REST Plugin

This plugin is only available with Struts 2.1.1 or later

Overview

The REST Pluginprovides high level support for the implementation of RESTful resource based web applicationsThe REST plugin can cooperate with the Convention Plugin to support a zero configuration approach to declaring your actions and results, but you can always use the REST plugin with XML style configuration if you like.

If you prefer to see a working code example, instead of reading through an explanation, you can download the struts2 sample apps and check out the `struts2-rest-showcase` application, a complete WAR file, that demonstrates a simple REST web program.

Features

- Ruby on Rails REST-style URLs
- Zero XML config when used with Convention Plugin
- Built-in serialization and deserialization support for XML and JSON
- Automatic error handling
- Type-safe configuration of the HTTP response
- Automatic conditional GET support

Mapping REST URLs to Struts 2 Actions

The main functionality of the REST plugin lies in the interpretation of incoming request URL's according the RESTful rules. In the Struts 2 framework, this 'mapping' of request URL's to Actions is handled by in implementation of the `ActionMapper` interface. Out of the box, Struts 2 uses the `DefaultActionMapper` to map URL's to Actions via the logic you are probably already familiar with.

Actions or Controllers? Most Struts 2 developers are familiar with the Action. They are the things that get executed by the incoming requests. In the context of the REST plugin, just to keep you on your toes, we'll adopt the RESTful lingo and refer to our Actions as *Controllers*. Don't be confused; it's just a name!

The REST plugin provides an alternative implementation, `RestActionMapper`, that provides the RESTful logic that maps a URL to a give action class (aka 'controller' in RESTful terms) and, more specifically, to the invocation of a method on that controller class. The following section, which comes from the Javadoc for the class, details this logic.

RESTful URL Mapping Logic

This Restful action mapper enforces Ruby-On-Rails REST-style mappings. If the method is not specified (via '!' or 'method:' prefix), the method is "guessed" at using REST-style conventions that examine the URL and the HTTP method. Special care has been given to ensure this mapper works correctly with the codebehind plugin so that XML configuration is unnecessary.

This mapper supports the following parameters:

- `struts.mapper.idParameterName` - If set, this value will be the name of the parameter under which the id is stored. The id will then be removed from the action name. Whether or not the method is specified, the mapper will try to truncate the identifier from the url and store it as a parameter.
- `struts.mapper.indexMethodName` - The method name to call for a GET request with no id parameter. Defaults to **index**.
- `struts.mapper.getMethodName` - The method name to call for a GET request with an id parameter. Defaults to **show**.
- `struts.mapper.postMethodName` - The method name to call for a POST request with no id parameter. Defaults to **create**.

- `struts.mapper.putMethodName` - The method name to call for a PUT request with an id parameter. Defaults to **update**.
- `struts.mapper.deleteMethodName` - The method name to call for a DELETE request with an id parameter. Defaults to **destroy**.
- `struts.mapper.editMethodName` - The method name to call for a GET request with an id parameter and the **edit** view specified. Defaults to **edit**.
- `struts.mapper.newMethodName` - The method name to call for a GET request with no id parameter and the **new** view specified. Defaults to **editNew**.

The following URL's will invoke its methods:

- `GET: /movies` => method=**index**
- `GET: /movies/Thrillers` => method=**show**, id=**Thrillers**
- `GET: /movies/Thrillers;edit` => method=**edit**, id=**Thrillers**
- `GET: /movies/Thrillers/edit` => method=**edit**, id=**Thrillers**
- `GET: /movies/new` => method=**editNew**
- `POST: /movies` => method=**create**
- `PUT: /movies/Thrillers` => method=**update**, id=**Thrillers**
- `DELETE: /movies/Thrillers` => method=**destroy**, id=**Thrillers**

To simulate the HTTP methods PUT and DELETE, since they aren't supported by HTML, the HTTP parameter "_method" will be used.

Or, expressed as a table:

HTTP method	URI	Class.method	parameters
GET	/movie	Movie.index	
POST	/movie	Movie.create	
PUT	/movie/Thrillers	Movie.update	id="Thrillers"
DELETE	/movie/Thrillers	Movie.destroy	id="Thrillers"
GET	/movie/Thrillers	Movie.show	id="Thrillers"
GET	/movie/Thrillers/edit	Movie.edit	id="Thrillers"
GET	/movie/new	Movie.editNew	

Content Types

In addition to providing mapping of RESTful URL's to Controller (Action) invocations, the REST plugin also provides the ability to produce multiple representations of the resource data. By default, the plugin can return the resource in the following content types:

- HTML
- XML
- JSON

There is nothing configure here, just add the conent type extension to your RESTful URL. The framework will take care of the rest. So, for instance, assuming a Controller called Movies and a movie with the id of superman, the following URL's will all hit the

```
http://my.company.com/myapp/movies/superman
http://my.company.com/myapp/movies/superman.xml
http://my.company.com/myapp/movies/superman.xhtml
http://my.company.com/myapp/movies/superman.json
```


Note, these content types are supported as incoming data types as well. And, if you need, you can extend the functionality by writing your own implementations of org.apache.struts2.rest.handler.ContentTypeHandler and registering them with the system.

Usage

This section will walk you through a quick demo. Here are the steps in the sequence that we will follow.

- Setting Up your Project
- Configuring your Project
- Writing your Controllers

Setting Up

Assuming you have a normal Struts 2 application, all you need to do for this REST demo is to add the following two plugins:

- Struts 2 Rest Plugin
- Struts 2 Convention Plugin

Note, you can download the jars for these plugins from Maven Central

Configuration (`struts.xml`)

Just dropping the plugin's into your application may not produce exactly the desired effect. There are a couple of considerations. The first consideration is whether you want to have any non-RESTful URL's coexisting with your RESTful URL's. We'll show two configurations. The first assumes all you want to do is REST. The second assumes you want to keep other non-RESTful URL's alive in the same Struts 2 application.

As with all configuration of Struts 2, we prefer using `<constant/>` elements in our `struts.xml`.

REST Only Configuration

Instruct Struts to use the REST action mapper:

```
<constant name="struts.mapper.class" value="rest" />
```

At this point, the REST mapper has replaced the DefaultActionMapper so all incoming URL's will be interpreted as RESTful URL's.

We're relying on the Convention plugin to find our controllers, so we need to configure the convention plugin a bit:

```
<constant name="struts.convention.action.suffix" value="Controller"/>
<constant name="struts.convention.action.mapAllMatches" value="true"/>
<constant name="struts.convention.default.parent.package" value="rest-default"/>
<constant name="struts.convention.package.locators" value="example"/>
```


Note, you don't have to use the Convention plugin just to use the REST plugin. The actions of your RESTful application can be defined in XML just as easily as by convention. The REST mapper doesn't care how the application came to know about your actions when it maps a URL to an invocation of one of it's methods.

REST and non-RESTful URL's Together Configuration

If you want to keep using some non-RESTful URL's alongside your REST stuff, then you'll have to provide for a configuration that utilizes to mappers.

Plugins contain their own configuration. If you look in the Rest plugin jar, you'll see the `struts-plugin.xml` and in that you'll see some configuration settings made by the plugin. Often, the plugin just sets things the way it wants them. You may frequently need to override those settings in your own `struts.xml`.

First, you'll need to re-assert the extensions that struts knows about because the rest plugin will have thrown out the default `action` extension.

```
  <constant name="struts.action.extension" value="xhtml,,xml,json,action"/>
```

Next, we will configure the `PrefixBasedActionMapper`, which is part of the core Struts 2 distribution, to have some URL's routed to the Rest mapper and others to the default mapper.

```
  <constant name="struts.mapper.class" value="org.apache.struts2.dispatcher.mapper.PrefixBasedActionMapper" />
  <constant name="struts.mapper.prefixMapping" value="/rest:rest,:struts"/>
```

And, again, we're relying on the Convention plugin to find our controllers, so we need to configure the convention plugin a bit:

```
<constant name="struts.convention.action.suffix" value="Controller"/>
<constant name="struts.convention.action.mapAllMatches" value="true"/>
<constant name="struts.convention.default.parent.package" value="rest-default"/>
<constant name="struts.convention.package.locators" value="example"/>
```

Write Your Controller Actions

Once everything is configured, you need to create the controllers. Controllers are simply actions created with the purpose of handling requests for a give RESTful resource. As we saw in the mapping logic above, various REST URL's will hit different methods on the controller. Traditionally, normal Struts 2 actions expose the `execute` method as their target method. Here's a sample controller for a *orders* resource. Note, this sample doesn't implement all of the methods that can be hit via the RESTful action mapper's interpretation of URL's.

```
package org.apache.struts2.rest.example;

public class OrdersController implements ModelDriven<Order> {

    private OrderManager orderManager;
    private String id;
    private Order model;

    // Handles /orders/{id} GET requests
    public HttpHeaders show() {
        model = orderManager.findOrder(id);
        return new DefaultHttpHeaders("show")
            .withETag(model.getUniqueStamp())
            .lastModified(model.getLastModified());
    }

    // Handles /orders/{id} PUT requests
    public String update() {
        orderManager.updateOrder(model);
        return "update";
    }

    // getters and setters
}
```

In this example, the `ModelDriven` interface is used to ensure that only my model, the Order object in this case, is returned to the client, otherwise, the whole `OrdersController` object would be serialized.

Where's ActionSupport? Normally, you extend ActionSupport when writing Struts 2 actions. In these case, our controller doesn't do that. Why, you ask? ActionSupport provides a bunch of important functionality to our actions, including support for i18n and validation. All of this functionality, in the RESTful case, is provided by the default interceptor stack defined in the REST plugin's struts-plugin.xml file. Unless you willfully break your controller's membership in the rest-default package in which that stack is defined, then you'll get all that functionality you are used to inheriting from ActionSupport.

You may wonder why the `show()` method returns a `HttpHeaders` object and the `update()` method returns the expected result code String. The REST Plugin adds support for action methods that return `HttpHeaders` objects as a way for the action to have more control over the response. In this example, we wanted to ensure the response included the ETag header and a last modified date so that the information will be cached properly by the client. The `HttpHeaders` object is a convenient way to control the response in a type-safe way.

Also, notice we aren't returning the usual "success" result code in either method. This allows us to use the special features of the Codebehind Plugin to intuitively select the result template to process when this resource is accessed with the `.xhtml` extension. In this case, we can provide a customized XHTML view of the resource by creating `/orders-show.jsp` and `/orders-update.jsp` for the respective methods.

Advanced Topics

The following sections describe some of the non-standard bells and whistles that you might need to utilize for your application's more non-standard requirements.

Custom ContentTypeHandlers

If you need to handle extensions that aren't supported by the default handlers, you can create your own `ContentTypeHandler` implementation and define it in your `struts.xml`:

```
<bean name="yaml" type="org.apache.struts2.rest.handler.ContentTypeHandler" class="com.mycompany.
MyYamlContentHandler" />
```

If the built-in content type handlers don't do what you need, you can override the handling of any extension by providing an alternate handler. First, define your own `ContentTypeHandler` and declare with its own alias. For example:

```
<bean name="myXml" type="org.apache.struts2.rest.handler.ContentTypeHandler" class="com.mycompany.
MyXmlContentHandler" />
```

Then, tell the REST Plugin to override the handler for the desired extension with yours. In `struts.properties`, it would look like this:

```
struts.rest.handlerOverride.xml=myXml
```

Use Jackson framework as JSON ContentTypeHandler

The default JSON Content Handler is build on top of the JSON-lib. If you prefer to use the Jackson framework for JSON serialisation, you can configure the JacksonLibHandler as Content Handler for your json requests.

First you need to add the jackson dependency to your web application by downloading the jar file and put it under WEB-INF/lib or by adding following xml snippet to your dependencies section in the pom.xml when you are using maven as build system.

```
<dependency>
    <groupId>org.codehaus.jackson</groupId>
    <artifactId>jackson-jaxrs</artifactId>
    <version>1.9.13</version>
</dependency>
```

Now you can overwrite the Content Handler with the Jackson Content Handler in the struts.xml:

```
<bean type="org.apache.struts2.rest.handler.ContentTypeHandler" name="jackson" class="org.apache.struts2.rest.
handler.JacksonLibHandler"/>
<constant name="struts.rest.handlerOverride.json" value="jackson"/>

<!-- Set to false if the json content can be returned for any kind of http method -->
<constant name="struts.rest.content.restrictToGET" value="false"/>

<!-- Set encoding to UTF-8, default is ISO-8859-1 -->
<constant name="struts.i18n.encoding" value="UTF-8"/>
```

Settings

The following settings can be customized. See the developer guide.
For more configuration options see the Convention Plugin Documentation

Setting	Description	Default	Possible Values
`struts.rest.handlerOverride.EXTENSION`	The alias for the `ContentTypeHandler` implementation that handles the EXTENSION value	N/A	Any declared alias for a `ContentTypeHandler` implementation
`struts.rest.defaultExtension`	The default extension to use when none is explicitly specified in the request	`xhtml`	Any extension
`struts.rest.validationFailureStatusCode`	The HTTP status code to return on validation failure	`400`	Any HTTP status code as an integer
`struts.rest.namespace`	Optional parameter to specify namespace for REST services	`/`	eg. /rest
`struts.rest.content.restrictToGET`	Optional parameter, if set to true blocks returning content from any other methods than GET, if set to false, the content can be returned for any kind of method	`true`	eg. put struts.rest.content.restrictToGET = false in struts.properties

Resources

- http://www.b-simple.de/documents - Short RESTful Rails tutorial (PDF, multiple languages)
- RESTful Web Services - Highly recommend book from O'Reilly
- Go Light with Apache Struts 2 and REST - Presentation by Don Brown at ApacheCon US 2008

Version History

From Struts 2.1.1+