RESTfulWeb服务教

极客学院出版

前言

RESTful Web 服务就是基于 REST 架构的 Web 服务。在 REST 架构中一切都是资源。RESTful Web 服务是轻量级的，高度可伸缩和可维护的，通常用于给基于 Web 的应用程序创建 APIs。

本教程将教会我们 RESTful Web 服务的基础知识，还包含讨论所有 RESTful Web 服务基本组成部分的章节和适当的例子。

适用人群

本教程是为那些愿意按照简易的步骤学习 RESTful Web 服务器的专业软件开发人员设计的。本教程会为我们理解 RESTful Web 服务提供很大的帮助，完成本教程之后你会处在专业水平的中级，然后可以自己提升到更高的水平。

学习前提

在继续学习本教程之前，你应该对 Java 语言，文本编辑器等有一个基本的了解。因为我们将会使用 RESTful 开发 Web 服务应用程序，如果你熟悉其他 Web 技术比如 HTML，CSS，AJAX 等等，这会更有利。

更新日期	更新内容
2015-05-26	第一版发布

目录

1

RESTful Web 服务 – 介绍

什么是 REST?

REST 是 REpresentational State Transfer 的缩写。REST 是一种基于 Web 标准的软件架构，它使用 HTTP 协议处理数据通信。它以资源为中心，其中每个组成部分都是一个资源，并且资源通过使用 HTTP 标准方法的公共接口访问。REST 由 Roy Fielding 在 2000 年首次提出。

在 REST 架构中，一个 REST 服务器只提供对资源的访问，REST 客户端访问并呈现资源。这里每个资源都通过 URIs/ 全局 ID 标识。REST 使用各种不同的表现形式表示资源，比如文本，JSON 和 XML。目前，JSON 是用于 Web 服务最流行的格式。

HTTP 方法

下面是常用于基于 REST 架构中的众所周知的 HTTP 方法：

- GET – 提供资源的只读访问。
- PUT – 用于创建一个新资源。
- DELETE – 用于移除一个资源。
- POST – 用于更新现有资源或者创建一个新资源。
- OPTIONS – 用于获取资源上支持的操作。

RESTFul Web 服务

一个 Web 服务就是一个用于在应用程序或系统之间交换数据的开放协议和标准的集合。使用不同语言编写以及运行在不同平台上的软件应用可以使用 Web 服务跨计算机网络交换数据，比如互联网的方式类似于一台计算机上的进程通信。这种互操作性（比如，Java 和 Python，或者 Windows 和 Linux 应用程序之间）归功于开放标准的使用。

这种基于 REST 架构的 Web 服务就被称为 RESTful Web 服务。这些 Web 服务使用 HTTP 方法实现 REST 架构的概念。一个 RESTful Web 服务通常定义了一个 URI，即统一资源标示符服务；提供资源表示形式比如 JSON 和设置 HTTP 方法。

创建 RESTFul Web 服务

本教程将会创建一个带以下功能的用户管理 Web 服务：

编号	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	获取 ID 为 1 的用户	只读
3	PUT	/UserService/users/2	插入 ID 为 2 的用户	幂等
4	POST	/UserService/users/2	更新 ID 为 2 的用户	N/A
5	DELETE	/UserService/users/1	删除 ID 为 1 的用户	幂等
6	OPTIONS	/UserService/users	列出 Web 服务所支持的操作	只读

2

RESTful Web 服务 – 环境设置

本教程将会指导我们如何准备开发环境，使用 Jersey 框架启动我们的工作以创建一个 RESTful Web 服务。Jersey 框架实现了 JAX-RS 2.0 API，这是创建 RESTful Web 服务的标准规范。在安装 Jersey 框架之前本教程还会教授我们如何在我们的机器上安装 JDK，Tomcat 和 Eclipse：

步骤 1 – 安装 Java 开发工具包（JDK）：

我们可以从 Oracle 的 Java 站点的 下载 Java SE (http://www.oracle.com/technetwork/java/javase/downloads/index.html) 页面下载最新版的 SDK。在下载的文件中可以找到安装 JDK 的说明，然后按照给定的说明安装和配置设置即可。最后，设置 PATH 和 JAVA_HOME 环境变量指向包含 java 和 javac 的目录，通常分别是 java_install_dir/bin 和 java_install_dir 目录。

如果你运行的是 Windows 并在 C:\jdk1.7.0_75 中安装了 JDK，那么你必须把下面的代码放到你的 C:\autoexec.bat 文件中。

```
set PATH=C:\jdk1.7.0_75\bin;%PATH%
set JAVA_HOME=C:\jdk1.7.0_75
```

或者，在 Windows NT/2000/XP 中，也可以右键我的电脑，选择属性，然后选择高级，最后选择环境变量。然后，更新 PATH 的值并按下 OK 按钮。

在 Unix（Solaris, Linux 等等）中，如果在 /usr/local/jdk1.7.0_75 中安装了 JDK 并使用的是 C Shell，你需要把下面的代码放到你的 .cshrc 文件中。

```
setenv PATH /usr/local/jdk1.7.0_75/bin:$PATH
setenv JAVA_HOME /usr/local/jdk1.7.0_75
```

或者，如果你使用集成开发环境（IDE），比如 Borland JBuilder， Eclipse,，IntelliJ IDEA 或者 Sun ONE Studio，请编译和运行一个简单的程序确认你的 IDE 知道你在哪里安装了 Java，否则请按照给定的文档对你的 IDE 做适当的设置。

步骤 2 – 安装 Eclipse IDE

本教程中的所有示例都是使用 Eclipse IDE 编写的。因此，我建议你在你的机器上安装最新版的 Eclipse。

要安装 Eclipse IDE，首先要从 http://www.eclipse.org/downloads/ 下载最新的 Eclipse 二进制文件。下载安装文件之后，解压二进制文件到某个方便的位置。比如 Windows 的 C:\eclipse 或者 Linux/Unix 的 /usr/local/exlipse 中，最后适当的设置 PATH 变量即可。

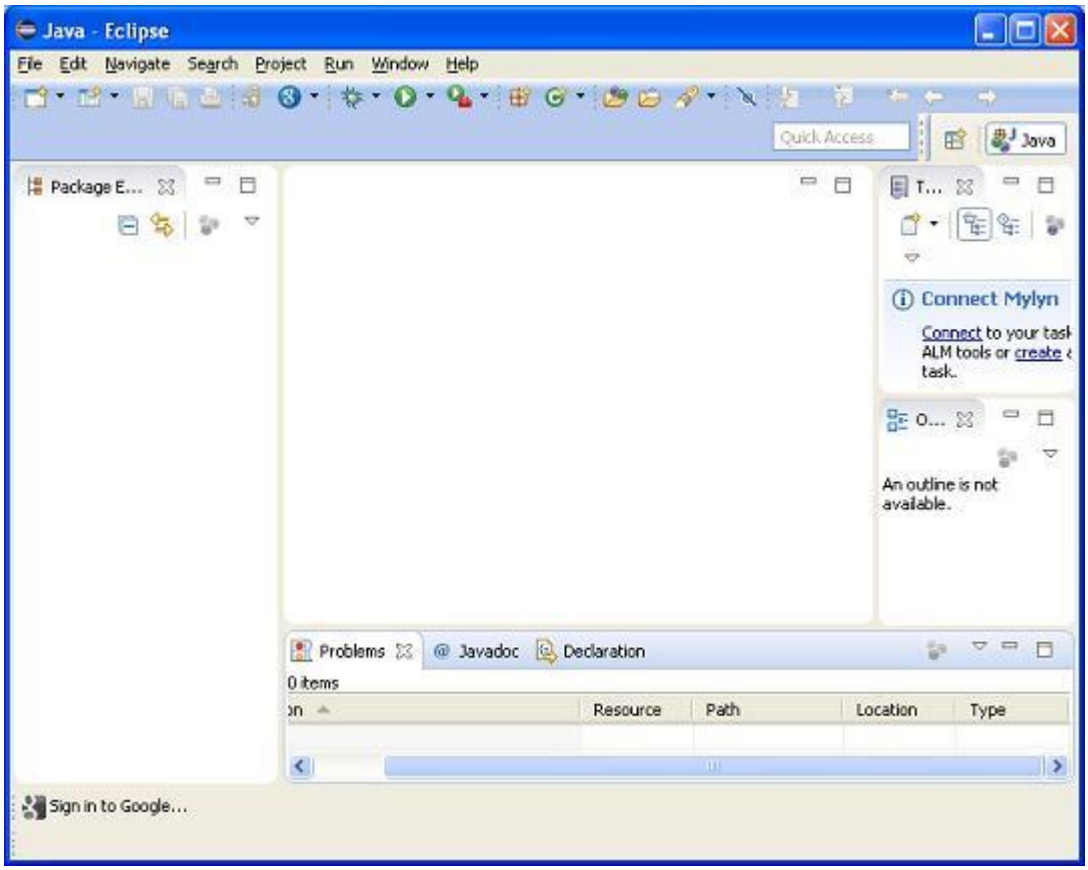
在 Windows 上可以通过执行如下命令或者简单的双击 eclipse.exe 启动 Eclipse。

```
%C:\eclipse\eclipse.exe
```

在 Unix（Solaris，Linux 等）机器上可以通过执行如下命令启动 Eclipse：

```
$/usr/local/eclipse/eclipse
```

启动成功之后，如果一切顺利，那么它应该显示如下所示结果：

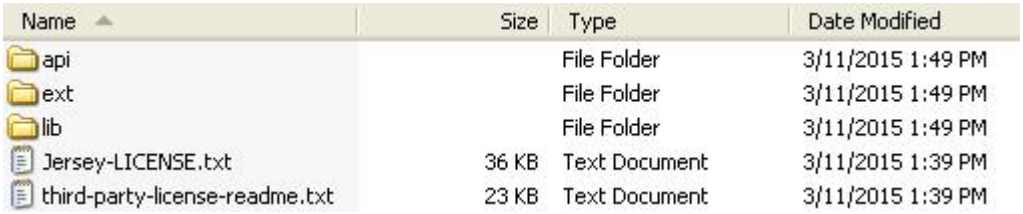


图片 2.1 eclipse

步骤 3 – 安装 Jersey 框架库

现在如果一切就绪，然后就可以安装 Jersey 框架了。以下是在你的机器上下载和安装这个框架的简单步骤。

- 选择是要在 Windows 上还是 Unix 上安装 Jersey，然后根据下一步为 Windows 下载 .zip 文件或者为 Unix 下载 .tz 文件。
- 从 https://jersey.java.net/download.html 下载最新版的 Jersey 框架二进制文件。
- 编写本教程时，我在我的 Windows 机器上下载的 **jaxrs-ri-2.17.zip**，解压下载的文件时它会在 E:\jaxrs-ri-2.17\jaxrs-ri 目录中生成如下所示的目录结构：



Name	Size	Type	Date Modified
api		File Folder	3/11/2015 1:49 PM
ext		File Folder	3/11/2015 1:49 PM
lib		File Folder	3/11/2015 1:49 PM
Jersey-LICENSE.txt	36 KB	Text Document	3/11/2015 1:39 PM
third-party-license-readme.txt	23 KB	Text Document	3/11/2015 1:39 PM

图片 2.2 jaxrs_directories

我们可以在 C:\jaxrs-ri-2.17\jaxrs-ri\lib 目录找到所有的 Jersey 库，在 C:\jaxrs-ri-2.17\jaxrs-ri\ext 中找到依赖。确保在这个目录正确设置了 CLASSPATH 变量，否则在运行应用程序时将会面临一些问题。如果你在使用 Eclipse，那么就不需要设置 CLASSPATH，因为所有的设置都会通过 Eclipse 完成。

步骤 4 – 安装 Apache Tomcat

可以从 http://tomcat.apache.org/ 上下载最新版的 Tomcat。下载安装文件之后，解压二进制文件到一个方便的位置。比如 Windows 的 C:\apache-tomcat-7.0.59 或者 Linux/Unix 的 /usr/local/apache-tomcat-7.0.59，然后设置 CATALINA_HOME 环境变量指向安装位置。

Windows 上可以通过执行如下命令或者简单的双击 startup.bat 文件启动 Tomcat：

```
%CATALINA_HOME%\bin\startup.bat

或者

C:\apache-tomcat-7.0.59\bin\startup.bat
```

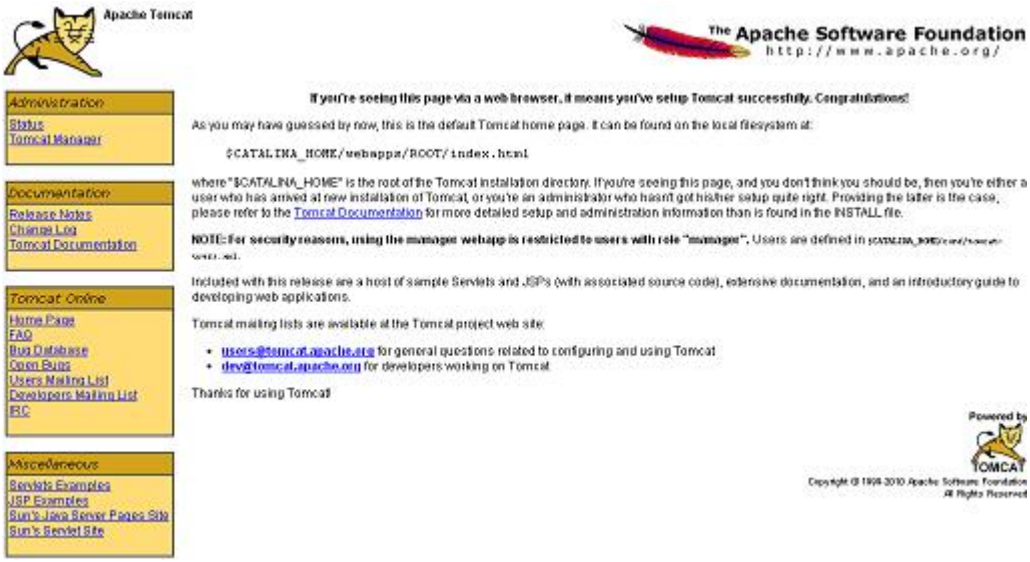
Unix（Solaris，Linux 等）机器上可以通过执行如下命令启动 Tomcat：

```
$CATALINA_HOME/bin/startup.sh

或者

/usr/local/apache-tomcat-7.0.59/bin/startup.sh
```

成功启动之后，可以通过访问 http://localhost:8080/ 查看 Tomcat 包含的默认 Web 应用程序。如果一切就绪，那么它应该显示如下结果：



图片 2.3 tomcat

关于配置和运行 Tomcat 的更多信息可以在包含的文章中找到，也可以在 Tomcat 的网页：http://tomcat.apache.org 上找到。

Windows 机器上可以通过执行如下命令停止 Tomcat：

```
%CATALINA_HOME%\bin\shutdown

或者

C:\apache-tomcat-7.0.59\bin\shutdown
```

Unix（Solaris，Linux 等）机器上可以通过执行如下命令停止 Tomcat：

```
$CATALINA_HOME/bin/shutdown.sh

或者

/usr/local/apache-tomcat-7.0.59/bin/shutdown.sh
```

一旦完成最后这一步，就可以开始准备下一章会看到的第一个 Jersey 示例了。

3

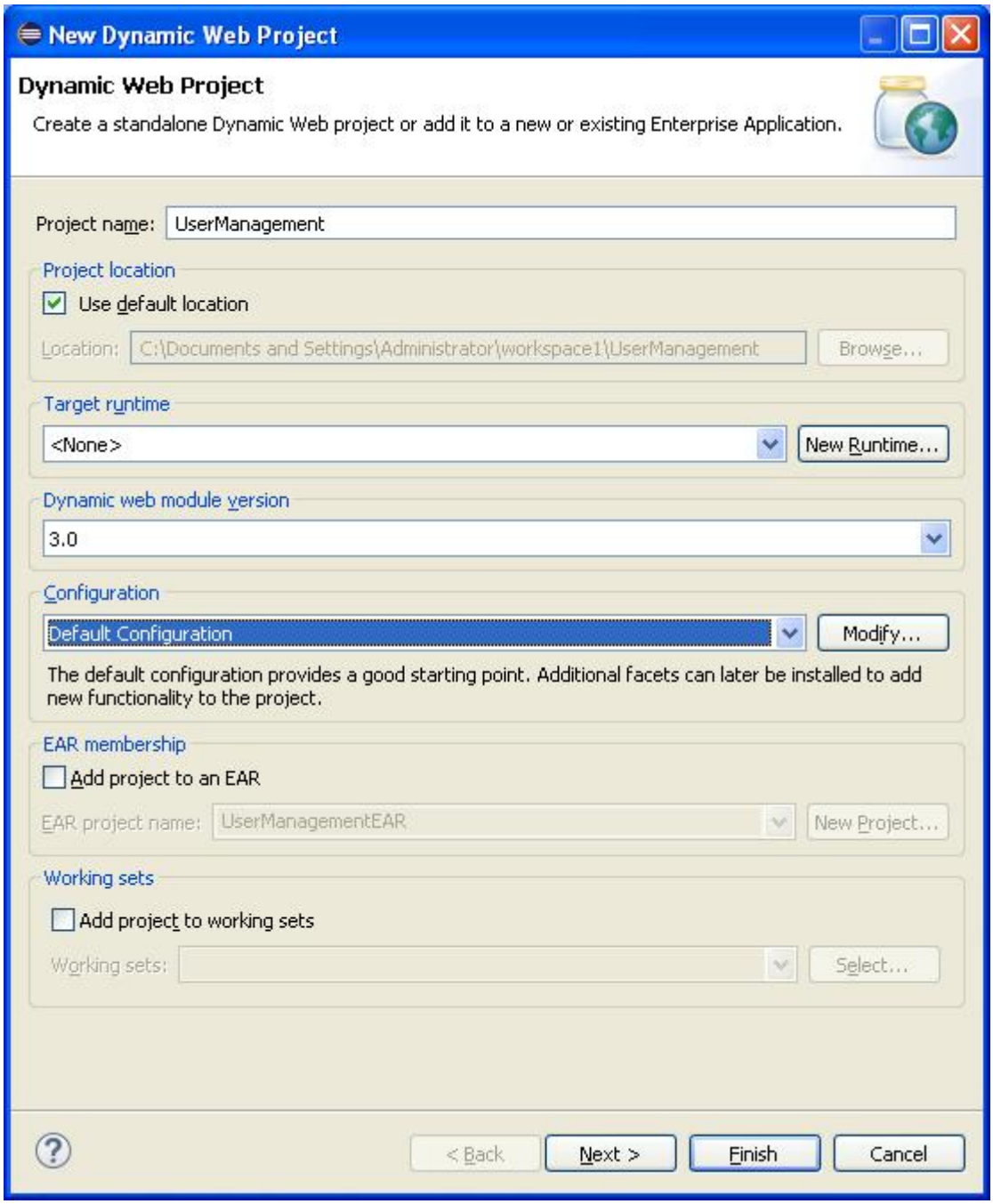
RESTful Web 服务 – 第一个应用

接下来我们开始使用 Jersey 框架编写真实的 RESTful Web 服务。在开始使用 Jersey 框架编写第一个示例之前，必须确保正确设置了 Jersey 环境，正如 RESTful Web 服务 – 环境设置 () 教程中所阐述的。我还假设你掌握了一点点使用 Eclipse IDE 的基础知识。

那么，让我们开始编写一个简单的 Jersey 应用程序吧，它将暴漏一个 Web 服务方法来显示用户列表。

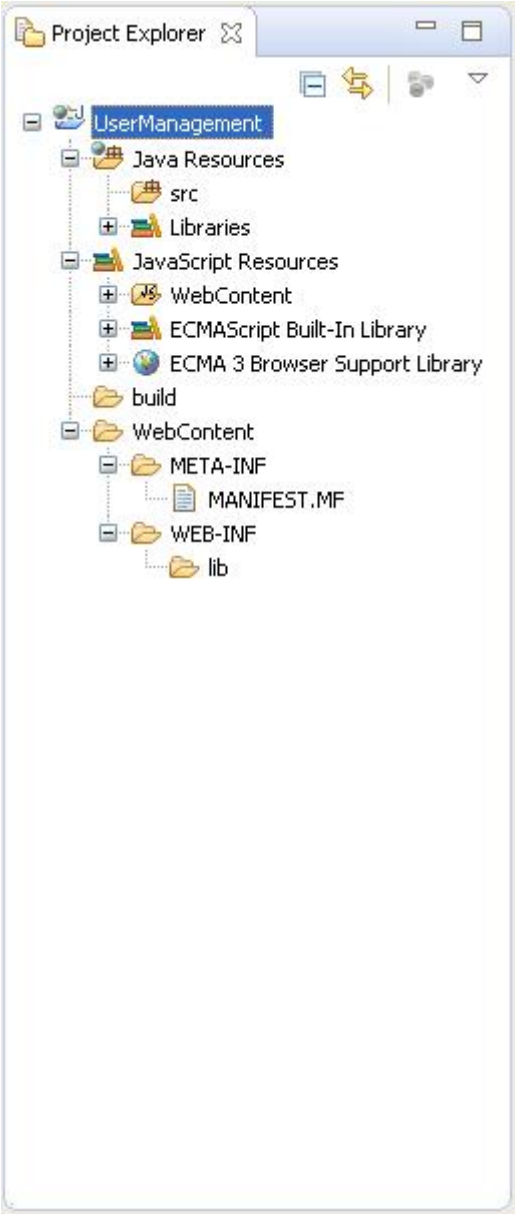
步骤 1 – 创建 Java 项目

第一步是使用 Eclipse IDE 创建一个动态 Web 项目。遵循选项 File -> New -> Project，最后在向导列表中选择 Dynamic Web Project 向导。然后在如下所示的向导窗口中将项目命名为 UserManagement：



图片 3.1 wizard

一旦项目创建成功，在 Project Explorer 中就会得到如下所示的目录结构：



图片 3.2 usermanagement_dir

步骤 2 – 添加需要的库

我们把在项目中添加 Jersey 框架和它的依赖（库）作为第二步。从下面下载的 jersey zip 文件夹目录中把所有的 jars 复制到项目的 WEB-INF/lib 目录中。

- \jaxrs-ri-2.17\jaxrs-ri\api
- \jaxrs-ri-2.17\jaxrs-ri\ext
- \jaxrs-ri-2.17\jaxrs-ri\lib

首先，右击项目名 UserManagement，选择上下文菜单中的选项：Build Path -> Configure Build Path 显示 Java Build Path 窗口。

最后使用 Libraries 选项卡下面的 Add JARs 按钮把 JARs 文件添加到 WEB-INF/lib 目录中。

步骤 3 – 创建源代码文件

现在，让我们在 UserManagement 项目下创建真实的源代码文件。首先我们需要创建一个叫做 com.tutorialspoint 的包。要做到这一点，在包管理器中的 src 上右击并遵循选项：New -> Package。

接下来我们会在 com.tutorialspoint 包下面创建 User.java，UserDao.java 以及 UserService.java 文件。

User.java 文件

```
package com.tutorialspoint;

import java.io.Serializable;

import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
@XmlRootElement(name = "user")
public class User implements Serializable {

  private static final long serialVersionUID = 1L;
  private int id;
  private String name;
  private String profession;

  public User(){}

  public User(int id, String name, String profession){
    this.id = id;
    this.name = name;
    this.profession = profession;
  }

  public int getId() {
    return id;
  }

  @XmlElement
  public void setId(int id) {
    this.id = id;
```

```
  }
  public String getName() {
    return name;
  }
  @XmlElement
  public void setName(String name) {
    this.name = name;
  }
  public String getProfession() {
    return profession;
  }
  @XmlElement
  public void setProfession(String profession) {
    this.profession = profession;
  }
}
```

UserDao.java 文件

下面的程序已经被硬编码以显示用户清单相关的功能。这里我们可以实现可能从数据库或者任意资源中选择用户或者其他所需数据需要的业务逻辑。

```
package com.tutorialspoint;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;
import java.util.List;

public class UserDao {
  public List<User> getAllUsers(){
    List<User> userList = null;
    try {
      File file = new File("Users.dat");
      if (!file.exists()) {
        User user = new User(1, "Mahesh", "Teacher");
        userList = new ArrayList<User>();
        userList.add(user);
        saveUserList(userList);
      }
      else{
```

```
         FileInputStream fis = new FileInputStream(file);
         ObjectInputStream ois = new ObjectInputStream(fis);
         userList = (List<User>) ois.readObject();
         ois.close();
       }
    } catch (IOException e) {
       e.printStackTrace();
    } catch (ClassNotFoundException e) {
       e.printStackTrace();
    }
    return userList;
  }
}
```

UserService.java 文件

```
package com.tutorialspoint;

import java.util.List;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

@Path("/UserService")
public class UserService {

  UserDao userDao = new UserDao();

  @GET
  @Path("/users")
  @Produces(MediaType.APPLICATION_XML)
  public List<User> getUsers(){
    return userDao.getAllUsers();
  }
}
```

这里有两个关于主程序需要注意的要点，UserService.java：

1. 第一步是使用 @Path 注释 UserService 给 Web 服务指定一个路径。
2. 第二步是使用 @Path 注释 UserService 方法给特定的 Web 服务方法指定一个路径。

步骤 4 – 创建 Web.xml 配置文件

我们需要创建一个 Web.xml 配置文件，这是一个 XML 文件，用于给我们的应用程序指定 Jersey 框架 servlet。

_web.xml 文件__

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
  http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
  id="WebApp_ID" version="3.0">
  <display-name>User Management</display-name>
  <servlet>
    <servlet-name>Jersey RESTful Application</servlet-name>
    <servlet-class>org.glassfish.jersey.servlet.ServletContainer</servlet-class>
      <init-param>
        <param-name>jersey.config.server.provider.packages</param-name>
        <param-value>com.tutorialspoint</param-value>
      </init-param>
    </servlet>
  <servlet-mapping>
  <servlet-name>Jersey RESTful Application</servlet-name>
    <url-pattern>/rest/*</url-pattern>
  </servlet-mapping>
</web-app>
```

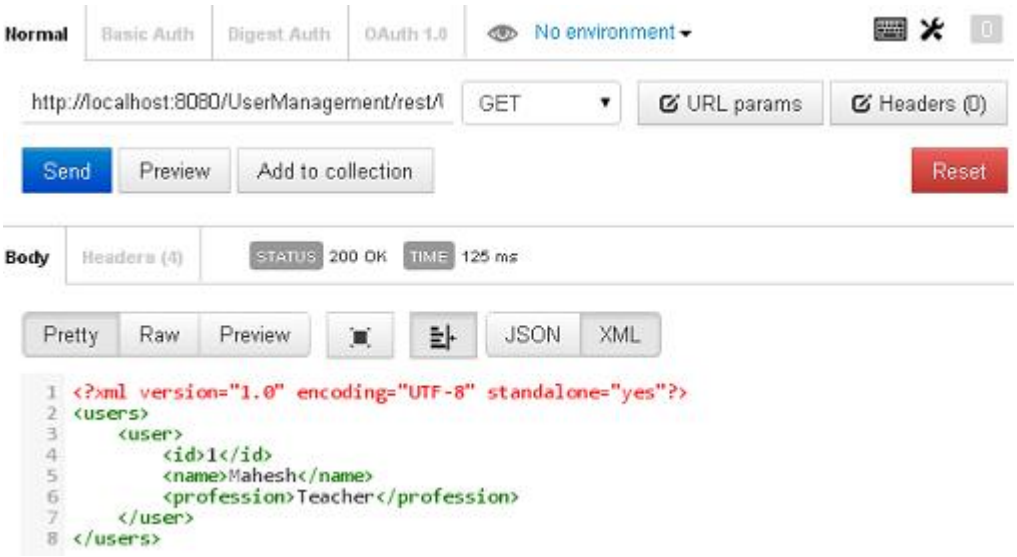
步骤 5 – 部署程序

一旦完成源代码和 Web 配置文件创建，就可以进入编译和运行程序这一步。要做到这一点，使用 Eclipse 将应用程序导出为一个 war 文件并部署到 Tomcat 中即可。要使用 eclipse 创建 WAR 文件，遵循选项 File -> export -> Web -> War File，最后选择项目 UserManagement 和目标文件夹即可。要在 Tomcat 中部署 war 文件，把 UserManagement.war 放到 Tomcat 安装目录的 webapps 目录中并启动 Tomcat 即可。

步骤 6 – 运行程序

我们使用 Postman (http://www.getpostman.com/) 来测试我们的 Web 服务，这是一个 Chrome 扩展。

建立一个到 UserManagement 获取所有用户列表的请求。把 `http://localhost:8080/UserManagement/rest/UserService/users` 放到 POSTMAN 中选择 GET 请求，可以看到下面的结果。



图片 3.3 restful_postman

恭喜，你已经成功创建你的第一个 RESTful 应用程序了。为了更进一步，让我们在接下来的几个章节中做一些有趣的事情。

4

RESTful Web 服务 – 资源

什么是资源?

REST 架构把所有内容都视为资源。这些资源可以是文本文件，html 页面，图像，视频或者动态业务数据。REST 服务器只提供对资源的访问，REST 客户端访问和修改资源。这里每个资源都通过 URIs/ 全局 IDs 标识。REST 使用不同的表示形式表示资源，比如文本，JSON，XML。XML 和 JSON 是最流行的资源表示形式。

资源表示形式

REST 中的资源类似于面向对象编程中的对象或者类似于数据库中的实体。一旦资源被确定，它的表现就会决定使用某个标准的格式，因此服务器可以按照上述格式发送这些资源，客户端也可以理解同样的格式。

比如，在 RESTful Web 服务 – 第一个应用 () 教程中，用户就是一个资源，其中使用如下所示的 XML 格式表示:

```
<user>
   <id>1</id>
   <name>Mahesh</name>
   <profession>Teacher</profession>
</user>
```

同样的资源也可以使用如下所示的 JSON 格式表示:

```
{
   "id":1,
   "name":"Mahesh",
   "profession":"Teacher"
}
```

良好的资源表示形式

REST 并不对资源表示格式施加任何限制。对于服务器上的同一资源，一个客户端可以要求使用 JSON 表示，而另一个客户端可能会要求使用 XML 表示。REST 服务器的责任就是以客户端理解的格式传递客户端资源。

下面是在 RESTful Web 服务中设计资源表示格式时要考虑的重点。

- **可理解性**：服务器端和客户端都应该能理解和利用资源表示格式。
- **完整性**：格式应该能够完整地表示一个资源。比如，资源可以包含另一个资源。格式应该能够表示简单以及复杂的资源结构。

- **可连接性**：资源可以链接到另一个资源，格式应该能够处理这种情况。

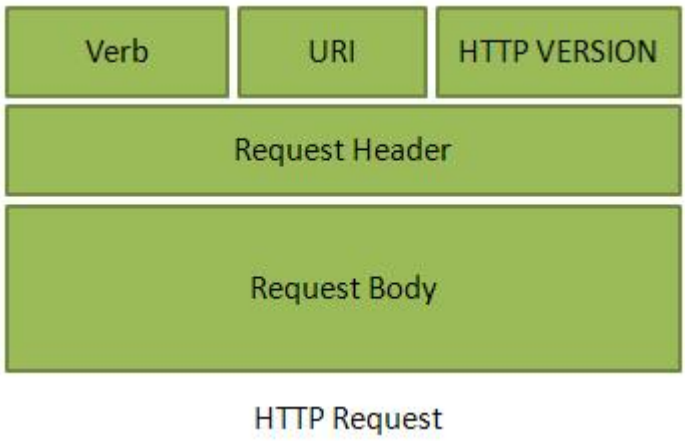
然而，目前大多数 Web 服务都使用 XML 或者 JSON 格式表示资源。有很多库和工具都可以用来理解，解析和修改 XML 和 JSON 数据。

5

RESTful Web 服务 – 消息

RESTful Web 服务使用 HTTP 协议作为客户端和服务器之间的通信媒介。客户端发送一个 HTTP 请求形式的消息，然后服务器按照 HTTP 响应形式的响应。这种技术被称为消息传递。这些消息包含消息数据和元数据，比如消息本身相关的信息。我们来看看 HTTP 1.1 中的 HTTP 请求和 HTTP 消息响应。

HTTP 请求

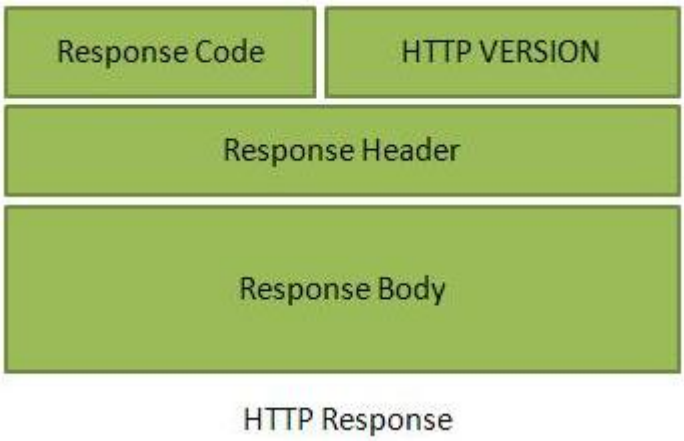


图片 5.1 http request

一个 HTTP 请求有五个主要部分：

- **动作（Verb）** – 表明 HTTP 方法，比如 GET，POST，DELETE，PUT 等等。
- **URI** – 用来标识服务器上资源的统一资源标示符（URI）。
- **HTTP 版本** – 表明 HTTP 版本，比如 HTTP v1.1。
- **请求头** – 包含 HTTP 请求消息的元数据，它是键–值对形式的。比如，客户端（或者浏览器）类型，客户端支持的格式，消息体格式，缓存设置等等。
- **请求体** – 消息内容或者资源表示形式。

HTTP 响应



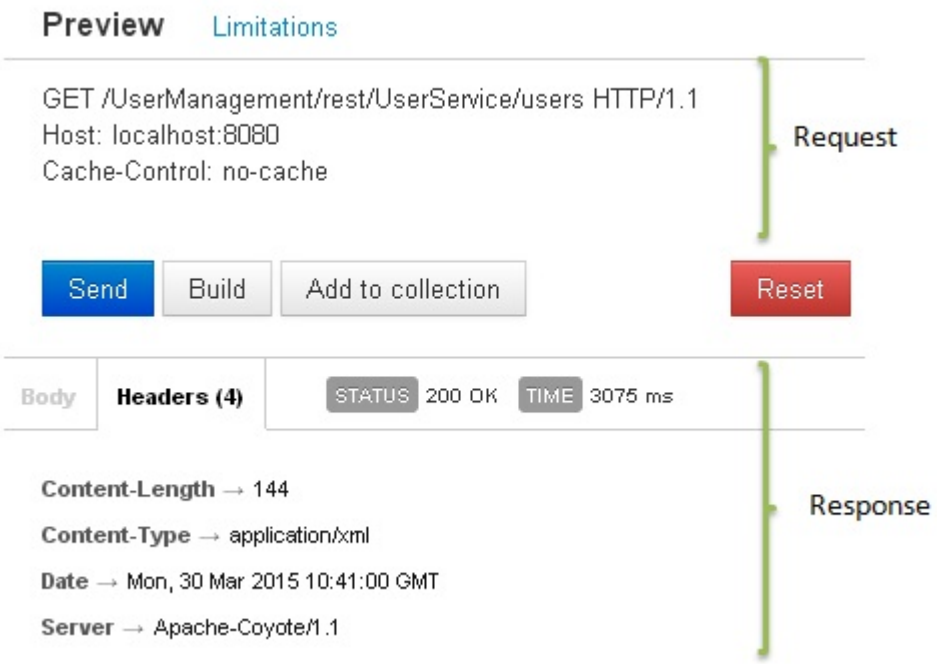
图片 5.2 http response

HTTP 响应有四个主要部分：

- **状态/响应码** – 表明请求资源的服务器状态。比如 404 意味着资源没有找到或者 200 意味着响应 OK。
- **HTTP 版本** – 表明 HTTP 版本，比如 HTTP v1.1。
- **响应头** – 包含 HTTP 响应消息的元素数据，它是键–值对形式的。比如，内容长度，内容类型，响应日期，服务器类型等等。
- **响应体** – 响应消息内容或者资源表示形式。

示例

正如 RESTful Web 服务 – 第一个应用 () 教程中所阐述的。让我们把 `http://localhost:8080/UserManagement/rest/UserService/users` 放到 POSTMAN 中并设置为 GET 请求。如果我们点击 Postman 发送按钮附近的预览按钮，然后点击发送按钮，你可能看到如下所示输出：



图片 5.3 postman request and response

这里可以看到，浏览器发送了一个 GET 请求并收到一个 XML 形式的响应体。

6

RESTful Web 服务 – 寻址

寻址指的是定位存储在服务器上的一个或多个资源。类似于定位某个人的邮寄地址。

REST 架构中的每个资源都通过它的 URI（统一资源标示符）标识。URI 格式如下：

```
<protocol>://<service-name>/<ResourceType>/<ResourceID>
```

URI 的目的是定位托管 Web 服务的服务器上的资源。请求的另一个重要的属性是 VERB，它用于标识要在资源上执行的操作。比如，在 RESTful Web 服务 – 第一个应用 () 教程中，URI 就是 `http://localhost:8080/UserManagement/rest/UserService/users`，VERB 是 GET。

构建一个标准的 URI

下面是设计 URI 时要考虑的要点：

- **使用复数名词** – 使用复数名词定义资源。比如，我们使用 users 标识用户资源。
- **避免使用空格** – 处理长资源名时使用下划线（_）或者连字符（-），比如，用 authorized_users 而不是 authorized%20users。
- **使用小写字母** – 尽管 URI 不区分带小写，但是在 url 中使用小写字母是一种很好的做法。
- **保持向后兼容** – 由于 Web 服务是一种公共服务，URI 一旦公开之后应该始终可用。这种情况下，要更新 URI，请使用 HTTP 状态码 – 300 重定向老的 URI 到新的 URI。
- **使用 HTTP Verb** – 始终使用 HTTP Verb，比如 GET，PUT 以及 DELETE 处理资源操作。在 URL 中使用操作名并不好。

示例

下面是一个获取用户的不好的 URI 示例：

```
http://localhost:8080/UserManagement/rest/UserService/getUser/1
```

下面是一个获取用户的好的 URI 示例：

```
http://localhost:8080/UserManagement/rest/UserService/users/1
```

7

RESTful Web 服务 – 方法

正如目前为止我们所讨论的，RESTful Web 服务大量使用 HTTP 动词确定要对指定资源进行的操作。下面的表格演示了常用 HTTP 动词的例子。

编号	HTTP 方法，URI 和操作
1	GET http://localhost:8080/UserManagement/rest/UserService/users 获取用户列表 （只读）
2	GET http://localhost:8080/UserManagement/rest/UserService/users/1 获取 ID 为 1 的用户 (只读)
3	PUT http://localhost:8080/UserManagement/rest/UserService/users/2 插入 ID 为 2 的用户 （幂等）
4	POST http://localhost:8080/UserManagement/rest/UserService/users/2 更新 ID 为 2 的用户 (N/A)
5	DELETE http://localhost:8080/UserManagement/rest/UserService/users/1 删除 ID 为 1 的用户 （幂等）
6	OPTIONS http://localhost:8080/UserManagement/rest/UserService/users 列出 Web 服务所支持的操作 （只读）
7	HEAD http://localhost:8080/UserManagement/rest/UserService/users 只返回 HTTP 头，不返回 HTTP 体 （只读）

这里是要考虑的要点：

- GET 操作是只读且安全的。
- PUT 和 DELETE 操作是幂等的意味着它们的结果总是相同的，无论这个操作被调用多少次。
- PUT 和 POST 操作几乎是相同的，区别在于 PUT 操作的结果是幂等的，而 POST 操作会导致不同的结果。

示例

让我们更新一下 RESTful Web 服务 – 第一个应用 () 教程中创建的例子来创建一个可以执行 CRUD（创建，读取，更新，移除）操作的 Web 服务。简单起见，这里我们使用文件 I/O 替代数据库操作。

我们来更新一下 com.tutorialspoint 包下面的 User.java，UserDao.java 和 UserService.java 文件。

User.java

```
package com.tutorialspoint;

import java.io.Serializable;

import javax.xml.bind.annotation.XmlElement;
import javax.xml.bind.annotation.XmlRootElement;
@XmlRootElement(name = "user")
public class User implements Serializable {

  private static final long serialVersionUID = 1L;
  private int id;
  private String name;
  private String profession;

  public User(){}

  public User(int id, String name, String profession){
    this.id = id;
    this.name = name;
    this.profession = profession;
  }

  public int getId() {
    return id;
  }
  @XmlElement
  public void setId(int id) {
    this.id = id;
  }
  public String getName() {
    return name;
  }
  @XmlElement
    public void setName(String name) {
```

```
    this.name = name;
  }
  public String getProfession() {
    return profession;
  }
  @XmlElement
  public void setProfession(String profession) {
    this.profession = profession;
  }

  @Override
  public boolean equals(Object object){
    if(object == null){
      return false;
    }else if(!(object instanceof User)){
      return false;
    }else {
      User user = (User)object;
      if(id == user.getId()
        && name.equals(user.getName())
        && profession.equals(user.getProfession())
      ){
        return true;
      }
    }
    return false;
  }
}
```

UserDao.java

```
package com.tutorialspoint;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.ArrayList;
import java.util.List;

public class UserDao {
  public List<User> getAllUsers(){
    List<User> userList = null;
```

```
   try {
      File file = new File("Users.dat");
      if (!file.exists()) {
         User user = new User(1, "Mahesh", "Teacher");
         userList = new ArrayList<User>();
         userList.add(user);
         saveUserList(userList);
      }
      else{
         FileInputStream fis = new FileInputStream(file);
         ObjectInputStream ois = new ObjectInputStream(fis);
         userList = (List<User>) ois.readObject();
         ois.close();
      }
   } catch (IOException e) {
      e.printStackTrace();
   } catch (ClassNotFoundException e) {
      e.printStackTrace();
   }
   return userList;
}

public User getUser(int id){
   List<User> users = getAllUsers();

   for(User user: users){
      if(user.getId() == id){
         return user;
      }
   }
   return null;
}

public int addUser(User pUser){
   List<User> userList = getAllUsers();
   boolean userExists = false;
   for(User user: userList){
      if(user.getId() == pUser.getId()){
         userExists = true;
         break;
      }
   }
   if(!userExists){
      userList.add(pUser);
      saveUserList(userList);
```

```
      return 1;
    }
    return 0;
  }

  public int updateUser(User pUser){
    List<User> userList = getAllUsers();

    for(User user: userList){
      if(user.getId() == pUser.getId()){
        int index = userList.indexOf(user);
        userList.set(index, pUser);
        saveUserList(userList);
        return 1;
      }
    }
    return 0;
  }

  public int deleteUser(int id){
    List<User> userList = getAllUsers();

    for(User user: userList){
      if(user.getId() == id){
        int index = userList.indexOf(user);
        userList.remove(index);
        saveUserList(userList);
        return 1;
      }
    }
    return 0;
  }

  private void saveUserList(List<User> userList){
    try {
      File file = new File("Users.dat");
      FileOutputStream fos;

      fos = new FileOutputStream(file);

      ObjectOutputStream oos = new ObjectOutputStream(fos);
      oos.writeObject(userList);
      oos.close();
    } catch (FileNotFoundException e) {
      e.printStackTrace();
```

```
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

UserService.java

```
package com.tutorialspoint;

import java.io.IOException;
import java.util.List;

import javax.servlet.http.HttpServletResponse;
import javax.ws.rs.Consumes;
import javax.ws.rs.DELETE;
import javax.ws.rs.FormParam;
import javax.ws.rs.GET;
import javax.ws.rs.OPTIONS;
import javax.ws.rs.POST;
import javax.ws.rs.PUT;
import javax.ws.rs.Path;
import javax.ws.rs.PathParam;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;

@Path("/UserService")
public class UserService {

  UserDao userDao = new UserDao();
  private static final String SUCCESS_RESULT="<result>success</result>";
  private static final String FAILURE_RESULT="<result>failure</result>";


  @GET
  @Path("/users")
  @Produces(MediaType.APPLICATION_XML)
  public List<User> getUsers(){
    return userDao.getAllUsers();
  }

  @GET
  @Path("/users/{userid}")
  @Produces(MediaType.APPLICATION_XML)
  public User getUser(@PathParam("userid") int userid){
```

```
  return userDao.getUser(userid);
}

@PUT
@Path("/users")
@Produces(MediaType.APPLICATION_XML)
@Consumes(MediaType.APPLICATION_FORM_URLENCODED)
public String createUser(@FormParam("id") int id,
  @FormParam("name") String name,
  @FormParam("profession") String profession,
  @Context HttpServletResponse servletResponse) throws IOException{
  User user = new User(id, name, profession);
  int result = userDao.addUser(user);
  if(result == 1){
    return SUCCESS_RESULT;
  }
  return FAILURE_RESULT;
}

@POST
@Path("/users")
@Produces(MediaType.APPLICATION_XML)
@Consumes(MediaType.APPLICATION_FORM_URLENCODED)
public String updateUser(@FormParam("id") int id,
  @FormParam("name") String name,
  @FormParam("profession") String profession,
  @Context HttpServletResponse servletResponse) throws IOException{
  User user = new User(id, name, profession);
  int result = userDao.updateUser(user);
  if(result == 1){
    return SUCCESS_RESULT;
  }
  return FAILURE_RESULT;
}

@DELETE
@Path("/users/{userid}")
@Produces(MediaType.APPLICATION_XML)
public String deleteUser(@PathParam("userid") int userid){
  int result = userDao.deleteUser(userid);
  if(result == 1){
    return SUCCESS_RESULT;
  }
  return FAILURE_RESULT;
}
```

```
  @OPTIONS
  @Path("/users")
  @Produces(MediaType.APPLICATION_XML)
  public String getSupportedOperations(){
    return "<operations>GET, PUT, POST, DELETE</operations>";
  }
}
```

现在，使用 Eclipse 将你的应用程序导出为 war 文件然后部署到 Tomcat 中。要使用 eclipse 创建 WAR 文件，遵循选项 **File -> export -> Web -> War File**，最后选择项目 UserManagement 和目标文件夹即可。要在 Tomcat 中部署 war 文件，把 UserManagement.war 放到 **Tomcat 安装目录** 下的 webapps 目录中并启动 Tomcat 即可。

测试 Web 服务

Jersey 提供了创建 Web 服务客户端的 API 以测试 Web 服务。我们在同一项目的 com.tutorialspoint 包下面创建了一个范例测试类 **WebServiceTester.java**。

WebServiceTester.java

```
package com.tutorialspoint;

import java.util.List;

import javax.ws.rs.client.Client;
import javax.ws.rs.client.ClientBuilder;
import javax.ws.rs.client.Entity;
import javax.ws.rs.core.Form;
import javax.ws.rs.core.GenericType;
import javax.ws.rs.core.MediaType;

public class WebServiceTester {

  private Client client;
  private String REST_SERVICE_URL = "http://localhost:8080/UserManagement/rest/UserService/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 );
  }
}
```

现在，使用 Eclipse 运行这个测试。右击文件，遵循选项 Run as –> Java Application。在 Eclipse 控制台中我们会看到如下所示结果：

```
Test case name: testGetAllUsers, Result: pass
Test case name: testGetUser, Result: pass
Test case name: testUpdateUser, Result: pass
Test case name: testAddUser, Result: pass
Test case name: testDeleteUser, Result: pass
```

8

RESTful Web 服务 – 无状态

根据 REST 架构，一个 RESTful Web 服务不应该在服务器上保持客户端状态。这种约束被称为无状态。客户端的职责是传递其上下文给服务器，然后服务器存储这个上下文以处理客户端的请求。比如，由服务器维护的会话是通过客户端传递的会话标示符识别的。

RESTful Web 服务应该遵守这一约束。我们已经在 RESTful Web 服务 – 方法 () 教程中见过，Web 服务方法不会存储调用它们的客户端的任意信息。

考虑如下 URI：

```
http://localhost:8080/UserManagement/rest/UserService/users/1
```

如果我们使用浏览器，使用基于 Java 的客户端或者使用 postman 访问上面的 url，结果始终是 User XML 并且它的 ID 为 1，因此服务器并没有存储客户端相关的任意信息。

```
<user>
<id>1</id>
<name>mahesh</name>
<profession>1</profession>
</user>
```

无状态的优势

下面是 RESTful Web 服务中无状态的好处：

- Web 服务可以独立对待每个请求方法。
- Web 服务不需要维护客户端先前的交互。简化了应用程序设计。
- HTTP 本身是一个无状态协议，RESTful Web 服务可与 HTTP 协议无缝协作。

无状态的缺点

下面是 RESTful Web 服务中无状态的缺点：

- Web 服务需要在每个请求中获取额外的信息，然后在客户端交互需要处理的情况下解读客户端状态。

9

RESTful Web 服务 – 缓存

缓存是指在客户端存储服务器响应，以便客户端不需要一次又一次的请求服务器上相同的资源。服务器响应应该有关于如何进行缓存的信息，以便客户端缓存一段时间内的响应或者永远不缓存服务器响应。

下面是可以用来配置客户端缓存的服务器响应头：

编号	头信息 & 描述
1	Date 创建资源的日期和时间。
2	Last Modified 最后修改资源的日期和时间。
3	Cache-Control 控制缓存的主要头信息。
4	Expires 缓存到期的日期和时间。
5	Age 从服务器获取资源持续的秒数。

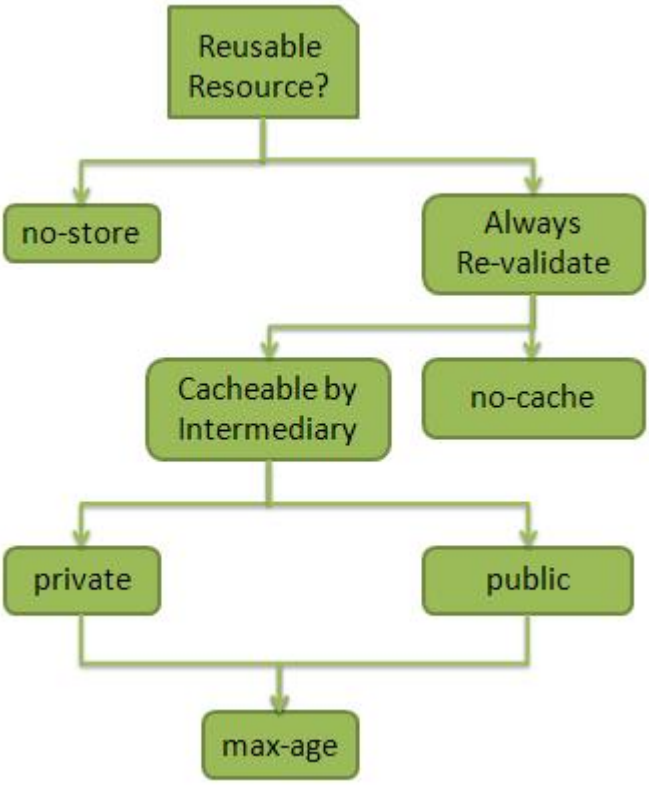
Cache-Control Header

下面是 Cache-Control 头详情：

编号	指令 & 描述
1	Public 表明该资源可由任何组件缓存。
2	Private 表明该资源只能由客户端和服务器缓存，没有中介可以缓存该资源。
3	no-cache/no-store 表明该资源不可缓存。
4	max-age 表明缓存在 max-age 指定的秒数内有效。之后，客户端就必须发起另一个请求。
5	must-revalidate 表明如果 max-age 已经过去了服务器要重新验证资源。

最佳实践

- 始终保持静态内容比如图像，CSS，JavaScript 可缓存，设置到期日期为 2 到 3 天。
- 永远不要保持过长的有效期。
- 动态内容应该只缓存几个小时。



图片 9.1 cache control

10

RESTful Web 服务 – 安全性

因为 RESTful Web 服务使用 HTTP URLs 路径，因此以保护网站同样的方式维护 RESTful Web 服务是非常重要的。以下是设计 RESTful Web 服务时要遵循的最佳实践。

- **验证** – 验证服务器上的所有输入。保护服务器免受 SQL 或者 NoSQL 注入攻击。
- **基于会话的认证** – 请求一个 Web 服务方法时使用基于会话的认证对用户进行身份验证。
- **URL 不要有敏感数据** – 永远不要在 URL 中使用用户名，密码或者会话标记，这些值应该通过 POST 方法传递给 Web 服务。
- **限制方法执行** – 允许限制使用方法，比如 GET，POST，DELET。GET 方法不应该能够删除数据。
- **验证有缺陷的 XML/JSON** – 检查格式良好的输入传递给 Web 服务方法。
- **抛出通用错误消息** – Web 服务方法应该使用 HTTP 错误消息，比如 403 展示禁止访问等。

HTTP 状态码

编号	HTTP 状态码 & 描述
1	200 OK，显示成功。
2	201 CREATED，当资源使用 POST 或者 PUT 请求建立成功时。使用位置头返回新建资源的链接。
3	204 NO CONTENT，当响应体为空时。比如，DELETE 请求。
4	304 NOT MODIFIED 在有条件的 GET 请求的情况下用于减少网络带宽的使用。响应体应该为空。头信息应该包含日期，位置等。
5	400 BAD REQUEST，指出提供的输入无效。比如验证错误，数据缺失。
6	401 UNAUTHORIZED，指出用户正在使用无效的或者错误的认证令牌。
7	403 FORBIDDEN，指出用户没有使用访问方法。比如，没有管理员权限访问删除操作。
8	404 NOT FOUND，指出该方法不可用。
9	409 CONFLICT，指出执行方法时冲突，比如添加重复的条目。
10	500 INTERNAL SERVER ERROR，指出执行该方法时服务器抛出了一些异常。

11

RESTful Web 服务 – Java (JAX-RS)

JAX-RS 表示用于 RESTful Web 服务的 Java API。JAX-RS 是一种基于 Java 的编程语言 API 以及为创建 RESTful Web 服务提供支持的规范。2.0 版本发布于 2013 年 5 月 24 日。从 Java SE 5 开始大量使用 JAX-RS 注释以简化基于 Java 的 Web 服务的创建和部署。它还为创建 RESTful Web 服务客户端提供支持。

规范

以下是影射某个资源为 Web 服务资源的常用注释:

编号	注释 & 描述
1	@Path 资源类或方法的相对路径。
2	@GET HTTP Get 请求，用来提取资源。
3	@PUT HTTP PUT 请求，用来创建资源。
4	@POST HTTP POST 请求，用来创建或更新资源。
5	@DELETE HTTP DELETE 请求，用来删除资源。
6	@HEAD HTTP HEAD 请求，用来获取方法可用的状态。
7	@Produces 由 Web 服务生成的 HTTP 响应，比如 APPLICATION/XML，TEXT/HTML，APPLICATION/JSON 等。
8	@Consumes HTTP 请求类型，比如 application/x-www-form-urlencoded 在 POST 请求期间在 HTTP 体中接受表单数据。
9	@PathParam 绑定传递给方法的参数为路径中的某个值。
10	@QueryParam 绑定传递给方法参数为路径中的某个查询参数。
11	@MatrixParam 绑定传递给方法参数为路径中的某个 HTTP 矩阵参数。
12	@HeaderParam 绑定传递给方法的参数为 HTTP 头。
13	@CookieParam 绑定传递给方法的参数为某个 Cookie。
14	@FormParam 绑定传递给方法的参数为某个表单值。
15	@DefaultValue 给传递给方法的参数分配一个默认值。

16	@Context 资源上下文，比如将 HTTP 请求作为上下文。

在 RESTful Web 服务 – 第一个应用 () 和 RESTful Web 服务 – 方法 () 教程中我们使用的 Jersey，它是 Oracle 的 JAX-RS 2.0 的参考实现。