**RESTful api**

网络应用程序，分为前端和后端两个部分。当前的发展趋势，就是前端设备层出不穷（手机、平板、桌面电脑、其他专用设备......）。

因此，必须有一种统一的机制，方便不同的前端设备与后端进行通信。这导致API构架的流行，甚至出现"API First"的设计思想。RESTful API是目前比较成熟的一套互联网应用程序的API设计理论。

中文叫“表述性状态转移",在符合架构原理的前提下，理解和评估以网络为基础的应用软件的架构设计，得到一个功能强、性能好、适宜通信的架构。REST 指的是一组架构约束条件和原则。"如果一个架构符合 REST 的约束条件和原则，我们就称它为 RESTful 架构，REST 其实并没有创造新的技术、组件或服务，在我的理解中，它更应该是一种理念、一种思想，利用 Web 的现有特征和能力，更好地诠释和体现现有 web 标准中的一些准则和约束。

**基本原则一：**URL

* 应该将 api 部署在专用域名之下。
* URL 中尽量不用大写。
* URL中不应该出现动词，动词应该使用 HTTP 方法表示但是如果无法表示，也可使用动词，例如：search 没有对应的 HTTP 方法,可以在路径中使用 search，更加直观。
* URL中的名词表示资源集合，使用复数形式。
* URL可以包含 queryString，避免层级过深。

**基本原则二：HTTP 动词**

对于资源的具体操作类型，由 HTTP 动词表示，常用的 HTTP 动词有下面五个：

* GET：从服务器取出资源（一项或多项）。
* POST：在服务器新建一个资源。
* PUT：在服务器更新资源（客户端提供改变后的完整资源）。
* PATCH：在服务器更新资源（客户端提供改变的属性）。
* DELETE：从服务器删除资源。

还有两个不常用的 HTTP 动词：

* HEAD：获取资源的元数据。
* OPTIONS：获取信息，关于资源的哪些属性是客户端可以改变的

例子：

* 用户管理模块：
* 1. [POST] http：//lou.springboot.tech/users // 新增
* 2. [GET] http//lou.springboot.tech/users?page=1&rows=10 // 列表查询
* 3. [PUT] http：//lou.springboot.tech/users/12 // 修改
* 4. [DELETE] http：//lou.springboot.tech/users/12 // 删除

**基本原则三：状态码（Status Codes）**

处理请求后，服务端需向客户端返回的状态码和提示信息。

常见状态码**(状态码可自行设计，只需开发者约定好规范即可)**：

* 200：SUCCESS 请求成功。
* 401：Unauthorized 无权限。
* 403：Forbidden 禁止访问。
* 410：Gone 无此资源。
* 500：INTERNAL SERVER ERROR 服务器发生错误。 ...

**基本原则四：错误处理**

如果服务器发生错误或者资源不可达，应该向用户返回出错信息。

**基本原则五：服务端数据返回**

后端的返回结果最好使用 JSON 格式，且格式统一。

**基本原则六：版本控制**

* 规范的 api 应该包含版本信息，在 RESTful api 中，最简单的包含版本的方法是将版本信息放到 url 中，如：

[GET] http：//lou.springboot.tech/v1/users?page=1&rows=10

[PUT] http：//lou.springboot.tech/v1/users/12

* 另一种做法是，使用 HTTP header 中的 accept 来传递版本信息。

以下为接口安全原则的注意事项：

#### 安全原则一：Authentication 和 Permission

Authentication 指用户认证，Permission 指权限机制，这两点是使 RESTful api 强大、灵活和安全的基本保障。

常用的认证机制是 Basic Auth 和 OAuth，RESTful api 开发中，除非 api 非常简单，且没有潜在的安全性问题，否则，**认证机制是必须实现的**，并应用到 api 中去。Basic Auth 非常简单，很多框架都集成了 Basic Auth 的实现，自己写一个也能很快搞定，OAuth 目前已经成为企业级服务的标配，其相关的开源实现方案非常丰富。

#### 安全原则二：CORS

CORS 即 Cross-origin resource sharing，在 RESTful api 开发中，主要是为 js 服务的，解决调用 RESTful api 时的跨域问题。

由于固有的安全机制，js 的跨域请求时是无法被服务器成功响应的。现在前后端分离日益成为 web 开发主流方式的大趋势下，后台逐渐趋向指提供 api 服务，为各客户端提供数据及相关操作，而网站的开发全部交给前端搞定，网站和 api 服务很少部署在同一台服务器上并使用相同的端口，js 的跨域请求时普遍存在的，开发 RESTful api 时，通常都要考虑到 CORS 功能的实现，以便 js 能正常使用 api。

目前各主流 web 开发语言都有很多优秀的实现 CORS 的开源库，我们在开发 RESTful api 时，要注意 CORS 功能的实现，直接拿现有的轮子来用即可。

**实现：**

使用了 Ajax 技术进行后端接口的调用，但是返回结果的数据格式并不统一，这在实际的项目开发工作中一般不会出现，因此我们首先将返回结果进行抽象并封装。

新建 common 包，并封装 Result 结果类，代码如下

package com.ajaxuse.common;  
  
import org.springframework.util.StringUtils;  
  
import java.io.Serializable;  
  
public class Result*<*T*>* implements Serializable *{* //业务号码  
 private static final long *serialVersionUID*=1l;  
 private String message;  
  
 //返回信息  
 private int resultCode;  
 //数据结果  
 private T data;  
  
 public Result*(){}* public Result*(*int resultCode, String message*)  
 {* this.resultCode=resultCode;  
 this.message=message;  
 *}* public static long getSerialVersionUID*() {* return *serialVersionUID*;  
 *}* public String getMessage*() {* return message;  
 *}* public void setMessage*(*String message*) {* this.message = message;  
 *}* public int getResultCode*() {* return resultCode;  
 *}* public void setResultCode*(*int resultCode*) {* this.resultCode = resultCode;  
 *}* public T getData*() {* return data;  
 *}* public void setData*(*T data*) {* this.data = data;  
 *}* public Result failure*(*String code*) {* return new Result*(*500, "服务错误"*)*;  
 *}* @Override  
 public String toString*() {* return "Result{" +  
 "message='" + message + '\'' +  
 ", resultCode=" + resultCode +  
 ", data=" + data +  
 '}';  
 *}  
}*

每一次后端数据返回都会根据以上格式进行数据封装，包括业务码、返回信息、实际的数据结果，而不是像前一个实验中的不确定格式，前端接受到该结果后对数据进行解析，并通过业务码进行相应的逻辑操作，之后再将 data 中的数据获取到并进行页面渲染或者进行信息提示。

实际返回的数据格式如下：

* 列表数据

{

"resultCode": 200,

"message": "SUCCESS",

"data": [

{

"id": 2,

"name": "user1",

"password": "123456"

},

{

"id": 1,

"name": "13",

"password": "12345"

}

]

}

* 单条数据

{

"resultCode": 200,

"message": "SUCCESS",

"data": true

}

如上两个分别是列表数据和单条数据的返回，后端进行业务处理后将会返回给前端一串 json 格式的数据，resultCode 等于 200 表示数据请求成功，该字段也可以自行定义，比如 0、1001、500 等等，message 值为 SUCCESS，也可以自行定义返回信息，比如“获取成功”、“列表数据查询成功”等，这些都需要与前端约定好，一个码只表示一种含义，而 data 中的数据可以是一个对象数组、也可以是一个字符串、数字等类型，根据不同的业务返回不同的结果，之后的实践内容里都会以这种方式返回数据。

**后端接口实现**

按照 api 规范进行接口设计和接口调用，以对 bg\_users为例进行实践，在controller下新建 ApiController 类，代码如下：

@Controller  
@RequestMapping*(*"/api"*)*public class ApiController *{* static Map*<*Integer,Users*> usersMap*= Collections.*synchronizedMap(*new HashMap*<*Integer, Users*>())*;  
 //初始化usersmap  
 static *{* Users user = new Users*()*;  
 user.setId*(*2*)*;  
 user.setName*(*"user1"*)*;  
 user.setPassword*(*"123456"*)*;  
 Users user2 = new Users*()*;  
 user2.setId*(*5*)*;  
 user2.setName*(*"13-5"*)*;  
 user2.setPassword*(*"4"*)*;  
 Users user3 = new Users*()*;  
 user3.setId*(*6*)*;  
 user3.setName*(*"12"*)*;  
 user3.setPassword*(*"123"*)*;  
 *usersMap*.put*(*2, user*)*;  
 *usersMap*.put*(*5, user2*)*;  
 *usersMap*.put*(*6, user3*)*;  
 *}* //查询  
 @RequestMapping*(*value = "/users/{id}",method = RequestMethod.*GET)* @ResponseBody  
 public Result*<*Users*>* getOne*(*@PathVariable*(*"id"*)* Integer id*)  
 {* if *(*id == null || id < 1*) {* return ResultGenerator.genFailResult*(*"缺少参数"*)*;  
 *}* Users user = *usersMap*.get*(*id*)*;  
 if *(*user == null*) {* return ResultGenerator.genFailResult*(*"无此数据"*)*;  
 *}* return ResultGenerator.genSuccessResult*(*user*)*;  
 *}* // 删除一条记录  
 @RequestMapping*(*value = "/users/{id}", method = RequestMethod.*DELETE)* @ResponseBody  
 public Result*<*Boolean*>* delete*(*@PathVariable*(*"id"*)* Integer id*) {* if *(*id == null || id < 1*) {* return ResultGenerator.genFailResult*(*"缺少参数"*)*;  
 *}  
 usersMap*.remove*(*id*)*;  
 return ResultGenerator.genSuccessResult*(*true*)*;  
 *}  
  
}*

根据前端不同的资源请求，我们按照前文中 HTTP 动词的要求对接口的请求类型进行设置，用户数据查询方法使用 GET 请求，用户添加方法 使用 POST 请求，对应的修改和删除操作使用 PUT 和 DELETE 请求，同时对于 api 的请求路径也按照设计规范进行设置，虽然有些映射路径相同，但是会根据请求方法进行区分。

比如：同样是 /users 路径，如果请求方法为 POST 则表示添加资源会调用 insert() 方法，而请求方法为 PUT 时则表示修改资源会调用 update() 方法，还有 /users/{id} 路径，会根据 GET 请求方式和 DELETE 请求方式进行区分表示获取单个资源和删除单个资源。

同时，每一个返回结果我们都统一使用 Result 类进行包装之后再返回给前端，并使用 @ResponseBody 注解将其转换为 json 格式。

**前端页面和 js 方法实现**

接口定义完成后就可以进行前端页面和接口调用逻辑的实现了，新建 api-test.html（注：代码位于resources/static），代码如下：

<!DOCTYPE html>

<html>

<head>

<meta charset="utf-8" />

<meta http-equiv="X-UA-Compatible" content="IE=edge" />

<title>lou.SpringBoot | api 请求测试</title>

</head>

<body class="hold-transition login-page">

<div style="width:720px;margin:7% auto">

<div class="content">

<div class="container-fluid">

<div class="row">

<div class="col-lg-6">

<hr />

<div class="card">

<div class="card-header">

<h5 class="m-0">详情查询接口测试</h5>

</div>

<div class="card-body">

<input

id="queryId"

type="number"

placeholder="请输入id字段"

/>

<h6 class="card-title">查询接口返回数据如下：</h6>

<p class="card-text" id="result0"></p>

<a href="#" class="btn btn-primary" onclick="requestQuery()"

>发送详情查询请求</a

>

</div>

</div>

<br />

<hr />

<div class="card">

<div class="card-header">

<h5 class="m-0">列表查询接口测试</h5>

</div>

<div class="card-body">

<h6 class="card-title">查询接口返回数据如下：</h6>

<p class="card-text" id="result1"></p>

<a

href="#"

class="btn btn-primary"

onclick="requestQueryList()"

>发送列表查询请求</a

>

</div>

</div>

<br />

<hr />

<div class="card">

<div class="card-header">

<h5 class="m-0">添加接口测试</h5>

</div>

<div class="card-body">

<input id="addId" type="number" placeholder="请输入id字段" />

<input

id="addName"

type="text"

placeholder="请输入name字段"

/>

<input

id="addPassword"

type="text"

placeholder="请输入password字段"

/>

<h6 class="card-title">添加接口返回数据如下：</h6>

<p class="card-text" id="result2"></p>

<a href="#" class="btn btn-primary" onclick="requestAdd()"

>发送添加请求</a

>

</div>

</div>

<br />

<hr />

<div class="card">

<div class="card-header">

<h5 class="m-0">修改接口测试</h5>

</div>

<div class="card-body">

<input

id="updateId"

type="number"

placeholder="请输入id字段"

/>

<input

id="updateName"

type="text"

placeholder="请输入name字段"

/>

<input

id="updatePassword"

type="text"

placeholder="请输入password字段"

/>

<h6 class="card-title">修改接口返回数据如下：</h6>

<p class="card-text" id="result3"></p>

<a href="#" class="btn btn-primary" onclick="requestUpdate()"

>发送修改请求</a

>

</div>

</div>

<br />

<hr />

<div class="card">

<div class="card-header">

<h5 class="m-0">删除接口测试</h5>

</div>

<div class="card-body">

<input

id="deleteId"

type="number"

placeholder="请输入id字段"

/>

<h6 class="card-title">删除接口返回数据如下：</h6>

<p class="card-text" id="result4"></p>

<a href="#" class="btn btn-primary" onclick="requestDelete()"

>发送删除请求</a

>

</div>

</div>

<hr />

</div>

</div>

</div>

</div>

</div>

</body>

</html>

本功能主要包括对于 bg\_users表增删改查功能的调用和结果显示，每个接口测试模块包括信息录入的 input 框和发送请求按钮以及结果显示的 div，点击不同的按钮分别触发不同的 js 方法，我们计划在 js 方法中使用 Ajax 对后端发送不同的请求，实现如下：

<!-- jQuery -->

<script src="https://cdn.staticfile.org/jquery/1.12.0/jquery.min.js"></script>

<script type="text/javascript">

function requestQuery() {

var id = $('#queryId').val();

if (typeof id == 'undefined' || id == null || id == '' || id < 0) {

return false;

}

$.ajax({

type: 'GET', //方法类型

dataType: 'json', //预期服务器返回的数据类型

url: '/api/users/' + id,

contentType: 'application/json; charset=utf-8',

success: function (result) {

$('#result0').html(JSON.stringify(result));

},

error: function () {

$('#result0').html('接口异常，请联系管理员！');

},

});

}

function requestQueryList() {

$.ajax({

type: 'GET', //方法类型

dataType: 'json', //预期服务器返回的数据类型

url: '/api/users',

contentType: 'application/json; charset=utf-8',

success: function (result) {

$('#result1').html(JSON.stringify(result));

},

error: function () {

$('#result1').html('接口异常，请联系管理员！');

},

});

}

function requestAdd() {

var id = $('#addId').val();

var name = $('#addName').val();

var password = $('#addPassword').val();

var data = { id: id, name: name, password: password };

$.ajax({

type: 'POST', //方法类型

dataType: 'json', //预期服务器返回的数据类型

url: '/api/users',

contentType: 'application/json; charset=utf-8',

data: JSON.stringify(data),

success: function (result) {

$('#result2').html(JSON.stringify(result));

},

error: function () {

$('#result2').html('接口异常，请联系管理员！');

},

});

}

function requestUpdate() {

var id = $('#updateId').val();

var name = $('#updateName').val();

var password = $('#updatePassword').val();

var data = { id: id, name: name, password: password };

$.ajax({

type: 'PUT', //方法类型

dataType: 'json', //预期服务器返回的数据类型

url: '/api/users',

contentType: 'application/json; charset=utf-8',

data: JSON.stringify(data),

success: function (result) {

$('#result3').html(JSON.stringify(result));

},

error: function () {

$('#result3').html('接口异常，请联系管理员！');

},

});

}

function requestDelete() {

var id = $('#deleteId').val();

if (typeof id == 'undefined' || id == null || id == '' || id < 0) {

return false;

}

$.ajax({

type: 'DELETE', //方法类型

dataType: 'json', //预期服务器返回的数据类型

url: '/api/users/' + id,

contentType: 'application/json; charset=utf-8',

success: function (result) {

$('#result4').html(JSON.stringify(result));

},

error: function () {

$('#result4').html('接口异常，请联系管理员！');

},

});

}

</script>

每个请求按钮点击后会触发不同的 js 方法，以修改请求为例，该按钮点击后会触发 requestUpdate() 方法，该方法中首先会获取用户输入的数据，之后将其封装到 data 中，同时设置请求的 url 为 /api/users，由于是修改用户数据，因此将请求方法设置为 PUT 方法，之后向后端发送请求，而后端代码在前文中已经介绍，请求地址为 /api/users 且请求方法为 PUT 时会调用修改方法将用户数据进行修改。

删除请求也是如此，首先用户输入一个需要删除的 id，之后根据该 id 将其拼接到 url 中，同时设置请求的 url 为 /api/users/{id}，由于是删除用户数据，因此将请求方法设置为 DELETE 方法，之后向后端发送请求，后端在收到请求后会根据请求方法和请求地址进行映射匹配，可以通过后端代码得知，该请求最终会被控制器中的 delete() 方法处理