# Grails和Web Service

web service既是将应用的功能暴露成web api供其它系统调用，主要的实现方式有REST和SOAP两种。

## Grails内置的REST

REST并不是什么新技术，确切的说REST是一种架构模式。REST通过具有特定意义的URL模式和HTTP方法来发送和返回XML或者JSON格式的数据。主要的HTTP方法有：GET、PUT、POST和DELETE。

每一种HTTP方法都对应一种acton类型。例如GET用来获取数据，POST用来创建数据，PUT用来更新，DELETE用来删除。

使用Grails能非常简单的创建一个RESTful的API。创建一个RESTful资源甚至可以简单到只需要一行代码。

### 将Domain类作为RESTful资源

最简单的创建RESTful API的方式就是将一个Domain class转变成一个RESTful资源。只需要给domain class加上一个Resource annotation就行了。

|  |
| --- |
| import grails.rest.\*  @Resource(uri='/books')  class Book {  String title  static constraints = {  title blank:false  }  } |

@Resource自动将domain转变成一个同时支持XML和JSON数据格式的RESTful资源，并自动创建了一个叫做BookController的controller。

假设在BootStrap.groovy中创建了两个book实例：

|  |
| --- |
| def init = { servletContext ->  new Book(title:"The Stand").save()  new Book(title:"The Shining").save()  } |

发送请求：http://localhost:8080/appname/books/1，将返回如下response：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <book id="1">  <title>The Stand</title>  </book> |

如果发送请求：http://localhost:8080/appname/books/1.json，将返回如下response

|  |
| --- |
| {"id":1,"title":"The Stand"} |

如果希望将JSON作为默认的返回格式，可以在@Resource设置formats属性：

|  |
| --- |
| import grails.rest.\*  @Resource(uri='/books', formats=['json', 'xml'])  class Book {  …  } |

formats属性中用到的格式名在Config.grovy中定义

|  |
| --- |
| grails.mime.types = [  …  json: ['application/json', 'text/json'],  …  xml: ['text/xml', 'application/xml']  ] |

通过Rrails的[Content Negotiation](http://grails.org/doc/latest/guide/single.html" \l "contentNegotiation)也可以设置HTTP请求的ACCEPT头指定需要返回的格式而不需要在url请求中加上格式对应的后缀名。例如使用curl命令：

|  |
| --- |
| $ curl -i -H "Accept: application/json" localhost:8080/myapp/books/1  {"id":1,"title":"The Stand"} |

如果发送一个POST请求，则可以创建一个新的book资源：

|  |
| --- |
| $ curl -i -X POST -H "Content-Type: application/json" -d '{"title":"Along Came A Spider"}' localhost:8080/myapp/books  HTTP/1.1 201 Created  Server: Apache-Coyote/1.1 |

PUT请求用来更新资源，DELETE请求用来删除资源：

|  |
| --- |
| $ curl -i -X PUT -H "Content-Type: application/json" -d '{"title":"Along Came A Spider"}' localhost:8080/myapp/books/1  HTTP/1.1 200 OK  Server: Apache-Coyote/1.1  ... |
| $ curl -i -X DELETE localhost:8080/myapp/books/1  HTTP/1.1 204 No Content  Server: Apache-Coyote/1.1  ... |

通过设置@Resource的readOnly属性可以把资源变成只读的。

|  |
| --- |
| import grails.rest.\*  @Resource(uri='/books', readOnly=true)  class Book {  …  } |

### REST资源的URL Mapping

请参考URL Mapping的文档。

### 链接到REST资源

通过link标签，很容易创建一个链接的REST资源的url：

|  |
| --- |
| <g:link resource="${book}">My Link</g:link> |

### 实现REST controller

实现RESTful资源虽然很简单，但是如果想自定义RESTful controller的逻辑或者返回内容那么就需要实现REST controller。

#### 继承RestfulController

最简单的方式就是继承RestfulController，例如：

|  |
| --- |
| class BookController extends RestfulController {  static responseFormats = ['json', 'xml']  BookController() {  super(Book)  }  } |

RestfulController可用的action和对应的HTTP Method和URI如下：

|  |  |  |
| --- | --- | --- |
| **HTTP Method** | **URI** | **Controller Action** |
| GET | /books | index |
| GET | /books/create | create |
| POST | /books | save |
| GET | /books/${id} | show |
| GET | /books/${id}/edit | edit |
| PUT | /books/${id} | update |
| DELETE | /books/${id} | delete |

假如资源是嵌套的，例如获取某个author下的book，其URL Mapping如下：

|  |
| --- |
| "/authors"(resources:'author') {  "/books"(resources:'book')  } |

则可以实现RestfulController的queryForResource方法：

|  |
| --- |
| class BookController extends RestfulController {  static responseFormats = ['json', 'xml']  BookController() {  super(Book)  }  @Override  Book queryForResource(Serializable id) {  Book.where {  id == id && author.id = params.authorId  }.find()  }  } |

#### 将普通controller转变成REST controller

如果不想使用RestfulController，那就需要自己一步步实现各个HTTP Method对应的action。

第一步创建一个controller：

|  |
| --- |
| $ grails create-controller book |

给BookController导入部分imports，并且设置为只读的：

|  |
| --- |
| import grails.transaction.\*  import static org.springframework.http.HttpStatus.\*  import static org.springframework.http.HttpMethod.\*  @Transactional(readOnly = true)  class BookController {  …  } |

各个可用的action和对应的HTTP Method和URI如下：

|  |  |  |
| --- | --- | --- |
| **HTTP Method** | **URI** | **Controller Action** |
| GET | /books | index |
| GET | /books/${id} | show |
| GET | /books/create | create |
| GET | /books/${id}/edit | edit |
| POST | /books | save |
| PUT | /books/${id} | update |
| DELETE | /books/${id} | delete |

第二步，实现index action：

|  |
| --- |
| def index(Integer max) {  params.max = Math.min(max ?: 10, 100)  respond Book.list(params), model:[bookCount: Book.count()]  } |

respont方法会使用Content Negotiation来根据请求的ACCEPT头返回最合适格式的response。

第三步，实现show action。show action根据id返回book resource。实现很简单：

|  |
| --- |
| def show(Book book) {  respond book  } |

Grails会自动根据传入的id http参数查找book 实例，如果找不到对应id的book，则传null给action，这样respond方法就会返回404错误

第四步，实现save action。

|  |
| --- |
| @Transactional  def save(Book book) {  if(book.hasErrors()) {  respond book.errors, view:'create'  }  else {  book.save flush:true  withFormat {  html {  flash.message = message(code: 'default.created.message', args: [message(code: 'book.label', default: 'Book'), book.id])  redirect book  }  '\*' { render status: CREATED }  }  }  } |

第五步，实现update action。update action更新资源。同save action很类似。

|  |
| --- |
| @Transactional  def update(Book book) {  if(book == null) {  render status: NOT\_FOUND  }  else {  book.save flush:true  withFormat {  html {  flash.message = message(code: 'default.updated.message', args: [message(code: 'book.label', default: 'Book'), book.id])  redirect book  }  '\*' { render status: OK }  }  } |

第六步，实现delete action

|  |
| --- |
| @Transactional  def update(Book book) {  if(book == null) {  render status: NOT\_FOUND  }  else {  book.delete flush:true  withFormat {  html {  flash.message = message(code: 'default.deleted.message', args: [message(code: 'Book.label', default: 'Book'), book.id])  redirect action:"index", method:"GET"  }  '\*'{ render status: NO\_CONTENT }  }  } |

## 使用jaxrs插件支持REST

创建grails应用resting。

### 安装插件

对于grails2.1-2.2.X的版本:

|  |
| --- |
| compile ':jaxrs:0.9' |

对于grails2.3的版本:

|  |
| --- |
| compile ':jaxrs:0.10' |

同时,还需要在BuildConfig.groovy中加上maven仓库和Restlet的依赖库：

|  |
| --- |
| mavenLocal()  mavenCentral()  mavenRepo "http://10.100.133.140:8088/nexus/content/groups/public/"  mavenRepo 'http://maven.restlet.org' |

### 创建resource

使用命令create-resource来创建resource。例如：

|  |
| --- |
| grails create-resource com.tapi.demo.api.hello |

上面的命令将创建resource和对应的测试文件：

|  |
| --- |
| grails-app/resources/com/tapi/demo/api/HelloResource.groovy  test/unit/com/tapi/demo/api/HelloResourceTests.groovy |

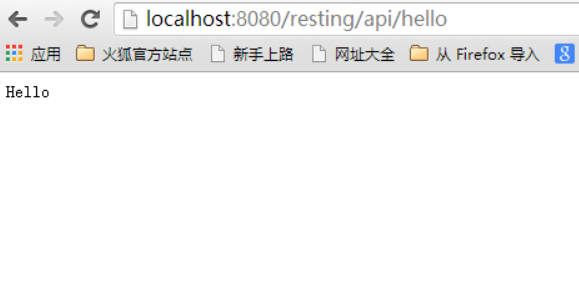
HelloResource.groovy的代码如下：

|  |
| --- |
| package com.tapi.demo.api  import javax.ws.rs.GET  import javax.ws.rs.Path  import javax.ws.rs.Produces  @Path('/api/hello')  class HelloResource {  @GET  @Produces('text/plain')  String getHelloRepresentation() {  'Hello'  }  } |

HelloResource定义一个响应Http GET操作的方法，Http Response包含方法的返回值，response的content type是text/plain。

同样，也可以手工创建resource，只需要将对应的XXXResource.groovy放在grails-app/resources目录下就行了。api/hello就是该resource的访问路径。

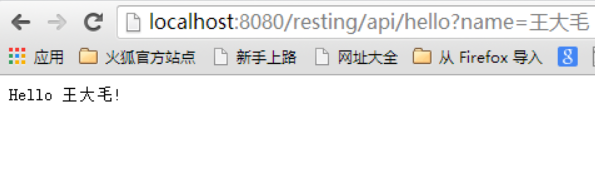
运行应用后就可以使用http://localhost:8080/resting/api/hello来访问该resource了。在浏览器中输入上面的url，请求后返回如下：



同样，我们可以给resource增加请求参数：

|  |
| --- |
| package com.tapi.demo.api  import javax.ws.rs.GET  import javax.ws.rs.Path  import javax.ws.rs.Produces  import javax.ws.rs.QueryParam  @Path('/api/hello')  class HelloResource {  @GET  @Produces('text/plain;charset=UTF-8')  String getHelloRepresentation(@QueryParam('name') String name) {  "Hello ${name}!"  }  } |

用url访问resource，就能得到如下response：



### 生成WADL

插件支持生成REST的WADL文档，访问url http://localhost:8080/resting/application.wadl，就可以得到如下的说明：

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"* standalone=*"yes"*?>  <application xmlns=*"http://wadl.dev.java.net/2009/02"*>  <doc xmlns:jersey=*"http://jersey.java.net/"* jersey:generatedBy=*"Jersey: 1.14 09/09/2012 05:39 PM"*/>  <grammars/>  <resources base=*"http://localhost:8080/resting/"*>  <resource path=*"/api/hello"*>  <method name=*"GET"* id=*"getHelloRepresentation"*>  <request>  <param xmlns:xs=*"http://www.w3.org/2001/XMLSchema"* type=*"xs:string"* style=*"query"* name=*"name"*/>  </request>  <response>  <representation mediaType=*"text/plain; charset=UTF-8"*/>  </response>  </method>  </resource>  </resources>  </application> |

### 将domain class成为RESTFul资源

插件也能很方便的将一个domain class变成一个REST resource。假如有如下一个domain class：

|  |
| --- |
| package com.tapi.demo.api  class Person {  String firstName  String lastName  static *constraints* = {  }  } |

#### 创建REST API

使用命令generate-resources可以创建domain rest相关的代码：

|  |
| --- |
| grails generate-resources com.tapi.demo.api.Person |

运行完毕后，会创建如下代码：

|  |
| --- |
| grails-app\services\com\tapi\demo\api\PersonResourceService.groovy  grails-app\resources\com\tapi\demo\api\PersonResource.groovy  grails-app\resources\com\tapi\demo\api\PersonCollectionResource.groovy |

PersonResourceService.groovy是对domain class Person的GRUD方法的封装：

|  |
| --- |
| package com.tapi.demo.api  import org.grails.jaxrs.provider.DomainObjectNotFoundException  class PersonResourceService {  def create(Person dto) {  dto.save()  }  def read(id) {  def obj = Person.get(id)  if (!obj) {  throw new DomainObjectNotFoundException(Person.class, id)  }  obj  }  def readAll() {  Person.findAll()  }  def update(Person dto) {  def obj = Person.get(dto.id)  if (!obj) {  throw new DomainObjectNotFoundException(Person.class, dto.id)  }  obj.properties = dto.properties  obj  }  void delete(id) {  def obj = Person.get(id)  if (obj) {  obj.delete()  }  }  } |

PersonResource.groovy是单个Person的REST API：

|  |
| --- |
| package com.tapi.demo.api  import static org.grails.jaxrs.response.Responses.\*  import javax.ws.rs.Consumes  import javax.ws.rs.DELETE  import javax.ws.rs.GET  import javax.ws.rs.Produces  import javax.ws.rs.PUT  import javax.ws.rs.core.Response  import org.grails.jaxrs.provider.DomainObjectNotFoundException  @Consumes(['application/xml','application/json'])  @Produces(['application/xml','application/json'])  class PersonResource {  def personResourceService  def id  @GET  Response read() {  ok personResourceService.read(id)  }  @PUT  Response update(Person dto) {  dto.id = id  ok personResourceService.update(dto)  }  @DELETE  void delete() {  personResourceService.delete(id)  }  } |

而PersonCollectionResource.groovy是Person集合的REST API：

|  |
| --- |
| package com.tapi.demo.api  import static org.grails.jaxrs.response.Responses.\*  import javax.ws.rs.Consumes  import javax.ws.rs.GET  import javax.ws.rs.Produces  import javax.ws.rs.Path  import javax.ws.rs.PathParam  import javax.ws.rs.POST  import javax.ws.rs.core.Response  @Path('/api/person')  @Consumes(['application/xml','application/json'])  @Produces(['application/xml','application/json'])  class PersonCollectionResource {  def personResourceService  @POST  Response create(Person dto) {  created personResourceService.create(dto)  }  @GET  Response readAll() {  ok personResourceService.readAll()  }  @Path('/{id}')  PersonResource getResource(@PathParam('id') Long id) {  new PersonResource(personResourceService: personResourceService, id:id)  }  } |

启动应用后查看WADL文件，其对resource的描述内容如下：

|  |
| --- |
| <resource path=*"/api/person"*>  <method name=*"POST"* id=*"create"*>  <request>  <representation mediaType=*"application/xml"* />  <representation mediaType=*"application/json"* />  </request>  <response>  <representation mediaType=*"application/xml"* />  <representation mediaType=*"application/json"* />  </response>  </method>  <method name=*"GET"* id=*"readAll"*>  <response>  <representation mediaType=*"application/xml"* />  <representation mediaType=*"application/json"* />  </response>  </method>  <resource path=*"/{id}"*>  <param xmlns:xs=*"http://www.w3.org/2001/XMLSchema"* type=*"xs:long"*  style=*"template"* name=*"id"* />  <method name=*"GET"* id=*"read"*>  <response>  <representation mediaType=*"application/xml"* />  <representation mediaType=*"application/json"* />  </response>  </method>  <method name=*"PUT"* id=*"update"*>  <request>  <representation mediaType=*"application/xml"* />  <representation mediaType=*"application/json"* />  </request>  <response>  <representation mediaType=*"application/xml"* />  <representation mediaType=*"application/json"* />  </response>  </method>  <method name=*"DELETE"* id=*"delete"* />  </resource>  </resource> |

### 使用REST的客户端插件

新建一个用于客户端测试的项目restingClient。

#### 安装插件

|  |
| --- |
| compile ":rest-client-builder:2.0.1" |

#### 启动应用

为了方便测试，用grails console方式启动应用，运行命令：

|  |
| --- |
| grails console |

运行如下的groovy 脚本：

|  |
| --- |
| import grails.plugins.rest.client.RestBuilder  def rest = new RestBuilder()  def resp = rest.post('http://localhost:8080/resting/api/person') {  accept 'application/xml'  contentType 'application/xml'  xml {  person {  firstName 'Foghorn'  lastName 'Leghorn'  }  }  }  println resp.text  println resp.status |

得到结果如下：

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?><person id="1"><firstName>Foghorn</firstName><lastName>Leghorn</lastName></person>  201 |

上面使用的xml的格式，JSON格式如下：

|  |
| --- |
|  |
|  |

## SOAP

### SOAP clients

调用SOAP web service有两种方式，一种是使用工具产生客户端代码，另一种是手工组建SOAP调用代码。

Grails的[CXF client插件](http://grails.org/plugin/cxf-client)可以使用wsdl2java产生客户端代码，并能支持在运行时动态更新客户端代码和缓存端口引用。

CXF client能提供比WS-client更好的性能。

#### WSDL2JAVA脚本

插件像运行其它命令一样运行WSDL2JAVA。

首先，需要在Config.groovy中定义一个WSDL节点。例如：

|  |
| --- |
| cxf {  client {  simpleServiceClient {  *//used in wsdl2java*  wsdl = "docs/SimpleService.wsdl" *//only used for wsdl2java script target*  namespace = "cxf.client.demo.simple"  client = false *//defaults to false*  bindingFile = "grails-app/conf/bindings.xml"  outputDir = "src/java"  allowChunking = true *//false*  *//used for invoking service*  clientInterface = cxf.client.demo.simple.SimpleServicePortType  serviceEndpointAddress = "${service.simple.url}"  }  *//Another example real service to use against wsd2java script*  stockQuoteClient {  wsdl = "http://www.webservicex.net/stockquote.asmx?WSDL"  clientInterface = net.webservicex.StockQuoteSoap  serviceEndpointAddress = "http://www.webservicex.net/stockquote.asmx"  }  }  } |

在版本1.2.4之后，还可以给wsdl2java提供运行所需的参数。例如：

|  |
| --- |
| cxf {  client {  simpleServiceClient {  *//used in wsdl2java*  wsdl = "docs/SimpleService.wsdl" *//only used for wsdl2java script target*  wsdlArgs = ['-autoNameResolution', '-validate']  *//wsdlArgs = '-autoNameResolution' //single param style*  namespace = "cxf.client.demo.simple"  client = false *//defaults to false*  bindingFile = "grails-app/conf/bindings.xml"  outputDir = "src/java"  *//used for invoking service*  clientInterface = cxf.client.demo.simple.SimpleServicePortType  serviceEndpointAddress = "${service.simple.url}"  }  }  } |

假如wsdl配置之后，就可以运行命令产生客户端代码了：

|  |
| --- |
| grails wsdl2java |

#### 手工运行WSDL2JAVA

也可以手工运行wsdl2java，例如：

|  |
| --- |
| wsdl2java -compile -client -d [output path] [path to wsdl] |

例如：

|  |
| --- |
| C:\projects\cxf-client-demo\docs>c:\apps\apache-cxf-2.4.2\bin\wsdl2java -compile -client -d . -p cxf.client.demo.complex ComplexService.wsdl |

#### 配置插件

安装插件，在BuildConfig.groovy中引入插件：

|  |
| --- |
| compile ":cxf-client:1.6.0" |

安装完插件后，就可以在Config.groovy中进行配置，可配置项有：

|  |
| --- |
| cxf {  client {  [beanName] {  clientInterface = [package and name of wsdl2java -client generated port interface class]  serviceEndpointAddress = [url for the service]  username = [username] //optional - used when secured is true - currently wss4j interceptor  password = [password] //optional - used when secured is true - currently wss4j interceptor  securityInterceptor = [text name of custom bean to use] //optional - defaults to wss4j interceptor  inInterceptors = [list of cxf in interceptors to add to the client] //optional - defaults to []  outInterceptors = [list of cxf out interceptors to add to the client] //optional - defaults to []  inFaultInterceptors = [list of cxf in fault interceptors to add to the client] //optional - defaults to []  outFaultInterceptors = [list of cxf out fault interceptors to add to the client] //optional - defaults to []  enableDefaultLoggingInterceptors = [turn on or off default in/out logging] //optional - defaults to true  secured = [true or false] //optional - defaults to false  connectionTimeout = [Number of milliseconds to wait for connection] //optional - Defaults to 60000 (use 0 to wait infinitely)  receiveTimeout = [Number of milliseconds to wait to receive a response] //optional - Defaults to 30000 (use 0 to wait infinitely)  allowChunking = [true or false] //optional - defaults to false  contentType = [String value of http content type] - defaults to 'text/xml; charset=UTF8'  httpClientPolicy = [text name of custom bean to use] //optional - defaults to null  proxyFactoryBindingId = [binding id uri if required] //optional - defaults to null  secureSocketProtocol = [socket protocol to use for secure service] //optional - defaults to null  wsdlServiceName = [set to enable mime type mapping] //optional - defaults to null  wsdlEndpointName = [may be needed for correct wsdl initialization] //optional - defaults to null  requestContext = [Setting a Request Context Property on the Client Side] //optional - defaults to [:]  tlsClientParameters = [conduit settings for secure services] //optional - defaults to [:]  //wsdl config  wsdl = [location of the wsdl either locally relative to project home dir or a url] //optional - only used by wsdl2java script  wsdlArgs = [custom list of args to pass in seperated by space such as ["-autoNameResolution", "-validate"]] //optional - only used by wsdl2java script  namespace = [package name to use for generated classes] //optional - uses packages from wsdl if not provided  client = [true or false] //optional - used to tell wsdl2java to output sample clients, usually not needed - defaults to false  bindingFile = [Specifies JAXWS or JAXB binding file or XMLBeans context file] //optional  outputDir = [location to output generated files] //optional - defaults to src/java  }  }  } |

在运行时调用service相关的配置有：

|  |  |  |
| --- | --- | --- |
| **Property** | **Description** | Required |
| beanName | 任何不重复的名字，这个是注入到controller或者service中用到bean name. | **Yes** |
| clientInterface | 生成port interface的包名和类名，例如：demo.simple.SimpleServicePortType | **Yes** |
| serviceEndpointAddress | service的调用url地址，可以是环境相关的url | **Yes** |
| username | 用于wss4j认证的用户。 (default: "") | No |
| password | 用于wss4j认证的用户密码。(default: "") | No |
| securityInterceptor | Provide a single bean name as a string to wire in as an out interceptor for apache cxf. If you provide a name for an interceptor, it will be implied that secured=true. If you require the default wss4j interceptor you will not need to set this property, simply set the secured=true and the username and password properties. If you set this to a value then the username and password fields will be ignored as it is expected that you will configure any required property injection in your resources.groovy file. You may also provide your custom security interceptor in the outInterceptors property as well. You would still be required to set secured=true. This is here as a convenience to any existing configured clients that do not wish to switch to using the newer outInterceptors property. See below for examples (default: "") | No |
| inInterceptors | 定义一个或多个bean用于cxf in 拦截器。多个bean时格式可以是"name", "name, name" or ["name","name"] 。bean本身在resource.groovy中定义。 | No |
| outInterceptors | 定义一个或多个bean用于cxf out 拦截器。多个bean时格式可以是"name", "name, name" or ["name","name"] 。bean本身在resource.groovy中定义。 | No |
| inFaultInterceptors | 定义一个或多个bean用于cxf in fault 拦截器。多个bean时格式可以是"name", "name, name" or ["name","name"] 。bean本身在resource.groovy中定义。 | No |
| outFaultInterceptors | 定义一个或多个bean用于cxf out fault 拦截器。多个bean时格式可以是"name", "name, name" or ["name","name"] 。bean本身在resource.groovy中定义。 | No |
| enableDefaultLoggingInterceptors | 如设置成true，则默认的in out 记日志拦截器将被使用。如想使用自定义的日志拦截器，则可以设置属性为false，并且在inInterceptor或者outInterceptor中设置自定义拦截器。如果不想记日志，则可以直接设置属性为false。默认值为true。 | No |
| connectionTimeout | 设置超时时间，单位是毫秒。默认值30000。 | No |
| receiveTimeout | 设置等待response返回的超时时间，单位是毫秒，默认值60000 | No |
| secured | 如果设置成true，cxf 增加username和password参数用于WSS4 (default: false) | No |
| allowChunking | If true will set the HTTPClientPolicy allowChunking for the clients proxy to true. (default: false) | No |
| contentType | 重置http content type，默认是'text/xml; charset=UTF8'. 例如改成"application/soap+xml; charset=UTF-8" 。 | No |
| httpClientPolicy | 使用在resources.groovy中配置的 HTTPClientPolicy bean替代 **connectionTimeout, receiveTimeout and allowChunking 配置.** (default: null) | No |
| proxyFactoryBindingId | The URI, or ID, of the message binding for the endpoint to use. For SOAP the binding URI(ID) is specified by the JAX-WS specification. For other message bindings the URI is the namespace of the WSDL extensions used to specify the binding. If you would like to change the binding (to use soap12 for example) set this value to "<http://schemas.xmlsoap.org/wsdl/soap12/>". (default: "") | No |
| secureSocketProtocol | 加密通道协议.对于https请求类型的web service，默认是"TLS"，也可以是 "SSL", "TLS" 或者 "TLSv1". (default: "") | No |
| wsdl | 远程或者本地的wsdl地址，(default: null) | No |
| wsdlServiceName | The QName of the service you will be accessing. Will be passed into JaxWsProxyFactoryBean. Only needed when using WSDL at run-time to handle things that cannot be captured in Java classes via wsdl2java. (example: '{[http://my.xml.namespace/}TheNameOfMyWSDLServicePorts'](http://my.xml.namespace/%7DTheNameOfMyWSDLServicePorts')) (default: null) | No |
| wsdlEndpointName | The QName of the endpoint/port in the WSDL you will be accessing. Will be passed into JaxWsProxyFactoryBean. May be needed when using WSDL at run-time to handle things that cannot be captured in Java classes via wsdl2java. (example: '{[http://my.xml.namespace/}TheNameOfMyWSDLServicePort'](http://my.xml.namespace/%7DTheNameOfMyWSDLServicePort')) (default: null) | No |
| requestContext | Setting a Request Context Property on the Client Side. (default: [:]) | No |
| tlsClientParameters | Configuration parameters for the secure conduit. (default: [:]) | No |

wsdl2java相关的配置项：

|  |  |  |
| --- | --- | --- |
| **Property** | **Description** | Required |
| wsdl | Location of the wsdl either locally relative to project home dir or a url. (default: "") | No |
| wsdlArgs | A custom list of args to pass in seperated by space such as ["-autoNameResolution","-validate"]. This can also be a single string value such as "-autoNameResolution", but when using multiple custom params you must specify each in a list ["-one val","-two","-three val"] due to limitations with ant. (default: "") | No |
| namespace | Specifies package names to use for the generated code. (default: "use wsdl provided schema") | No |
| client | Used to tell wsdl2java to output sample clients, usually not needed. (default: false) | No |
| bindingFile | Path of binding file to pass to wsdl2java. (default: "") | No |
| outputDir | Password to pass along with request in wss4j interceptor when secured is true. (default: "src/java") | No |

#### 例子

配置：

|  |
| --- |
| *//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**  *// IMPORTANT - these must be set externally to env if you want to refer to them later for use*  *// via cxf. You can also simply hard code the url in the cxf section and NOT refer to a variable*  *// as well.*  service.simple.url = ""  service.complex.url = ""  *// set per-environment service url*  environments {  production {  grails.serverURL = "http://www.changeme.com"  service.simple.url = "${grails.serverURL}/services/simple"  service.complex.url = "${grails.serverURL}/services/complex"  }  development {  grails.serverURL = "http://localhost:8080/${appName}"  service.simple.url = "${grails.serverURL}/services/simple"  service.complex.url = "${grails.serverURL}/services/complex"  }  test {  grails.serverURL = "http://localhost:8080/${appName}"  service.simple.url = "${grails.serverURL}/services/simple"  service.complex.url = "${grails.serverURL}/services/complex"  }  }  cxf {  client {  simpleServiceClient {  clientInterface = cxf.client.demo.simple.SimpleServicePortType  serviceEndpointAddress = "${service.simple.url}"  }  complexServiceClient {  clientInterface = cxf.client.demo.complex.ComplexServicePortType  serviceEndpointAddress = "${service.complex.url}"  }  }  }  *//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\** |

然后就可以在controller、service或者taglib使用：

|  |
| --- |
| class DemoController {  def simpleServiceClient  def complexServiceClient  def simpleServiceDemo = {  SimpleRequest request = new SimpleRequest(age: 100, name: "Bob")  SimpleResponse response = simpleServiceClient.simpleMethod(request)  render(view: '/index', model: [simpleRequest: request, simpleResponse: response])  }  } |

#### 自定义拦截器等内容

请查看插件文档：

<https://github.com/Grails-Plugin-Consortium/grails-cxf-client>

### SOAP servers

CXF插件用于将grails的service或者endpoints代码发布成Apache CXF SOAP web service。

#### 安装插件

在BuildConfig.groovy的plugins块中增加如下代码：

|  |
| --- |
| compile ":cxf:1.1.1" |

#### Endpoint和service

CXF插件引入了一个新的artefact类型endpoint，endpoint类型对象都位于grails-app/endpoints目录下。

插件会自动将grails-app/endpoints和grails-app/services目录下有webservice注解的对象发布成CXF web service。

可以使用create-endpoint和create-endpoint-simple命令分别创建endpoint类型是JAX-WS和SIMPLE类型的endpoint。例如：

|  |
| --- |
| grails create-endpoint com.tapi.grails.demo.hello |

运行上面的命令后，就会在grails-app/endpoints下新建一个HelloEndpoint.groovy类。

也可以使用create-cxf-service 和create-cxf-service-simple命令在grails-app/services目录新建可以发布成web service的service类。例如：

|  |
| --- |
| grails create-cxf-service com.tapi.grails.demo.greet |

上面的命令运行后，就会在grails-app/services目录下增加一个GreetService.groovy类。

通过命令创建的service和endpoint会在应用运行时被插件自动发布成web service。

#### WSDL2JAVA脚本

插件提供了一个方便的Gant脚本用来从wsdl生成java代码。脚本对应的命令名是wsdl-to-java，例如：grails wsdl-to-java "--wsdl=path --mark"

|  |
| --- |
| usage: grails wsdl-to-java --wsdl=<path to wsdl>  [--package=<package>] [--fe=frontend-name] [--db=databinding-name]  [--wv=wsdl-version] [--sn=service-name] [--b=binding-name]  [--catalog=catalog-file-name] [--d output-directory] [--compile]  [--classdir=compile-class-dir] [--client] [--server]  [--impl] [--all] [--ant] [--autoNameResolution] [--exsh=(true/false)]  [--dns=(true/false)] [--dex=(true/false)] [--validate] [--keep] [--noAddressBinding]  [--exceptionSuper] [--reserveClass=classname] [--allowElementReferences<=true>]  [--asyncMethods=foo,bar,...] [--bareMethods=foo,bar,...]  [--mimeMethods=foo,bar,...] [--mark]  Script Options:  -help, --help Prints this help message |

#### 配置插件

插件的默认配置都在插件的配置文件DefaultCxfConfig.groovy内。如果想要自定义配置替换默认配置，则可以将对应的属性配置到Config.groovy中。例如想让SOAP版本默认是1.2，则在Config.groovy中增加如下配置：

|  |
| --- |
| cxf.endpoint.soap12Binding = true |

下面是插件的默认配置：

|  |
| --- |
| */\*\**  *\* Default configuration values for the plugin. You can override in Config.groovy*  *\*/*  cxf {  servlet {  */\*\**  *\* cxf.servlet.loadOnStartup*  *\* <p>*  *\* Specifies the order in which to load the servlet. Lower positive*  *\* values load first, while negative or unspecified mean that the*  *\* sevlet can be loaded at anytime.*  *\*/*  loadOnStartup = 10  */\*\**  *\* cxf.servlet.defaultServlet*  *\* <p>*  *\* When multiple servlets are defined by the {@code cxf.servlets}*  *\* configuration value this specifies the default binding for endpoints*  *\* that don't explicitly define a {@code static servlet = name}. If*  *\* this value is not set then the first alphabetically will be used.*  *\*/*  *//defaultServlet = 'CxfServlet'*  }  */\*\**  *\* cxf.servlets*  *\* <p>*  *\* A map of Servlet Name -> Servlet Mapping Pattern. If multiple Cxf*  *\* servlets are required or a different mapping pattern is needed this*  *\* configuration allows that.*  *\*/*  servlets = [  CxfServlet: '/services/\*'  ]  endpoint {  */\*\**  *\* cxf.endpoint.soap12Binding*  *\* <p>*  *\* Sets the Cxf Server Factory to generate a Soap 1.2 binding. This can*  *\* also be set on a per endpoint basis using*  *\* {@code static soap12 = true}.*  *\*/*  soap12Binding = false  }  } |

#### 通过annotation暴露服务

插件在1.1.0版本之后增加了一个@GrailsCxfEndpoint()用来将class暴露成web service。以下是该annotation的详细信息：

|  |
| --- |
| @Retention(RetentionPolicy.RUNTIME)  @Target(ElementType.TYPE)  public @interface GrailsCxfEndpoint {  String address() default ""  EndpointType expose() default EndpointType.JAX\_WS  boolean soap12() default false  String wsdl() default ""  String[] excludes() default []  String[] inInterceptors() default []  String[] outInterceptors() default []  String[] inFaultInterceptors() default []  String[] outFaultInterceptors() default []  } |

##### address

属性address用来指定服务的发布路径。如果没有指定，则就用service或者endpoint本身的名字。例如，BookService会被发布为/services/book，VeryGoodEndpoint发布成/services/veryGood。

如果想发布成别的地址，则可以设置address属性，例如：

|  |
| --- |
| @GrailsCxfEndpoint(address='/v2/custom/path')  class CarService {  ...  } |

上面的service就会被发布到/services/v2/custom/path。

如果想要将service名字反应到发布路径中，则可以使用#name：

|  |
| --- |
| @GrailsCxfEndpoint(address='/v2/#name') *//as address='/v2/car'*  class CarService {  ...  } |

上面的service将被发布到/services/v2/car。

##### expose

expose属性告诉插件如何发布服务。默认是EndpointType.JAX\_WS。

|  |
| --- |
| @GrailsCxfEndpoint(expose=EndpointType.JAX\_WS)  class CarService {  ...  } |

注意使用JAX\_WS时，还需要对service的方法加上@WebMethod，@WebResult和@WebParam。

除了JAX\_WS以外，还有[SIMPLE](http://cxf.apache.org/docs/simple-frontend-configuration.html):

|  |
| --- |
| expose = EndpointType.SIMPLE |

[wsdl first](http://cxf.apache.org/docs/jax-ws-configuration.html)：

|  |
| --- |
| expose = EndpointType.JAX\_WS\_WSDL  wsdl = 'org/grails/cxf/test/soap/CustomerService.wsdl' *//your path (preferred) or url to wsdl* |

[jax rest](http://cxf.apache.org/docs/jax-rs.html)：

|  |
| --- |
| expose = EndpointType.JAX\_RS |

##### SOAP12

soap12属性用来确定是否使用soap 1.2版本，默认为false。

##### Excludes

如果在使用EndpointType.SIMPLE时有些service方法不希望发布成web service，则可以在excludes属性中列出来。例如：

|  |
| --- |
| @GrailsCxfEndpoint(expose=EndpointType.SIMPLE, excludes=['methodOne', 'methodTwo'])  class CarService {  ...  } |

##### WSDL

wsdl属性用于[wsdl first](http://cxf.apache.org/docs/jax-ws-configuration.html)方式发布web service。

|  |
| --- |
| @WebService(name = 'CustomerServiceWsdlEndpoint',  targetNamespace = 'http://test.cxf.grails.org/',  serviceName = 'CustomerServiceWsdlEndpoint',  portName = 'CustomerServiceWsdlPort')  @GrailsCxfEndpoint(wsdl = 'org/grails/cxf/test/soap/CustomerService.wsdl', expose = EndpointType.JAX\_WS\_WSDL)  class AnnotatedCustomerServiceWsdlEndpoint {  CustomerServiceEndpoint customerServiceEndpoint  List<Customer> getCustomersByName(final String name) {  customerServiceEndpoint.getCustomersByName(name)  }  } |

##### 其它属性

请查阅文档。

#### 注意事项

##### 发布成web service的方法不能使用带有默认值的参数。

例如下面的代码将会导致错误：

|  |
| --- |
| static expose = EndpointType.JAX\_WS  @WebResult(name = 'data')  @WebMethod(operationName = 'getData')  List<Medication> getData(@WebParam(name = 'id') String id, @WebParam(name = 'type') String type, @WebParam(name = 'isGeneric') Boolean isGeneric = true) {  ...  } |

这是因为对于有默认值参数的方法，groovy事实上生成了两个同名的方法，这样就会导致发布的web service有命名冲突。应该创建新的方法：

|  |
| --- |
| static expose = EndpointType.JAX\_WS  @WebResult(name = 'data')  @WebMethod(operationName = 'getDataWithGeneric')  List<Medication> getDataWithGeneric(@WebParam(name = 'id') String id, @WebParam(name = 'type') String type, @WebParam(name = 'isGeneric') Boolean isGeneric) {  this.getData(id, type, isGeneric)  }  @WebResult(name = 'data')  @WebMethod(operationName = 'getDataNoGeneric')  List<Medication> getDataNoGeneric(@WebParam(name = 'id') String id, @WebParam(name = 'type') String type) {  this.getData(id, type)  }  List<Medication> getData(String id, String type, Boolean isGeneric = true) {  ...  } |

### Demo应用

cxf-client：<http://10.100.133.101:8080/cxf-client-demo>

cxf-server: <http://10.100.133.101:8080/grails-cxf-test/services>