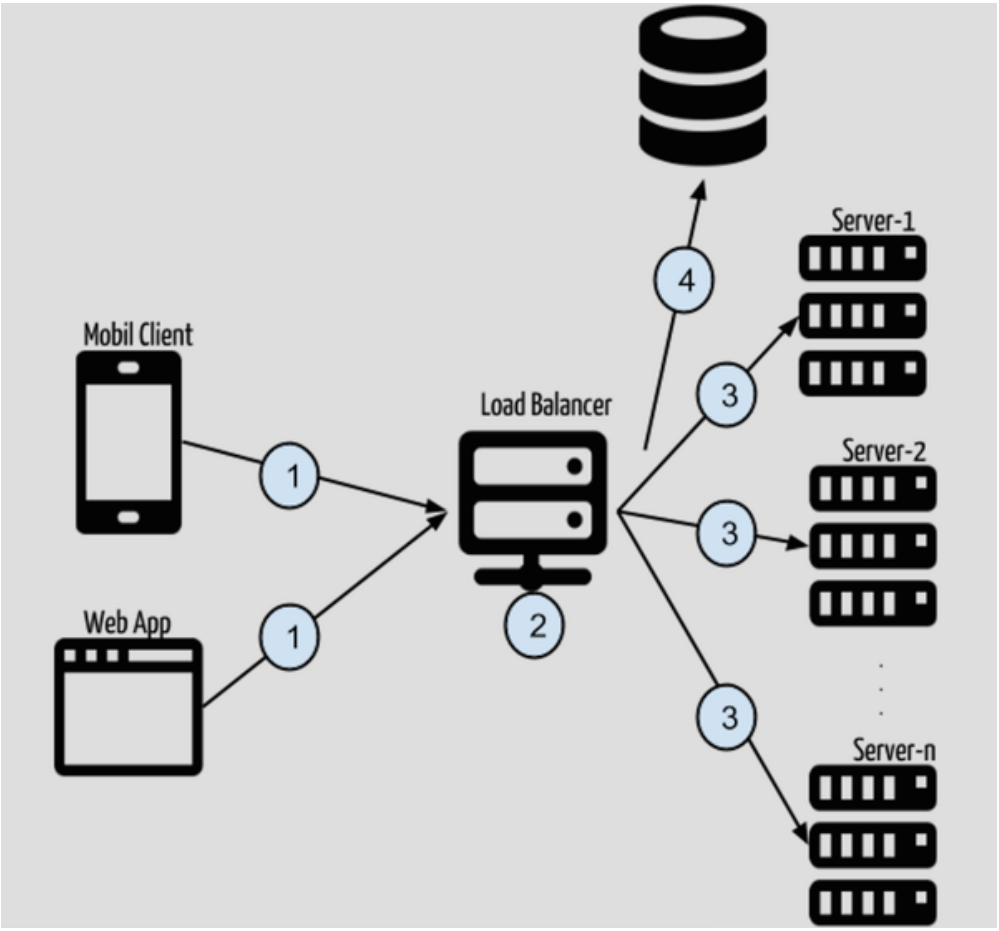


使用 AngularJS & NodeJS 实现基于 token 的认证应用

范洪春 · 10 个月前

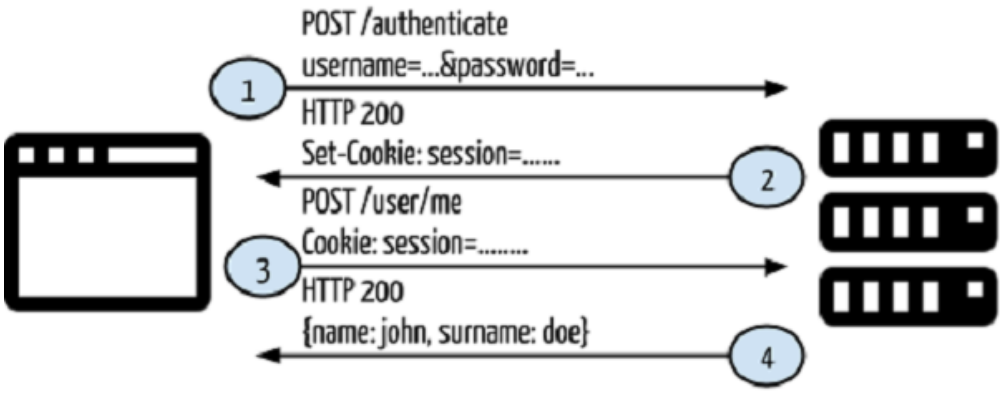
认证是任何 web 应用中不可或缺的一部分。在这个教程中，我们会讨论基于 token 的认证系统以及它和传统的登录系统的不同。这篇教程的末尾，你会看到一个使用 AngularJS 和 NodeJS 构建的完整的应用。

传统的认证系统

在开始说基于 token 的认证系统之前，我们先看一下传统的认证系统。

1. 用户在登录域输入 **用户名** 和 **密码** ，然后点击 **登录** ；
2. 请求发送之后，通过在后端查询数据库验证用户的合法性。如果请求有效，使用在数据库得到的信息创建一个 session，然后在响应头信息中返回这个 session 的信息，目的是把这个 session ID 存储到浏览器中；
3. 在访问应用中受限制的后端服务器时提供这个 session 信息；
4. 如果 session 信息有效，允许用户访问受限制的后端服务器，并且把渲染好的

HTML 内容返回。



在这之前一切都很美好。web 应用正常工作，并且它能够认证用户信息然后可以访问受限的后端服务器；然而当你在开发其他终端时发生了什么呢，比如在 Android 应用中？你还能使用当前的应用去认证移动端并且分发受限制的内容么？真相是，不可以。有两个主要的原因：

1. 在移动应用上 session 和 cookie 行不通。你无法与移动终端共享服务器创建的 session 和 cookie。
2. 在这个应用中，渲染好的 HTML 被返回。但在移动端，你需要包含一些类似 JSON 或者 XML 的东西包含在响应中。

在这个例子中，需要一个独立客户端服务。

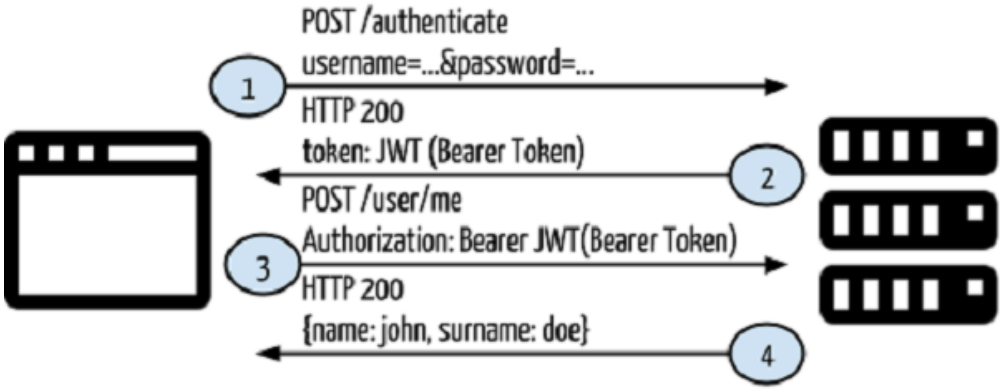
基于 token 的认证

在基于 token 的认证里，不再使用 cookie 和session。token 可被用于在每次向服务器请求时认证用户。我们使用基于 token 的认证来重新设计刚才的设想。

将会用到下面的控制流程：

1. 用户在登录表单中输入 **用户名** 和 **密码** ，然后点击 **登录** ；
2. 请求发送之后，通过在后端查询数据库验证用户的合法性。如果请求有效，使用在数据库得到的信息创建一个 token，然后在响应头信息中返回这个的信息，目的是把这个 token 存储到浏览器的本地存储中；
3. 在每次发送访问应用中受限制的后端服务器的请求时提供 token 信息；
4. 如果从请求头信息中拿到的 token 有效，允许用户访问受限制的后端服务器，并且返回 JSON 或者 XML。

在这个例子中，我们没有返回的 session 或者 cookie，并且我们没有返回任何 HTML 内容。那意味着我们可以把这个架构应用于特定应用的所有客户端中。你可以看一下面的架构体系：



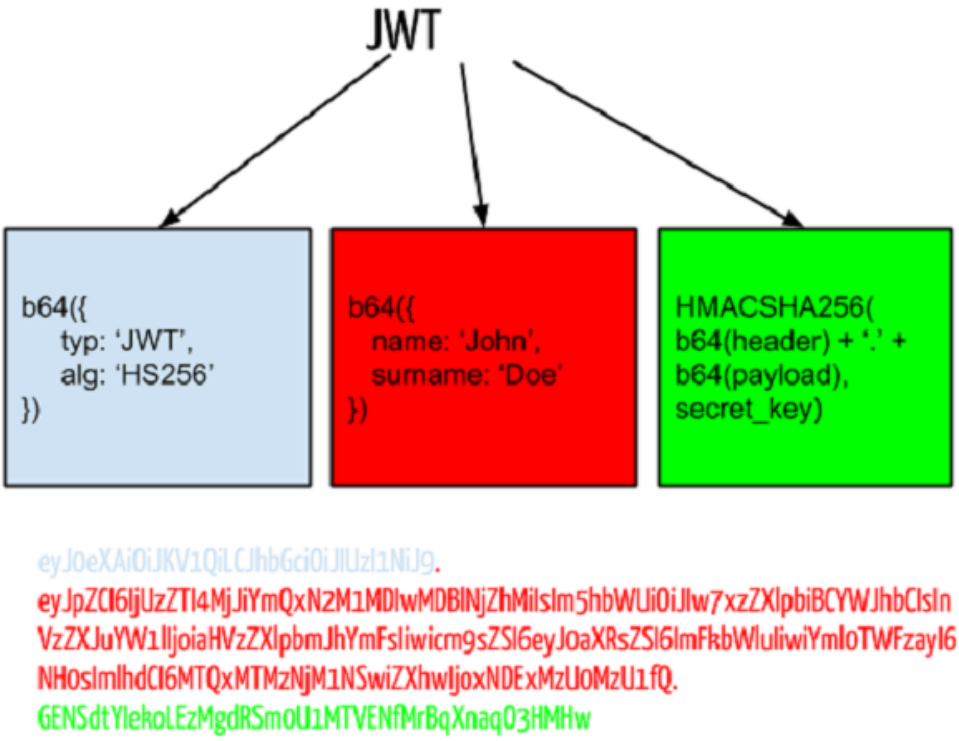
那么，这里的 JWT 是什么？

JWT

JWT 代表 **JSON Web Token** ，它是一种用于认证头部的 token 格式。这个 token 帮你实现了在两个系统之间以一种安全的方式传递信息。出于教学目的，我们暂且把 JWT 作为“不记名 token”。一个不记名 token 包含了三部分：header，payload，signature。

- header 是 token 的一部分，用来存放 token 的类型和编码方式，通常是使用 base-64 编码。
- payload 包含了信息。你可以存放任一种信息，比如用户信息，产品信息等。它们都是使用 base-64 编码方式进行存储。
- signature 包括了 header，payload 和密钥的混合体。密钥必须安全地保存储在服务端。

你可以在下面看到 JWT 刚要和一个实例 token：



你不必关心如何实现不记名 token 生成器函数，因为它对于很多常用的语言已经有多个版本的实现。下面给出了一些：

NodeJS: auth0/node-jsonwebtoken · GitHub

PHP: firebase/php-jwt · GitHub

Java: auth0/java-jwt · GitHub

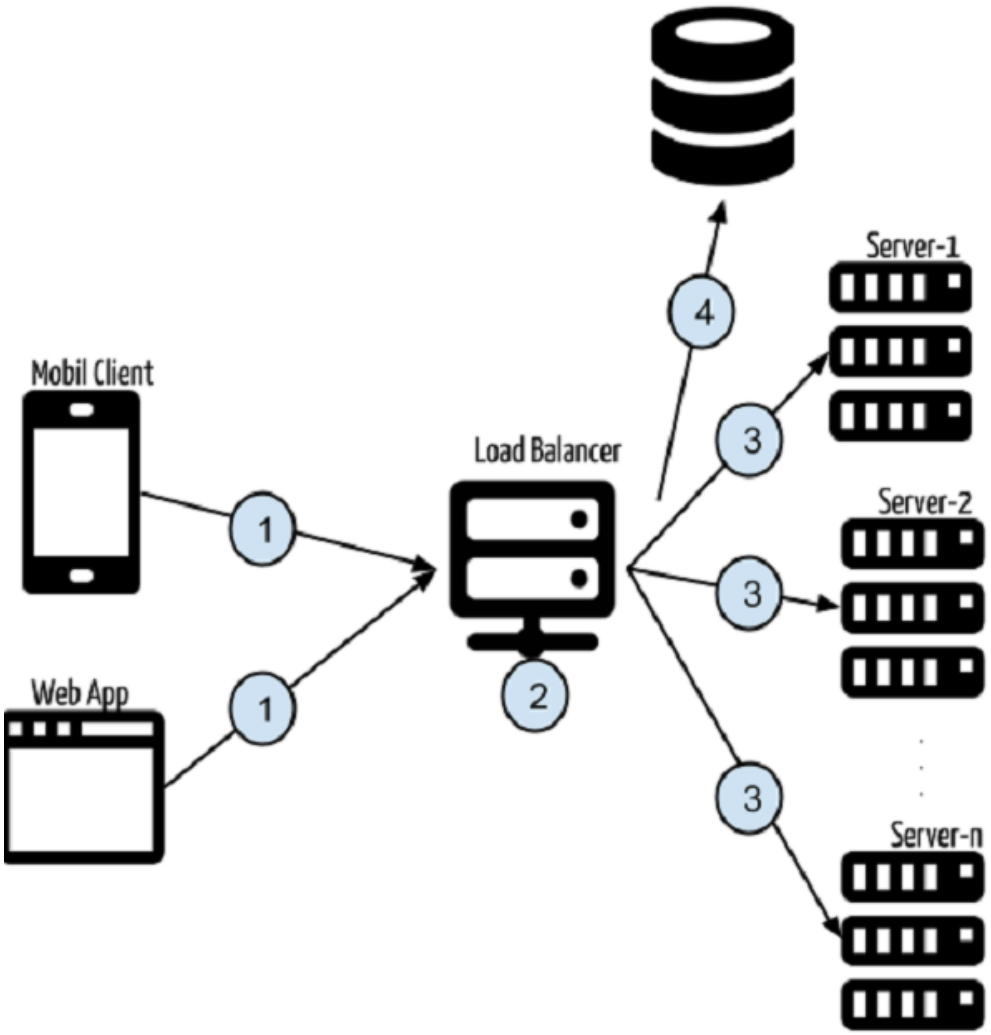
Ruby: progrium/ruby-jwt · GitHub

.NET: AzureAD/azure-activedirectory-identitymodel-extensions-for-dotnet · GitHub

Python: progrium/pyjwt · GitHub

一个实例

在讨论了关于基于 token 认证的一些基础知识后，我们接下来看一个实例。看一下下面的几点，然后我们会仔细的分析它：



1. 多个终端，比如一个 web 应用，一个移动端等向 API 发送特定的请求。
2. 类似 api.yourexampleapp.com 这样的请求发送到服务层。如果很多人使用了这个应用，需要多个服务器来响应这些请求操作。
3. 这时，负载均衡被用于平衡请求，目的是达到最优化的后端应用服务。当你向 api.yourexampleapp.com 发送请求，最外层的负载均衡会处理这个请求，然后重定向到指定的服务器。
4. 一个应用可能会被部署到多个服务器上（server-1, server-2, ..., server-n）。当有请求发送到[api.yourexampleapp.com](api.yourexampleapp.com) 时，后端的应用会拦截这个请求头部并且从认证头部中提取到 token 信息。使用这个 token 查询数据库。如果这个 token 有效并且有请求终端数据所必须的许可时，请求会继续。如果无效，会返回 403 状态码（表明一个拒绝的状态）。

优势

基于 token 的认证在解决棘手的问题时有几个优势：

- **Client Independent Services** 。在基于 token 的认证，token 通过请求头传输，而不是把认证信息存储在 session 或者 cookie 中。这意味着无状态。你可以从任意一种可以发送 HTTP 请求的终端向服务器发送请求。

- **CDN** 。在绝大多数现在的应用中，view 在后端渲染，HTML 内容被返回给浏览器。前端逻辑依赖后端代码。这中依赖真的没必要。而且，带来了几个问题。比如，你和一个设计机构合作，设计师帮你完成了前端的 HTML，CSS 和 JavaScript，你需要拿到前端代码并且把它移植到你的后端代码中，目的当然是为了渲染。修改几次后，你渲染的 HTML 内容可能和设计师完成的代码有了很大的不同。在基于 token 的认证中，你可以开发完全独立于后端代码的前端项目。后端代码会返回一个 JSON 而不是渲染 HTML，并且你可以把最小化，压缩过的代码放到 CDN 上。当你访问 web 页面，HTML 内容由 CDN 提供服务，并且页面内容是通过使用认证头部的 token 的 API 服务所填充。
- **No Cookie-Session (or No CSRF)** 。CSRF 是当代 web 安全中一处痛点，因为它不会去检查一个请求来源是否可信。为了解决这个问题，一个 token 池被用在每次表单请求时发送相关的 token。在基于 token 的认证中，已经有一个 token 应用在认证头部，并且 CSRF 不包含那个信息。
- **Persistent Token Store** 。当在应用中进行 session 的读，写或者删除操作时，会有一个文件操作发生在操作系统的temp 文件夹下，至少在第一次时。假设有多台服务器并且 session 在第一台服务上创建。当你再次发送请求并且这个请求落在另一台服务器上，session 信息并不存在并且会获得一个“未认证”的响应。我知道，你可以通过一个粘性 session 解决这个问题。然而，在基于 token 的认证中，这个问题很自然就被解决了。没有粘性 session 的问题，因为在每个发送到服务器的请求中这个请求的 token 都会被拦截。

这些就是基于 token 的认证和通信中最明显的优势。基于 token 认证的理论和架构就说到这里。下面上实例。

应用实例

你会看到两个用于展示基于 token 认证的应用：

1. token-based-auth-backend
2. token-based-auth-frontend

在后端项目中，包括服务接口，服务返回的 JSON 格式。服务层不会返回视图。在前端项目中，会使用 AngularJS 向后端服务发送请求。

token-based-auth-backend

在后端项目中，有三个主要文件：

- package.json 用于管理依赖；
- models\User.js 包含了可能被用于处理关于用户的数据库操作的用户模型；
- server.js 用于项目引导和请求处理。

就是这样！这个项目非常简单，你不必深入研究就可以了解主要的概念。

```
{
    "name": "angular-restful-auth",
```

```
    "version": "0.0.1",
    "dependencies": {
        "express": "4.x",
        "body-parser": "~1.0.0",
        "morgan": "latest",
        "mongoose": "3.8.8",
        "jsonwebtoken": "0.4.0"
    },
    "engines": {
        "node": ">=0.10.0"
    }
}
```

package.json包含了这个项目的依赖：express 用于 MVC，body-parser 用于在 NodeJS 中模拟 post 请求操作，morgan 用于请求登录，mongoose 用于为我们的 ORM 框架连接 MongoDB，最后 jsonwebtoken 用于使用我们的 User 模型创建 JWT 。如果这个项目使用版本号 >= 0.10.0 的 NodeJS 创建，那么还有一个叫做 engines 的属性。这对那些像 HeroKu 的 PaaS 服务很有用。我们也会在另外一节中包含那个话题。

```
var mongoose     = require('mongoose');
var Schema       = mongoose.Scema;

var UserSchema   = new Schema({
    email: String,
    password: String,
    token: String
});

module.exports = mongoose.model('User', UserSchema);
```

上面提到我们可以通过使用用户的 payload 模型生成一个 token。这个模型帮助我们处理用户在 MongoDB 上的请求。在User.js，user-schema 被定义并且 User 模型通过使用 mogoose 模型被创建。这个模型提供了数据库操作。

我们的依赖和 user 模型被定义好，现在我们把那些构想成一个服务用于处理特定的请求。

```
// Required Modules
var express    = require("express");
var morgan     = require("morgan");
var bodyParser = require("body-parser");
var jwt        = require("jsonwebtoken");
var mongoose   = require("mongoose");
var app        = express();
```

在 NodeJS 中，你可以使用 require 包含一个模块到你的项目中。第一步，我们需要把必要的模块引入到项目中：

```
var port = process.env.PORT || 3001;
var User     = require('./models/User');
// Connect to DB
mongoose.connect(process.env.MONGO_URL);
```

服务层通过一个指定的端口提供服务。如果没有在环境变量中指定端口，你可以使用那个，或者我们定义的 3001 端口。然后，User 模型被包含，并且数据库连接被建立用来处理一些用户操作。不要忘记定义一个 MONGO_URL 环境变量，用于数据库连接 URL。

```
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());
app.use(morgan("dev"));
app.use(function(req, res, next) {
    res.setHeader('Access-Control-Allow-Origin', '*');
    res.setHeader('Access-Control-Allow-Methods', 'GET, POST');
    res.setHeader('Access-Control-Allow-Headers', 'X-Requested-With,cor
    next();
});
```

上一节中，我们已经做了一些配置用于在 NodeJS 中使用 Express 模拟一个 **HTTP** 请求。我们允许来自不同域名的请求，目的是建立一个独立的客户端系统。如果你没这么做，可能会触发浏览器的 CORS（跨域请求共享）错误。

- Access-Control-Allow-Origin 允许所有的域名。
- 你可以向这个设备发送 POST 和 GET 请求。
- 允许 X-Requested-With 和 content-type 头部。

```
app.post('/authenticate', function(req, res) {
    User.findOne({email: req.body.email, password: req.body.password},
        if (err) {
            res.json({
                type: false,
                data: "Error occured: " + err
            });
        } else {
            if (user) {
               res.json({
                    type: true,
                    data: user,
                    token: user.token
                });
            } else {
                res.json({
                    type: false,
                    data: "Incorrect email/password"
                });
            }
        }
    });
});
```

我们已经引入了所需的全部模块并且定义了配置文件，所以是时候来定义请求处理函数了。在上面的代码中，当你提供了用户名和密码向 /authenticate 发送一个 POST 请求时，你将会得到一个 JWT。首先，通过用户名和密码查询数据库。如果用户存

在，用户数据将会和它的 token 一起返回。但是，如果没有用户名或者密码不正确，要怎么处理呢？

```
app.post('/signin', function(req, res) {
    User.findOne({email: req.body.email, password: req.body.password},
        if (err) {
            res.json({
                type: false,
                data: "Error occured: " + err
            });
        } else {
            if (user) {
                res.json({
                    type: false,
                    data: "User already exists!"
                });
            } else {
                var userModel = new User();
                userModel.email = req.body.email;
                userModel.password = req.body.password;
                userModel.save(function(err, user) {
                    user.token = jwt.sign(user, process.env.JWT_SECRET)
                    user.save(function(err, user1) {
                        res.json({
                            type: true,
                            data: user1,
                            token: user1.token
                        });
                    });
                })
            }
        }
    });
});
```

当你使用用户名和密码向 /signin 发送 POST 请求时，一个新的用户会通过所请求的用户信息被创建。在 第 19 行，你可以看到一个新的 JSON 通过 jsonwebtoken 模块生成，然后赋值给 jwt 变量。认证部分已经完成。我们访问一个受限的后端服务器会怎么样呢？我们又要如何访问那个后端服务器呢？

```
app.get('/me', ensureAuthorized, function(req, res) {
    User.findOne({token: req.token}, function(err, user) {
        if (err) {
            res.json({
                type: false,
                data: "Error occured: " + err
            });
        } else {
            res.json({
                type: true,
                data: user
            });
        }
    });
```

```
});
```

当你向 /me 发送 GET 请求时，你将会得到当前用户的信息，但是为了继续请求后端服务器， ensureAuthorized 函数将会执行。

```
function ensureAuthorized(req, res, next) {
    var bearerToken;
    var bearerHeader = req.headers["authorization"];
    if (typeof bearerHeader !== 'undefined') {
        var bearer = bearerHeader.split(" ");
        bearerToken = bearer[1];
        req.token = bearerToken;
        next();
    } else {
        res.send(403);
    }
}
```

在这个函数中，请求头部被拦截并且 authorization 头部被提取。如果头部中存在一个不记名 token，通过调用 next()函数，请求继续。如果 token 不存在，你会得到一个 403（Forbidden）返回。我们回到 /me 事件处理函数，并且使用req.token 获取这个 token 对应的用户数据。当你创建一个新的用户，会生成一个 token 并且存储到数据库的用户模型中。那些 token 都是唯一的。

这个简单的例子中已经有三个事件处理函数。然后，你将看到；

```
process.on('uncaughtException', function(err) {
    console.log(err);
});
```

当程序出错时 NodeJS 应用可能会崩溃。添加上面的代码可以拯救它并且一个错误日志会打到控制台上。最终，我们可以使用下面的代码片段启动服务。

```
// Start Server
app.listen(port, function () {
    console.log( "Express server listening on port " + port);
});
```

总结一下：

- 引入模块
- 正确配置
- 定义请求处理函数
- 定义用来拦截受限终点数据的中间件
- 启动服务

我们已经完成了后端服务。到现在，应用已经可以被多个终端使用，你可以部署这个简单的应用到你的服务器上，或者部署在 Heroku。有一个叫做 Procfile 的文件在项目的根目录下。现在把服务部署到 Heroku。

Heroku 部署

你可以在这个 GitHub 库 下载项目的后端代码。

我不会教你如何在 Heroku 如何创建一个应用；如果你还没有做过这个，你可以查阅这篇文章 。创建完 Heroku 应用，你可以使用下面的命令为你的项目添加一个地址：

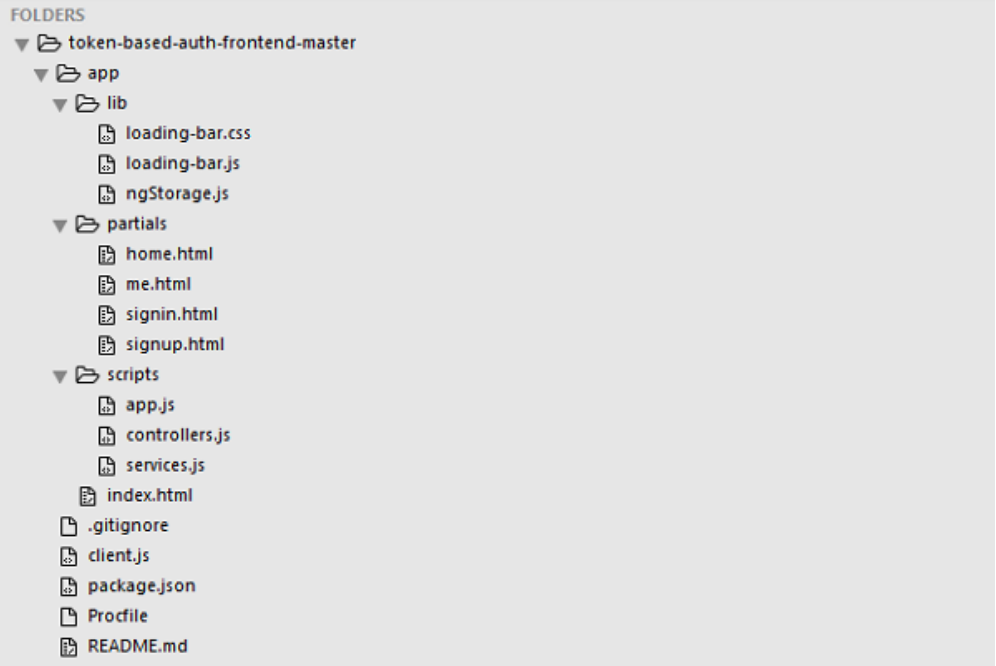
```
git remote add heroku <your_heroku_git_url>
```

现在，你已经克隆了这个项目并且添加了地址。在 git add 和 git commit 后，你可以使用 git push heroku master 命令将你的代码推到 Heroku。当你成功将项目推送到仓库，Heroku 会自动执行 npm install 命令将依赖文件下载到 Heroku 的 temp 文件夹。然后，它会启动你的应用，因此你就可以使用 HTTP 协议访问这个服务。

token-based-auth-frontend

在前端项目中，将会使用 AngularJS。在这里，我只会提到前端项目中的主要内容，因为 AngularJS 的相关知识不会包括在这个教程里。

你可以在这个 GitHub 库 下载源码。在这个项目中，你会看下下面的文件结构：



ngStorage.js 是一个用于操作本地存储的 AngularJS 类库。此外，有一个全局的 layout 文件 index.html 并且在 partials 文件夹里还有一些用于扩展全局 layout 的部分。 controllers.js 用于在前端定义我们 controller 的 action。 services.js 用于向我们在上一个项目中提到的服务发送请求。还有一个 app.js 文件，它里面有配置文件和模块引入。最后，client.js 用于服务静态 HTML 文件（或者仅仅 index.html，在这里例子中）；当你没有使用 Apache 或者任何其他的 web 服务器时，它可以为静态的 HTML 文件提供服务。

```
...
```

```
<script src="//cdnjs.cloudflare.com/ajax/libs/jquery/2.1.1/jquery.min.j
<script src="//maxcdn.bootstrapcdn.com/bootstrap/3.2.0/js/bootstrap.mir
<script src="//cdnjs.cloudflare.com/ajax/libs/angular.js/1.2.20/angular
<script src="//cdnjs.cloudflare.com/ajax/libs/angular.js/1.2.20/angular
<script src="/lib/ngStorage.js"></script>
<script src="/lib/loading-bar.js"></script>
<script src="/scripts/app.js"></script>
<script src="/scripts/controllers.js"></script>
<script src="/scripts/services.js"></script>
</body>
```

在全局的 layout 文件中，AngularJS 所需的全部 JavaScript 文件都被包含，包括自定义的控制器，服务和应用文件。

```
'use strict';

/* Controllers */

angular.module('angularRestfulAuth')
    .controller('HomeCtrl', ['$rootScope', '$scope', '$location', '$loc

        $scope.signin = function() {
            var formData = {
                email: $scope.email,
                password: $scope.password
            }

            Main.signin(formData, function(res) {
                if (res.type == false) {
                    alert(res.data)
                } else {
                    $localStorage.token = res.data.token;
                    window.location = "/";
                }
            }, function() {
                $rootScope.error = 'Failed to signin';
            })
        };

        $scope.signup = function() {
            var formData = {
                email: $scope.email,
                password: $scope.password
            }

            Main.save(formData, function(res) {
                if (res.type == false) {
                    alert(res.data)
                } else {
                    $localStorage.token = res.data.token;
                    window.location = "/"
                }
            }, function() {
                $rootScope.error = 'Failed to signup';
            })
```

```
        };

        $scope.me = function() {
            Main.me(function(res) {
                $scope.myDetails = res;
            }, function() {
                $rootScope.error = 'Failed to fetch details';
            })
        };

        $scope.logout = function() {
            Main.logout(function() {
                window.location = "/"
            }, function() {
                alert("Failed to logout!");
            });
        };
        $scope.token = $localStorage.token;
    }])
```

在上面的代码中，HomeCtrl 控制器被定义并且一些所需的模块被注入（比如 $rootScope 和 $scope）。依赖注入是 AngularJS 最强大的属性之一。 $scope 是 AngularJS 中的一个存在于控制器和视图之间的中间变量，这意味着你可以在视图中使用 test，前提是你在特定的控制器中定义了 $scope.test=.....。

在控制器中，一些工具函数被定义，比如：

- signin 可以在登录表单中初始化一个登录按钮；
- signup 用于处理注册操作；
- me 可以在 layout 中生生一个 Me 按钮；

在全局 layout 和主菜单列表中，你可以看到 data-ng-controller 这个属性，它的值是 HomeCtrl。那意味着这个菜单的 dom 元素可以和 HomeCtrl 共享作用域。当你点击表单里的 sign-up 按钮时，控制器文件中的 sign-up 函数将会执行，并且在这个函数中，使用的登录服务来自于已经注入到这个控制器的 Main 服务。

主要的结构是 view -> controller -> service。这个服务向后端发送了简单的 Ajax 请求，目的是获取指定的数据。

```
'use strict';

angular.module('angularRestfulAuth')
    .factory('Main', ['$http', '$localStorage', function($http, $localS
        var baseUrl = "your_service_url";
        function changeUser(user) {
            angular.extend(currentUser, user);
        }

        function urlBase64Decode(str) {
            var output = str.replace('-', '+').replace('_', '/');
            switch (output.length % 4) {
                case 0:
```

```
                    break;
                case 2:
                    output += '==';
                    break;
                case 3:
                    output += '=';
                    break;
                default:
                    throw 'Illegal base64url string!';
            }
            return window.atob(output);
        }

        function getUserFromToken() {
            var token = $localStorage.token;
            var user = {};
            if (typeof token !== 'undefined') {
                var encoded = token.split('.')[1];
                user = JSON.parse(urlBase64Decode(encoded));
            }
            return user;
        }

        var currentUser = getUserFromToken();

        return {
            save: function(data, success, error) {
                $http.post(baseUrl + '/signin', data).success(success).
            },
            signin: function(data, success, error) {
                $http.post(baseUrl + '/authenticate', data).success(suc
            },
            me: function(success, error) {
                $http.get(baseUrl + '/me').success(success).error(error
            },
            logout: function(success) {
                changeUser({});
                delete $localStorage.token;
                success();
            }
        };
    }
]);
```

在上面的代码中，你会看到服务函数请求认证。在 controller.js 中，你可能已经看到了有类似 Main.me 的函数。这里的Main 服务已经注入到控制器，并且在它内部，属于这个服务的其他服务直接被调用。

这些函数式仅仅是简单地向我们部署的服务器集群发送 Ajax 请求。不要忘记在上面的代码中把服务的 URL 放到 baseUrl。当你把服务部署到 Heroku，你会得到一个类似 appname.herokuapp.com 的服务 URL。在上面的代码中，你要设置 var baseUrl = "appname.herokuapp.com"。

在应用的注册或者登录部分，不记名 token 响应了这个请求并且这个 token 被存储到

本地存储中。当你向后端请求一个服务时，你需要把这个 token 放在头部中。你可以使用 AngularJS 的拦截器实现这个。

```
$httpProvider.interceptors.push(['$q', '$location', '$localStorage', fu
            return {
                'request': function (config) {
                    config.headers = config.headers || {};
                    if ($localStorage.token) {
                        config.headers.Authorization = 'Bearer ' + $loc
                    }
                    return config;
                },
                'responseError': function(response) {
                    if(response.status === 401 || response.status === 4
                        $location.path('/signin');
                    }
                    return $q.reject(response);
                }
            };
        }]);
```

在上面的代码中，每次请求都会被拦截并且会把认证头部和值放到头部中。

在前端项目中，会有一些不完整的页面，比如 signin，signup，profile details 和 vb。这些页面与特定的控制器相关。你可以在 app.js 中看到：

```
angular.module('angularRestfulAuth', [
    'ngStorage',
    'ngRoute'
])
.config(['$routeProvider', '$httpProvider', function ($routeProvider, $

    $routeProvider.
        when('/', {
            templateUrl: 'partials/home.html',
            controller: 'HomeCtrl'
        }).
        when('/signin', {
            templateUrl: 'partials/signin.html',
            controller: 'HomeCtrl'
        }).
        when('/signup', {
            templateUrl: 'partials/signup.html',
            controller: 'HomeCtrl'
        }).
        when('/me', {
            templateUrl: 'partials/me.html',
            controller: 'HomeCtrl'
        }).
        otherwise({
            redirectTo: '/'
        });
```

如上面代码所示，当你访问 /，home.html 将会被渲染。再看一个例子：如果你访问 /signup，signup.html 将会被渲染。渲染操作会在浏览器中完成，而不是在服务端。

结论

你可以通过检出这个实例看到我们在这个教程中所讨论的项目是如何工作的。

基于 token 的认证系统帮你建立了一个认证/授权系统，当你在开发客户端独立的服务时。通过使用这个技术，你只需关注于服务（或者 API）。

认证/授权部分将会被基于 token 的认证系统作为你的服务前面的层来处理。你可以访问并且使用来自于任何像 web 浏览器，Android，iOS 或者一个桌面客户端这类服务。

原文：Token-Based Authentication With AngularJS & NodeJS

关注微博：前端外刊评论

何为 Io.js

JavaScript：回眸 2014年

10 条评论