ValueStack Outjection Developer's Guide

Version 1.2.1

February 1, 2014

Table of Contents

Disclaimer

The ValueStack Outjection plugin is under Apache License. It's free for anyone to use, modify, or add as part of a commercial or personal project. Many of the packages start with "com.allanshoulders". It is fully understood that no one wants another person's name all over his or her source code. You may change the names of the packages as you like, but please do not remove the credits from the author information in the comments or Javadocs.

I am still very new to contributing to the open source community so please bear with me on mistakes or any non-conformance to open source standards within the Struts and Apache communities.

Finally, please understand that as of now I am the only person on this so-called project. The documentation will have some mistakes here and there. I'll do the best I can to make sure they don't cause confusion or make the document unreadable. I'll try to keep things as brief (but detailed) as possible, as I am very aware that long documents can be a turn off!

Introduction

Before continuing with this document, it is assumed that you are familiar with Struts 2. It assumed that you are comfortable with configuration, Annotations, Interceptors, and finally the ValueStack itself. All of the examples used in this guide are available for download on Github (valuestack-outjection-sample). The Spring framework is used in parts of the sample code. Despite it's popularity, not everyone knows the Spring framework, so I will briefly explain the constructs when needed. The sample code referenced in this guide is built using Maven. If you are unfamiliar with Maven, they have 5 minute tutorial on the Maven website (http://maven.apache.org). There is also an appendix that explains other aspects of the code for the very curious or detail oriented.

When working with Struts 2, I prefer to take advantage of the ValueStack as much as I can. The `ModelDriven` interface allows the developer to push an object onto the `ValueStack` by simply overriding a method named `getModel()`. The problem is that you can only do this once per Action class. This isn't a problem if you are pushing the same object on the ValueStack for every request that the Action class is going to handle.

What happens when you want more than one, or different objects on the ValueStack? What happens when you want to push different objects onto the ValueStack depending on the Action being requested? For the first question, the immediate solution that comes to mind is place your objects in a collection and then have the `getModel()` method return the collection to be pushed onto the ValueStack. That doesn't sound like very viable or clean solution on both the front and back ends. For the second question, you can use horrible if-else logic in the `getModel()` method to return the object you want based on some condition. I don't think that would be a very viable solution either.

A more viable solution could be to simply add getters/setters to your Action class. From there, you can access the objects on the front end by property name. Most importantly, you don't have to implement any `getModel()` methods or deal with its limitations. This seems fine at first, but what if you are using the JSON plugin provided by Struts 2? Since your Action class is always at the top of the ValueStack by default, it will serialize all of your data associated with that Action class instance to JSON. What if the actual data you need is buried deep within a nested object returned by some DAO? You are now stuck with having to sift through a glut of JSON and come up with the appropriate expression in your front end to access it. The JSON plugin allows you filter out objects and such, but do you want to write regular expressions to do your filtering within XML?

In case you are wondering, this is a true story. I needed a simple way to get the objects I wanted on top of the ValueStack when I needed it. I always felt like the Struts 2 framework should provide access to the ValueStack in a much simpler way.

Setup

Like any other Struts 2 plugin, you will need to have the vsoutjection-1.2.1.jar in your WEB-INF/lib. The JAR is not in the Maven Central Repository yet, and that will be a goal for future release. For now, you can download the JAR, place it in your local repository, and then list it as a dependency in your project. Next, you will have to modify your Struts 2 Interceptor stacks to include the ValueStackOutjectionInterceptor. You can do this the following ways:

1. Have your package extend the "vsoutject-default" included with distribution. From there, you can configure which Actions within that package will use the Interceptor stack "vsoutjectStack" included in the distribution. It extends the "defaultStack" so it has most of what you would typically need in most Struts 2 applications.
2. Define your own custom Interceptor stack or override one of the Struts 2 provided Interceptor stacks. Consult the Struts 2 Guides on the Struts website if you aren't sure how to do this.

Setup Examples – Using the default

The sample project uses both XML and Annotations. I believe that the two should be used together rather one versus the other, but that's been debated several times. The sample project has a configuration file named vso-default.xml and it contains the following package definition:

```
<struts>
<package name="vso-sample-default" namespace="/" extends="vsoutject-default,json-default">
       <default-interceptor-ref name="vsoutjectStack"/>
</package>
</struts>
```

All of my actions in the "vso-sample-default" package will now recognize any `@VSOutject` annotations in my Action classes.

Setup Examples – Custom Interceptor stack

If you prefer or have a need that requires a custom Interceptor stack, you can simply add the `ValueStackOutjectionInterceptor` to the stack.

```
<interceptor-stack name="defaultStack">
        .
        .
        .
  <interceptor-ref name="vsoutject"/>
  <interceptor-ref name="params">
       <!-- params interceptor parameters -->
   </interceptor-ref>
        .
        .
        .
</interceptor-stack>
```

Above is an example of overriding the defaultStack and adding the vsoutject Interceptor to the stack. Note that it has been declared **before** the params Interceptor. This is a must if you want things like form values to populate objects pushed to the top of the stack.

Notes About Annotations

As with most frameworks, when taking advantage of annotations (sometimes referred to as "zero-configuration") the classes that utilize annotations must be scanned. Struts 2 uses a plugin called the Convention plugin to activate and provide annotations. If you take a peek at the struts.xml file in the example source code, you will see the Struts constant named "`struts.convention.default.parent.package`" being overridden. The value is the same as the package definition in the vso-default.xml we described above. Struts will take all of your annotated classes and puts them in the package defined in the value of the `struts.convention.default.parent.package` constant. All the annotated classes will by default see only the interceptors defined by that package. I chose this way, because I wanted to keep the code verbosity to minimum. Also, a default setup should suffice for most of our examples. Please consult the Struts 2 guides for more details on the Convention plugin and it's setup requirements. This is only a concern when your project uses annotations.

ValueStack Outjection Rules and Features

This section briefly describes the rules and features of the ValueStack Outjection plugin.

Features

The `@VSOutject` annotation can be applied to both methods and field values since version 1.2.1. You can apply the following attributes: *newInstance, isTopLevel,* and finally *onAction*.

- The *newInstance* attribute is a Boolean value indicating that you want the plugin to create a new instance for you and place it on the ValuesStack. A true value will create a new instance of the object while a false value will not. The default is false.
- The *isTopLevel* attribute is a Boolean value that will determine if the object should be at the very top of the ValueStack. A Boolean true will make this object the very top-level object on the ValueStack. A false value will ensure that it is second from the top-level object on the ValueStack. This value is true by default.
- The *onAction* attribute is String value that should be an Action class alias. If the Action class alias to be executed matches the *onAction* attribute, the annotated object will be push onto the ValueStack. If not, it will be ignored.

Rules

Below are some rules of `@VSOutject` annotation you should take note of:

- The *newInstance* attribute must be a concrete class with a no-arg constructor. An exception will be thrown if the class is not a concrete class (such as an interface) or does not contain a no-arg constructor.
- Annotated fields declared in super classes will **not** be processed.
- When the *newInstance* attribute is used with a method level `@VSOutjection` annotation, the method must be an appropriate getter/setter. Failure to do so will result in an exception.
- The processing of any fields or methods annotated with `@VSOutjection` is unordered.

Basic Examples

Introduction

As stated earlier, all the examples used in this guide are available at the project's home page on Github (http://github.com/itsajskid/vsoutjection-sample). The project is setup as a Maven project, which helps resolve dependencies as well as provides templates to get started with several applications including Struts 2. Maven resolves dependencies by downloading the jars from a central location. Unfortunately, some dependencies of the sample code are not in the Maven central repository. To resolve this issue, the jars not available in the Maven central repository are bundled with the project code. The src/main/resources/dependencies path contains all the jars not in the Maven central repository at the time of this writing. The project references these dependent jars through the use of Maven's system path option. You should be able to compile, test, and deploy the code from the project without any changes. However, you have the option to add the dependencies to you local repository manually, and change the dependencies accordingly.

We used Tomcat 7.0 and Eclipse (Kepler) to run all of the sample code. You can choose whatever Application Server (or container) and IDE you like, but we are not aware of any issues in other tools outside the aforementioned ones.

Example 1 – ModelDriven Style

Our first example uses the `@VSOutject` annotation similar to how you would use the ModelDriven interface. The only difference is that by using @VSOutject, we don't have to implement an interface and override any methods.

```
@Namespace("/basic")
@SuppressWarnings("serial")
public class ModelDrivenStyleVSOutjectionAction extends ActionSupport {

        @Autowired
        private ContactsDao contactsDao;

        @Action("modeldrivenstyle")
        @Override
        public String execute() throws Exception {
                return SUCCESS;
        }

        @VSOutject
        public List<User> getModel() {
                return contactsDao.getAllUsers();
        }

}
```

In the code above located in the ModelDrivenStyleVSOutjectionAction.java source code file, you'll probably first notice `@Autowired` annotation. This is a Spring framework annotation that will inject references into our code for us. We won't don't go into detail about the Spring framework, but trust that the `ContactsDao` object won't be null at runtime. The `ContactsDao` object as its name implies will provide us with operations to obtain our data. The real interesting things happen when the Action class is instantiated, wrapped in an ActionProxy, and makes its way down the Struts 2 workflow.

After instantiation, the class is given to Spring, which injects our ContactsDao with an actual reference. Afterwards, the interceptors defined in the Struts 2 interceptor stack will fire off. The `ValueStackOutjectionInterceptor` class is registered on the interceptor stack in this project. Once this interceptor fires off, it will inspect our class for any field or method level `@VSOutject` annotations and process them. In the case of our `getModel()` method in the example above, it will take the returned value and place it on top of the ValueStack. In the case of example above, it will be a `java.util.List` of `com.allanshoulders.vsoutjection.sample.dao.User` objects.

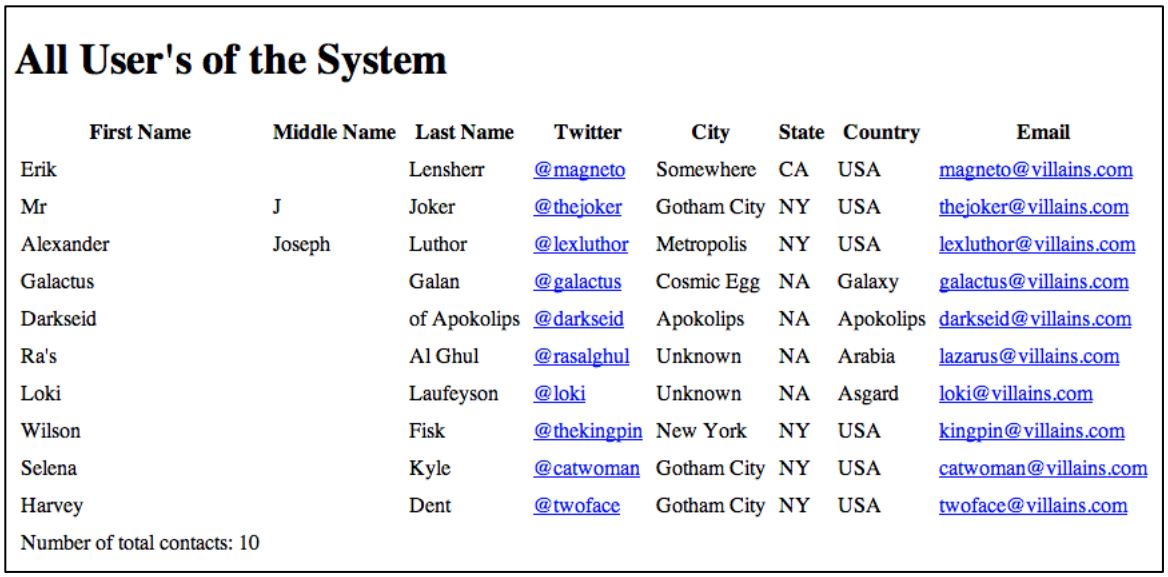
As you are keenly aware, Struts 2 uses OGNL, an expression language. When the result is a dispatcher type, rendering a JSP, Velocity, or FreeMarker template, OGNL expressions are resolved prior to the client receiving any HTML. Most of the time, OGNL expressions are resolved against the properties on the ValueStack. In the case of our example above, let's take a look at the resulting JSP page.

```
<table border="0" cellpadding="5" cellspacing="0">
	<thead>
		<tr>
			<th>First Name</th>
			<th>Middle Name</th>
			<th>Last Name</th>
			<th>Twitter</th>
			<th>City</th>
			<th>State</th>
			<th>Country</th>
			<th>Email</th>
		</tr>
	</thead>

<s:iterator value="iterator()">
	<tr>
		<td><s:property value="firstName"/></td>
		<td><s:property value="middleName"/></td>
		<td><s:property value="lastName"/></td>
		<td>
			<s:a action="getsetcontacts">
				<s:param name="twitterName" value="twitterName"/>
				<s:property value="twitterName"/>
			</s:a>
		</td>
		<td><s:property value="city"/></td>
		<td><s:property value="state"/></td>
		<td><s:property value="country"/></td>
		<td>
			<s:a action="fieldlevelcontacts">
				<s:param name="email" value="email"/>
				<s:property value="email"/>
			</s:a>
		</td>
	</tr>
</s:iterator>

<tfoot>
<tr><td>Number of total contacts: <s:property value="size()"/></td></tr>
</tfoot>
```

The first thing to notice is the `<s:iterator>` tag. The expression refers to the `iterator()` method belonging to a `java.util.List` interface implementation. Since the very top-level object is our `List<User>` object, the `iterator()` expression get resolved against that object. Notice in our `<s:property>` tags, we only list the name of the property that we want, further evidence of the `List<User>` object being at the very top of the ValueStack as we expect it to. Finally, we see the "`size()`" property. This is a required method of any List interface implementation. This is again resolved since our `List<User>` is at the very top of the ValueStack. We'll complete this example with a view of the front-end.



All User's of the System

First Name	Middle Name	Last Name	Twitter	City	State	Country	Email
Erik		Lensherr	@magneto	Somewhere	CA	USA	magneto@villains.com
Mr	J	Joker	@thejoker	Gotham City	NY	USA	thejoker@villains.com
Alexander	Joseph	Luthor	@lexluthor	Metropolis	NY	USA	lexluthor@villains.com
Galactus		Galan	@galactus	Cosmic Egg	NA	Galaxy	galactus@villains.com
Darkseid		of Apokolips	@darkseid	Apokolips	NA	Apokolips	darkseid@villains.com
Ra's		Al Ghul	@rasalghul	Unknown	NA	Arabia	lazarus@villains.com
Loki		Laufeyson	@loki	Unknown	NA	Asgard	loki@villains.com
Wilson		Fisk	@thekingpin	New York	NY	USA	kingpin@villains.com
Selena		Kyle	@catwoman	Gotham City	NY	USA	catwoman@villains.com
Harvey		Dent	@twoface	Gotham City	NY	USA	twoface@villains.com

Number of total contacts: 10

Example 2 – Getter/Setter Approach

The previous example showed how the `@VSOutjection` annotation could be used similar to the `ModelDriven` interface. The `@VSOutjection` annotation can also be used on getter and setter methods defined within an Action class. An example of this kind of usage can be seen in GetterSetterStyleVSOutjectionAction.java source file. A snippet of the code is shown below.

```
@Namespace("/basic")
@SuppressWarnings("serial")
public class GetterSetterStyleVSOutjectionAction extends ActionSupport {

	private User user;
	private ArrayList<User> contacts;

	@Autowired
	private ContactsDao contactsDao;

	@VSOutject(newInstance=true)
	public User getUser() {
		return user;
	}

	public void setUser(User user) {
		this.user = user;
	}

	@VSOutject(newInstance=true)
	public ArrayList<User> getContacts() {
		return contacts;
	}

	public void setContacts(ArrayList<User> contacts) {
		this.contacts = contacts;
	}

	@Action("getsetcontacts")
	@Override
	public String execute() throws Exception {
		User user =
		contactsDao.getUserByTwitterName(this.user.getTwitterName());
		BeanUtils.copyProperties(this.user, user);
		contacts.addAll(contactsDao.getUserContactsById(user.getId()));

		return SUCCESS;
	}
}
```

Building off the previous example, we already know what @Autowired annotation does and what the ContactsDao object's role is. The obvious difference is the presence of getter and setter methods. Another glaring difference is where the @VSOutjection annotation is placed, as well as the options being specified.

When using getter/setter methods, the @VSOutjection should be placed above the getter method. The ValueStackOutjectionInterceptor will call the getter method, and place the returned reference on top of the ValueStack. If the @VSOutjection annotation were placed on the setter method, the ValueStackOutjectionInterceptor would attempt make a call to the setter method. Since the ValueStackOutjectionInterceptor assumes that there is no arguments (as with a valid getter method) it will attempt to call the setter method as such, and an exception will be thrown as a result.

As we mentioned earlier in the *ValueStack Outjection Rules and Features* section, when using the *newInstance* attribute with getter/setter methods, you must have a valid setter method with an assignable or matching type of your getter method. Our setter method meets those criteria. Notice how we are using a `java.util.ArrayList` instead of the `java.util.List` interface. We must use

concrete types with the *newInstance* attribute since the type will be created for us. Remember, the class that will be created automatically must have a no argument constructor.

Limitations such as these will be resolved in future versions of the ValueStackOutjection plugin. We present some solutions to this problem in the *Limitations* section.

Another difference worth looking at is that we have two getter methods that are annotated with @VSOutjection. This means that we will have two objects on top of the ValueStack, an `com.allanshoulders.vsoutjection.sample.dao.User` instance, and an `java.util.ArrayList` instance.

Like the previous example, let's take a look at the JSP page and see what our expressions will look like.

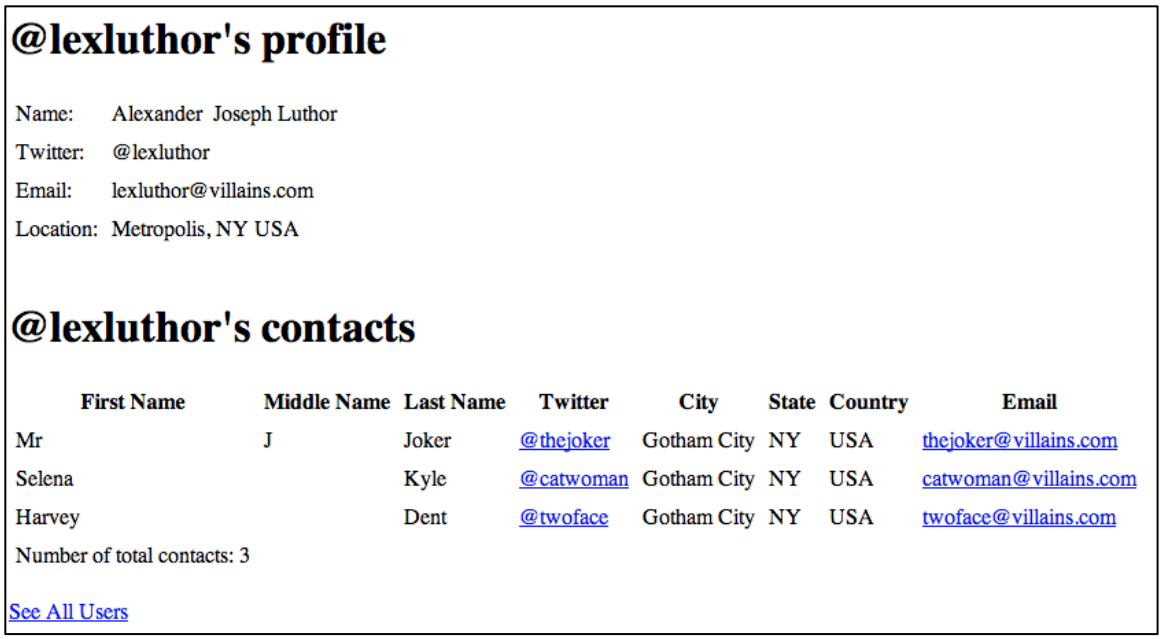
```
<table cellspacing="0" cellpadding="5" border="0">
	<tr>
		<td>Name:</td>
		<td>
			<s:property value="firstName"/> 
			<s:if test="middleName.length > 0">
				<s:property value="middleName"/>
			</s:if>
			<s:property value="lastName"/>
		</td>
	</tr>
	<tr>
		<td>Twitter:</td>
		<td><s:property value="twitterName"/></td>
	</tr>
	<tr>
		<td>Email:</td>
		<td><s:property value="email"/>
	</tr>
	<tr>
		<td>Location:</td>
		<td>
			<s:if test="city != 'Unknown'">
				<s:property value="city"/>,
			</s:if>
			<s:if test="state != 'NA'">
				<s:property value="state"/>
			</s:if>
			<s:property value="country"/>
		</td>
	</tr>
</table>
```

The JSP page part 1.

In part 1 of our JSP page, you can see we have a standard table that displays property values. Just like the previous example, we have explicit property names of objects as our expressions. All of these properties have similar names of our User object. Since our User object has been pushed directly on the ValueStack, the expressions are resolved against our user object.

Part 2 of the JSP page (which we will omit) is the same JSP page shown in the previous example. Again, our ValueStack has an implementation of java.util.List pushed onto it.

We'll close this example similar to the previous example with the resulting page from the client view.



@lexluthor's profile

Name: Alexander Joseph Luthor
Twitter: @lexluthor
Email: lexluthor@villains.com
Location: Metropolis, NY USA

@lexluthor's contacts

First Name	Middle Name	Last Name	Twitter	City	State	Country	Email
Mr	J	Joker	@thejoker	Gotham City	NY	USA	thejoker@villains.com
Selena		Kyle	@catwoman	Gotham City	NY	USA	catwoman@villains.com
Harvey		Dent	@twoface	Gotham City	NY	USA	twoface@villains.com

Number of total contacts: 3

See All Users

Example 3 – Field Level Approach

Perhaps you are in a situation where you would prefer to not have getter/setters for every field within an Action class, but would like to push them onto the ValueStack for some reason. As of version 1.2.1, the @VSOutject can be used on the field level. The class FieldLevelStyleVSOutjection.java below shows an example of this.

```
@Namespace("/basic")
@SuppressWarnings("serial")
public class FieldLevelStyleVSOutjectionAction extends ActionSupport {

        @Autowired
        private ContactsDao contactsDao;

        @VSOutject (newInstance=true)
        private User user;

        @VSOutject (newInstance=true)
        private ArrayList<User> contacts;

        @Action("fieldlevelcontacts")
        @Override
        public String execute() throws Exception {
                User user = contactsDao.getUserByEmail(this.user.getEmail());
                BeanUtils.copyProperties(this.user, user);
                contacts.addAll(contactsDao.getUserContactsById(user.getId()));
                return SUCCESS;
        }
}
```

The code above is the same as the previous example except for the @VSOutject annotations on the two private field methods.

The front-end JSP pages are the exact same as the last example, so we won't show them again.

Struts2-Jquery Examples

Introduction

Struts2-Jquery is an open-source Struts 2 plugin that wraps jQuery and jQuery UI within JSP tags. It also integrates nicely with the Struts 2 architecture (ValueStack, ONGL expressions, etc.) For more information on the framework see the projects home page: http://code.google.com/p/struts2-jquery/

The previous examples used a very basic UI that did not provide basic things like sorting, pagination, column shifting etc. All of this can be accomplished with jQuery/jQuery UI, MooTools, ExtJS, or any other JavaScript widget framework. The Struts2-jQuery project however, allows you access to a library of JavaScript widgets with rich UI capabilities out of the box. All the examples in this section use the Grid widget.

Using this library on a real project was my motivation to write this plugin.

How the Grid plugin works

The Grid plugin is very straightforward. We won't detail all of its features here, but we'll show the most important ones to get it working correctly. This is especially important to understanding where @VSOutjection annotation comes into play. If you already understand how the Grid plugin works, please feel free to skip this section.

The Grid plugin's most important attributes are:

- The *href* attribute identifies the URL of the action class that will be responsible for making the data available to the Grid plugin widget.
- The *dataType* attribute identifies the format of the expected data made available by the action class. The available formats are JSON, XML, HTML, and Text. All of our examples in this section of the guide use JSON.
- The *gridModel* attribute contains the expression used to find your data. More on this in our examples that follow.

The examples we show have attributes in addition to the ones above.

The JsonTable Object

The example shown in the Struts2-jQuery wiki shows all several properties that can be sent to the client and to the server. These properties configure your Grid. For example, if we wanted to set the default page to 2 instead of 1, we could set the page attribute to 2 within the Action class logic. Instead of placing these properties in every Action class (creating boilerplate code), or using Inheritance (not much better than using boilerplate), it's pretty simple to create a POJO and use that as a property within the Action class. The drawback is that you would have to use the dot-property notation in your front end. In most cases that's usually not a big deal.

JSON Consumption

The Grid plugin does not consume data from the ValueStack. The Grid plugin consumes JSON data (other formats are supported, see last section or the Struts2-jQuery wiki for details). That's where *href* attribute comes into play. The Grid makes a request to that URL, and expects JSON data to be returned to it.

The simplest way to resolve this requirement is to have the *url* attribute point to an Action class that will return the JSON data when complete. The easiest way to achieve this is to use the JSON plugin bundled with Struts 2. You would simply have to make sure the packages that declare your action extend the *json-default* package. You can then change the Action class' result type to "json". For more information on the JSON plugin, see the Guides on the Struts 2 website: http://struts.apache.org.

By default, the JSON plugin will serialize the object on the top of the ValueStack. The top object in most cases is the Action class itself. Since the Action class has all the data we need, we should be able to get to it using dot-notation. For example, using our JsonTable object described above, an expression would look like: "jsonTable.gridModel".

For some the dot-property notation doesn't bring to much woe. The real issue is that the JSON plugin will serialize the entire Action class by default. To get around this, you declare the root object. So it's possible we can make the JsonTable the root object giving us back the ability to request our data by property name only ("gridModel"). The JSON plugin allows you to specify regular expressions to include or exclude properties from a class as well as other options.

From my perspective, I'd rather use as much of the JSON plugin's default as possible. I would like to limit how much of the data gets serialized to JSON also. The examples in the next section show how to accomplish this.

Example 4 – Introducing onAction

In this example we introduce a few things not seen in the previous examples:

- Use of the "onAction" attribute.
- Use of Struts2-jQuery Grid plugin.
- Use of JsonTable object described in the previous section.

Let us begin by taking a look at the source file `AllUsersStruts2JQueryAction.java`. A snippet of the code is provided below.

```
@Namespace("/struts2jquery")
@SuppressWarnings("serial")
public class AllUsersStruts2JQueryAction extends ActionSupport {

        @Autowired
        private ContactsDao contactsDao;

        @VSOutject(newInstance=true, onAction="allusers-json")
        private JsonTable<User> usersJsonTable;

        @Action(value="allusers-json", results={@Result(type="json")})
        public String getAllUsersAsJSON() throws Exception {
                usersJsonTable.setGridModel(contactsDao.getAllUsers());
                return SUCCESS;
        }

        @Action("allusers")
        public String getAllUsers() throws Exception {
                return SUCCESS;
        }

}
```

Notice the namespace change from "basic" to "struts2jquery". Don't forget this when you execute the code.

The first @VSOutjection annotation is placed on our `JsonTable` object designed to carry `User`s. We ask the ValueStackOutjectionInterceptor to create a new instance of the JsonTable for us. Since JsonTable has a no-argument constructor, this should be a simple task to carry out. What's new however, is the *onAction* attribute. When specified, this tells the ValueStackOutjectionInterceptor to process this field only when the Action class alias matches the *onAction* attribute. In this case the Action class alias must be "allusers-json". Without specifying the *onAction* attribute, the JsonTable object would have been created and pushed on top of the ValueStack for every Action method alias call in the class. Understanding how the ValueStack works, this could lead to confusing bugs, or unexpected outcomes.

Notice that we have two Action methods: `getAllUsersAsJSON()` and `getAllUsers()`. The names of these methods convey the purpose of each method clearly. One returns JSON and the other returns the typical Dispatcher type (the default). When the "allusers-json" Action class alias is requested, it asks the DAO for all users of the

system. The returned `List` of `User`s are passed on the `JsonTable` object's `setGridModel()` access method. We'll see the importance of this in our examples soon. Since the `JsonTable` object is pushed on top of the ValueStack, it is the object that will be serialized by the JSON plugin provided by the framework. We now have a means to getting all of our data in JSON format.

The `getAllUsers()` method does nothing but return the "SUCCESS " value. You might be wondering the purpose of such a method. It is simply the method that will dispatch the JSP page containing our Grid plugin. The following jsontable.jsp view page will show how all of this is put together.

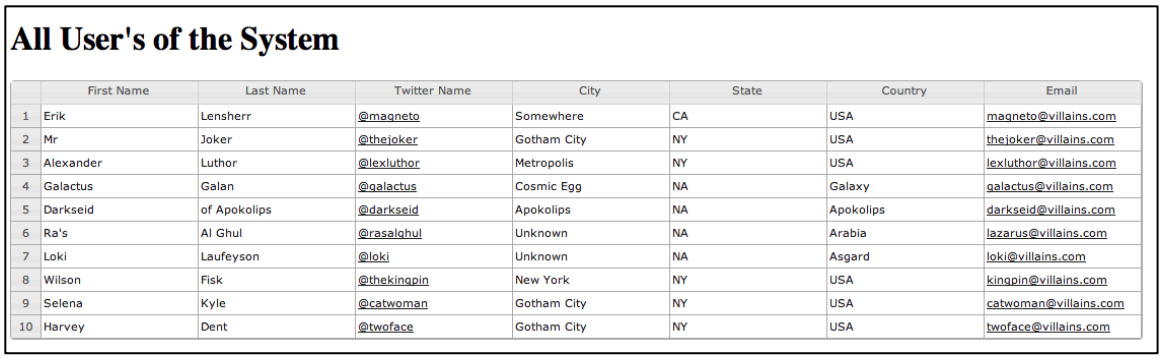
```
<!-- From calling page (allusers.jsp) -->
<s:url var="url" action="allusers-json"/>

<sjg:grid
        id="gridtable"
        gridModel="gridModel"
        dataType="json"
        rownumbers="true"
        href="%{url}"
        loadonce="true"
        sortable="true">

        <sjg:gridColumn name="firstName" index="firstName" title="First Name"/>
        <sjg:gridColumn name="lastName" index="lastName" title="Last Name"/>
        <sjg:gridColumn name="twitterName" index="twitterName" title="Twitter Name"
                formatter="userContactsByTwitterLink"/>
        <sjg:gridColumn name="city" index="city" title="City"/>
        <sjg:gridColumn name="state" index="state" title="State"/>
        <sjg:gridColumn name="country" index="country" title="Country"/>
        <sjg:gridColumn name="email" index="email" title="Email"
        formatter="userContactsByEmailLink"/>

</sjg:grid>
```

You can see where we specify the dataType attribute as "json" and where the href attribute is set to contain our Action class URL. The gridModel attribute contains an expression as it relative to the JSON we get back from the Action class URL. The name attribute in each column is a property of the gridModel's JSON object. The allusers.jsp page is dispatched by the Action class' `getAllUsers()` metho.d Like the other sections, we'll conclude this example with a look at the view from the client.



All User's of the System

	First Name	Last Name	Twitter Name	City	State	Country	Email
1	Erik	Lensherr	@magneto	Somewhere	CA	USA	magneto@villains.com
2	Mr	Joker	@thejoker	Gotham City	NY	USA	thejoker@villains.com
3	Alexander	Luthor	@lexluthor	Metropolis	NY	USA	lexluthor@villains.com
4	Galactus	Galan	@galactus	Cosmic Egg	NA	Galaxy	galactus@villains.com
5	Darkseid	of Apokolips	@darkseid	Apokolips	NA	Apokolips	darkseid@villains.com
6	Ra's	Al Ghul	@rasalghul	Unknown	NA	Arabia	lazarus@villains.com
7	Loki	Laufeyson	@loki	Unknown	NA	Asgard	loki@villains.com
8	Wilson	Fisk	@thekingpin	New York	NY	USA	kingpin@villains.com
9	Selena	Kyle	@catwoman	Gotham City	NY	USA	catwoman@villains.com
10	Harvey	Dent	@twoface	Gotham City	NY	USA	twoface@villains.com

If you notice we have a different look and feel. If you run this example, you can click on the column headers to sort the headers. You drag the columns to shift them, changing the order.

The other Struts2-jQuery examples in the project works similar to this one. You can run it in your own environment if you want to see it work in action. We'll omit the details due to the similarity.

Limitations

Introduction

If you have seen some of the source code or Javadocs pertaining to the ValueStack Outjection Plugin, then you have probably already spotted some flaws. This section serves as guide to possible solutions and workarounds to these flaws and limitations. Many of these will be corrected in future releases of the plugin.

Concrete Types

In many of the examples, we have seen mostly concrete types when it came to field level annotations of @VSOutjection. In particular when we used the *newInstance* attribute. Since that attribute creates an instance of the class type of the field (or if you are using getter/setters the return type of the setter) using Interface types will not work.

To solve this issue you can:

- Forgo the *newInstance* attribute and initialize the annotated field within a constructor or at the field level.
- Use a Dependency Injection framework (i.e. Spring, Weld) that will inject these values for you before the Action class is sent down the Interceptor Stack.

A precautionary note to the second option (DI framework): many of these DI frameworks make use of proxy objects. We have not tested the ValueStack Outjection Plugin with proxy objects. We predict there would be no issues pushing these object onto the ValueStack, but we cannot predict how OGNL expressions, or other plugins that serialize data (i.e. JSON) will work against a proxy object. Many DI frameworks come with options to force real references instead of proxies. We believe the safest thing to do is to force real references if you use these frameworks in conjunction with the ValueStack Outjection Plugin.

Out Synch problems

The infamous out of synch reference is a problem for both the @VSOutjection and ModelDriven interface. The out of synch problem occurs when a reference assigned to a variable is different than the reference on the ValueStack. The obvious (and easiest) way this can happen is to assign a new value to a property after the previous reference has already been pushed onto the ValueStack. For example:

```
@VSOutject(newInstance=true, onAction="allusers-json")
private JsonTable<User> usersJsonTable;

@Action(value="allusers-json", results={@Result(type="json")})
public String getAllUsersAsJSON() throws Exception {
        usersJsonTable = new JsonTable<User>();
        usersJsonTable.setGridModel(contactsDao.getAllUsers());
        return SUCCESS;
}
```

The `JsonTable` object on the ValueStack has a different reference than the one in the Action class. All the users that have been assigned to the gridModel property via setter method will only be visible to Action class. The ValueStack with have a reference with a null gridModel property since we asked the `ValueStackOutjectionInterceptor` to create one for us using the no-argument constructor.

We cannot give a recommendation on how to deal with this problem other than to avoid it. We mentioned it here just to make you aware, and to try to avoid this issue at all costs.

Appendix – Sample Code Further Details

Data Layer

The data module of the sample code uses the Spring framework's JDBC library. The JDBC library from the Spring framework provides templates and other features that allow us to avoid boilerplate code. Spring JDBC is beyond the scope of this document. If you want more information please consult Spring's documentation online at Spring's website: http://spring.io/.

We use JDBC for these examples. While Hibernate is the most popular ORM, JDBC suits our needs just fine. Especially since our queries are very simple, as we are only performing select queries. We can avoid the painful boilerplate and checked exceptions since we are using Spring JDBC's templates.

Finally, the database of choice is Hypersonic. Hypersonic is an in-memory relational database written in Java. It can be used as a persistent database server as well. We chose this because it is popular and very powerful database that can be stored in JAR files. This allows us to use a database without you having to connect to one or set one up yourself. It's also a much better alternative to XML marshaling and un-marshaling at startup, or storing data in file structures. For more information see Hypersonic's website: http://hsqldb.org/.

The most important class in this whole module is the `DaoConfiguration` class. This class contains important objects needed to connect to the database as well as methods to get instances to the DAO class itself. The class is instantiated within the Spring context as a bean. This allows the DAO class (interface) `ContactsDao` to be injected several places in the Struts 2 Action classes.

Project Structure

The project was created using Maven. As stated earlier, Maven is a project management/build tool that yields enormous benefits when used for Java projects. If you want to know more about Maven, please visit the Maven website at http://maven.apache.org.

The project was created using Java SE 7. If you want to change this, you can do it in Maven by simply opening the pom.xml and change the target, source, and fork elements in the plugin section under the artifactId: maven-compiler-plugin. The Maven website introduces more options to take advantage of also.