参考：<http://blog.rainy.im/2015/06/10/react-jwt-pretty-good-practice/>

JWT 在前后端分离中的应用与实践

[Yusheng](http://blog.rainy.im/author/yu/) · Jun 10, 2015

本文主要介绍JWT（[JSON Web Token](http://jwt.io/)）授权机制在前后端分离中的应用与实践，包括以下三部分：

JWT原理介绍

JWT的安全性

React.js+Flux架构下的实践（[React-jwt example](https://github.com/rainyear/react-jwt-example.git)）

0 关于前后端分离

前后端分离是一个很有趣的议题，它不仅仅是指前后端工程师之间的相互独立的合作分工方式，更是前后端之间开发模式与交互模式的模块化、解耦化。计算机世界的经验告诉我们，对于复杂的事物，模块化总是好的，无论是后端API开发中越来越成为规范的[RESTful API](http://en.wikipedia.org/wiki/Representational_state_transfer)风格，还是Web前端越来越多的模板、框架（参见[MVC，MVP 和 MVVM 的图示](http://www.ruanyifeng.com/blog/2015/02/mvcmvp_mvvm.html)），包括移动应用中前后端天然分离的特质，都证实了前后端分离的重要性与必要性（更生动的细节与实例说明可以参看赫门分享的主题[淘宝前后端分离实践](http://2014.jsconf.cn/slides/herman-taobaoweb/" \l "/)）。

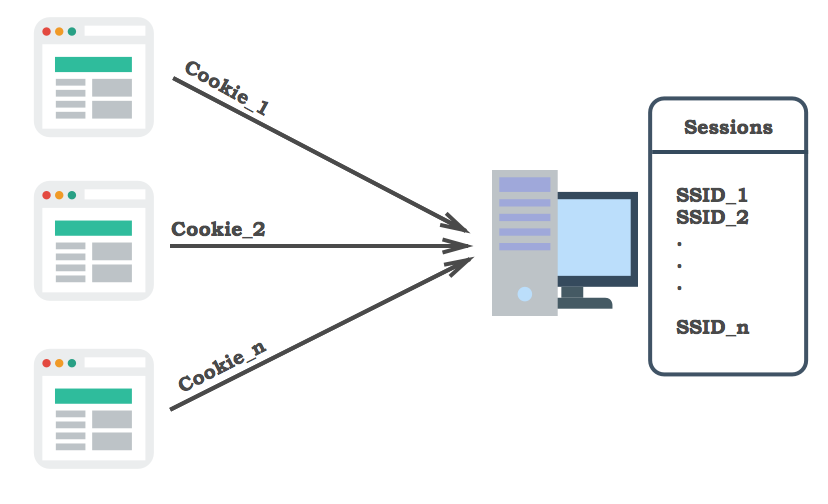
实现前后端分离，对于后端开发人员来说是一件很幸福的事情，因为不需要再考虑怎样在HTML中套入数据，只关心数据逻辑的处理；而前端则需要承担接收数据之后界面呈现、用户交互、数据传递等所有任务。虽然这看起来加重了前端的工作量，但实际上有越来越多丰富多样的前端框架可供选择，这让前端开发变得越来越结构化、系统化，前端工程师也不再只是“套版的”。

在所有前端框架中，Facebook推出的[React](http://facebook.github.io/react)无疑是当下最热门（之一），然而React只负责界面渲染层面，相当于MVC中的V（View），因此只靠React无法完成一个完整的单页应用（[Single Page App](http://en.wikipedia.org/wiki/Single-page_application)）。Facebook另外推出与之配套的[Flux](http://facebook.github.io/flux)架构，主要为了避免Angular.js之类MVC的架构模式，规避数据双向绑定而采用单向绑定的数据传递方式。实际上React无论是学习还是使用都是非常简单的，而Flux则需要花更多时间去理解消化，本文第3部分我采用Flux架构的一种实现[Reflux.js](https://github.com/spoike/refluxjs)，做了一个基于JWT授权机制的登入、登出的例子，顺便介绍Flux架构的细节。

1 JWT 介绍及其原理

JWT是我之前做Android应用的时候了解到的一种用户授权机制，虽然原生的移动手机应用与基于浏览器的Web应用之间存在很多差异，但很多情况下后端往往还是沿用已有的架构跟代码，所以用户授权往往还是采用Cookie+Session的方式，也就是需要原生应用中模拟浏览器对Cookie的操作。

Cookie+Session的存在主要是为了解决HTTP这一无状态协议下服务器如何识别用户的问题，其原理就是在用户登录通过验证后，服务端将数据加密后保存到客户端浏览器的Cookie中，同时服务器保留相对应的Session（文件或DB）。用户之后发起的请求都会携带Cookie信息，服务端需要根据Cookie寻回对应的Session，从而完成验证，确认这是之前登陆过的用户。其工作原理如下图所示：



JWT是[Auth0](https://auth0.com/)提出的通过对JSON进行加密签名来实现授权验证的方案，编码之后的JWT看起来是这样的一串字符：

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.TJVA95OrM7E2cBab30RMHrHDcEfxjoYZgeFONFh7HgQ

由.分为三段，通过解码可以得到：

// 1. Headers

// 包括类别（typ）、加密算法（alg）；

{

"alg": "HS256",

"typ": "JWT"

}

// 2. Payload

// 包括需要传递的用户信息；

{

"sub": "1234567890",

"name": "John Doe",

"admin": true

}

// 3. Signature

// 根据alg算法与私有秘钥进行加密得到的签名字串；

// 这一段是最重要的敏感信息，只能在服务端解密；

HMACSHA256(

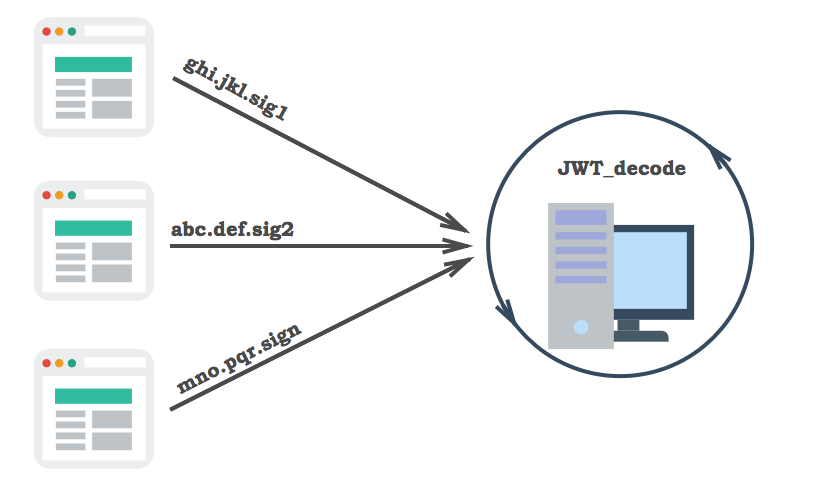
base64UrlEncode(header) + "." +

base64UrlEncode(payload),

SECREATE\_KEY

)

在使用过程中，服务端通过用户登录验证之后，将Header+Payload信息加密后得到第三段签名，然后将签名返回给客户端，在后续请求中，服务端只需要对用户请求中包含的JWT进行解码，即可验证是否可以授权用户获取相应信息，其原理如下图所示：



通过比较可以看出，使用JWT可以省去服务端读取Session的步骤(me：需要解码计算，但是相对于io的节省是很可观的)，这样更符合RESTful的规范。但是对于客户端（或App端）来说，为了保存用户授权信息，仍然需要通过Cookie或类似的机制进行本地保存。因此JWT是用来取代服务端的Session而非客户端Cookie的方案，当然对于客户端本地存储，HTML5提供了Cookie之外更多的解决方案（localStorage/sessionStorage），究竟采用哪种存储方式，其实从Js操作上来看没有本质上的差异，不同的选择更多是出于安全性的考虑。

2 JWT 安全性

用户授权这样敏感的信息，安全性当然是首先需要考虑的因素。这里主要讨论在使用JWT时如何防止XSS和XSRF两种攻击。

XSS是Web中最常见的一种漏洞（我们的\*\*学报官网就存在这个漏洞这件事我就不说了=.=），其主要原因是对用户输入信息不加过滤，导致用户（被误导）恶意输入的Js代码在访问该网页时被执行，而Js可以读取当前网站域名下保存的Cookie信息。针对这种攻击，无论是Cookie还是localStorage中的信息都有可能被窃取，但防止XSS也相对简单一些，对用户输入的所有信息进行过滤即可。另外，现在越来越多的CDN服务，让我们可以节省服务器流量，但同时也有可能引入不安全的Js脚本，例如前段时间Github被Great Cannon轰击的案例，则需要提高对某度之类服务的警惕。

另外一种更加棘手的XSRF漏洞主要利用Cookie是按照域名存储，同时访问某域名时浏览器会自动携带该域名所保存的Cookie信息这一特征。如果执意要将JWT存储在Cookie中，服务端则需要额外验证请求来源，或者在提交表单中加入随机签名并在处理表单时进行验证。

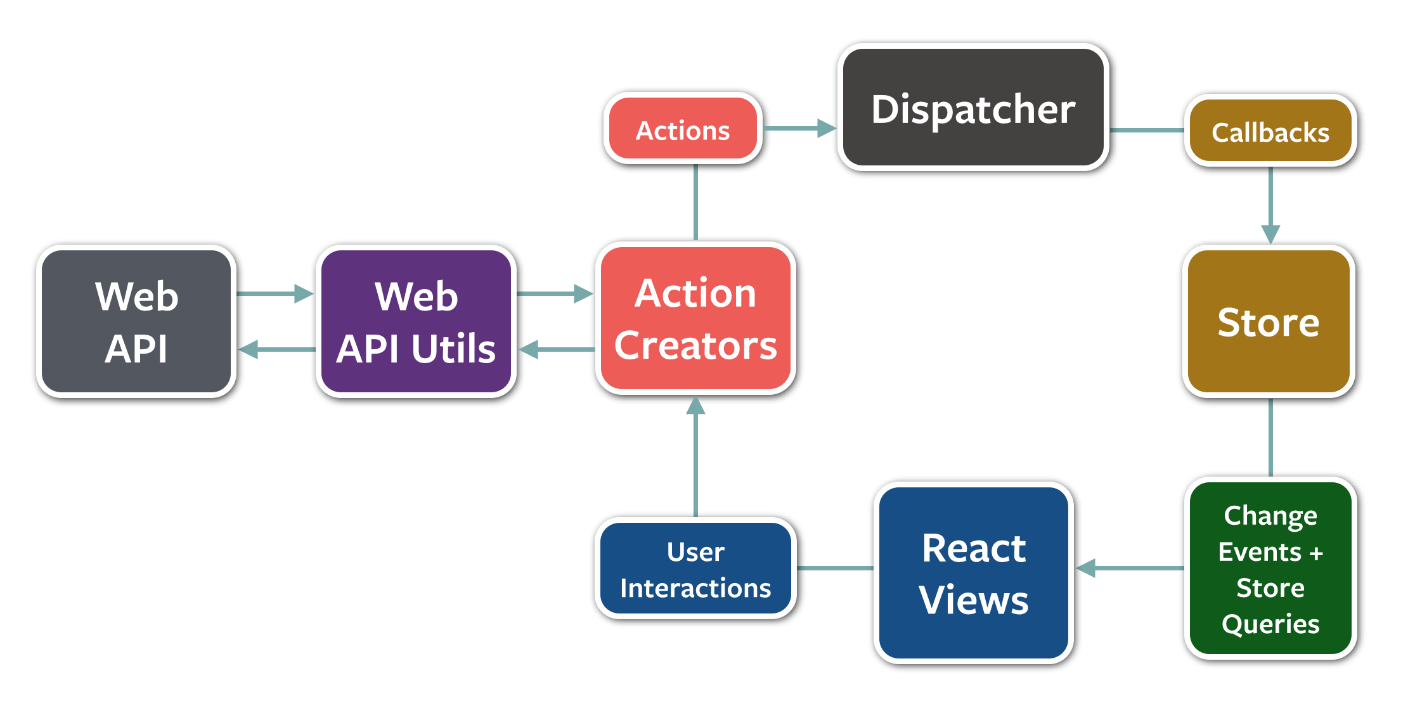
我在后面的实例中采用将JWT保存在localStorage中的方案，请求时将JWT放入Request Header中的Authorization位。对JWT安全性问题想要了解更多可以参考下面几篇文章：

1. [Where to Store Your JWTs - Cookies vs HTML5 Web Storage](https://stormpath.com/blog/where-to-store-your-jwts-cookies-vs-html5-web-storage/)
2. [Use JWT the Right Way!](https://stormpath.com/blog/jwt-the-right-way/)
3. [10 Things You Should Know about Tokens](https://auth0.com/blog/2014/01/27/ten-things-you-should-know-about-tokens-and-cookies/)
4. [Where to store JWT in browser? How to protect against CSRF?](http://stackoverflow.com/questions/27067251/where-to-store-jwt-in-browser-how-to-protect-against-csrf)

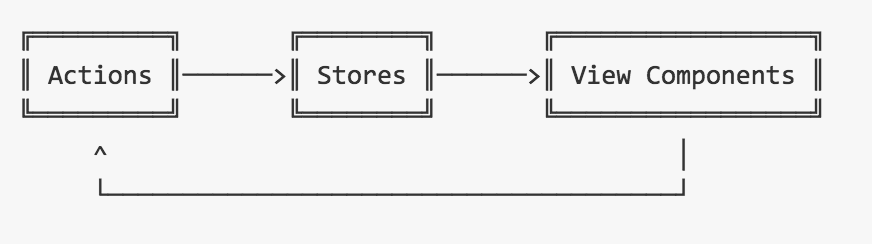
**3 React-jwt Example**

本节源码可见[Github: react-jwt-example](https://github.com/rainyear/react-jwt-example)。

前面提到的React.js框架学习成本其实非常低，只要跟着官方教程走一遍，搞清楚props、states、virtual DOM几个概念，就可以开始用了。但是只有View层什么都做不了，Facebook推出配套的Flux架构，一开始看到下面这张架构图，当时我就懵逼了。



好在Flux只是一种理论架构，虽然官方也提供了实现方案，但是我更倾向于[Reflux.js](https://github.com/spoike/refluxjs)的实现方式，如下图所示：



其中View Components即视图层由React负责，Stores用于存储数据，Actions则用于监听所有动作，所有数据的传递都是单向绑定的，在分割不同模块时，可以清楚地看到数据的流动方向。

我尝试写了一个简单的登录、登出以及获取用户个人数据的例子，除了Reflux之外，还用到如下模块：

1. [react-router](https://github.com/rackt/react-router): SPA路由；
2. [react-bootstrap](http://react-bootstrap.github.io/): React化的Bootstrap，UI样式；
3. [reqwest](https://www.npmjs.com/package/reqwest): Ajax请求；
4. [jwt-decode](https://www.npmjs.com/package/jwt-decode): 客户端的JWT解码；

另外服务端API采用[Go gin](https://gin-gonic.github.io/gin)框架，依赖于[jwt-go](http://github.com/dgrijalva/jwt-go)。代码目录结构如下：

tree -I 'node\_modules|.git'

.

├── README.md

├── gulpfile.js

├── index.html

├── package.json

├── scripts

│   ├── actions

│   │   └── actions.js

│   ├── app.js

│   ├── build

│   │   └── dist.js

│   ├── components

│   │   └── HelloWorld.js

│   ├── stores

│   │   ├── loginStore.js

│   │   └── userStore.js

│   └── views

│   ├── home.js

│   ├── login.js

│   └── profile.js

└── server.go

完整的页面放在view中，可复用的组件放在components，用户的动作包括login、logout以及getBalance，因此需要创建相应的action来监听这些动作：

// actions.js

var actions = Reflux.createActions({

"login": {},

"updateProfile": {}, // login成功更新用户数据

"loginError": {}, // login失败错误信息

"logout": {},

"getBalance": {asyncResult: true}

});

actions.login.listen(function(data){});

用户点击view中的Submit Button时，将表单信息提交给login action：

// views/login.js

var Login = React.createClass({

...

login: function (e) {

e.preventDefault();

actions.login({

name: this.refs.name.getValue(),

pass: this.refs.pass.getValue(),

}),

...

});

// actions.js

var req = require('reqwest');

actions.login.listen(function(data){

req({

url: HOST+"/user/token",

method: "post",

data: JSON.stringify(data),

type: 'json',

contentType: 'application/json',

headers: {'X-Requested-With': 'XMLHttpRequest'},

success: function (resp) {

if(resp.code == 200){

actions.updateProfile(resp.jwt)

}else{

actions.updateProfile(resp.msg)

}

},

})

});

根据API返回结果，将再次触发updateProfile或updateProfile action，而分别由userStore和loginStore接收：

// stores/userStore.js

var userStore = Reflux.createStore({

listenables: actions, // 声明userStore所监听的action

updateProfile: function(jwt){

// 注册监听actions.updateProfile

localStorage.setItem('jwt', jwt);

this.user = jwt\_decode(jwt);

this.user.logd = true;

this.trigger(this.user);

},

})

// stores/loginStore.js

var loginStore = Reflux.createStore({

listenables: actions,

loginError: function(msg){

this.trigger(msg);

},

});

store接收action数据后，通过this.trigger(msg)将处理过后的数据重新传递会view：

var Login = React.createClass({

mixins : [

Router.Navigation,

Reflux.listenTo(userStore, 'onLoginSucc'),

Reflux.listenTo(loginStore, 'onLoginErr')

],

onLoginSucc: function(){

// 登录成功，跳转回首页

this.transitionTo('home');

},

onLoginErr: function (msg) {

// 登录失败，显示错误信息

this.setState({

errorMsg: msg,

});

},

...

});

至此，从用户点击登录到登录结果传回，整个流程数据在View->Action->Store->View中完成单向传递，这就是Flux架构的基本概念。

在完成登录后，API会将验证通过的JWT传回：

// server.go

token := jwt.New(jwt.SigningMethodHS256)

// Headers

token.Header["alg"] = "HS256"

token.Header["typ"] = "JWT"

// Claims

token.Claims["name"] = validUser.Name

token.Claims["mail"] = validUser.Mail

token.Claims["exp"] = time.Now().Add(time.Hour \* 72).Unix()

tokenString, err := token.SignedString([]byte(mySigningKey))

if err != nil {

c.JSON(200, gin.H{"code": 500, "msg": "Server error!"})

return

}

c.JSON(200, gin.H{"code": 200, "msg": "OK", "jwt": tokenString})

当登录之后的用户在profile页面发起getBalance请求时，存储于本地的jwt将一起传递，我这里采用Header的方式传递，具体取决于API端的协议：

// actions.js

actions.getBalance.listen(function(){

var jwt = localStorage.getItem('jwt');

req({

url: HOST+"/user/balance",

method: "post",

type: "json",

headers: {

'Authorization': "Bearer "+jwt,

},

success: function (resp) {

if (resp.code == 200) {

actions.updateProfile(resp.jwt);

}else{

actions.loginError(resp.msg);

}

}

})

})

而服务端面对任何需要验证权限的请求需要通过Token验证：

//server.go

token, err := jwt.ParseFromRequest(c.Request, func(token \*jwt.Token) (interface{}, error) {

b := ([]byte(mySigningKey))

return b, nil