案例：<https://gitee.com/zx19890628/spring-boot-example/tree/master/lab_008_web_Jersey>

# 一、REST是什么 ?

RESTful Web服务是基于REST架构的Web服务。在REST架构一切都是一种资源。

RESTful Web服务是轻量级的，高度可扩展性和可维护性，并且非常常用于创建基于API的Web应用程序。

REST表示代表性状态传输。REST是一种基于Web标准的架构，并使用HTTP协议进行数据通信。

它围绕着资源，其中每一个组件是资源和一个资源是由一个共同的接口，使用HTTP的标准方法获得。

REST最初是由Roy Fielding在2000年推出。

在REST架构中，REST服务器只是提供的资源以及在REST客户端访问时提供资源。这里每个资源由URI标识/全局标识。REST采用各种申述表示类似文本，JSON和XML资源。现在JSON是Web服务所使用的最流行的格式。

# 二、HTTP 方法

下列著名的HTTP方法是常用于基于REST的架构。

GET - 提供的资源的只读访问。

PUT - 用于创建一个新的资源。

DELETE - 用于删除资源。

POST - 用于更新现有资源或创建新的资源。

OPTIONS - 用于得到支持的操作上的资源。

# 三、HTTP请求中参数的类型

1、header：就是写在请求头中的参数，多用于一些验证信息，如token,cookie。

2、query ：写在url后面的参数，url=http://www.baidu.com?a=1&b=2，a和b就是query参数。

3、path ：写在url里面的参数，url=http://www.baidu.com/user/{id}，参数就是id=3,用于替换路径。

4、body ：请求参数的数据包，一个http请求中只能有一个body,多用和post和put请求。

# 四、RESTful Web服务

Web服务是用于交换应用程序或系统之间的数据开放的协议和标准的集合。

编写各种编程语言和运行在不同平台上的软件应用程序可以使用Web服务来以类似于进程间通信在单台计算机上的方式通过计算机网络进行交换数据，如因特网。

这份互联特性（如Java和Python，或Windows和Linux应用程序之间），是由于使用开放标准。

基于REST架构的Web服务称为RESTful web服务。这些Web服务使用HTTP方法来实现REST架构的概念。

RESTful Web服务通常定义URI，统一资源标识符的服务，提供资源的表示，如JSON，并设置HTTP方法。

# 五、创建RESTful Web服务

以下是创建一个Web服务使用的用户管理功能：

No. HTTP方法 URI 操作 操作类型

1 GET /UserService/users 获取用户列表 只读

2 GET /UserService/users/1 读取用户ID为1用户信息 只读

3 PUT /UserService/users/2 插入Id为2的用户信息 等幂

4 POST /UserService/users/2 更新Id为2的用户信息 N/A

5 DELETE /UserService/users/1 删除Id为1的用户信息 等幂

6 OPTIONS /UserService/users 列出Web服务支持的操作 只读

本项目使用Jersey框架创建基于REST的Web服务。Jersey框架实现JAX-RS2.0 API，这是建立标准规范REST Web服务。

# 六、创建简单的RestFul服务

1.下载

http://repo1.maven.org/maven2/org/glassfish/jersey/bundles/jaxrs-ri/

或是https://jersey.github.io/download.html

这里下载的是jaxrs-ri-2.17.zip，用于window下开发。

2.创建一个java web项目。

将api、ext、lib下所有的lib包添加到项目中

例如：20170827\_Jersey项目中

通过创建User.java UserDao.java Server.java

并配置web.xml，创建一个简单的restFul服务。但是官方提供的jar只能实现XML数据的传输。

# 七、创建应用级的RestFul服务

高级的应用参考20170902\_Jersey，使用maven搭建环境。

Jersey是JAX-RS（JSR311）开源参考实现用于构建RESTful Web service，它包含三个部分：

1.核心服务器（Core Server） 通过提供JSR 311中标准化的注释和API标准化，可以用直观的方式开发RESTful Web服务；

2.核心客户端（Core Client） Jersey客户端API能够帮助开发者与RESTful服务轻松通信；

3.集成（Integration） Jersey还提供可以轻松继承Spring、Guice、Apache Abdera的库。

1.maven环境

<build>

<plugins>

<plugin>

<groupId>org.apache.tomcat.maven</groupId>

<artifactId>tomcat7-maven-plugin</artifactId>

<version>2.1</version>

<configuration>

<port>8080</port>

<path>/jersey</path>

<uriEncoding>UTF-8</uriEncoding>

<finalName>jersey</finalName>

<server>tomcat7</server>

</configuration>

</plugin>

</plugins>

<finalName>jersey</finalName>

</build>

<dependencies>

<dependency><!--jersey 核心服务-->

<groupId>org.glassfish.jersey.containers</groupId>

<artifactId>jersey-container-servlet-core</artifactId>

<version>2.0</version>

</dependency>

<dependency><!--JAXB API 用于发布接口-->

<groupId>javax.xml.ws</groupId>

<artifactId>jaxws-api</artifactId>

<version>2.1</version>

</dependency>

<dependency><!-- Json支持，在Jersey2.0中需要使用Jackson1.9才能支持json。 -->

<groupId>org.codehaus.jackson</groupId>

<artifactId>jackson-core-asl</artifactId>

<version>1.9.12</version>

</dependency>

<dependency>

<groupId>org.codehaus.jackson</groupId>

<artifactId>jackson-mapper-asl</artifactId>

<version>1.9.12</version>

</dependency>

<dependency>

<groupId>org.codehaus.jackson</groupId>

<artifactId>jackson-jaxrs</artifactId>

<version>1.9.12</version>

</dependency>

</dependencies>

2.简单的资源

@Path("/hello")

public class HelloResource {

@GET

@Produces(MediaType.TEXT\_PLAIN)

public String sayHello() {

return "Hello Jersey";

}

}

3.编写JAX-RS application

//Jersey资源配置

public class APIApplication extends ResourceConfig {

public APIApplication() {

//加载Resource

register(HelloResource.class);

//注册数据转换器

register(JacksonJsonProvider.class);

// Logging.

register(LoggingFilter.class);

}

}

4.web.xml中配置jersey servlet的入口，以及配置初始化JAX-RS application。

<!--用于定义 RESTful Web Service 接口 -->

<servlet>

<servlet-name>JerseyServlet</servlet-name>

<servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>

<init-param>

<param-name>javax.ws.rs.Application</param-name>

<param-value>com.zx.APIApplication</param-value>

</init-param>

<load-on-startup>1</load-on-startup>

</servlet>

<servlet-mapping>

<servlet-name>JerseyServlet</servlet-name>

<url-pattern>/rest/\*</url-pattern>

</servlet-mapping>

5.maven build ———>tomcat7:run

6.浏览器：http://localhost:8080/jersey/rest/hello 显示Hello Jerse

# 八、注解

## 1.@Produces

就是发送到客户端的数据标识类型（MIME）。

常用方式：

返回xml：@Produces(MediaType.APPLICATION\_XML)

返回json：@Produces(MediaType.APPLICATION\_JSON)

返回自定义返回类型及编码：@Produces("application/json;charset=utf-8")

返回多个类型：@Produces({“application/json”,“application/xml”})

## @Consumes

就是接受只客户端的的数据标识类型（MIME）。多用于post和put请求。

用法和@Produces响应

## @PathParam

可以获取URI中指定规则的参数。

@Path("/user/{id}")

@GET

@Produces(MediaType.APPLICATION\_XML)

public User getUserById(@PathParam("id") Integer id){

return this.userDao.getById(id);

}

## @QueryParam

用于获取GET请求中的查询参数。

@GET

@Path("/user")

@Produces("text/plain")

public User getUser(@QueryParam("name") String name, @QueryParam("age") int age) {
}

## @DefaultValue

为参数设置默认值

@GET

@Path("/user")

@Produces("text/plain")

public User getUser(@DefaultValue("张三") @QueryParam("name") String name) {
}

## @FormParam

从POST请求的表单参数中获取数据

@POST

@Consumes("application/x-www-form-urlencoded")

public void post(@FormParam("name") String name) {
}

## @BeanParam

将请求中的参数，保存为对象中。

@POST

@Consumes("application/x-www-form-urlencoded")

public void post(@BeanParam  User user) {
}

## @Context

因为参数的多变，参数结构的调整都会因为以上几种方式而遇到问题，这时可以考虑使用@Context 注释，并获取UriInfo实例

@GET

public String get(@Context UriInfo ui) {

MultivaluedMap<String, String> queryParams = ui.getQueryParameters();
MultivaluedMap<String, String> pathParams = ui.getPathParameters();

}

同样还可以通过@Context 注释获取ServletConfig、ServletContext、HttpServletRequest、HttpServletResponse和HttpHeaders等

@Path("/")

public class Resource {

@Context

HttpServletRequest req;

@Context

ServletConfig servletConfig;

@Context

ServletContext servletContext;

@GET

public String get(@Context HttpHeaders hh) {
MultivaluedMap<String, String> headerParams = hh.getRequestHeaders();
Map<String, Cookie> pathParams = hh.getCookies();

}

}

## @XmlRootElement

数据为XML是，用于和Javabean绑定，

@XmlRootElement

public class User{

private String name;

@XmlElement(name = "user\_name")//这个必须写在set方法上。

public void setName(String name) {

this.name = name;

}

}

## 注解列表

|  |  |
| --- | --- |
| S.N. | 注解 & 描述 |
| 1 | **@Path** 资源类/方法的相对路径。 |
| 2 | **@GET** HTTP GET请求，用于获取资源。 |
| 3 | **@PUT** HTTP PUT请求，用于建立资源。 |
| 4 | **@POST** HTTP POST请求，用于创建/更新资源。 |
| 5 | **@DELETE** HTTP DELETE请求，用于删除的资源。 |
| 6 | **@HEAD** HTTP HEAD请求，用来获取方法可用状态。 |
| 7 | **@Produces** 状态通过Web服务产生的HTTP响应，例如APPLICATION/XML, TEXT/HTML, APPLICATION/JSON等。 |
| 8 | **@Consumes** 状态的HTTP请求类型，例如 application/x-www-form-urlencoded 接受POST请求的过程中HTTP主体表单数据。 |
| 9 | **@PathParam** 结合传递到方法，在路径中的值的参数。 |
| 10 | **@QueryParam** 结合传递给方法的路径中的查询参数的参数。 |
| 11 | **@MatrixParam** 结合传递给方法的路径中的HTTP矩阵参数的参数。 |
| 12 | **@HeaderParam** 结合传递到方法到HTTP头中的参数。 |
| 13 | **@CookieParam** 结合传递给方法一个Cookie参数。 |
| 14 | **@FormParam** 结合传递到方法的一种形式值的参数。 |
| 15 | **@DefaultValue** 分配一个默认值，以传递一个参数给方法。 |
| 16 | @Context 上下文中资源，例如HTTP请求的上下文。 |

# 九、配置中心

public class APIApplication extends ResourceConfig {

public APIApplication() {

packages(Server.class.getPackage().getName());

//扫描包下的所有class，用于加载的注解，加载与Server同一个包的类

// packages("com.zx");//需要扫描的包

// 注册数据转换器

register(JacksonJsonProvider.class);

// Logging日志

register(LoggingFilter.class);

//单独加载一个类

//register(OptionResult.class);

// 注册上传/下载

register(MultiPartFeature.class);

}

}

# 十、REST的优点

1. 可更高效利用缓存来提高响应速度

2. HTTP连接是无状态的（也就是不记录每个连接的信息），而REST传输会包含应用的所有状态信息，因此可以大幅降低对HTTP连接的重复请求资源消耗。

3. 浏览器即可作为客户端，简化软件需求

4. 相对于其他叠加在[HTTP协议](http://zh.wikipedia.org/wiki/%E8%B6%85%E6%96%87%E6%9C%AC%E4%BC%A0%E8%BE%93%E5%8D%8F%E8%AE%AE)之上的机制，REST的软件依赖性更小

5. 不需要额外的资源发现机制

6. 在软件技术演进中的长期的兼容性更好

# 十一、json与xml的JavaBean

## 1.XML对应的bean

@XmlRootElement//这个是必须有的注解

**public** **class** User **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

**private** **int** id;

**private** String name;

**private** String profession;

**public** **int** getId() {

**return** id;

}

@XmlElement(name = "user\_id")//这个注解没有的话，就会解析为id

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

@XmlElement(name = "user\_name")

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getProfession() {

**return** profession;

}

@XmlElement(name = "user\_acl")

**public** **void** setProfession(String profession) {

**this**.profession = profession;

}

}

## 2.json对应的bean

**public** **class** Person **implements** Serializable {

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

@JsonProperty("user\_id")//这个注解与@XmlElement作用相同

**private** **int** id;

@JsonProperty("user\_name")

**private** String name;

@JsonProperty("user\_acl")

**private** String profession;

**public** **int** getId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

}

**public** String getProfession() {

**return** profession;

}

**public** **void** setProfession(String profession) {

**this**.profession = profession;

}

}

# 十二、文件上传下载

### 1.增加maven依赖

<!-- 上传文件需要该依赖-->

<dependency>

<groupId>org.glassfish.jersey.media</groupId>

<artifactId>jersey-media-multipart</artifactId>

<version>2.0</version>

</dependency>

<!-- 这个用于上传文件工具操作-->

<dependency>

<groupId>commons-io</groupId>

<artifactId>commons-io</artifactId>

<version>2.4</version>

</dependency>

## Jersey资源配置

**public** APIApplication() {

...........

// 注册上传/下载

register(MultiPartFeature.**class**);

}

## 上传代码

@POST

@Path("/import")

@Consumes(MediaType.***MULTIPART\_FORM\_DATA***)

@Produces(MediaType.***APPLICATION\_JSON***)

**public** **void** upload(

@FormDataParam("file") InputStream fileInputStream,

@FormDataParam("file") FormDataContentDisposition disposition){

String fileName = disposition.getFileName();

**try** {

//使用common io的文件写入操作

FileUtils.*copyInputStreamToFile*(fileInputStream, **new** File("D://" + fileName));

} **catch** (IOException ex) { }

}

## 下载代码

@GET

@Path("/download")

@Produces(MediaType.***APPLICATION\_OCTET\_STREAM***)

**public** Response download(){

File file = **new** File("D://1.jpg");

//如果文件不存在，提示404

**if**(!file.exists()){

**return** Response.*status*(Response.Status.***NOT\_FOUND***).build();

}

String fileName = **null**;

**try** {

fileName = URLEncoder.*encode*("测试.jpg", "UTF-8");

} **catch** (UnsupportedEncodingException e1) {

e1.printStackTrace();

}

**return** Response.*ok*(file)

.header("Content-disposition","attachment;filename=" +fileName)

.header("Cache-Control", "no-cache").build();

}

## 5.html

<h1>上传文件</h1>

<form action=*"rest/server/import"*

method=*"post"* enctype=*"multipart/form-data"*>

<input name=*"file"* type=*"file"*/><input type=*"submit"* value=*"提交"*/>

</form>

<a href=*"rest/server/download"*>下载图片</a>

# 十三、RESTful Web Services缓存

缓存是指存储服务器响应客户端本身，使客户端无须做出了对同样的资源服务器的请求连连。服务器的响应有关于缓存是如何做到使客户端缓存的响应一段时间或永远捕捉服务器响应信息。

## 1.响应头

以下是其中一个服务器响应可以具有以配置客户端的缓存的报头：

|  |  |
| --- | --- |
| **S.N.** | **消息头和说明** |
| 1 | **Date** 日期和创建时的资源的时间。 |
| 2 | **Last Modified** 资源的日期和时间，它的最后修订。 |
| 3 | **Cache-Control** 主要头控制缓存。 |
| 4 | **Expires** 到期日和缓存时间 |
| 5 | **Age** 在几秒钟时间，从资源服务器获取时。 |

Cache-Control头

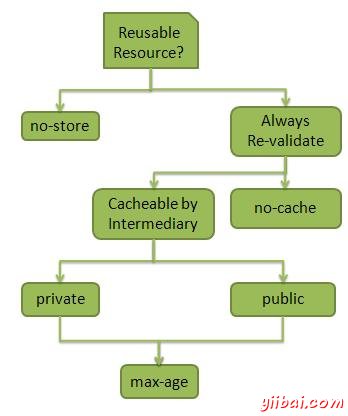
|  |  |
| --- | --- |
| **S. N.** | **指令和说明** |
| 1 | **Public** 指示资源是可缓存任何组件。 |
| 2 | **Private** 指示资源是可缓存只有客户端和服务器，无中介可以缓存资源。 |
| 3 | **no-cache/no-store** 表示资源是不可缓存 |
| 4 | **max-age** 表示缓存有效期为max-age以秒为单位。在此之后，客户端必须使另一个请求。 |
| 5 | **must-revalidate** 指示服务器来验证资源，如果最大到期时间已经过去了。 |

## 最佳实践

（1）.始终保持静态内容，如图像，CSS，JavaScript的缓存，以2〜3天有效期。

（2）.不要让过期日期太长。

（3）.动态内容应只能缓存几个小时。



# 十四、RESTful Web服务安全

正如RESTfulWeb服务以HTTP URL的路径工作，所以这是非常重要的，在相同的方式维护一个RESTful Web服务的网站是安全的。

以下是在设计一个RESTful Web服务应遵循的最佳实践。

## 1.最佳实践

（1）验证 - 验证服务器上的所有输入。防止SQL或NoSQL注入攻击你的服务器。

（2）基于会话的认证 - 使用基于会话的认证，每当一个请求到Web服务的方法来验证用户的身份。

（3）URL没有敏感数据 - 千万不要使用用户名，密码或会话令牌的URL，这些值应通过POST方法传递给Web服务。

（4）方法执行限制 - 允许限制使用像GET，POST，DELETE等方法。 GET方法不应该用于删除数据。

（5）验证格式错误 XML/JSON - 检查以及表单输入传递到Web服务方法。

（6）抛出通用错误信息 - Web服务方法应使用HTTP错误信息像403，显示禁止访问等。

## 响应码

|  |  |
| --- | --- |
| S.N. | HTTP 代码 & 描述 |
| 1 | 200 正确/正常, 显示成功。 |
| 2 | 201 创建，当使用POST或PUT请求成功创建了资源。返回链接使用位置标头新创建的资源。 |
| 3 | 204 无内容, 当响应主体是空的，例如，一个DELETE请求。 |
| 4 | 304 不修改, 用于减少以防条件GET请求的网络带宽的使用。响应主体应该是空的。头应该有日期，位置等。 |
| 5 | 400 错误的请求, 状态表示无效输入，例如提供验证错误，数据丢失。 |
| 6 | 401 未经授权, 指出用户正在使用无效或错误的身份验证令牌。 |
| 7 | 403 被禁止, 指出用户没有访问方法的使用，例如，删除权限没有管理员权限。 |
| 8 | 404 未找到, 状态表示这个方法不可用。 |
| 9 | 409 冲突, 执行方法，例如，添加重复条目状态的冲突的情况。 |
| 10 | 500 内部服务器错误,状态该服务器已经在执行方法抛出了一些异常。 |

# Jersey Client

对用于测试和在其他java引用中调用restAPI

**public** **class** WebServiceTester {

**private** Client client;

**private** String REST\_SERVICE\_URL = "http://localhost:8080/jersey/rest/users";

**private** **static** **final** String ***SUCCESS\_RESULT*** = "<result>success</result>";

**private** **static** **final** String ***PASS*** = "pass";

**private** **static** **final** String ***FAIL*** = "fail";

**private** **void** init() {

**this**.client = ClientBuilder.*newClient*();

}

**public** **static** **void** main(String[] args) {

WebServiceTester tester = **new** WebServiceTester();

// initialize the tester

tester.init();

// test get all users Web Service Method

tester.testGetAllUsers();

// test get user Web Service Method

tester.testGetUser();

// test update user Web Service Method

tester.testUpdateUser();

// test add user Web Service Method

tester.testAddUser();

// test delete user Web Service Method

tester.testDeleteUser();

}

// Test: Get list of all users

// Test: Check if list is not empty

**private** **void** testGetAllUsers() {

GenericType<List<User>> list = **new** GenericType<List<User>>() {

};

List<User> users = client.target(REST\_SERVICE\_URL).request(MediaType.***APPLICATION\_XML***).get(list);

String result = ***PASS***;

**if** (users.isEmpty()) {

result = ***FAIL***;

}

System.***out***.println("Test case name: testGetAllUsers, Result: " + result);

}

// Test: Get User of id 1

// Test: Check if user is same as sample user

**private** **void** testGetUser() {

User sampleUser = **new** User();

sampleUser.setId(1);

User user = client.target(REST\_SERVICE\_URL).path("/{userid}")

.resolveTemplate("userid", 1)

.request(MediaType.***APPLICATION\_XML***).get(User.**class**);

String result = ***FAIL***;

**if** (sampleUser != **null** && sampleUser.getId() == user.getId()) {

result = ***PASS***;

}

System.***out***.println("Test case name: testGetUser, Result: " + result);

}

// Test: Update User of id 1

// Test: Check if result is success XML.

**private** **void** testUpdateUser() {

Form form = **new** Form();

form.param("id", "1");

form.param("name", "suresh");

form.param("profession", "clerk");

String callResult = client.target(REST\_SERVICE\_URL).request(MediaType.***APPLICATION\_XML***)

.post(Entity.*entity*(form, MediaType.***APPLICATION\_FORM\_URLENCODED\_TYPE***), String.**class**);

String result = ***PASS***;

**if** (!***SUCCESS\_RESULT***.equals(callResult)) {

result = ***FAIL***;

}

System.***out***.println("Test case name: testUpdateUser, Result: " + result);

}

// Test: Add User of id 2

// Test: Check if result is success XML.

**private** **void** testAddUser() {

Form form = **new** Form();

form.param("id", "2");

form.param("name", "naresh");

form.param("profession", "clerk");

String callResult = client.target(REST\_SERVICE\_URL).request(MediaType.***APPLICATION\_XML***)

.put(Entity.*entity*(form, MediaType.***APPLICATION\_FORM\_URLENCODED\_TYPE***), String.**class**);

String result = ***PASS***;

**if** (!***SUCCESS\_RESULT***.equals(callResult)) {

result = ***FAIL***;

}

System.***out***.println("Test case name: testAddUser, Result: " + result);

}

// Test: Delete User of id 2

// Test: Check if result is success XML.

**private** **void** testDeleteUser() {

String callResult = client.target(REST\_SERVICE\_URL).path("/{userid}").resolveTemplate("userid", 2)

.request(MediaType.***APPLICATION\_XML***).delete(String.**class**);

String result = ***PASS***;

**if** (!***SUCCESS\_RESULT***.equals(callResult)) {

result = ***FAIL***;

}

System.***out***.println("Test case name: testDeleteUser, Result: " + result);

}

}