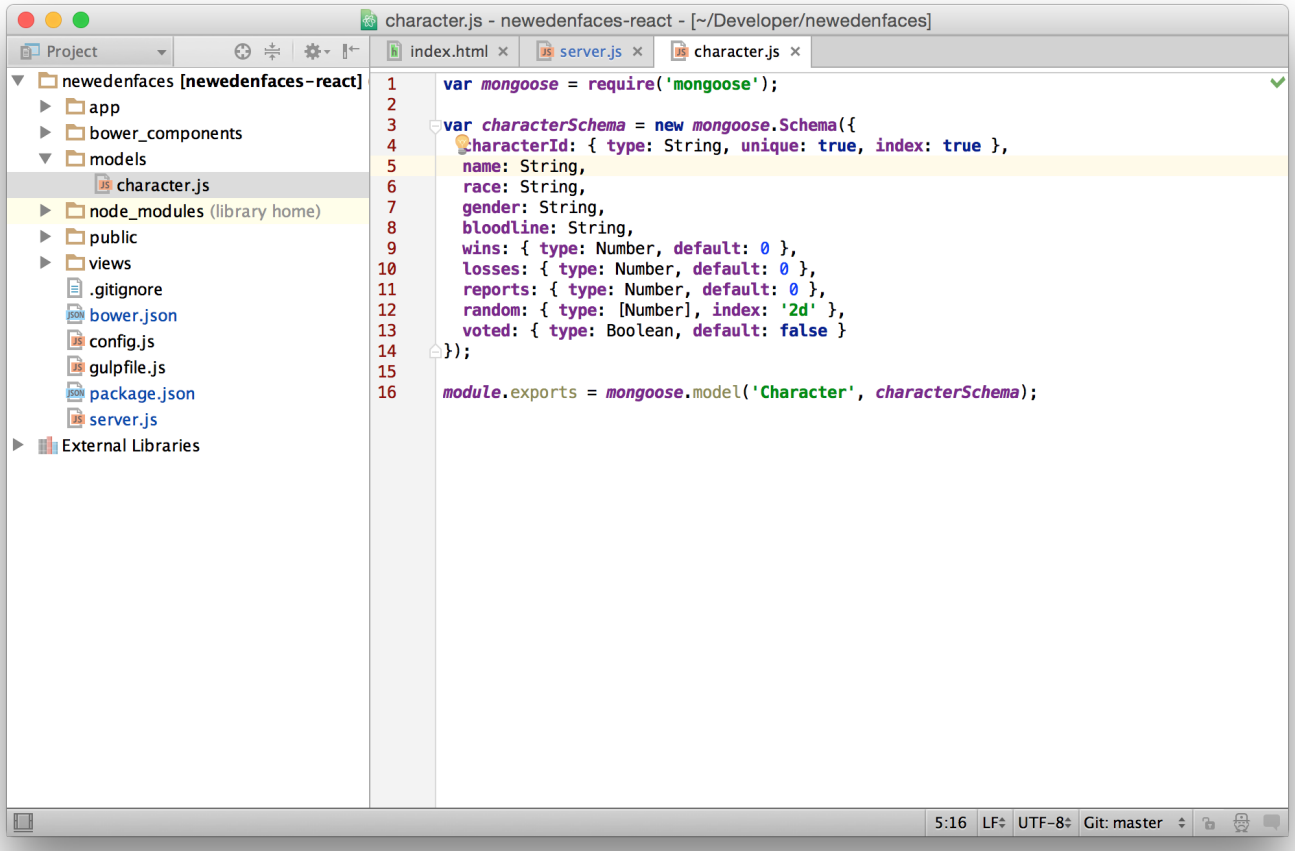
在下一节，我们将实现添加和保存新character到数据库的后端代码。

第十二步：数据库模式

第十二步：数据库模式

在根目录新建目录models，然后进入目录并新建文件*character.js*：

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

var characterSchema = new mongoose.Schema({
  characterId: { type: String, unique: true, index: true },
  name: String,
  race: String,
  gender: String,
  bloodline: String,
  wins: { type: Number, default: 0 },
  losses: { type: Number, default: 0 },
  reports: { type: Number, default: 0 },
  random: { type: [Number], index: '2d' },
  voted: { type: Boolean, default: false }
});

module.exports = mongoose.model('Character', characterSchema);
```



一个模式（schema）是你的MongoDB数据库中的数据的一个表示，你能强迫某些字段必须为特定的类型，甚至决定该字段是否必需、唯一或者仅包含指定的元素。

和抽象的模式相比，一个模型（model）是和实践更接近的对象，包含添加、删除、查询、更新数据的方法，在上面，我们创建了一个Character模型并将它暴露出来。

> 注意：为什么这个教程仍然使用MongoDB？为什么不使用MySQL、PostgreSQL、CouchDB甚至RethinkDB？这是因为对于要构建的应用来说，我并不真正关心数据库层到底是什么样的。我更关注在前端的技术栈，因为这是我最感兴趣的部分。MongoDB也许并适合所有的使用场景，但它是一个合适的通用数据库，并且过去3年来我和它相处良好。

这里大多数字段都能自我解释，不过 `random` 和 `voted` 也许需要更多解释：

- `random` – 从 `[Math.random(), 0]` 生成的包含两个数字的数组，这是一个MongoDB相关的地理标记，为了从数据库随机抓取一些角色，我们将使用 `$near` 操作符，我是从StackOverflow上Random record from MongoDB学到这个技巧。
- `voted` – 一个布尔值，为确定角色是否已被投票。如果不设置的话，人们可能会给同一角色反复刷票，现在当请求两个角色时，只有那些没有被投票的角色会被获取。即使有人直接使用API，已投票的角色也不会再次被投票。

回到server.js，在文件开头添加下面的代码：

```
var mongoose = require('mongoose');
var Character = require('./models/character');
```

为了保证一致性和系统性，我经常按照下面的顺序导入模块：

1. 核心Node.js模块——path、querystring、http
2. 第三方NPM库——mongoose、express、request
3. 应用本身文件——controllers、models、config

最后，为链接到数据库，在依赖模块和Express中间件之间添加下面的代码，它将在我们启动Express app的时候发起一个到MongoDB的连接池：

```
mongoose.connect(config.database);
mongoose.connection.on('error', function() {
  console.info('Error: Could not connect to MongoDB. Did you forget to run `mongod`?');
});
```

> 注意：我们将在config.js中设置数据库的hostname以避免硬编码。

在根目录新建另一个文件*config.js*：

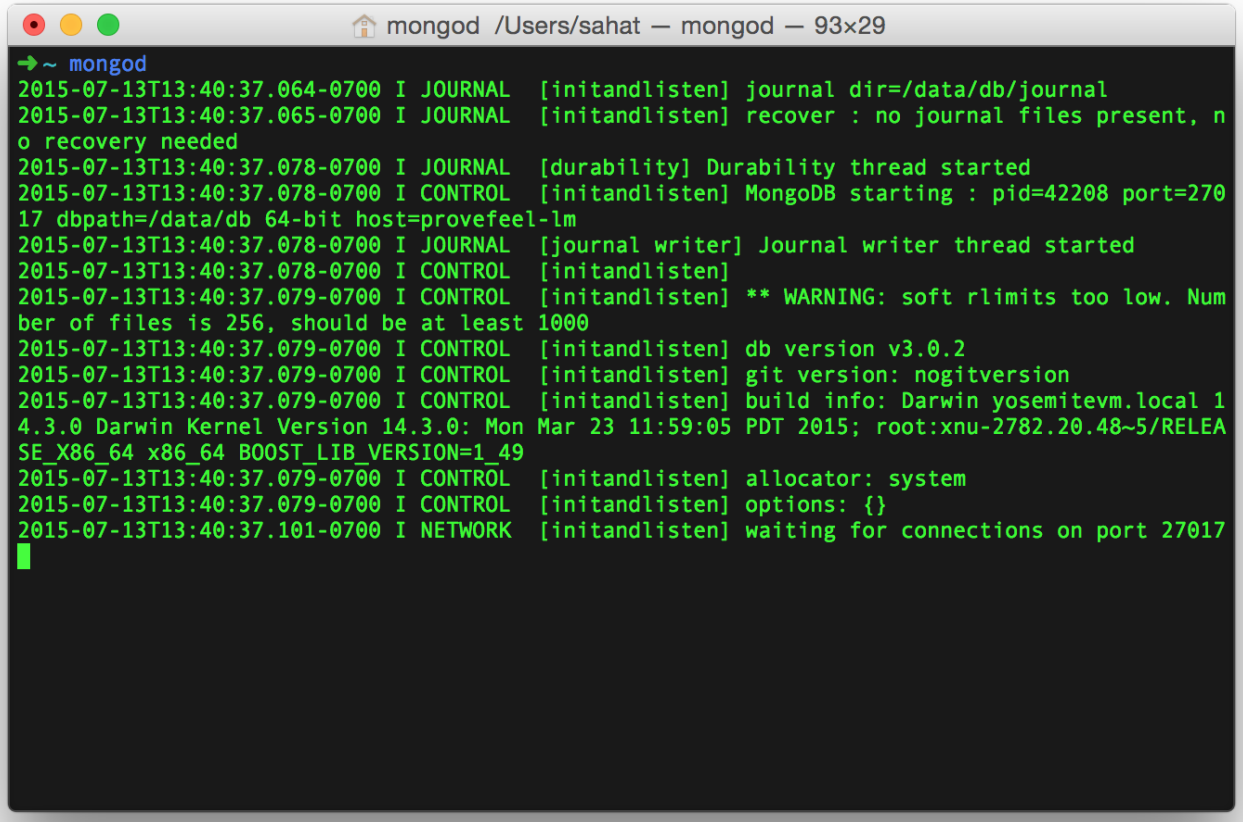
```
module.exports = {
  database: process.env.MONGO_URI || 'localhost'
};
```

它将使用一个环境变量（如果可用）或降级到localhost，这将允许我们在本地开发时使用一个hostname，而在生产环境使用另一个，同时无需修改任何代码。这种方法对于处理OAuth客户端key和secret时特别有用。

现在让我们将它导入到server.js中：

```
var config = require('./config');
```

在终端中打开一个新的标签并运行 `mongod` 。



第十三步：Express API 路由（1/2）

第十三步：Express API 路由（1/2）

在这一节我们将实现一个Express路由，以获取角色信息并存储进数据库。我们将使用EVE Online API来获取给定character name的Character ID，Race以及Bloodline。

> 注意：角色性别并不是公开数据，它需要一个API key。在我看来，让New Eden Faces变得非常棒的是它的开放生态：用户并不需要登录即可添加、查看EVE中的角色。这也是为什么我在表单里添加了性别选项让用户自己填写的缘故，虽然它的准确性的确依赖于用户的诚信。

下面的表格列出了每个路由的职责。不过，我们不会实现所有的路由，如果需要的话你可以自己实现它们。

Route	POST	GET	PUT	DELETE
/api/characters	添加新角色	获取随机两个角色	更新角色投票胜负信息	删除所有角色
/api/characters/:id	N/A	获取角色	更新角色	删除角色

在server.js文件前面添加下列依赖：

```
var async = require('async');var request = require('request');
```

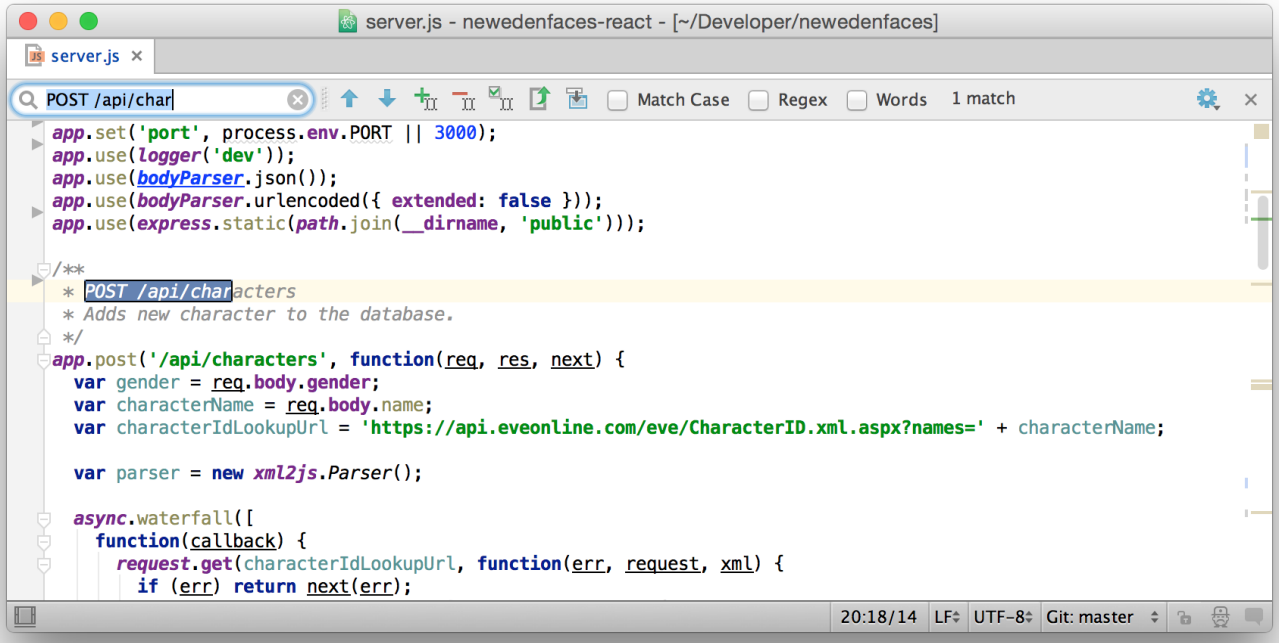
我们将使用 async.waterfall 来管理多异步操作，使用request来向EVE Online API发起HTTP请求。

将我们的第一个路由添加到Express中间件后面，在第8步创建的React中间件前面：

```
/**
 * POST /api/characters
 * Adds new character to the database.
 */
app.post('/api/characters', function(req, res, next) {
  var gender = req.body.gender;
  var characterName = req.body.name;
  var characterIdLookupUrl = 'https://api.eveonline.com/eve/CharacterID.xml.aspx?names=' + charact
erName;

  var parser = new xml2js.Parser();

  async.waterfall([
    function(callback) {
      request.get(characterIdLookupUrl, function(err, request, xml) {
        if (err) return next(err);
        parser.parseString(xml, function(err, parsedXml) {
```

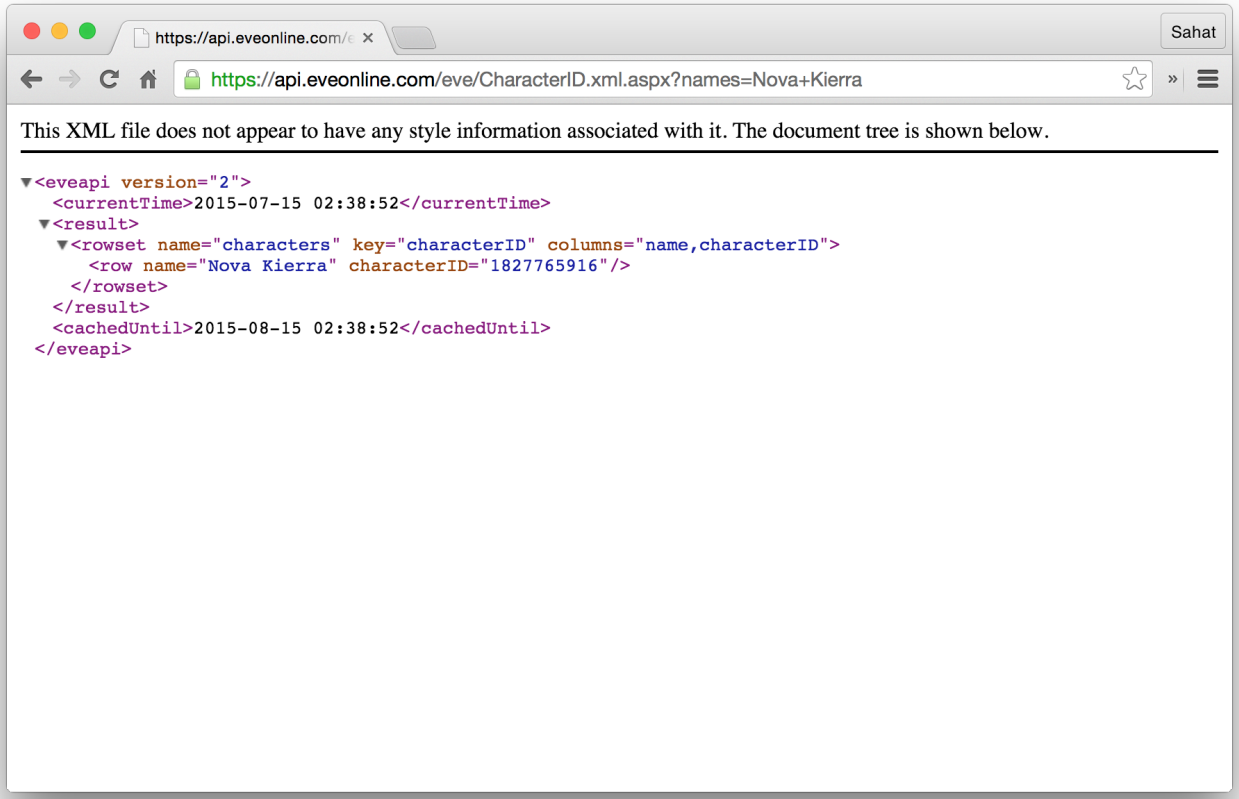
```
        if (err) return next(err);
        try {
          var characterId = parsedXml.eveapi.result[0].rowset[0].row[0].$.characterID;

          Character.findOne({ characterId: characterId }, function(err, character) {
            if (err) return next(err);

            if (character) {
              return res.status(409).send({ message: character.name + ' is already in the database.' });
            }

            callback(err, characterId);
          });
        } catch (e) {
          return res.status(400).send({ message: 'XML Parse Error' });
        }
      });
    });
  },
  function(characterId) {
    var characterInfoUrl = 'https://api.eveonline.com/eve/CharacterInfo.xml.aspx?characterID=' + char
acterId;

    request.get({ url: characterInfoUrl }, function(err, request, xml) {
      if (err) return next(err);
      parser.parseString(xml, function(err, parsedXml) {
        if (err) return res.send(err);
        try {
          var name = parsedXml.eveapi.result[0].characterName[0];
          var race = parsedXml.eveapi.result[0].race[0];
          var bloodline = parsedXml.eveapi.result[0].bloodline[0];

          var character = new Character({
            characterId: characterId,
            name: name,
            race: race,
            bloodline: bloodline,
            gender: gender,
            random: [Math.random(), 0]
          });

          character.save(function(err) {
            if (err) return next(err);
            res.send({ message: characterName + ' has been added successfully!' });
          });
        } catch (e) {
          res.status(404).send({ message: characterName + ' is not a registered citizen of New Eden.' });
        }
      });
    });
  }
]);
});
```

> 注意：我一般在路由上面写块级注释，包括完整路径和简介，这样我就能使用查找功能（Command+F）来快速寻找路由，如下图所示：

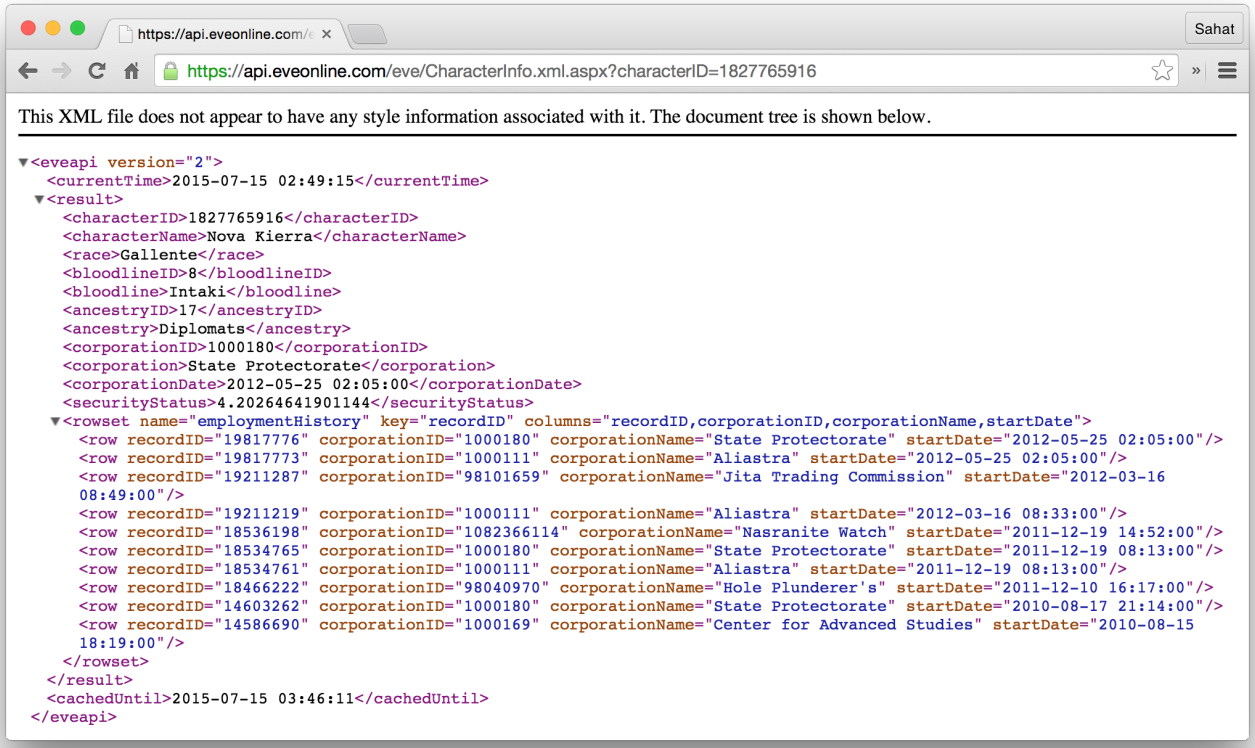


下面就来按部就班的看一下它是如何工作的：

1. 利用Character Name获取Character ID。



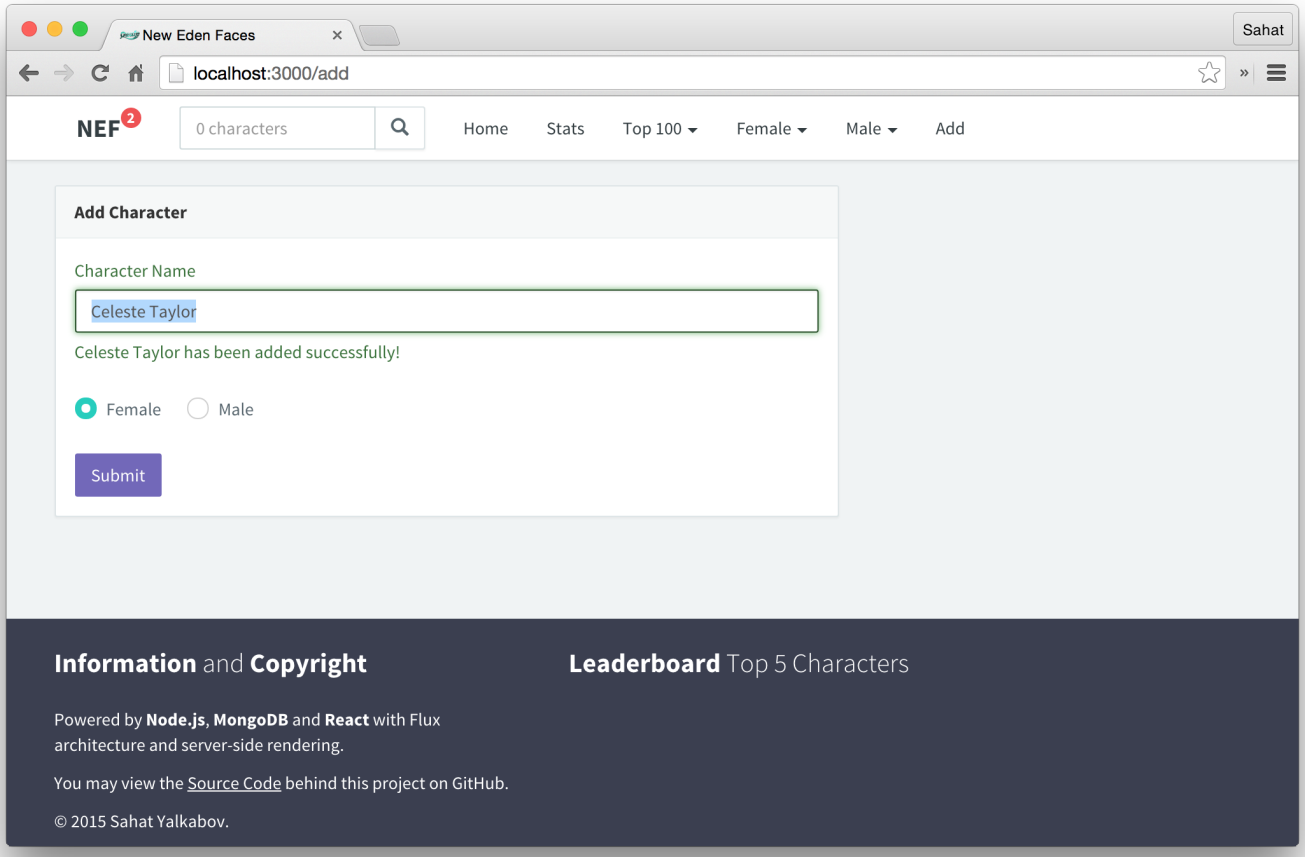
2. 解析XML响应。
3. 查询数据库看这个角色是否已经存在。
4. 将Character ID传递给 `async.waterfall` 中的下一个函数。
5. 利用Character ID获取基本的角色信息。



6. 解析XML响应。
7. 添加新角色到数据库。

在浏览器打开http://localhost:3000/add 并添加一些角色，你可以使用下面的名字：

- Daishan Auergni
- CCP Falcon
- Celeste Taylor



或者，你也可以下载这个MongoDB文件dump并将它导入到你的数据库，它包含4000以上个角色。如果你之前添加了其中的一些角色，可能会出现“duplicate key errors”错误，不用管它：

- newedenfaces.bson

下载之后使用下面的命令将它导入到数据库：

```
$ mongorestore newedenfaces.bson
```

鉴于我们还没有实现相关的API，现在你还不能看到总的角色数，我们将在下下节来实现。

下面让我们先创建Home组件，这是一个初始页面，上面会并排显示两个角色。

（*译者注：下面会先第15节再第14节，原文如此*）

第十五步：Home组件

第十五步：Home组件

这是一个稍微简单些的组件，它唯一的职责就是显示两张图片并且处理点击事件，用于告知哪个角色胜出。

组件

在components目录下新建文件*Home.js*：

```
import React from 'react';
import {Link} from 'react-router';
import HomeStore from '../stores/HomeStore'
import HomeActions from '../actions/HomeActions';
import {first, without, findWhere} from 'underscore';

class Home extends React.Component {

  constructor(props) {
    super(props);
    this.state = HomeStore.getState();
    this.onChange = this.onChange.bind(this);
  }

  componentDidMount() {
    HomeStore.listen(this.onChange);
    HomeActions.getTwoCharacters();
  }

  componentWillUnmount() {
    HomeStore.unlisten(this.onChange);
  }

  onChange(state) {
    this.setState(state);
  }

  handleClick(character) {
    var winner = character.characterId;
    var loser = first(without(this.state.characters, findWhere(this.state.characters, { characterId: winner })
)).characterId;
    HomeActions.vote(winner, loser);
  }

  render() {
    var characterNodes = this.state.characters.map((character, index) => {
      return (
        <div key={character.characterId} className={index === 0 ? 'col-xs-6 col-sm-6 col-md-5 col-md
```

```
-offset-1' : 'col-xs-6 col-sm-6 col-md-5'}>
          <div className='thumbnail fadeInUp animated'>
            <img onClick={this.handleClick.bind(this, character)} src={'http://image.eveonline.com/Charac
ter/' + character.characterId + '_512.jpg'}/>
            <div className='caption text-center'>
              <ul className='list-inline'>
                <li><strong>Race:</strong> {character.race}</li>
                <li><strong>Bloodline:</strong> {character.bloodline}</li>
              </ul>
              <h4>
                <Link to={'/characters/' + character.characterId}><strong>{character.name}</strong></Lin
k>
              </h4>
            </div>
          </div>
        </div>
      );
    });

    return (
      <div className='container'>
        <h3 className='text-center'>Click on the portrait. Select your favorite.</h3>
        <div className='row'>
          {characterNodes}
        </div>
      </div>
    );
  }
}

export default Home;
```

2015年7月27日更新：修复“Cannot read property ‘characterId’ of undefined”错误，我更新了在 handleClick() 方法里获取“失败”的Character ID。它使用 _.findWhere 在数组里查找“获胜”的角色对象，然后使用 _.without 获取不包含“获胜”角色的数组，因为数组只包含两个角色，所以这就是我们需要的，然后使用 _.first 获取数组第一个元素，也就是我们需要的对象。

鉴于角色数组只有两个元素，其实没有必要非要使用map方法不可，虽然这也能达到我们的目的。另一种做法是为 characters[0] 和 characters[1] 各自创建标记。

```
render() {
    return (
      <div className='container'>
        <h3 className='text-center'>Click on the portrait. Select your favorite.</h3>
        <div className='row'>
          <div className='col-xs-6 col-sm-6 col-md-5 col-md-offset-1'>
            <div className='thumbnail fadeInUp animated'>
              <img onClick={this.handleClick.bind(this, characters[0])} src={'http://image.eveonline.com/Ch
aracter/' + characters[0].characterId + '_512.jpg'}/>
              <div className='caption text-center'>
                <ul className='list-inline'>
                  <li><strong>Race:</strong> {characters[0].race}</li>
                  <li><strong>Bloodline:</strong> {characters[0].bloodline}</li>
                </ul>
                <h4>
                  <Link to={'/characters/' + characters[0].characterId}><strong>{characters[0].name}</stron
g></Link>
                </h4>
              </div>
            </div>
          </div>
          <div className='col-xs-6 col-sm-6 col-md-5'>
            <div className='thumbnail fadeInUp animated'>
              <img onClick={this.handleClick.bind(this, characters[1])} src={'http://image.eveonline.com/Ch
aracter/' + characters[1].characterId + '_512.jpg'}/>
              <div className='caption text-center'>
                <ul className='list-inline'>
                  <li><strong>Race:</strong> {characters[1].race}</li>
                  <li><strong>Bloodline:</strong> {characters[1].bloodline}</li>
                </ul>
                <h4>
                  <Link to={'/characters/' + characters[1].characterId}><strong>{characters[1].name}</stron
g></Link>
                </h4>
              </div>
            </div>
          </div>
        </div>
      </div>
    );
  }
```

第一张图片使用Bootstrap中的`col-md-offset-1`位移，所以两张图片是完美居中的。

注意我们在点击事件上绑定的不是`this.handleClick`，而是`this.handleClick.bind(this, character)`。简单的传递一个事件对象是不够的，它不会给我们任何有用的信息，不像文本字段、单选、复选框元素等。

MSDN文档中的解释：

```
function.bind(thisArg[, arg1[, arg2[, ...]]])
```

- thisARG(必须) – 使用this的一个对象，能在新函数内部指向当前对象的上下文
- arg1, arg2, ... (可选) – 传递给新函数的一系列参数

简单的来说，因为我们需要在 `handleClick` 方法里引用 `this.state` ，所以需要将 `this` 上下文传递进去。另外我们还传递了被点击的角色对象，而不是当前的event对象。

`handleClick` 方法里的 `character` 参数代表的是获胜的角色，因为它是被点击的那一个。因为我们仅有两个角色需要判断，所以不难分辨谁是输的那个。接下来将获胜和失败的角色Character ID传递给 `Character ID` action。

Actions

在actions目录下新建*HomeActions.js* :

```
import alt from '../alt';

class HomeActions {
  constructor() {
    this.generateActions(
      'getTwoCharactersSuccess',
      'getTwoCharactersFail',
      'voteFail'
    );
  }

  getTwoCharacters() {
    $.ajax({ url: '/api/characters' })
      .done(data => {
        this.actions.getTwoCharactersSuccess(data);
      })
      .fail(jqXhr => {
        this.actions.getTwoCharactersFail(jqXhr.responseJSON.message);
      });
  }

  vote(winner, loser) {
    $.ajax({
      type: 'PUT',
      url: '/api/characters' ,
      data: { winner: winner, loser: loser }
    })
      .done(() => {
        this.actions.getTwoCharacters();
      })
      .fail((jqXhr) => {
        this.actions.voteFail(jqXhr.responseJSON.message);
      });
  }
}

export default alt.createActions(HomeActions);
```

这里我们不需要`voteSuccess` action，因为`getTwoCharacters`已经满足了我们的需求。换句话说，在一次成功的投票之后，我们需要从数据库获取两个新的随机角色显示出来。

Store

在stores目录下新建文件*HomeStore.js*：

```
import alt from '../alt';
import HomeActions from '../actions/HomeActions';

class HomeStore {
  constructor() {
    this.bindActions(HomeActions);
    this.characters = [];
  }

  onGetTwoCharactersSuccess(data) {
    this.characters = data;
  }

  onGetTwoCharactersFail(errorMessage) {
    toastr.error(errorMessage);
  }

  onVoteFail(errorMessage) {
    toastr.error(errorMessage);
  }
}

export default alt.createStore(HomeStore);
```

下一步，让我们实现剩下的Express路由，来获取并更新Home组件中的两个角色、获得总角色数量等等。

第十四步：Express API 路由（2/2）

第十四步：Express API 路由（2/2）

让我们回到server.js。我希望现在你已经明白下面这些路由该放在哪里——在Express中间件后面和React中间件前面。

> 注意：请理解我们这里将所有的路由都放在server.js，是为了这个教程的方便。在我工作期间所构建的仪表盘项目里，所有的路由都被拆开分散到不同的文件，并放在routes目录下面，并且，所有的路由处理程序也都被打散，分成不同的文件放到controllers目录下。

让我们以获取Home组件中两个角色的路由作为开始。

GET /api/characters

```
/**
 * GET /api/characters
 * Returns 2 random characters of the same gender that have not been voted yet.
 */
app.get('/api/characters', function(req, res, next) {
  var choices = ['Female', 'Male'];
  var randomGender = _.sample(choices);

  Character.find({ random: { $near: [Math.random(), 0] } })
    .where('voted', false)
    .where('gender', randomGender)
    .limit(2)
    .exec(function(err, characters) {
      if (err) return next(err);

      if (characters.length === 2) {
        return res.send(characters);
      }

      var oppositeGender = _.first(_.without(choices, randomGender));

      Character
        .find({ random: { $near: [Math.random(), 0] } })
        .where('voted', false)
        .where('gender', oppositeGender)
        .limit(2)
        .exec(function(err, characters) {
          if (err) return next(err);

          if (characters.length === 2) {
            return res.send(characters);
          }

          Character.update({}, { $set: { voted: false } }, { multi: true }, function(err) {
            if (err) return next(err);
            res.send([]);
          });
        });
    });
});
```

别忘了在最顶部添加Underscore.js模块，因为我们需要使用它的几个函数 `_.sample()` 、 `_.first()` 和 `_.without()` 。

```
var _ = require('underscore');
```

我已经尽力让这段代码易于理解，所以你应该很清楚如何获取两个随机角色。它将随机选择Male或Female性别并查询数据库以获取两个角色，如果获得的角色少于2个，它将尝试用另一个性别进行查询。比如，如果我们有10个男性角色但其中9个已经被投票过了，只显示一个角色没有意义。如果无论是男性还是女性角色查询返回都不足两个角色，说明我们已经耗尽了所有未投票的角色，应该重置投票计数，通

过设置所有角色的 voted:false 即可办到。

PUT /api/characters

这个路由和前一个相关，它会分别更新获胜的 wins 字段和失败角色的 losses 字段。

```
/**
 * PUT /api/characters
 * Update winning and losing count for both characters.
 */
app.put('/api/characters', function(req, res, next) {
  var winner = req.body.winner;
  var loser = req.body.loser;

  if (!winner || !loser) {
    return res.status(400).send({ message: 'Voting requires two characters.' });
  }

  if (winner === loser) {
    return res.status(400).send({ message: 'Cannot vote for and against the same character.' });
  }

  async.parallel([
      function(callback) {
        Character.findOne({ characterId: winner }, function(err, winner) {
          callback(err, winner);
        });
      },
      function(callback) {
        Character.findOne({ characterId: loser }, function(err, loser) {
          callback(err, loser);
        });
      }
    ],
    function(err, results) {
      if (err) return next(err);

      var winner = results[0];
      var loser = results[1];

      if (!winner || !loser) {
        return res.status(404).send({ message: 'One of the characters no longer exists.' });
      }

      if (winner.voted || loser.voted) {
        return res.status(200).end();
      }

      async.parallel([
        function(callback) {
          winner.wins++;
          winner.voted = true;
          winner.random = [Math.random(), 0];
```

```
          winner.save(function(err) {
            callback(err);
          });
        },
        function(callback) {
          loser.losses++;
          loser.voted = true;
          loser.random = [Math.random(), 0];
          loser.save(function(err) {
            callback(err);
          });
        }
      ], function(err) {
        if (err) return next(err);
        res.status(200).end();
      });
    });
});
```

这里我们使用 async.parallel 来同时进行两个数据库查询，因为这两个查询并不相互依赖。不过，因为我们有两个独立的MongoDB文档，还要进行两个独立的异步操作，因此我们还需要另一个 async.parallel 。一般来说，我们仅在两个角色都完成更新并没有错误后给出一个success的响应。

GET /api/characters/count

MOngoDB有一个内建的 count() 方法，可以返回所匹配的查询结果的数量。

```
/**
 * GET /api/characters/count
 * Returns the total number of characters.
 */
app.get('/api/characters/count', function(req, res, next) {
  Character.count({}, function(err, count) {
    if (err) return next(err);
    res.send({ count: count });
  });
});
```

> 注意：从这个返回总数量的一次性路由上，你可能注意到我们开始与RESTful API设计模式背道而驰。很不幸这就是现实。我还没有在一个能完美实现RESTful API的项目中工作过，你可以参看Apigee写的这篇文章来进一步了解为什么会这样。

GET /api/characters/search

我上次检查时MongoDB还不支持大小写不敏感的查询，所以这里我们需要使用正则表达式，不过还好MongoDB提供了 $regex 操作符。

```
/**
 * GET /api/characters/search
 * Looks up a character by name. (case-insensitive)
 */
app.get('/api/characters/search', function(req, res, next) {
  var characterName = new RegExp(req.query.name, 'i');

  Character.findOne({ name: characterName }, function(err, character) {
    if (err) return next(err);

    if (!character) {
      return res.status(404).send({ message: 'Character not found.' });
    }

    res.send(character);
  });
});
```

GET /api/characters/:id

这个路由是供角色资料页面使用的（我们将在下一节创建角色组件），教程最开始的图片就是这个页面。

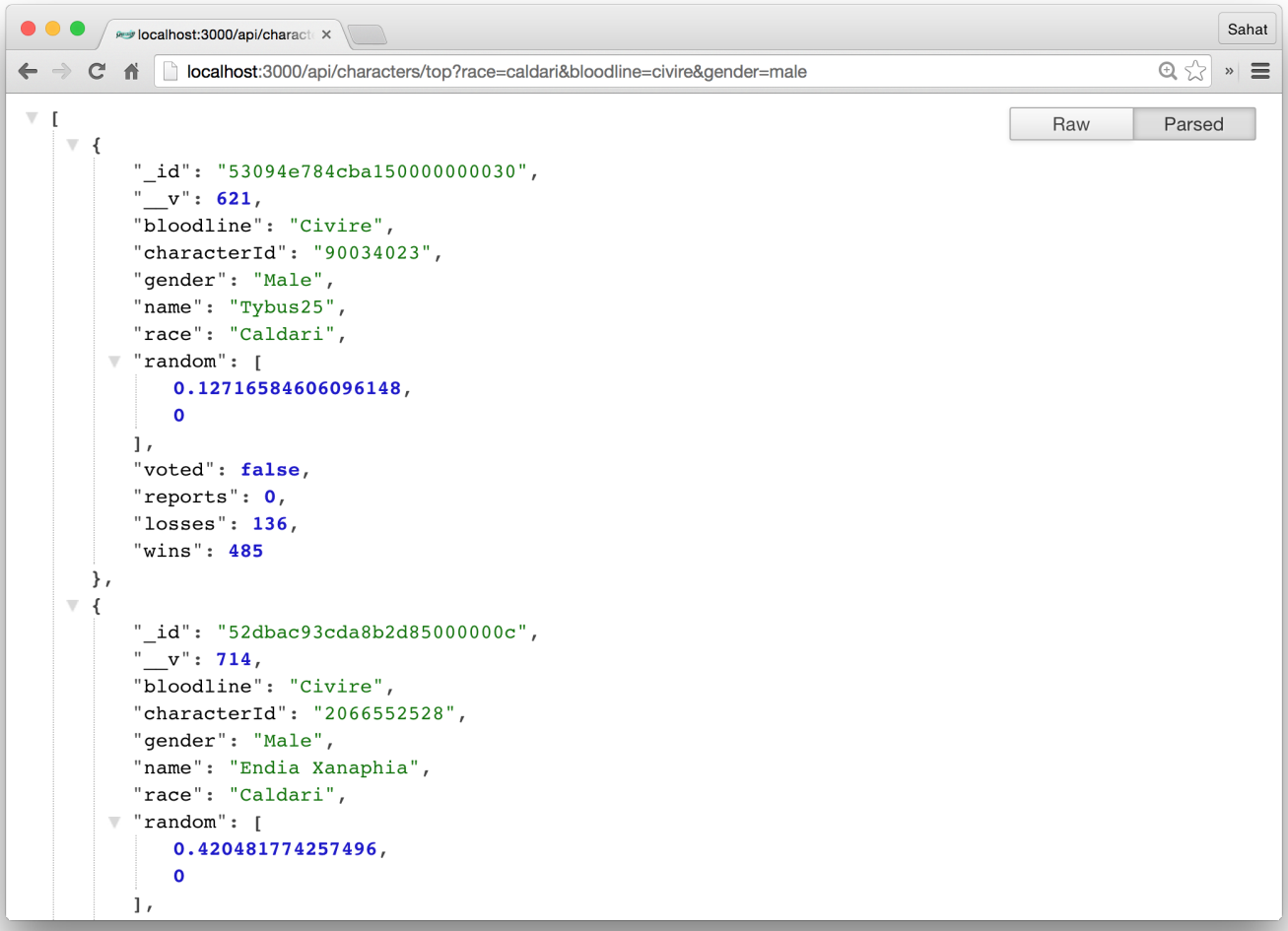
```
/**
 * GET /api/characters/:id
 * Returns detailed character information.
 */
app.get('/api/characters/:id', function(req, res, next) {
  var id = req.params.id;

  Character.findOne({ characterId: id }, function(err, character) {
    if (err) return next(err);

    if (!character) {
      return res.status(404).send({ message: 'Character not found.' });
    }

    res.send(character);
  });
});
```

当我开始构建这个项目时，我大概有7-9个几乎相同的路由来检索Top 100的角色。在经过一些代码重构后我仅留下了下面这一个：

```
/**
 * GET /api/characters/top
 * Return 100 highest ranked characters. Filter by gender, race and bloodline.
 */
app.get('/api/characters/top', function(req, res, next) {
  var params = req.query;
  var conditions = {};

  _.each(params, function(value, key) {
    conditions[key] = new RegExp('^' + value + '$', 'i');
  });

  Character
    .find(conditions)
    .sort('-wins') // Sort in descending order (highest wins on top)
    .limit(100)
    .exec(function(err, characters) {
      if (err) return next(err);

      // Sort by winning percentage
      characters.sort(function(a, b) {
        if (a.wins / (a.wins + a.losses) < b.wins / (b.wins + b.losses)) { return 1; }         if (a.wins / (a.wins +
a.losses) > b.wins / (b.wins + b.losses)) { return -1; }
        return 0;
      });

      res.send(characters);
    });
});
```

比如，如果我们对男性、种族为Caldari、血统为Civire的Top 100角色感兴趣，你可以构造这样的URL路径：

```
GET /api/characters/top?race=caldari&bloodline=civire&gender=male
```



如果你还不清楚如何构造 conditions 对象，这段经过注释的代码应该可以解释：

```
// Query params object
req.query = {
  race: 'caldari',
  bloodline: 'civire',
  gender: 'male'
};

var params = req.query;
var conditions = {};

// This each loop is equivalent...
_.each(params, function(value, key) {
  conditions[key] = new RegExp('^' + value + '$', 'i');
});

// To this code
conditions.race = new RegExp('^' + params.race + '$', 'i'); // /caldari$/i
conditions.bloodline = new RegExp('^' + params.bloodline + '$', 'i'); // /civire$/i
conditions.gender = new RegExp('^' + params.gender + '$', 'i'); // /male$/i

// Which ultimately becomes this...
Character
    .find({ race: /caldari$/i, bloodline: /civire$/i, gender: /male$/i })
```

在我们取回获胜数最多的角色后，我们会对胜率进行一个排序，不让最老的角色始终显示在前面。

GET /api/characters/shame

和前一个路由差不多，这个路由会取回失败最多的100个角色：

```
/**
 * GET /api/characters/shame
 * Returns 100 lowest ranked characters.
 */
app.get('/api/characters/shame', function(req, res, next) {
  Character
    .find()
    .sort('-losses')
    .limit(100)
    .exec(function(err, characters) {
      if (err) return next(err);
      res.send(characters);
    });
});
```

POST /api/report

有些角色没有一个有效的avatar（一般是灰色轮廓），另有些角色的avatar是漆黑一片，它们在一开始就不应该添加到数据库中。但因为任何人都能添加任何角色，因此有些时候你需要从数据库移除一些异常角

色。这里设置当一个角色被访问者举报4次后将被删除。

```
/**
 * POST /api/report
 * Reports a character. Character is removed after 4 reports.
 */
app.post('/api/report', function(req, res, next) {
  var characterId = req.body.characterId;

  Character.findOne({ characterId: characterId }, function(err, character) {
    if (err) return next(err);

    if (!character) {
      return res.status(404).send({ message: 'Character not found.' });
    }

    character.reports++;

    if (character.reports > 4) {
      character.remove();
      return res.send({ message: character.name + ' has been deleted.' });
    }

    character.save(function(err) {
      if (err) return next(err);
      res.send({ message: character.name + ' has been reported.' });
    });
  });
});
```

GET /api/stats

最后，为角色的统计创建一个路由。是的，下面的代码可以用 async.each 或 promises 来简化，不过记住，我在两年前开始创建New Eden Faces时对这些方案还不熟悉，到现在绝大部分的后端代码没怎么动过。不过即使这样，这些代码还是足够鲁棒，最少它很明确并且易读。

```
/**
 * GET /api/stats
 * Returns characters statistics.
 */
app.get('/api/stats', function(req, res, next) {
  async.parallel([
      function(callback) {
        Character.count({}, function(err, count) {
          callback(err, count);
        });
      },
      function(callback) {
        Character.count({ race: 'Amarr' }, function(err, amarrCount) {
          callback(err, amarrCount);
        });
```

```
    },
    function(callback) {
      Character.count({ race: 'Caldari' }, function(err, caldariCount) {
        callback(err, caldariCount);
      });
    },
    function(callback) {
      Character.count({ race: 'Gallente' }, function(err, gallenteCount) {
        callback(err, gallenteCount);
      });
    },
    function(callback) {
      Character.count({ race: 'Minmatar' }, function(err, minmatarCount) {
        callback(err, minmatarCount);
      });
    },
    function(callback) {
      Character.count({ gender: 'Male' }, function(err, maleCount) {
        callback(err, maleCount);
      });
    },
    function(callback) {
      Character.count({ gender: 'Female' }, function(err, femaleCount) {
        callback(err, femaleCount);
      });
    },
    function(callback) {
      Character.aggregate({ $group: { _id: null, total: { $sum: '$wins' } } }, function(err, totalVotes) {
          var total = totalVotes.length ? totalVotes[0].total : 0;
          callback(err, total);
        }
      );
    },
    function(callback) {
      Character
        .find()
        .sort('-wins')
        .limit(100)
        .select('race')
        .exec(function(err, characters) {
          if (err) return next(err);

          var raceCount = _.countBy(characters, function(character) { return character.race; });
          var max = _.max(raceCount, function(race) { return race });
          var inverted = _.invert(raceCount);
          var topRace = inverted[max];
          var topCount = raceCount[topRace];

          callback(err, { race: topRace, count: topCount });
        });
    },
    function(callback) {
      Character
        .find()
        .sort('-wins')
```

```
          .limit(100)
          .select('bloodline')
          .exec(function(err, characters) {
            if (err) return next(err);

            var bloodlineCount = _.countBy(characters, function(character) { return character.bloodline; });
            var max = _.max(bloodlineCount, function(bloodline) { return bloodline });
            var inverted = _.invert(bloodlineCount);
            var topBloodline = inverted[max];
            var topCount = bloodlineCount[topBloodline];

            callback(err, { bloodline: topBloodline, count: topCount });
          });
      }
    ],
    function(err, results) {
      if (err) return next(err);

      res.send({
        totalCount: results[0],
        amarrCount: results[1],
        caldariCount: results[2],
        gallenteCount: results[3],
        minmatarCount: results[4],
        maleCount: results[5],
        femaleCount: results[6],
        totalVotes: results[7],
        leadingRace: results[8],
        leadingBloodline: results[9]
      });
    });
});
```

最后使用 `aggregate()` 方法的操作比较令人费解。必须承认，到这一步我也曾去寻求过帮助。在MongoDB里，聚合（aggregation）操作处理数据记录并且返回计算后的结果。在这里它通过将所有 `wins` 数量相加，来计算所有投票的总数。因为投票是一个零和游戏，获胜总数总是和失败总数相同，所以我们同样也可以使用 `losses` 数量来计算。

项目到这里基本就完成了。在教程的最后我还将给项目添加更多特性，给它稍稍扩展一下。

第十六步：角色(资料)组件

第十六步：角色(资料)组件

在这一节里我们将为角色创建资料页面。它和其它组件有些不同，其不同之处在于：

1. 它有一个覆盖全页面的背景图片。
2. 从一个角色页面导航至另一个角色页面并不会卸载组件，因此，在 `componentDidMount` 内部的 `getCharacter` action仅会被调用一次，比如它更新了URL但并不获取新数据。

Component

在app/components目录新建文件*Character.js*：

```
import React from 'react';
import CharacterStore from '../stores/CharacterStore';
import CharacterActions from '../actions/CharacterActions'

class Character extends React.Component {
  constructor(props) {
    super(props);
    this.state = CharacterStore.getState();
    this.onChange = this.onChange.bind(this);
  }

  componentDidMount() {
    CharacterStore.listen(this.onChange);
    CharacterActions.getCharacter(this.props.params.id);

    $('.magnific-popup').magnificPopup({
      type: 'image',
      mainClass: 'mfp-zoom-in',
      closeOnContentClick: true,
      midClick: true,
      zoom: {
        enabled: true,
        duration: 300
      }
    });
  }

  componentWillUnmount() {
    CharacterStore.unlisten(this.onChange);
    $(document.body).removeClass();
  }

  componentDidUpdate(prevProps) {
    // Fetch new charachter data when URL path changes
    if (prevProps.params.id !== this.props.params.id) {
```

```
      CharacterActions.getCharacter(this.props.params.id);
    }
  }

  onChange(state) {
    this.setState(state);
  }

  render() {
    return (
      <div className='container'>
        <div className='profile-img'>
          <a className='magnific-popup' href={'https://image.eveonline.com/Character/' + this.state.ch
aracterId + '_1024.jpg'}>
            <img src={'https://image.eveonline.com/Character/' + this.state.characterId + '_256.jpg'} />
          </a>
        </div>
        <div className='profile-info clearfix'>
          <h2><strong>{this.state.name}</strong></h2>
          <h4 className='lead'>Race: <strong>{this.state.race}</strong></h4>
          <h4 className='lead'>Bloodline: <strong>{this.state.bloodline}</strong></h4>
          <h4 className='lead'>Gender: <strong>{this.state.gender}</strong></h4>
          <button className='btn btn-transparent'
                  onClick={CharacterActions.report.bind(this, this.state.characterId)}
                  disabled={this.state.isReported}>
            {this.state.isReported ? 'Reported' : 'Report Character'}
          </button>
        </div>
        <div className='profile-stats clearfix'>
          <ul>
            <li><span className='stats-number'>{this.state.winLossRatio}</span>Winning Percentage<
/li>
            <li><span className='stats-number'>{this.state.wins}</span> Wins</li>
            <li><span className='stats-number'>{this.state.losses}</span> Losses</li>
          </ul>
        </div>
      </div>
    );
  }
}

Character.contextTypes = {
  router: React.PropTypes.func.isRequired
};

export default Character;
```

在`componentDidMount`里我们将当前Character ID（从URL获取）传递给`getCharacter` action并且初始化Magnific Popup lightbox插件。

> 注意：我从未成功使用`ref="magnificPopup"`进行插件初始化，这也是我采用代码中方法的原因。这也许不是最好的办法，但它能正常工作。

另外你需要注意，角色组件包含一个全页面背景图片，并且在 componentWillUnmount 时移除，因为其它组件不包含这样的背景图。它又是什么时候添加上去的呢？在store中当成功获取到角色数据时。

最后值得一提的是在 componentDidUpdate 中发生了什么。如果我们从一个角色页面跳转至另一个角色页面，我们仍然处于角色组件内，它不会被卸载掉。而因为它没有被卸载， componentDidMount 不会去获取新角色数据，所以我们需要在 componentDidUpdate 中获取新数据，只要我们仍然处于同一个角色组件且URL是不同的，比如从/characters/1807823526跳转至/characters/467078888。 componentDidUpdate 在组件的生命周期中，每一次组件状态变化后都会触发。

Actions

在app/actions目录新建文件*CharacterActions.js* :

```
import alt from '../alt';

class CharacterActions {
  constructor() {
    this.generateActions(
      'reportSuccess',
      'reportFail',
      'getCharacterSuccess',
      'getCharacterFail'
    );
  }

  getCharacter(characterId) {
    $.ajax({ url: '/api/characters/' + characterId })
      .done((data) => {
        this.actions.getCharacterSuccess(data);
      })
      .fail((jqXhr) => {
        this.actions.getCharacterFail(jqXhr);
      });
  }

  report(characterId) {
    $.ajax({
      type: 'POST',
      url: '/api/report',
      data: { characterId: characterId }
    })
      .done(() => {
        this.actions.reportSuccess();
      })
      .fail((jqXhr) => {
        this.actions.reportFail(jqXhr);
      });
  }
}

export default alt.createActions(CharacterActions);
```

Store

在app/store目录新建文件*CharacterStore.js* :

```
import {assign, contains} from 'underscore';
import alt from '../alt';
import CharacterActions from '../actions/CharacterActions';

class CharacterStore {
  constructor() {
    this.bindActions(CharacterActions);
    this.characterId = 0;
    this.name = 'TBD';
    this.race = 'TBD';
    this.bloodline = 'TBD';
    this.gender = 'TBD';
    this.wins = 0;
    this.losses = 0;
    this.winLossRatio = 0;
    this.isReported = false;
  }

  onGetCharacterSuccess(data) {
    assign(this, data);
    $(document.body).attr('class', 'profile ' + this.race.toLowerCase());
    let localData = localStorage.getItem('NEF') ? JSON.parse(localStorage.getItem('NEF')) : {};
    let reports = localData.reports || [];
    this.isReported = contains(reports, this.characterId);
    // If is NaN (from division by zero) then set it to "0"
    this.winLossRatio = ((this.wins / (this.wins + this.losses) * 100) || 0).toFixed(1);
  }

  onGetCharacterFail(jqXhr) {
    toastr.error(jqXhr.responseJSON.message);
  }

  onReportSuccess() {
    this.isReported = true;
    let localData = localStorage.getItem('NEF') ? JSON.parse(localStorage.getItem('NEF')) : {};
    localData.reports = localData.reports || [];
    localData.reports.push(this.characterId);
    localStorage.setItem('NEF', JSON.stringify(localData));
    toastr.warning('Character has been reported.');
  }

  onReportFail(jqXhr) {
    toastr.error(jqXhr.responseJSON.message);
  }
}

export default alt.createStore(CharacterStore);
```

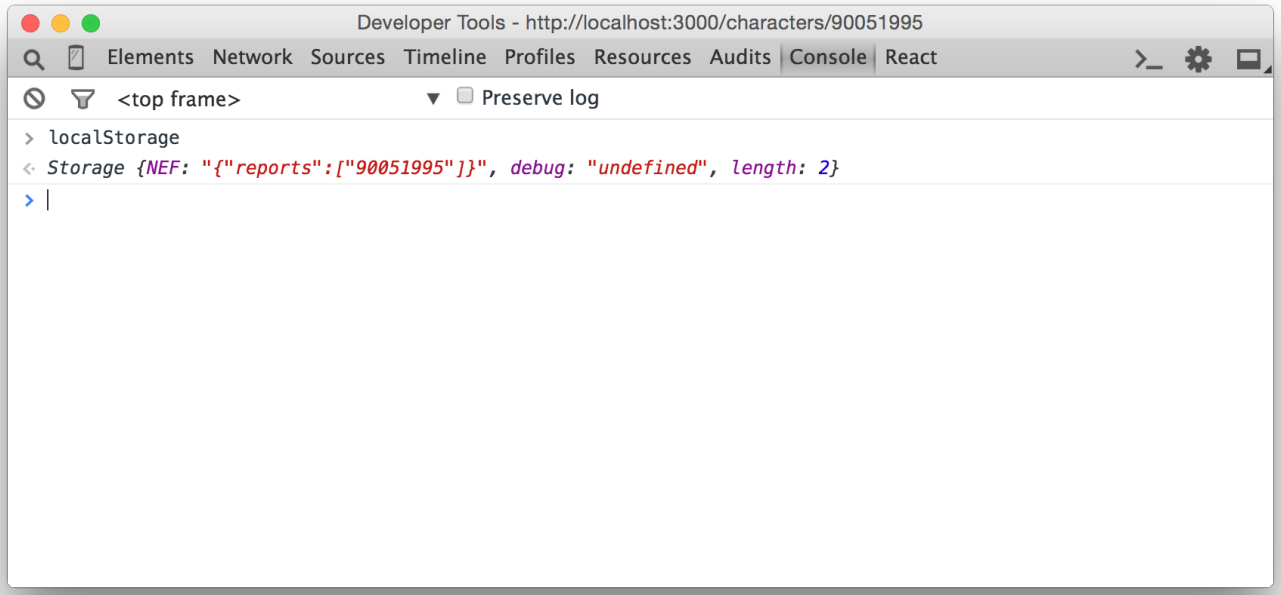
这里我们使用了Underscore的两个辅助函数 `assign` 和 `contains` ，来合并两个对象并检查数组是否包含指定值。

> 注意：在我写本教程时Babel.js还不支持 `Object.assign` 方法，并且我觉得 `contains` 比相同功能的

> Array.indexOf() > -1 可读性要好得多。

就像我在前面解释过的，这个组件在外观上和其它组件有显著的不同。添加 profile 类到 <body> 改变了页面整个外观和感觉，至于第二个CSS类，可能是 caldari 、 gallente 、 minmatar 、 amarr 其中的一个，将决定使用哪一个背景图片。我一般会避免与组件 render() 之外的DOM直接交互，但这里为简单起见还是允许例外一次。最后，在 onGetCharacterSuccess 方法里我们需要检查角色在之前是否已经被该用户举报过。如果举报过，举报按钮将设置为disabled。*因为这个限制很容易被绕过，所以如果你想严格对待举报的话，你可以在服务端执行一个IP检查。*

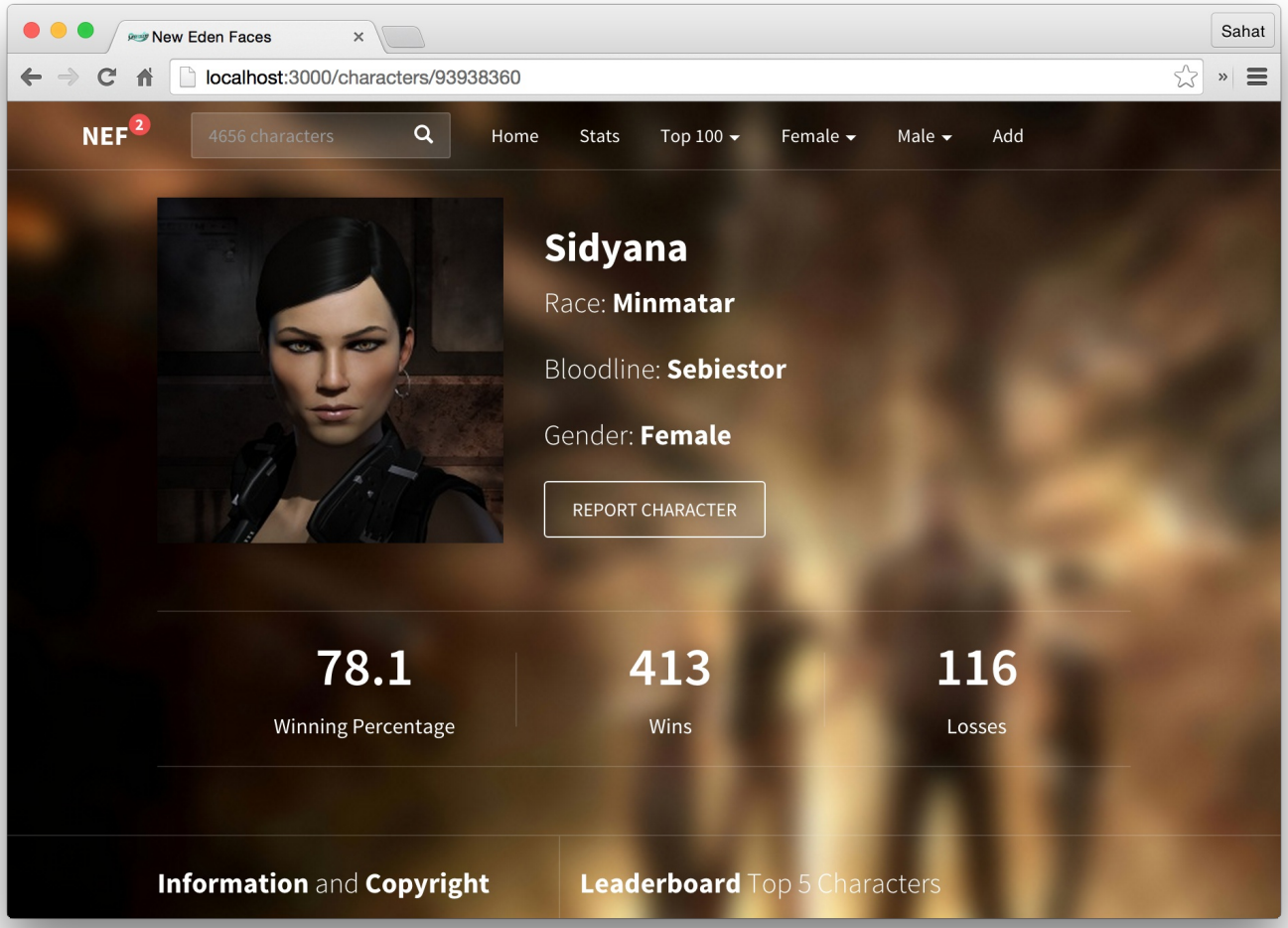
如果角色是第一次被举报，相关信息会被存储到Local Storage里，因为我们不能在Local Storage存储对象，所以我们需要先用 JSON.stringify() 转换一下。



最后，打开routes.js并且为 /characters/:id 添加一个新路由。这个路由使用了动态区段 id 来匹配任意有效Character ID，同时，别忘了导入Character组件。

```
import React from 'react';
import {Route} from 'react-router';
import App from './components/App';
import Home from './components/Home';
import AddCharacter from './components/AddCharacter';
import Character from './components/Character';

export default (
  <Route handler={App}>
    <Route path='/' handler={Home} />
    <Route path='/add' handler={AddCharacter} />
    <Route path='/characters/:id' handler={Character} />
  </Route>
);
```

刷新浏览器，点击一个角色，现在你应该能看到新的角色资料页面。

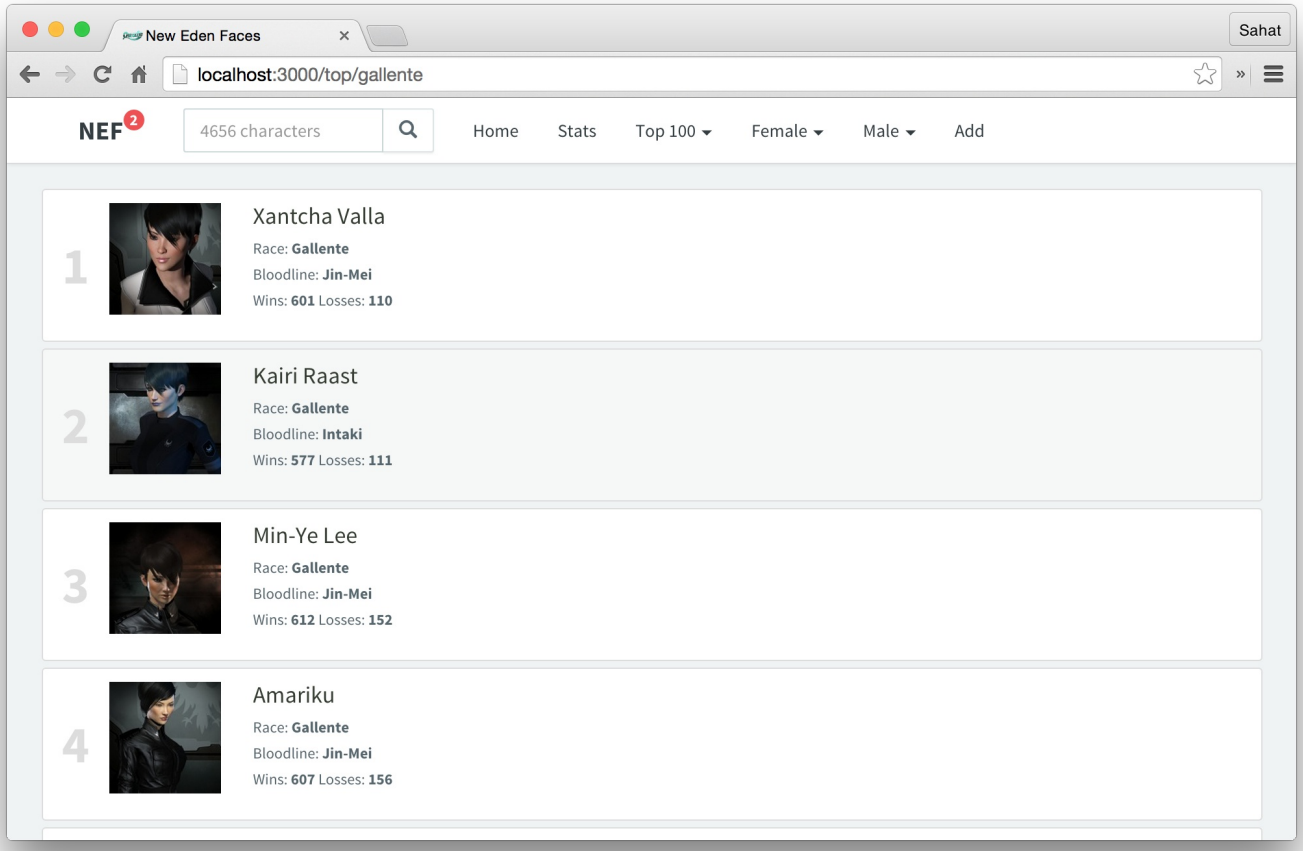


下一节我们将介绍如何为Top100角色构建CharacterList组件，并且能够根据性别、种族、血统进行过滤。耻辱墙(Hall of Shame)同样也是该组件的一部分。

第十七步：Top 100 组件

第十七步：Top 100 组件

这个组件使用Bootstrap的Media控件作为主要的界面，下面是它看起来的样子：



Component

在app/components目录新建文件*CharacterList.js*：

```
import React from 'react';
import {Link} from 'react-router';
import {isEqual} from 'underscore';
import CharacterListStore from '../stores/CharacterListStore';
import CharacterListActions from '../actions/CharacterListActions';

class CharacterList extends React.Component {
  constructor(props) {
    super(props);
    this.state = CharacterListStore.getState();
    this.onChange = this.onChange.bind(this);
  }

  componentDidMount() {
```

```
  componentDidMount() {
    CharacterListStore.listen(this.onChange);
    CharacterListActions.getCharacters(this.props.params);
  }

  componentWillUnmount() {
    CharacterListStore.unlisten(this.onChange);
  }

  componentDidUpdate(prevProps) {
    if (!isEqual(prevProps.params, this.props.params)) {
      CharacterListActions.getCharacters(this.props.params);
    }
  }

  onChange(state) {
    this.setState(state);
  }

  render() {
    let charactersList = this.state.characters.map((character, index) => {
      return (
        <div key={character.characterId} className='list-group-item animated fadeIn'>
          <div className='media'>
            <span className='position pull-left'>{index + 1}</span>
            <div className='pull-left thumb-lg'>
              <Link to={'/characters/' + character.characterId}>
                <img className='media-object' src={'http://image.eveonline.com/Character/' + character.c
haracterId + '_128.jpg'} />
              </Link>
            </div>
            <div className='media-body'>
              <h4 className='media-heading'>
                <Link to={'/characters/' + character.characterId}>{character.name}</Link>
              </h4>
              <small>Race: <strong>{character.race}</strong></small>
              <br />
              <small>Bloodline: <strong>{character.bloodline}</strong></small>
              <br />
              <small>Wins: <strong>{character.wins}</strong> Losses: <strong>{character.losses}</stron
g></small>
            </div>
          </div>
        </div>
      );
    });

    return (
      <div className='container'>
        <div className='list-group'>
          {charactersList}
        </div>
      </div>
    );
  }
```

```
}

CharacterList.contextTypes = {
  router: React.PropTypes.func.isRequired
};

export default CharacterList;
```

鉴于角色数组已经按照胜率进行了排序，我们可以使用`index + 1`（从1到100）来作为数组下标直接输出角色。

Actions

在app/actions目录新建*CharacterListActions.js*：

```
import alt from '../alt';

class CharacterListActions {
  constructor() {
    this.generateActions(
      'getCharactersSuccess',
      'getCharactersFail'
    );
  }

  getCharacters(payload) {
    let url = '/api/characters/top';
    let params = {
      race: payload.race,
      bloodline: payload.bloodline
    };

    if (payload.category === 'female') {
      params.gender = 'female';
    } else if (payload.category === 'male') {
      params.gender = 'male';
    }

    if (payload.category === 'shame') {
      url = '/api/characters/shame';
    }

    $.ajax({ url: url, data: params })
      .done((data) => {
        this.actions.getCharactersSuccess(data);
      })
      .fail((jqXhr) => {
        this.actions.getCharactersFail(jqXhr);
      });
  }
}

export default alt.createActions(CharacterListActions);
```

这里的 payload 包含React Router参数，这些参数我们将在routes.js中指定。

```
<Route path=':category' handler={CharacterList}>
  <Route path=':race' handler={CharacterList}>
    <Route path=':bloodline' handler={CharacterList} />
  </Route>
</Route>
```

比如，如果我们访问http://localhost:3000/female/gallente/intaki ，则 payload 对象将包括下列数据：

```
{
  category: 'female',
  race: 'gallente',
  bloodline: 'intaki'
}
```

Store

在app/store目录下新建文件*CharacterListStore.js* :

```
import alt from '../alt';
import CharacterListActions from '../actions/CharacterListActions';

class CharacterListStore {
  constructor() {
    this.bindActions(CharacterListActions);
    this.characters = [];
  }

  onGetCharactersSuccess(data) {
    this.characters = data;
  }

  onGetCharactersFail(jqXhr) {
    toastr.error(jqXhr.responseJSON.message);
  }
}

export default alt.createStore(CharacterListStore);/pre>
```

打开route.js并添加下列路由。所有内嵌路由都使用动态区段，所以不用重复输入。确保它们在路由的最后面，否则:category将会覆盖其它路由如/stats、/add和/shame。不要忘了导入CharacterList组件：

```
import React from 'react';
import {Route} from 'react-router';
import App from './components/App';
import Home from './components/Home';
import AddCharacter from './components/AddCharacter';
import Character from './components/Character';
import CharacterList from './components/CharacterList';

export default (
  <Route handler={App}>
    <Route path='/' handler={Home} />
    <Route path='/add' handler={AddCharacter} />
    <Route path='/characters/:id' handler={Character} />
    <Route path='/shame' handler={CharacterList} />
    <Route path=':category' handler={CharacterList}>
      <Route path=':race' handler={CharacterList}>
        <Route path=':bloodline' handler={CharacterList} />
      </Route>
    </Route>
  </Route>
);
```

下面是所有动态区段可以取的值：

- `:category` — male, female, top.
- `:race` — caldari, gallente, minmatar, amarr.
- `:bloodline` — civire, deteis, achura, intaki, gallente, jin-mei, amarr, ni-kunni, khanid, brutor, sebiestor, vherokior.

可以看到，如果我们使用硬编码的话，将如此多的路由包含进去将使route.js变得很长很长。

第十八步：Stats组件

第十八步：Stats组件

我们最后一个组件非常简单，仅仅是一个包含一般统计的表格，比如角色总数，按种族、性别、总投票等等统计出来的数据。这些代码甚至都无需解释，因为它们很简单。

Component

在app/components新建文件*Stats.js*：

```
import React from 'react';
import StatsStore from '../stores/StatsStore'
import StatsActions from '../actions/StatsActions';

class Stats extends React.Component {
  constructor(props) {
    super(props);
    this.state = StatsStore.getState();
    this.onChange = this.onChange.bind(this);
  }

  componentDidMount() {
    StatsStore.listen(this.onChange);
    StatsActions.getStats();
  }

  componentWillUnmount() {
    StatsStore.unlisten(this.onChange);
  }

  onChange(state) {
    this.setState(state);
  }

  render() {
    return (
      <div className='container'>
        <div className='panel panel-default'>
          <table className='table table-striped'>
            <thead>
            <tr>
              <th colSpan='2'>Stats</th>
            </tr>
            </thead>
            <tbody>
            <tr>
              <td>Leading race in Top 100</td>
              <td>{this.state.leadingRace.race} with {this.state.leadingRace.count} characters</td>
```

```
            </tr>
            <tr>
              <td>Leading bloodline in Top 100</td>
              <td>{this.state.leadingBloodline.bloodline} with {this.state.leadingBloodline.count} characters
              </td>
            </tr>
            <tr>
              <td>Amarr Characters</td>
              <td>{this.state.amarrCount}</td>
            </tr>
            <tr>
              <td>Caldari Characters</td>
              <td>{this.state.caldariCount}</td>
            </tr>
            <tr>
              <td>Gallente Characters</td>
              <td>{this.state.gallenteCount}</td>
            </tr>
            <tr>
              <td>Minmatar Characters</td>
              <td>{this.state.minmatarCount}</td>
            </tr>
            <tr>
              <td>Total votes cast</td>
              <td>{this.state.totalVotes}</td>
            </tr>
            <tr>
              <td>Female characters</td>
              <td>{this.state.femaleCount}</td>
            </tr>
            <tr>
              <td>Male characters</td>
              <td>{this.state.maleCount}</td>
            </tr>
            <tr>
              <td>Total number of characters</td>
              <td>{this.state.totalCount}</td>
            </tr>
            </tbody>
          </table>
        </div>
      </div>
    );
  }
}

export default Stats;
```

Actions

在app/actions目录新建*Stats.js*：

```
import alt from '../alt';

class StatsActions {
  constructor() {
    this.generateActions(
      'getStatsSuccess',
      'getStatsFail'
    );
  }

  getStats() {
    $.ajax({ url: '/api/stats' })
      .done((data) => {
        this.actions.getStatsSuccess(data);
      })
      .fail((jqXhr) => {
        this.actions.getStatsFail(jqXhr);
      });
  }
}

export default alt.createActions(StatsActions);
```

Store

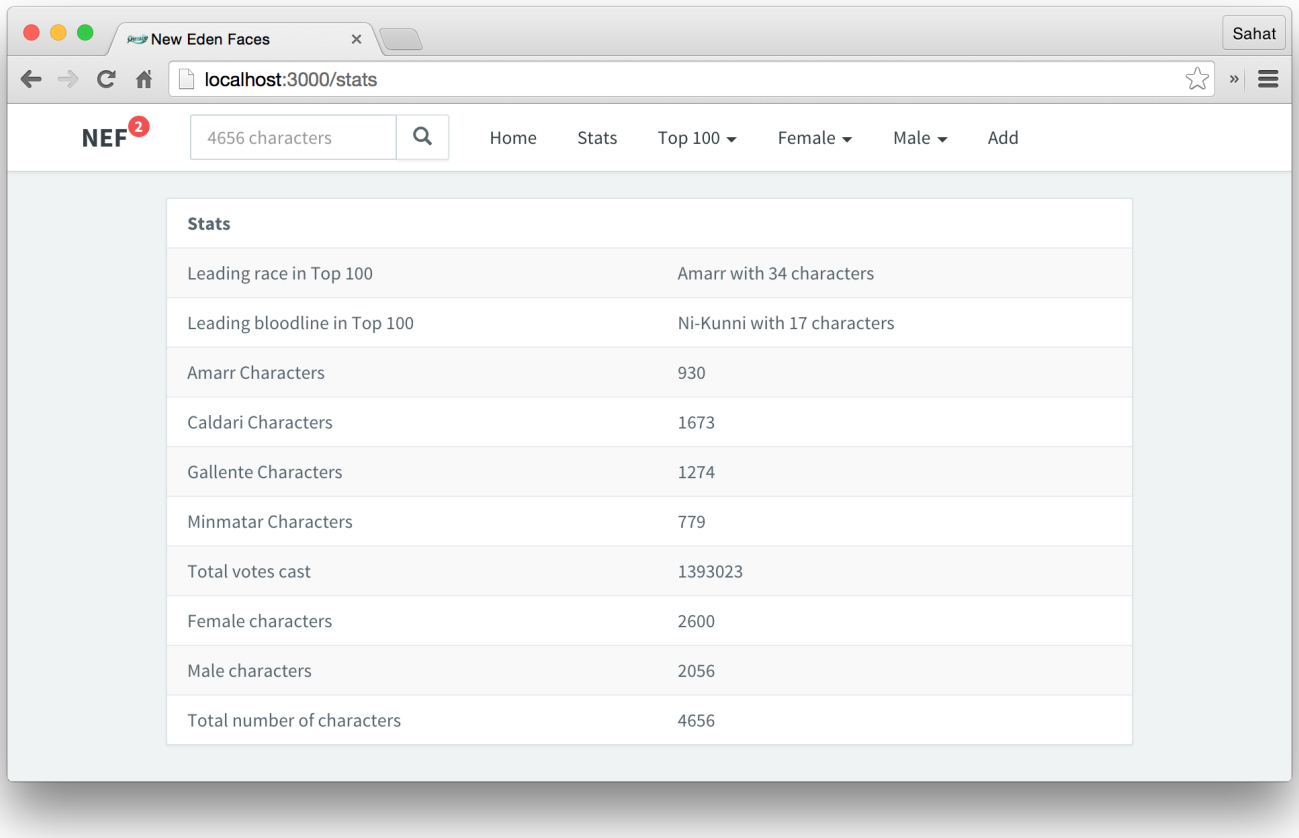
在app/store目录新建*Stats.js*：

```
import {assign} from 'underscore';
import alt from '../alt';
import StatsActions from '../actions/StatsActions';

class StatsStore {
  constructor() {
    this.bindActions(StatsActions);
    this.leadingRace = { race: 'Unknown', count: 0 };
    this.leadingBloodline = { bloodline: 'Unknown', count: 0 };
    this.amarrCount = 0;
    this.caldariCount = 0;
    this.gallenteCount = 0;
    this.minmatarCount = 0;
    this.totalVotes = 0;
    this.femaleCount = 0;
    this.maleCount = 0;
    this.totalCount = 0;
  }

  onGetStatsSuccess(data) {
    assign(this, data);
  }

  onGetStatsFail(jqXhr) {
    toastr.error(jqXhr.responseJSON.message);
  }
}

export default alt.createStore(StatsStore);
```

打开routes.js并添加新路由 `/stats` 。我们必须将它放在 `:category` 路由之前，这样它会被优先执行。

```
import React from 'react';
import {Route} from 'react-router';
import App from './components/App';
import Home from './components/Home';
import AddCharacter from './components/AddCharacter';
import Character from './components/Character';
import CharacterList from './components/CharacterList';
import Stats from './components/Stats';

export default (
  <Route handler={App}>
    <Route path='/' handler={Home} />
    <Route path='/add' handler={AddCharacter} />
    <Route path='/characters/:id' handler={Character} />
    <Route path='/shame' handler={CharacterList} />
    <Route path='/stats' handler={Stats} />
    <Route path=':category' handler={CharacterList}>
      <Route path=':race' handler={CharacterList}>
        <Route path=':bloodline' handler={CharacterList} />
      </Route>
    </Route>
  </Route>
);
```

刷新浏览器，你应该看到如下的新Stats组件：



第十九步：部署

第十九步：部署

现在我们的项目已经完成，而我们终于可以开始部署它了。网上有不少的托管服务提供商，不过如果你关注过我之前的项目或者教程的话，就会知道我为什么这么喜欢Heroku了，不过其它托管商的部署流程应该和它差不太多。

让我们先在根目录创建一个.gitignore文件。然后添加下面的内容，其中大多数来自于Github的gitignore仓库。

```
# Logs
logs
*.log

# Runtime data
pids
*.pid
*.seed

# Directory for instrumented libs generated by jscoverage/JSCover
lib-cov

# Coverage directory used by tools like istanbul
coverage

# Grunt intermediate storage (http://gruntjs.com/creating-plugins#storing-task-files)
.grunt

# Compiled binary addons (http://nodejs.org/api/addons.html)
build/Release

# Dependency directory
# Commenting this out is preferred by some people, see
# https://www.npmjs.org/doc/misc/npm-faq.html#should-i-check-my-node_modules-folder-into-git-
node_modules
bower_components

# Users Environment Variables
.lock-wscript

# Project
.idea
*.iml
.DS_Store

# Compiled files
public/css/*
public/js/*
```

> 注意：我们仅签入源代码到Git中，不包括编译后的CSS和Gulp生成的JavaScript代码。

你还需要在package.json的 `"scripts"` 中添加下列代码：

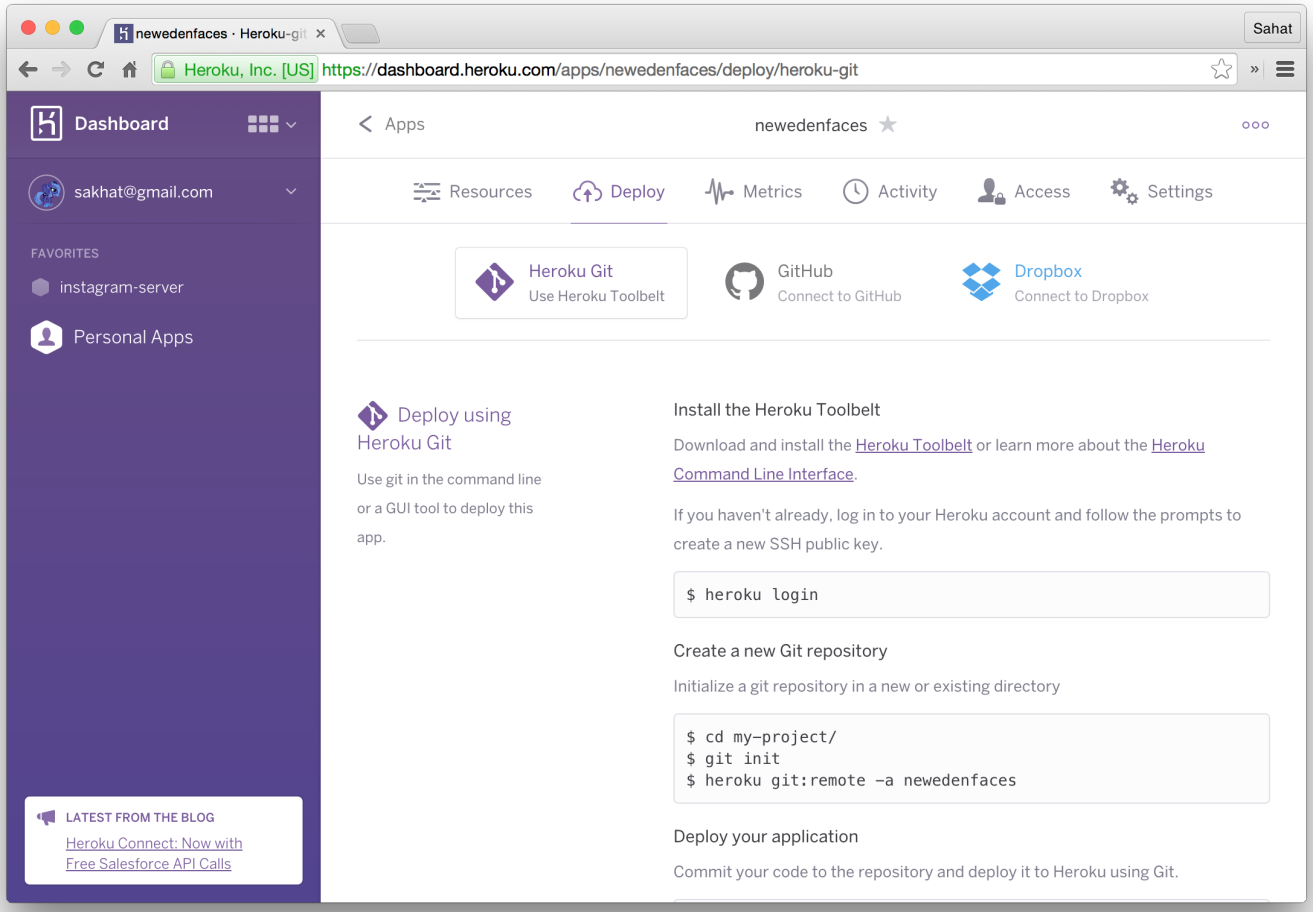
```
"postinstall": "bower install && gulp build"
```

因为我们没有签入编译后的CSS和JavaScript，以及第三方库，我们需要使用 `postinstall` 命令，让Heroku在部署后编译应用并下载Bower包，否则它将不包含main.css、vendor.js、vendor.bundle.js和bundle.js文件。

下一步，让我们在项目根目录下初始化一个新Git仓库：

```
$ git init
$ git add .gitignore
$ git commit -m 'initial commit'
```

现在我们已经准备好将代码推送到Heroku了，不过，我们需要先在Heroku上新建一个应用。在新建应用后顺着下面这个页面的指南操作：



准备完毕后，现在运行下面的命令，这里newedenfaces是我所建应用的名称，把它替换为你在Heroku上新建的应用名称：

```
$ heroku git:remote -a newedenfaces
```

然后，点击Settings标签，顺次点击Reveal Config Vars和Edit按钮，添加下面的环境变量，和我们在config.js中的设置相匹配：

KEY	VALUE
MONGO_URI	mongodb://admin:1234@ds061757.mongolab.com:61757/newedenfaces-tutorial

上面是我为这个教程提供的沙箱数据库，但如果你想创建自己的数据库的话，可以

从MongoLab或Compose甚至直接从Heroku Addons免费获取。

运行下面的命令，然后我们就大功告成！

```
$ git push heroku master
```

现在，你可以从 `http://<app_name>.herokuapp.com` 这样的链接看到你的应用了。

第二十步： 附加资源

第二十步： 附加资源

下面是一些我在学习React、Flux和ES6过程中找到的一些资源，大部分很用，有一些则很有趣。

Link	Description
Elemental UI	漂亮的React UI组件库，包含按钮、表单、旋钮、模态框等等
Navigating the React Ecosystem	非常棒的博文，由Tomas Holas所作，探索了ES6、Generator、Babel、React、React Router、Alt、Flux、React表单、Typeahead等等，从很多层面上说它补足了这篇教程，非常推荐。
A Quick Tour Of ES6	关于ES6新特性的追加资源，非常注重实践并且易于阅读的博文
Atomic CSS	一个激进的新方式来设置你的App的样式。它需要花时间适应，不过一旦你适应，它的优点就显现出来了。你不用再使用CSS类，而是直接在组件中使用“原子的”CSS来设置样式。
classnames	一个React插件用于优雅的设置类名
Iso	Alt的辅助类，用于从服务器传递原始数据到客户端

总结

总结

在我去年底发布的上一篇博客中，我说道：

> 祝贺你成功坚持到了最后！这是我发布过最长的博客了。有趣的是，在TV Show Tracker博客中我也说过同样的话。

但是现在，这篇文章比那一篇还要长！我的确没有想到会写这么长，也绝不是为了打破记录而特意这么做。但我的确希望这篇教程对读者有帮助，并且内容丰富。如果你从本文中学到任何一点东西，那么我的辛苦就没有白费。

如果你喜欢这个项目，可以考虑扩展它，甚至基于New Eden Faces创建一个全新的应用。所有这些代码都放在Github上并且是完全免费的，所以你可以按照你的想法使用或修改它。下面是我想到的一些主意：

- 为重置统计数据、修正性别、删除角色创建一个后台管理界面。
- 为每周统计数据创建一个邮件订阅程序，类似Fitbit Weekly Progress Report。
- 为两个角色创建一对一的竞选投票。
- 更智能的匹配算法，比如高胜率角色应该匹配别的高胜率角色。
- 使用分页列出所有角色。
- 将图片存储到Amazon S3或者MongoDB GridFS来避免每次都请求EVE Online API。
- 研发图片处理算法，以拒绝添加新角色的默认角色形象。
- 每进行X轮后自动重置统计。
- 在角色资料页面显示投票历史。
- 一个归档页面，以显示之前几轮投票的Top 100人物角色。

从我发布的TV Show Tracker教程所收到的邮件，我很高兴看到这些文章几乎对所有水平的人都有用。无论对刚开始编程的初学者，还是对资深的JavaScript专家，或者两者中间的人。

最后是我的一些学习经验，送给那些还在迷茫的人。

如果你还在迷茫是否要学习JavaScript：

- 相信我，我也经历过这个阶段。在学校学习C++和Java后，我难以理解JS中的异步和回调那一套东西。我曾经感到如此愤怒和挫败，以致于我以为以后再也不会用JavaScript了。当然，最后我还是学会了它。这里的技巧是，不要假装你会JavaScript，而是以一个开放的心态从头开始扎实的学习它。

如果你还在迷茫是否要学习和使用ES6：

- 我曾经讨厌ES6.它根本不像我在过去2-3年里逐渐爱上的JavaScript。尽管ES6从很大程度上不过是一套语法糖，但它对我来说就像外星人一样。你需要的是给它一些时间，最终你会爱上它的。并且，不

管你喜不喜欢它，它就是JavaScript发展的方向。

如果你还在迷茫是否要使用React：

- 我记得第一次使用React时的想法是：“这些HTML跑我JavaScript里干嘛？去死吧，我还是坚持用AngularJS。”不过现在是2015年了，我想不用花时间去说服你React是一个很棒的库了。1年前还没什么人用它，但是现在瞅瞅这个使用React的网站的长长的列表。React并不需要你用一种新思维方式去构建应用，而一旦你跨过最初的学习障碍，使用React构建应用其实是很有意思的。我读过很多React和Flux教程，但老实说，直到我开始构建自己的应用，我才真正的理解了它。这里我只想再次重复我的想法：搭建一个小项目是学习任何技术最好的方式，而不是被动的阅读一堆教程和书籍，也不是观看录屏或教学视频。

如果你还在为如何学习编程而挣扎：

- 你应该学习如何坚持，并且应对学习路上一定会产生的沮丧和挫败感，不要放弃。如果2009年我放弃了，我也不会进入大学主修计算机专业；如果2012年我放弃了，我不会获得大学学位；如果2014年我放弃了Hacker School项目，也不会有后来的Satellizer，到现在被全世界的数千开发者使用。挣扎和挫败始终存在，特别是在这个发展特别迅速的行业。不管你怎么想，我不认为我是一个专家，我仍然和大多数人一样每天都有迷茫和挣扎。我很少有走进办公室，并且清楚知道需要做什么、怎么去做的时候，如果工作对于我来说很轻松，那说明我不再进步，该考虑换个工作了。

如果你是一个寻求建议的大学生：

- 现在就开始打造你的代表作品。去创建一个GitHub账号，并且开始为开源项目做贡献或者开发你的个人项目。不要期望学校会教你市场所需要的所有技能。如果你的GPA成绩不好也不要担心，只要你拥有一个好的代表作品，或者对知名开源项目做了重大贡献，那就没什么。对GPA成绩和学校知名度过于重视的公司过于拘泥于传统，你可能不太想为它们工作，除非你也很重视这些。为生活确定一个长期目标，并且为之而努力。我今天所获得的成绩并不是因为我有多聪明或有多天才，我也并不是那些幸运儿。我能取得成绩仅仅是因为那是我想要的，并且为了获得它而持续的努力工作。

（全文完）