Manual:RESTful Web Services (Chinese)

From Ext JS - Learn

Jump to: navigation, search

Summary: This article is all about using Ext to talk to RESTful Web Services, although it should also be (hopefully) useful to people who just want to learn a bit more about bending Ext's Ajax request capabilities to their will.

Author: Patrick Donelan (译者Frank Cheung、协助：朱古力豆)

Published: 2008-01-28

Ext Version: 2.0

Languages: English Chinese

重要消息：按照Ext Roadmap的规划， Ext 2.1（2008春天）已经完整支持REST了！不错的消息！

Contents

[hide]

REST与RESTful Web Services

表述性状态传送（**REST**）是一种架构上的风格。此术语由Roy Fielding（联合制定 HTTP标准联合作者之一）所创造。在他的博士论文的第五章中， 焦点的内容是关于*现代Web架构的设计底层原理和与其他架构风格所不同的地方*。

对于REST粗浅的理解可以这样地形容:你拥有**一些资源（Resources）**（概念化一些对象，就像数据库中实体）， 资源统一经由某些接口所暴露出来（web之上便是**HTTP协议**和五个常用的标准HTTP动词：**GET**、**POST**、**PUT**、**DELETE**和**OPTIONS**）。资源的状态**表述（Representations）**经统一的接口访问和操作。比如，要查看用户的账号你就要GET账号的状态表述，要更新它就要PUT一次新的表述，要移除它就要DELETE资源等等...

REST并不限制应用于HTTP上（HTTP一种基于消息的协议，驱动着整个互联网），而且纯技术而言，为实现REST风格，你的统一接口不一定要使用以上五个动词，也可达到真的"RESTful"。但目前大多数基于WEB的"RESTful"服务（web-based services）**却是**构建于采用这五个标准动词的HTTP协议，因此我们会着重关心。

我深受REST的影响缘起于Leonard Richardson和Sam Ruby有关该主题的权威书籍（07年期间）RESTful Web Services（O'Reilly出版）。欲对REST了解更深的话我就强烈推荐你阅读这本书。为不偏离本文主旨，我不打算讨论REST的理论和解释为什么要采用RESTful风格的Web Services（总之本文的侧重点不在这里），所以，笔者这里假设阁下已经权衡分析REST的利、弊并作出决定投身REST的革命中:)

A Disclaimer of Sorts

无论是ExtJS还是其他Ajax库，凡涉及RESTful Web Services之处并非神秘不可测，终究我们所研究的REST只是在利用HTTP协议通讯下的，一组最佳实践而已。好的消息是，在主流浏览器中，让Ajax成为可能的XmlHttpRequest对象均可完整地支持五种标准的HTTP动词。坏消息是几乎是所有的WEB开发者到最近才意识到REST的重要性，这样就导致了大多数的Ajax的库、框架到现在还是认为你只会发起GET 或POST的请求。有鉴于此，你就必须了解AJAX库的底层是怎么样的，去操作PUT/DELETE/OPTIONS的请求。总之，本文可被看做是进一步挖掘ExtJS API的教程，竭尽全力使你懂得API的原理（借此能对非REST的ExtJS用户有所帮助）。

HTTP方法

当需要在Ajax请求中指定某个HTTP方法是不太困难的，尤其在使用底层方法Ext.Ajax.request()方法的时候。由于我们多数是在较高级的API下工作的（像配置Ext.grid.GridPanel中的Ext.data.Store），所以接触Ext.Ajax.request方法的机会比较小，但并不说明在调用这种级别的API是麻烦的，相反这是很方便地让你控制Ext发出各种请求（当调试RESTful Web服务时，能更方便地直接生成这些请求，——不过你可能发现基于命令行的curl库会更为方便）。

以下是一些例子（使用Firebug的控制台来了解例子运行的状况并实时观察XHR正在发出的请求）。

从/users资源处获取（GET）列表:

```
Ext.Ajax.request({
 url: '/users',
 method: 'GET'
})
```

POST（提交）一篇新的文章到/articles资源:

```
Ext.Ajax.request({
 url: '/articles',
 method: 'POST',
```

```
 params: {
  author: 'Patrick Donelan',
  subject: 'RESTful Web Services'
 }
})
```


在/articles/restful-web-services 资源处，对文章PUT（执行）一次representation（表述性）的更新：

```
Ext.Ajax.request({
 url: '/articles/restful-web-services',
 method: 'PUT',
 params: {
  author: 'Patrick Donelan',
  subject: 'RESTful Web Services are easy with Ext!'
 }
})
```

DELETE（删除）位于/articles/rpc-is-the-best-web-architecture的资源：

```
Ext.Ajax.request({
 url: '/articles/rpc-is-the-best-web-architecture',
 method: 'DELETE',
 success: function(){ alert('Begone!'); }
})
```

HTTP状态代码
HTTP规范中定义了许多状态代码（Status Codes），用于代码表示HTTP传输过程是怎样的。下面都一些你熟悉的代码：

'404 Not Found' - 找不到请求的资源
'401 Unauthorized' - 请求需要用户认证
'200 OK' - 请求发生成功
也有可能是：

'500 Internal Server Error' - 服务器发生错误
'503 Service Unavailable' - web服务器有可能超负荷了
如果你对过去五年的WEB开发有所了解，你会发现为什么仅仅有这些Status Codes是情有可原的。话又说回来，HTML文件构成的静态站点只会让你遇到以上的status code。而由编程语言（如Perl、PHP、Ruby、Java等）动态生成的站点，你可能遇到其他更多的Status Codes。

综合来说：HTTP status codes（都是三位数字）可归纳为：

1xx - 信息 Informational（可能与你无关）
2xx - 成功 Success（例如'200 OK'和'201 Created'）
3xx - 转向 Redirection
4xx - 客户端错误 Client Error（例如一般的"400 Bas Request"）
5xx - 服务端错误 Server Error （例如"500 Internal Server Error"）

有人会问，有必要这么详细的分类吗？当然有，不过为了不涉及其他过多的话题我假设你懂得原因（参见文章底部的#延伸阅读）部分以了解更多的内容）。

使用HTTP状态代码的一个很好的理由就是（另外一个是因为我们是JavaScript的开发者）可以让我们无须理会请求的内容是什么、返回的Response的具体又是什么就可以处理分析这次的Response成功操作与否。

例如，脚本无须了解响应的消息体（response body）的格式究竟是什么就可得知请求成功与否（4xx有可能验证错误，2xx操作成功，5xx服务器不能工作）。

你完全可以在你的脚本中构建一个清晰而合理的方式与服务端通讯，而且你会越来越感觉与HTTP系统打交道是一个非常自然的事情（例如你脚本中有错误了那么服务端就可能反馈你一个5xx的状态代码）。

另外重要的一方面是，你会发现不需要将状态信息放到整个body身上，这样对于你(one less non-standard thing to document)和其他使用你Web服务的人都带来了方便，因为在统一的接口界面下（interface）他们能够利用现有的知识而不需要再学习你的语法。

牵涉一个相关的例子，就是，Ext API其中一部分我觉得要强调的是（尽管这里表达是良性的批评-见#A Disclaimer of Sorts），在服务端的响应中，BasicForm与Form的actions应接收某些特定的Ext参数，以便正确自动地作出表单验证和回调函数success/failure的处理。

例如，一次成功的请求应该会返回'200 OK'和下面的实体（尤其注意"success"属性）：

```
{
 "success": true,
 "data": {
  "field_id_1": "Field 1 Value",
  "field_id_2": "Field 2 Value"
 }
}
```

实体好像是多余的，因为你会觉得200的状态代码是代表成功的。

另外，即使请求不获验证的通过，也是返回'200 OK'（不返回4XX的代码，那样的意思是请求包含客户端无效的数据），实体会是这样的：

```
{
 "success": false,
 "errors": [
  { "id": "field_id_1", "msg": "Field 1 Error Message" },
```

```
  { "id": "field_id_2", "msg": "Field 2 Error Message" }
 ]
}
```

当中最大的问题是，如果你只是想提交表单，却在JSON/xml实体body中忘记包含Ext指定的属性"success: true"就会返回一个成功的status code但是空内容的body：

```
{
}
```

即是服务端告诉你这是一个成功的请求，form所属的提交的handler却不会执行相应的回调函数！ 这样会使得不熟悉ExtJS的程序员狂抓（或者像我这样的人有时会忘记强制性的"success"属性），为什么Ext会忽略那些状态代码。

总之，回到务实的态度上，你可以用Ext.form.BasicForm或Ext.form.Form提交Aajx表单，而不需要在实体上指定Ext特有的success属性，仍然可以提交如执行Ext.Ajax.request()时相类似，参数做了有关http状态码与回调success/failure的方面的事情：

```
Ext.Ajax.request({
 url: '/some_resource',
 method: 'POST',
 form: 'my-form-id',
 success: function(){alert('Must have been 2xx http status code')},
 failure: function(){alert('Must have been 4xx or a 5xx http status code')},
});
```

这意味着验证错误的信息将不能自动显示（却是一个好的功能）你可以在Ex底层代码中轻松地分离这一块的功能，若你遭遇到4xx的代码就直接调用该方法。下面我就我组件项目中的实际需求贴出来给大家看看，我先没有使用Ext.form.BasicForm因为我希望实现Http状态码感知（HTTP Status Code aware）！

Busting the Cache-buster

使用REST其中一个好处是，你会与HTTP一起工作而不是排斥它（又来RPC?!）比如，你很可能会依靠HTTP headers来控制内容的有效期和缓存。默认下，为每次得到刷新内容，Ext会在每次的GET中加入一个特别的"cache-buster"参数。要禁用这项功能，你可以在javascript代码中加入这一行(which those fond of double-negatives will especially enjoy)：

```
Ext.Ajax.disableCaching = false;
```

表述式的资源，文档类型（Content Types）和Accept: header

假设我们有一处表述式的资源位于 http://server.com/resource 你打算用GET方法获取它。 一些RESTful Web Service会根据你请求中HTTP头部"Accept:"字段以决定返回序列化的格式。 比如，请求是这样的：

```
GET /resource HTTP/1.1
Host: server.com
Accept: application/json
```

这是服务端会返回（JSON格式）：

```
[
 {a:1, b:2},
 {a:3, b:4}
]
```

若你的请求是这样的：

```
GET /resource HTTP/1.1
Host: server.com
Accept: */*
```

服务端就会返回默认的格式，如XML:

```
<opt>
 <data a="1" b="2" />
 <data a="3" b="4" />
</opt>
```

Ext中的一个常见的场景是，在获取所有的表述内容的都固定某种的格式（很可能是JSON）,最快的方法是在每个请求中设置一个缺省的头部（"Accept:" header）：

```
Ext.Ajax.defaultHeaders = {
 'Accept': 'application/json'
};
```

之后发生的所有请求将把"Accept:"的头部内容设置为"application/json"。 例如，我们发起一次GET的请求到/resource并观察Firebug 检查发出的headers：

```
Ext.Ajax.request({
 url: '/resource',
 params: {
  test: 1
 },
 method: 'GET'
})
```

得到：

```
Host: server.com
Accept: application/json
X-Requested-With: XMLHttpRequest
..
```

Data Stores，自定义HTTP Methods（HTTP方法）和Ext.data.JsonStore的实践例子

Ext具备清晰而完整对象层次。一个例子便是Ext.data.Store把Record对象"封装"为客户端缓存，用于某些组件，如GridPanel、ComboBox或DataView提取数据。下面会讲到用 Ext.data.JsonStore类动态生成 Ext.form.ComboBox组件，假设有一个Web Service，能够返回JSON格式的资源：

```
var cb = new Ext.form.ComboBox({
 store: new Ext.data.JsonStore({
  url: '/resource?query=something',
  fields: ['id']
  }),
 displayField: 'id',
 valueField: 'id',
 triggerAction: 'all',
 renderTo: document.body
})
```

要留意查询的参数那里（通常给人第一的感觉这应该是GET的请求），在EXT中实际还是使用了POST的方法获取数据！

如果我们就这样发送到一个RESTful的Web Service，你很可能就会发现返回的是"405 -Not Implemented"的HTTP状态码（假设资源仅支持GET）。 即使你的面对的不是一个RESTfule的Web Service，只要你的服务端语言能够分辨GET或POST的参数的话，你仍有机会使用GET，让服务器能够读取请求的参数。

Ext.data.JsonStore实际上来说是混合了Ext.data.Proxy与Ext.data.Reader的一个强大的Ext.data.Store类，减少了不必要的对象耦合。不过接下来我们不使用Ext.data.JsonStore而是重新定义我们的Ext.data.JsonStore,这是增加了几行代码但能更好地说明问题。 如果你经常使用这个store你可以重新定义个Ext.data.MorePowerfulJsonStore对象，将多个对象封装在内（instead of hard-wiring things）：

```
var cb = new Ext.form.ComboBox({
 store: new Ext.data.Store({
  proxy: new Ext.data.HttpProxy({
   url: '/resource',
   method: 'GET',
   params: {
        query: 'something'
   }
  }),
  reader: new Ext.data.JsonReader({
   fields: ['id']
  })
 }),
 displayField: 'id',
 valueField: 'id',
 triggerAction: 'all',
 renderTo: document.body
})
```

正如所见，通过定义Ext.data.HttpProxy可允许你指定任意一种HTTP方法。因此除了你习惯的GET方法获取Data Store对象，还可以使用其他任意你喜欢的HTTP方法。

HTTP认证（Authentication）
HTTP协议中相关的认证头部，都是可扩展和定义良好的字段，依靠这些字段来完成HTTP的认证机制（HTTP Authention scheme），是基于RESTful风格的Web Services的常见做法。实现的例子有HTTP Basic Authentication、HTTP Digest Authentication和Amazon的custom S3 authentication scheme（这些都是相应到公钥/密钥的概念）。

这是一个HTTP Basic 认证的应用例子，--通过Apacheht .htaccess文件保护目录/网站的账号和密码，假设我们处于http://mysite.com/protected_content 的资源是密码保护的：

```
GET /protected_content HTTP/1.1
Host: mysite.com
```

web service 返回以下头部（header）：

```
HTTP/1.1 401 Authorization Required
WWW-Authenticate: Basic
..
```

头部里的www-Authenticate项设置为Basic，即服务端希望您使用HTTP的Basic认证（WWW-Authenticate，这是应用在RESTful Web Services中的表示HTTP状态代码HTTP Status Codes的一个例子），现在，HTTP Basic认证可以在Authorization的位置观察到，用户名和密码base64编码后，前置一个"Basic"的字符串，用分号":"与Authorization相分隔开。假设现在你的用户名是"Aladdin"和密码是"open sesame"，那你所需要提交的认证是这样的：

```
GET /protected_content HTTP/1.1
Host: mysite.com
Authorization: Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==
```

服务器接受请求后，会对用户名和密码进行base64的解码（本例中你的密码明显是以普通文本去发送的，不过你亦可以是用HTTP的Basic认证来运行实际的web service）然后根据**Authorization**规则以决定应该返回什么内容（很希望接下来的是被保护的内容）。

理论上，只要在Authorzation头部进行相关设置，通过一个Ajax调用也可以完成一个这个的请求，达到相同的目的，下面就是这样一个例子，也是按照一般做法发起请求 如：

```
Ext.Ajax.defaultHeaders.Authorization = "Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==";
Ext.Ajax.request({
 url: '/protected_content',
 method: 'GET'
})
```

这样可以正常运行（另外很明显是你或者会定义一个快捷的函数来生成Authorization的字符串，论坛上有base64编码相关的帖子）。

有一个问题便是，一涉及到HTTP认证，浏览器就会自动自觉地提供帮助。如果你欲通过**WWW-Authenticate**对资源发送Basic的请求，那资源会返回401Authorization Required的状态代码，Web浏览器这时就会提示一个登陆的对话框，并会为你进行base64的字符编码，并保存在本地缓存中，方便在每一次请求都将这个Authorization加入头部中。本身来说就是比较方便的，但也存在下面相关的问题：

你不可以遵循你程序外观设置样式；
你不可以通过修改Ext.Ajax.defaultHeaders的属性来重写Authorization的头部，意味着你想以别的用户身份就不太方便；
你不能从浏览器缓存中删除用户自己信任条目，意味着你记录用户登出时不太方便。
由于是使用了Ajax无刷新的表单提交，即使你提交后假设服务器不返回**401 Authorization Required**的相应，浏览器并不会自己作进一步动作，而用户也会毫不知情。在使用RESTful风格的HTTP Basic认证效验机制下，我将用户在表单填好username/password定义到我之前安排好资源位置称作"/my_account" HTTP Basic 认证效验中，逻辑句柄会向我之前定义好的称作"/my_account"位置获取（GETS）带有Authorization头部的表征信息（respresentation）。如果这段respresention是空白的话，逻辑句柄就判断为安全效验的信任不获取通过。若得到的是一段通过用户的表征信息，那么则认为登陆成功，并设置Ext.Ajax.defaultHeaders.Authorization为已认证的字符串。

但是如果用户想绕过登录界面直接去访问受保护的资源时，会发生什么情形呢？可以看出，这对单页面的Ajax Web程序不存在问题，但如果说是传统的多页面的Web程序的场景，用户很可能会收藏这个资源地址直接访问。在单页面的Web程序也有可能调用一个RESTful的API（例如通过JavaScript或许会发送一个XHR的z请求获取一组用户列表，生成UI上的comboBox）。假设用户在浏览器地址栏输入/api/users将会得到**401 Authorization Required**的回应，显示登录的对话框并缓存结果，需要再次输入信息。所幸的是，XmlHttpRequst的设计者已经想到过这个问题，在请求的参数上加两个可选的参数，指定用户名称密码（亦进行base64的编码），**不过遗憾的是，当前标准的ExtJs Ajax调用并不支持这两个可选的参数。**直到有解决方案出现之前（我想这需要一点时间）你有这些可选方案：

1 在一些浏览器上在url后面加上用户名/密码：注意一些浏览器不支持（包括IE6以后的版本）；
2 借助Doug Hendricks优秀的ext-baseX.js库；
3 返回一个非标准的HTTP状态代码，而不是401Authorization Required这样浏览器就不会提示，例如你可返回403Forbidden典型把这个403的代码涉及到HTTP/101标准，的内容即是Authorization不会帮助而且不应重复要求，但是你会打算取巧地使用这种方法（不足的是使用其他的Web Service会有所限制）
Cookies与Sessions
虽然能使用Cookies，来方便地维护Session，但是由于这是把无态的协议切换为一个有态的协议，所以并不符合RESTful的风格，本文将不会集中谈论HTTP当中的缺失之处（若读者有兴趣了解，可阅读文章#延伸阅读部分）。这时如果在某些场合应用Cookies，如客户端的本地储存使用，仍不算违反RESTful的建议要求。

举一个例子，为了减少在不同的浏览器中来回输入登陆的信息，可以登录的信息，可在登录的表单加入"记住我"的单选框（checkbox），以方便用户下次登录。即使在多页面的ajax程序中你也需要这项功能来解决每次页面刷新后丢死记录的问题。

要实现这种方案你可以设置在Cookies中保存用户认证的详细信息：

```
Ext.state.Manager.setProvider(new Ext.state.CookieProvider({
 path: "/",
 expires: new Date(new Date().getTime()+(1000*60*60*24*14)) //remember for 2 weeks
}));
Ext.state.Manager.set('auth', "Basic QWxhZGRpbjpvcGVuIHNlc2FtZQ==");
```

然后对某个程序的cookie现场测试，你会发现就会怎么通过cookies保存认证信息了。

本质上这很明显是不安全的，任何人都可以访问包含用户名和密码的cookie缓存，要最小限度避免这种情况你应考虑使用更高级的HTTP认证机制，值得指出的是无数网站所普遍使用的是session-key-in-a-cookie（即Session的键名称值放在Cookie中），会很容易受到大量自身安全漏洞的攻击的，此类攻击诸如：Session hijacking、Cross-site request forgery等等。

完毕之标记
This article covers the topics where I've found myself heading slightly off the beaten path when using Ext to talk to my own RESTful Web Services. If you find other areas that aren't covered, as I'm sure will happen as REST becomes more widespread, please drop me a line as I'd love to hear how your own experiences with REST and Ext go.

延伸阅读
RESTful Web Services, by Leonard Richardson and Sam Ruby, 2007 (O'Reilly) - provides a definitive guide to all things REST, as well as the **Resource Oriented Architecture** which you can use to implement highly functional, real-world RESTful Web Services.
Roy Fielding's 2000 PhD Dissertation which first coined the term REST
curl library
HTTP Status Codes
REST on Wikipedia (contains links to other REST resources, no pun intended)