# java Restlet

restlet是java框架中用于开发RESTful风格的web应用程序的框架。在restlet中，我们可以方便的使用Router来管理URI。

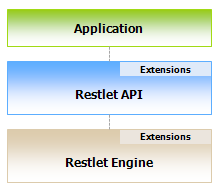
一个HTTP请求包含一个URI来标识要访问的资源。这个信息被存储在Request.resourceRef属性中，并作为我们路由请求的基础。我们的第一个目标是找目标资源，它可以是一个ServerResource类及其对象或ServerResource类的子类及其对象。如果使用类而不是对象，则在有请求到来并路由成功（匹配route成功）之后，会自动实例化该类。

在Router类中有一个attach()方法，它需要两个参数，一个是URI模版，一个是ServerResource。attach()方法会创建相应的routes，以便在有请求到来时进行匹配。在使用Router时，我们不必关心HTTP请求的Method，因为我们只需要在Resource类中使用注解（比如：@put，@post等），Router就会自动根据HTTP请求中的Method，去调用有着相应注解的方法。

## Restlet概述

Restlet框架由两个重要部分组成：Restlet API和Restlet Engine 。Restlet API是支持REST的API，并能处理来自用户侧和服务器侧应用程序的调用。Restlet API的后端是Restlet Engine，这两部分都在org.restlet.jar包中。

在API和实现之间的区别，类似Servlet API和Web contatiners（Jetty或Tomcat）或JDBC API和具体的JDBC drivers之间的区别。



Restlet框架**既是客户框架也是服务器框架**。比如，Restlet能使用HTTP client connector去操作远程资源。在REST中的一个connector是来保证两个组件之间正常通信的软件元素，通常由网络协议实现。Restlet提供几个基于已有开源项目的client connectors实现。在connectors一章中，列出了所有可用的client和server connectors，并解释了如何去使用和配置。

|  |
| --- |
| // Create the client resource  ClientResource resource = new ClientResource("http://restlet.com");  // Customize the referrer property  resource.setReferrerRef("http://www.mysite.org");  // Write the response entity on the console  resource.get().write(System.out); |

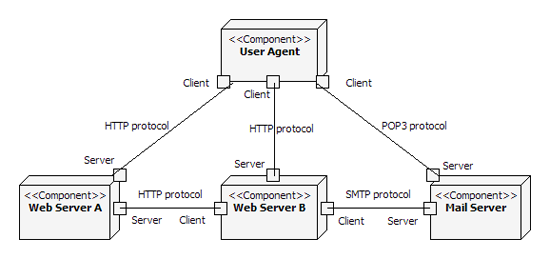
现在，我们想要知道Restlet框架如何监听客户请求和相应。我们将用Internal Restlet HTTP服务器connector，并返回一个简单的字符串表示“hello, world”。

|  |
| --- |
| public class Part03 extends ServerResource {  public static void main(String[] args) throws Exception {  // Create the HTTP server and listen on port 8182  new **Server**(Protocol.HTTP, 8182, Part03.class).start();  }  @Get  public String toString() {  return "hello, world";  }  } |

如果运行这个代码和启动你的服务器，你能打开一个Web 浏览器，并登录<http://localhost:8182>。事实上，任何URI都能正常工作，尝试使用<http://localhost:8182/test/tutorial>。注意，如果你从另一台机器测试你的服务器，你需要替换“localhost”。

## REST架构概述

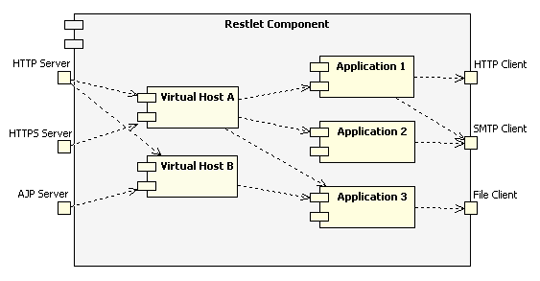
让我们从REST的角度来考虑典型的web架构。这下图中，ports表示connector，links表示具体的协议（HTTP，SMTP等）。



## Components，virtual hosts和applications

除了支持标准REST软件架构之外，Restlet 框架也提供一个可以极大简化在一个JVM内部驻留多个应用程序的类集。目标是提供RESTful、portable和更灵活的选择来替代已有Servlet API。在下图中，我们能看到Restlets提供了三个类型来管理这些复杂情形。Components能管理几个Virtual Hosts和Applications。

Virutal Hosts支持灵活的配置，比如，几个域名共享相同IP地址，或者相同域名被负载均衡到几个不同的IP地址。最后，我们用Applications来管理相关Restlets、Resources和Representations的集合。除此之外，Applications被确保能在不同的Restlet实现和不同的Virual Hosts上portable和reconfigurable。除此之外，他们也提供像访问日志、请求实体的自动编码、可配置状态页设置等等的重要服务。



为了说明这些类，我们使用一个简单的例子。在这里我们创建一个Component，然后增加一个HTTP服务器Connector到它上面，并监听在端口8182。然后我们创建一个简单的trace Restlet，并把它增加到Component的默认Virtual Host上。默认host会抓住任何还没有路由到一个声明VirtualHost的请求（看Component.hosts属性）。在之后的例子中，我们也引入Application类的用法。注意现在还没有任何的访问日志显示在终端上。

|  |
| --- |
| public class **Part05** extends ServerResource {  public static void main(String[] args) throws Exception {  // Create a new Restlet component and add a HTTP server connector to it  Component component = new Component();  component.getServers().add(Protocol.HTTP, 8182);    // Then attach it to the local host  component.getDefaultHost().attach("/trace", **Part05**.class);    // Now, let's start the component!  // Note that the HTTP server connector is also automatically started.  component.start();  }  @Get  public String toString() {  // Print the requested URI path  return "Resource URI : " + getReference() + '\n' + "Root URI : "  + getRootRef() + '\n' + "Routed part : "  + getReference().getBaseRef() + '\n' + "Remaining part: "  + getReference().getRemainingPart();  }  } |

现在，我们通过在Web浏览器上输入http://localhost:8182/trace/abc/def?param=123来测试。得到的结果如下：

|  |
| --- |
| Resource URI : http://localhost:8182/trace/abc/def?param=123  Root URI : http://localhost:8182/trace  Routed part : http://localhost:8182/trace  Remaining part: /abc/def?param=123 |

## Serving static files

你的Web应用程序会提供像Javadocs一样的静态页面吗？如果有，不需要建立一个Apache服务器，使用Directory类即可。

|  |
| --- |
| // URI of the root directory.  public static final String ROOT\_URI = "file:///c:/restlet/docs/api/";  [...]  // Create a component  Component component = new Component();  component.getServers().add(Protocol.HTTP, 8182);  component.getClients().add(Protocol.FILE);  // Create an application  Application application = new Application() {  @Override  public Restlet createInboundRoot() {  return new **Directory**(getContext(), ROOT\_URI);  }  };  // Attach the application to the component and start it  component.getDefaultHost().attach(application);  component.start(); |

为了运行这个例子，你需要指明ROOT\_URI的有效值，在这里，它是被设置成<file:///c:/restlet/docs/api/>。注意不需要额外的配置，如果想要自定义在文件扩展和元数据之间的映射，或如果想要指明一个index名，就需要使用Application的metadataservice属性。

## Access logging

能够适当地记录Web应用程序的活动是一个常见需求。Restlet Components默认知道如何去生成类似Apache的日志，或者可以自定义。通过充分利用建立在JDK上的日志系统，可以像配置任意标准JDK日志一样配置logger。

为了充分配置日志，需要通过设置系统属性声明一个配置文件：

|  |
| --- |
| System.setProperty("java.util.logging.config.file",  "/your/path/logging.config"); |

配置文件格式请参见：<http://java.sun.com/j2se/1.5.0/docs/api/index.html?java/util/logging/LogManager.html>

## Displaying error pages

另外一个常见需求是自定义状态页。

## 引导对敏感资源的访问

如果需要确保对一些Restlets的访问 的安全，有几个方法可用。一个通用的方法是依靠cookies来识别客户（或客户会话），并检查违反应用程序声明的用户ID和会话ID来决定接入是否应该被授予。

Restlets通过Request或Response中的Cookie和CookieSetting 对象来支持cookie。

也有另外一种基于标准HTTP认证机制的方法。Restlet Engine当前接收证书并放到Basic HTTP scheme中，证书也可以被送到Amazon Web Service Scheme。

当接收到一个调用，开发者能通过ChallengeAuthenticator filter来使用在Request.challengeResponse.identifier/secret中有效且分析过的证书。Filters是一类具体的Restlets，它能在唤醒之前预处理一个调用，并附着在Restlet上，或者在附着的Restlet对调用返回之后处理调用。如果你熟悉Servlet AI，这个概念有点类似Filter接口。看以下的例子，我们如何修改之前的例子来确保访问的安全：

|  |
| --- |
| // Create a simple password verifier  MapVerifier verifier = new MapVerifier();  verifier.getLocalSecrets().put("scott", "tiger".toCharArray());  // Create a Guard  ChallengeAuthenticator guard = new ChallengeAuthenticator(  getContext(), ChallengeScheme.HTTP\_BASIC, "Tutorial");  guard.setVerifier(verifier);  // Create a Directory able to return a deep hierarchy of files  **Directory** directory = new **Directory**(getContext(), ROOT\_URI);  directory.setListingAllowed(true);  guard.setNext(directory);  return guard; |

注意认证和授权决定可以通过继承自Authenticator（比如ChallengeAuthenticator）和Authorizer抽象类的filters来自定义。我们在这简单设置一个用户和密码。为了测试，我们需要使用客户侧Restlet API。

|  |
| --- |
| // Prepare the request  ClientResource resource = new ClientResource("http://localhost:8182/");  // Add the client authentication to the call  ChallengeScheme scheme = ChallengeScheme.HTTP\_BASIC;  ChallengeResponse authentication = new ChallengeResponse(scheme,  "scott", "tiger");  resource.setChallengeResponse(authentication);  // Send the HTTP GET request  resource.get();  if (resource.getStatus().isSuccess()) {  // Output the response entity on the JVM console  resource.getResponseEntity().write(System.out);  } else if (resource.getStatus()  .equals(Status.CLIENT\_ERROR\_UNAUTHORIZED)) {  // Unauthorized access  System.out  .println("Access authorized by the server, check your credentials");  } else {  // Unexpected status  System.out.println("An unexpected status was returned: "  + resource.getStatus());  } |

你能改变被测试client发送的用户ID或密码来检查来自服务器的返回。记住在启动你的client之前，先启动之前的Restlet服务器。注意如果从另外一台机器上测试你的服务器，你需要替换“localhost”。

## URI重写和重定向

Restlet框架的另外一个优势是对cool URIs的支持。

第一个有效工具是Redirector，它可以把一个cool URI重写到另一个URI，并遵循自动化重定向。

它也支持多个重定向类型，通过client/browser的外部重定向和为类似代理行为的connector重定向。在以下例子中，我们将定义一个对我们网站（命名为“mysite.org”）的基于google的搜索服务。“/search”URI识别搜索服务，通过“kwd”参数接收一些关键字。

|  |
| --- |
| // Create a root router  Router router = new Router(getContext());  // Create a Redirector to Google search service  String target = "http://www.google.com/search?q=site:mysite.org+{keywords}";  Redirector redirector = new Redirector(getContext(), target,  Redirector.MODE\_CLIENT\_TEMPORARY);  // While routing requests to the redirector, extract the "kwd" query  // parameter. For instance :  // http://localhost:8182/search?kwd=myKeyword1+myKeyword2  // will be routed to  // http://www.google.com/search?q=site:mysite.org+myKeyword1%20myKeyword2  Extractor extractor = new Extractor(getContext(), redirector);  extractor.extractQuery("keywords", "kwd", true);  // Attach the extractor to the router  router.attach("/search", extractor); |

注意Redirector只需要三个参数。第一个是parent上下文。第二个是定义URI应该如何被重写，基于URI模版。第三个参数定义重定向类型，在这里我们选择客户重定向。

我们也依靠Router类来从原始请求中抽取查询参数“kwd”。如果参数被找到，它被复制进名叫“keywords”的请求属性，并准备在当格式化它的目标URIs时被Redirector使用。

## Routers和分层URIs

除了重定向，我们有另外一个工具来管理cool URIs，即Routers。他们是一类具体的Restlet并且有一些其他Restlets（比如Finders和Filters）能附着在他们上面，并且能自动代表基于一个URI模版的调用。通常，你将设置一个Router作为你应用程序的root。

下面我们向解释如何处理以下URI模式：

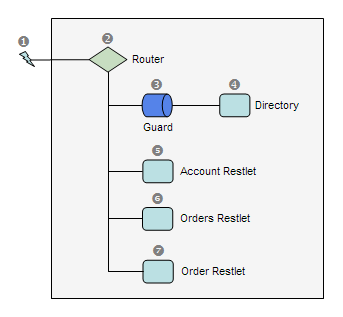
/docs/ to display static files

/users/{user} to display a user account

/users/{user}/orders to display the orders of a particular user

/users/{user}/orders/{order} to display a specific order

事实上这些URIs包含可变部分（在accolades之间），并且没有使用在典型Web container中很难处理的文件扩展。在这里，你仅仅需要使用URI模版把目标Restlets附着到一个Router上。在运行时，最佳匹配请求URI的规则将接收调用，并能唤醒附着在它上面的Restlet。同时，请求中的属性映射将随着URI模版变量的值而被自动更新。



看以下的代码实现，在真实的应用程序中，你将可能要创建分离的子类来代替匿名的。

|  |
| --- |
| // Create a root router  Router router = new Router(getContext());  // Attach a guard to secure access to the directory  Guard guard = new Guard(getContext(), ChallengeScheme.HTTP\_BASIC,  "Restlet tutorial");  guard.getSecrets().put("scott", "tiger".toCharArray());  router.attach("/docs/", guard);  // Create a directory able to expose a hierarchy of files  Directory directory = new Directory(getContext(), ROOT\_URI);  guard.setNext(directory);  // Create the account handler  Restlet account = new Restlet() {  @Override  public void handle(Request request, Response response) {  // Print the requested URI path  String message = "Account of user \""  + request.getAttributes().get("user") + "\"";  response.setEntity(message, MediaType.TEXT\_PLAIN);  }  };  // Create the orders handler  Restlet orders = new Restlet(getContext()) {  @Override  public void handle(Request request, Response response) {  // Print the user name of the requested orders  String message = "Orders of user \""  + request.getAttributes().get("user") + "\"";  response.setEntity(message, MediaType.TEXT\_PLAIN);  }  };  // Create the order handler  Restlet order = new Restlet(getContext()) {  @Override  public void handle(Request request, Response response) {  // Print the user name of the requested orders  String message = "Order \""  + request.getAttributes().get("order")  + "\" for user \""  + request.getAttributes().get("user") + "\"";  response.setEntity(message, MediaType.TEXT\_PLAIN);  }  };  // Attach the handlers to the root router  router.attach("/users/{user}", account);  router.attach("/users/{user}/orders", orders);  router.attach("/users/{user}/orders/{order}", order); |

注意routing假设请求包含一个能识别一个目标资源的绝对目标URI。在请求处理期间，在routers分层中的每一层，资源的基本URI都会被更新。这解释了为什么routers的默认行为是只匹配剩余URI部分的开始，而不是全部。在这种情形中，你也许想要改变默认模式，通过在Router上的“defaultMatchingMode”属性可以很容易做到这一点，或者修改与被Router.attach()方法创建的route相关的模版的“matchingMode”属性。对于这些模式，你能用Template.MODE\_EQUALS或Template.MODE\_STARTS\_WITH常数。

请注意变量的值是直接从URI中抽取，因此没有百分百解码，为了实现这个任务，看看Reference#decode(String)方法。

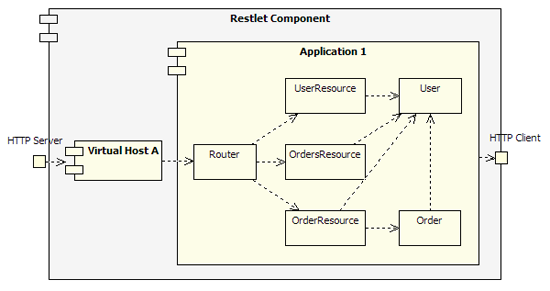
## Reaching target Resources

在之前的例子中，我们充分利用框架的灵活routing特征来路由请求，同时从目标URI中抽取感兴趣的内容。但是，我们不关心请求方法，也不关心客户偏好，会忽略掉客户期待的响应设置。

我们如何连接我们的Restlet资源与后端系统呢？

到目前为止，我们介绍了比传统Servlet API的更好的特征。

总结下来，一个请求包含能识别目标资源的URI。这个信息被存储在Request.resourceRef属性中，并且正如我们所看到的那样，它作为路由的基础。所以，处理请求时的第一个目标是找到在框架中的目标资源，这是一个ServerResource类的实例或更准确的说是ServerResource类的子类的实例。为了帮助我们完成这个任务，我们能使用Finder（Restlet的子类），它从参数中获取一个ServerResource类，并且会在有请求到达时自动实例化它。资源将动态分发调用到（1）匹配上被注解的方法（2）预定义的方法（get()，post()，put()，delete()等）。当然，这个行为能被定制。在Router上有一个attach()方法，它需要两个参数，一个是URI模版，另一个是ServerResource类，它能为你创建一个Finder。现在，让我们看看全局图，它显示了在主框架类之间的关系：



回到代码上来，这里是我们对Application.createRoot()方法的重构。为了简化，我们没有使用Directory来服务静态文件。你能注意到资源类被直接附到router上。

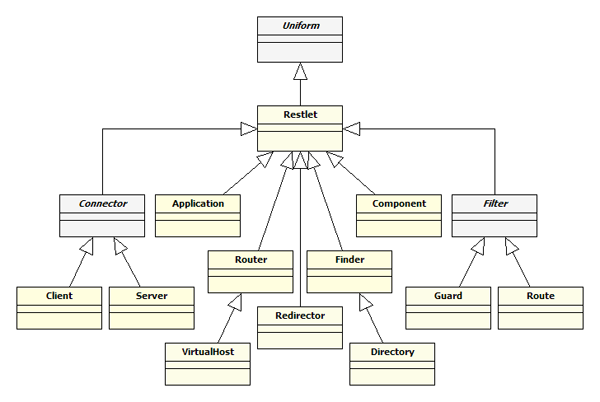
|  |
| --- |
| // Create a router  Router router = new Router(getContext());  // Attach the resources to the router  router.attach("/users/{user}", UserResource.class);  router.attach("/users/{user}/orders", OrdersResource.class);  router.attach("/users/{user}/orders/{order}", OrderResource.class); |

我们最后再看看UserResource资源类。这个类来自org.restlet.resource.ServerResource。我们重写init()方法来取得“user”属性，它会自动从“/users/{user}”URI模版中被抽取出来，并存储它的值在一个成员变量中。在完整的应用程序中，我们查询“user”域对象。最后，我们声明一个toString()方法来支持由@Get注解指示的GET方法。

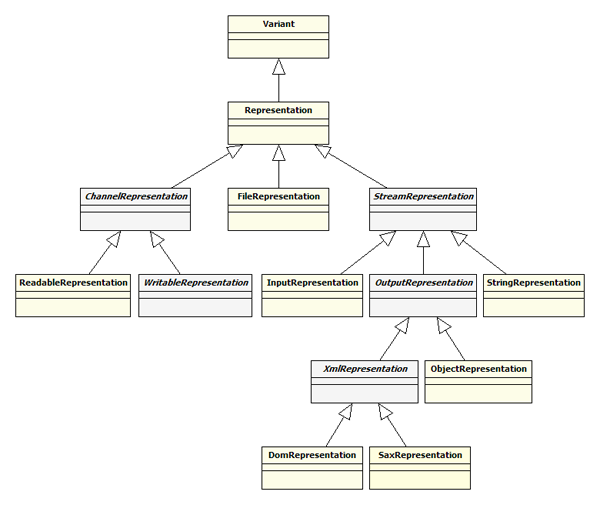
|  |
| --- |
| public class UserResource extends ServerResource {  String userName;  Object user;  @Override  public void init() {  this.userName = (String) getRequestAttributes().get("user");  this.user = null; // Could be a lookup to a domain object.  }  @Get  public String toString() {  return "Account of user \"" + this.userName + "\"";  }  } |

## 总结

本文中主要概念及其关系：



核心表示类的关系：



除了这个教程之外，你需要看看：

Restlet API ：<http://restlet.com/learn/javadocs/2.2/jse/api/>

Restlet Extension：<http://restlet.com/learn/javadocs/2.2/jse/ext/>

Restlet engine：<http://restlet.com/learn/javadocs/2.2/jse/engine/>

所有可用客户和服务器connectors：<http://restlet.com/learn/guide/2.2/core/base/connectors/>

提供可插拔特征（比如servlet containers的整合，动态表示的生成等等）的所有可用extensions：<http://restlet.com/learn/guide/2.2/extensions/editions-matrix>