Struts Result JSON Serializer 1.0 Component Specification

1. Design

We are going to move Direct(www.topcoder.com/direct) from flex to HTML based platform. Release 1 contains Launch Contest page(s). Clients can use this page to create/update studio (nonsoftware) and software contests. Existing back end services will be used.

1.1.1 Configuration approach and format

- Configuration will be done using ONLY IoC. We will not utilize any TopCoder components for object creation or parameter setting.

1.1.1.1 Setter injection

We will also utilize rudimentary setter injection when initializing instances of actions, results, or interceptors. In that sense we will also go with the POJO pattern.

1.1.2 Method Argument Handling

This section describes the generic convention for expected handling of arguments for this component. Any operation that needs to diverge from this convention will be explicitly mentioned in the TCUML documentation. Here are the generic assumptions:

- Any list handled by the component must not contain null elements.
- Unless stated, all input arguments will be required (non-null). String arguments must not be null/empty.

If input parameter conditions are not meet, `IllegalArgumentException` will be used if the argument is null or illegal.

1.2 Design Patterns

MVC – This is the foundation of this framework. STRUTS will do everything using this pattern and the actions, and results (including the AJAXResult defined in this component which plugs in as a strategy) will plug into this framework. In addition the data model is specified as `AggregateDataModel` for the response data. Note that actions will be placed on the ValueStack with their data already properly set.

Strategy – Different implementations of `AJAXDataSerializer` could be provided. The users are expected to provide concrete implementations of the serialization for anything other than JSON. In addition the pre-processing and post-processing can also be plugged in through the `AJAXDataPreProcessor` and `AJAXDataPostProcessor` implementations (both optional)

In addition, to provide some rudimentary IoC we will utilize the ***POJO*** pattern, which requires that each instance of an object that needs to be initialized will have corresponding getters and setters for the specific properties.

1.3 Industry Standards

- HTTP
- Servlets
- AJAX
- JavaBean (POJO)
- IoC
- XML (for configuration)
- JSON
- Gzip compression

1.4 Required Algorithms

There are no complex algorithms here to show. Please consult the TCUML for all development needs.

1.4.1 JSON String structure

The actual JSON strings that we will be returning will have the following general format:

Successful Result Format

```
{
     "result": {
        "name": "<action name>",
        "return": <json_data>
     }
}
```

Error Result Format

```
{
   "error": {
      "name": "<action name>",
      "errorMessage": "<error message>"
   }
}
```

Note that both of these formats will be configurable in the following manner:

- Each of the two strings will be done as a configurable template where tokens can be defined as follows ${token_name} where the token_name would be the specific token being substituted in the string.
- Users will be able to add additional tokens through the setFormatData method in the JSONDataSerializer class. Each of those tokens will need to be set in the format templates using the syntax specified above. Thus if we want to pass some new data into the template under for example extraData key then we place the data under the "extraData" key and we also create a token named "${extraData}" in the template and the serializer will fetch the data form the map and replace the token with this data in the string.
- Three special tokens will be used for the actual JSON data that is retrieved from the aggregate mode:
  - ${result} which will simply mean the result data from the aggregate model to be placed in the JSON result string.
  - ${error} which will simply mean the error data we got from the aggregate data model
  - ${action} which will stand for the action name that we fetch from AbstractAction.

1.4.1.1 Serialization and the json_data format

The actual serialization is done through a third-part lib:
JSON-lib (http://json-lib.sourceforge.net/) library.
Note that the library already properly serializes both POJOs as well as Lists of POJOs so we do not need to do anything more than the following assuming that we have a List or an Object data:

```
JSON json = JSONSerializer.toJSON( data );
String json_data = json.toString();
```

This is all that we need to do.

What we will get on the JSON side will basically depend on the specific bean or list of beans that we are serializing. Here we will present the specific rules, but the front-end developers will need to consult the JSON-lib documentation for the specifics.

- JavaBean Properties are serialized simply as normal “name”: “value” pairs in the JSON string where name will be the field name (for example for a field named “file” we will have the JSON name of “file”.
- Java Lists will be serialized as JSON arrays where the name of the array will be the name of the property under which the list was stored. In case the List is a top level list (i.e. the actual object being serialized is a list_ then in your case it will always be serialized under “result”
- Java Maps will also be serialized as an array of simple values.

Please consult the examples here for more information: http://json-lib.sourceforge.net/usage.html

Note that the RS requires that we be able to take care of the following entities:
- StudioCompetition
- SoftwareCompetition
- List<Category>
- List<Technology>
- ValidationErrors
- Exception (only the message)
- Project
- ProjectData
- List<UploadedDocument> (without the content bytes)
- List< CompDocumentation>
- List<CapacityData>

In fact JSON-Lib is not specific to any given object type as long as it follows simple rules (i.e. for POJOs, Lists, and Maps) as specified above we do not need to know anything about these entities and thus this solution is agnostic.

The only entity that we are aware of is the `UploadedDocument`, which we prune in the provided pre processor.

1.5 Component Class Overview

1.5.1 com.topcoder.service.actions.ajax

AJAXResult
This result processes the result data into an AJAX compatible response that can be then The injected AJAXModelSerializer performs the actual serialization to specific AXAJ representation. In addition a preprocessing and post processing modules can also be injected for complete processing abstraction. This result also provides the abilities to compress response, disable response cache, and set custom response content type and charset.

AJAXDataSerializer
This interface defines the contract to serialize an Object to a string representation, typically a JSON string.

AJAXResultPreProcessor
This interface defines the contract to pre-process an Object. Such preprocessing will simply do something with the input (like a translation, conversion, or perhaps pruning of

sensitive data) and return the resulting output. This will typically be utilized *before* serialization with AJAXDataSerializer

AJAXResultPostProcessor
This interface defines the contract to post-process a string. Such post-processing will simply do something with the input (like a additional translation, conversion, or perhaps pruning of sensitive data) and return the resulting output.
This will typically be utilized *after* serialization with AJAXDataSerializer. Currently there is no default implementation provided.

1.5.2 com.univision.guessandwin.actions.ajax.serializers

JSONDataSerializer
This implementation provides a JSON serialization of a Java Object.
There are two possible outcomes: a successful result or an error result.
A JSON response would have the following two formats when serialized (which are configurable as specified in CS section 1.4.1):

Successful Result Format

```
{
  "result": {
    "name": “$[action]”,
    "return": $[result]
  }
}
```

Error Result Format

```
{
  "error": {
    "name": “$[action]”,
    "errorMessage": “$[error]”
  }
}
```

NOTE: action_json_data will be simply the data object as serialized by the Json-lib (http://json-lib.sourceforge.net/) library that we are utilizing

1.5.3 com.topcoder.service.actions.ajax.processors

DefaultAJAXResultPreProcessor
This is an implementation of the AJAXResultPreProcessor contract, which simply removes some (byte based) data from the objects before serialization. In this implementation we will simply remove the byte[] contents from the UploadedDocument instances.

1.6 Component Exception Definitions

1.6.1 Custom exceptions

AJAXDataSerializationException
This is a serialization exception, which alerts the caller that specific data could not be serialized for AJAX-based processing.

AJAXDataPostProcessingException
This is a post-processing exception, which will alert the caller that the specific data object could not be post-processed.

AJAXDataPreProcessingException
This is a pre-processing exception, which will alert the caller that the specific data object could not be pre-processed.

1.7 Thread Safety

The data model that STRUTS places on the ValueStack is not thread safe. This should be fine since each user has data model different instances with life-cycle bound to the request/response round-trip, thus the data model will not be shared across user threads. So it can be considered being used in a thread safe manner.

The AJAXResult is technically not thread safe since it has mutable states, and those states are intent to be injected by Spring. But it is possible that the configuration of AJAXResult may differ when servicing different actions. So in order to be thread-safe, the administrator will need to ensure that different instances of AJAXResult are used when servicing different actions. Thus we can assume that on these two conditions AJAXResult implementation will be thread-safe.

Neither the AJAXResultPreProcessor nor the AJAXResultPostProcessor are required to be thread-safe. AJAXDataSerializer implementation should be conditionally thread-safe which means that they should be thread safe after initialization (i.e. we can assume that any configuration will only be done once)

2. Environment Requirements

2.1 Environment

- Development language: Java 1.5, J2EE 1.5
- Compile target: Java 1.5, J2EE 1.5
- QA Environment: JBoss 4.0.2, Informix 11

2.2 TopCoder Software Components

- **Struts Framework Version 1.0** – This is where the `AbstractAction` and `AggregateDataModel` that we use in this component are defined.
- **Studio Service 1.3** – This is where the entities are defined. Specifically the one we use explicitly is the `UploadedDocument` entity. Note that all the other entities are currently not specified as this component is agnostic to their structure (i.e. it goes through Object)

NOTE: The default location for TopCoder Software component jars is../lib/tcs/COMPONENT_NAME/COMPONENT_VERSION relative to the component installation. Setting the tcs_libdir property in topcoder_global.properties will overwrite this default location.

2.3 Third Party Components

- Struts 2.1.1+
- Spring 3.0 (used for create objects and configure them by IoC injection)
- Json-lib (http://json-lib.sourceforge.net/) - is be used for the JSON serialization.

NOTE: The default location for 3rd party packages is ../lib relative to this component installation. Setting the ext_libdir property in topcoder_global.properties will overwrite this default location.

3. Installation and Configuration

3.1 Package Name

com.topcoder.service.actions.ajax
com.topcoder.service.actions.ajax.serializers

3.2 Configuration Parameters

3.2.1 Configure Objects by Spring

This component uses Spring to create objects. To Integrate Spring & Struts:

1. Add "*struts.objectFactory* = *spring*" in *struts.properties* to indicate Struts to use Spring for object creation.
2. Add "*org.springframework.web.context.ContextLoaderListener*" listener class in *web.xml*.
3. And the objects have proper setters so that Spring can inject configured values.

3.3 Dependencies Configuration

See CS of dependencies components for their configuration.

4. Usage Notes

4.1 Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute 'ant test' within the directory that the distribution was extracted to.

4.2 Required steps to use the component

Deploy the application to J2EE container, for example:

```
sample.war
    -- WEB-INF
       -- applicationContext.xml
       -- web.xml
       -- classes
          -- struts.xml
          -- struts.properties
          -- ……
       -- lib
          -- ….. libraries
     -- example
          -- …… jsp pages
```

4.3 Demo

Since the actual classes are used within STRUTS we cannot easily show the user interaction. Thus we will mostly describe it.

4.3.1 Configuration example

Note that this is just a sample which highlight the specifics of AJAXResult configuration:

A snippet of **application.xml**:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
"http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans default-autowire="byName">
<!-- A custom serializer -->
      <bean id="dataSerializer"
class="com.topcoder.service.actions.ajax.serializers.JSONDataSeri
alizer" scope="prototype">
            <property name="jsonResultTemplate" value="It's a
demo : {result: {name: $[action], return:
$[result]}}"></property>
      </bean>
<!-- AJAXResult. Note the scope is "prototype". -->
      <bean id="ajaxresult"
class="com.topcoder.service.actions.ajax.CustomFormatAJAXResult"
scope="prototype">
            <property name="enabledGzip" value="true"></property>
            <property name="noCache" value="false"></property>
            <property name="contentType"
value="application/json"></property>
            <property name="charset" value="UTF-8"></property>
            <property name="dataSerializer">
                  <ref local="dataSerializer"/>
            </property>
      </bean>

      <bean id="pj"
class="com.topcoder.service.actions.ajax.Project"
scope="prototype">
        <property name="id" value="100"></property>
        <property name="name" value="project_name"></property>
            <property name="description"
value="project_description"></property>
      </bean>

      <bean id="mockAction"
class="com.topcoder.service.actions.ajax.MockAction"
scope="prototype">
            <property name="ajaxResult">
                  <ref local="ajaxresult"/>
            </property>
            <property name="project">
                  <ref local="pj"/>
            </property>
      </bean>

</beans>
```

The configuration of struts, here is the struts.xml :

```
<?xml version="1.0" encoding="UTF-8" ?>

<!DOCTYPE struts PUBLIC
    "-//Apache Software Foundation//DTD Struts Configuration
2.0//EN"
    "http://struts.apache.org/dtds/struts-2.0.dtd">
```

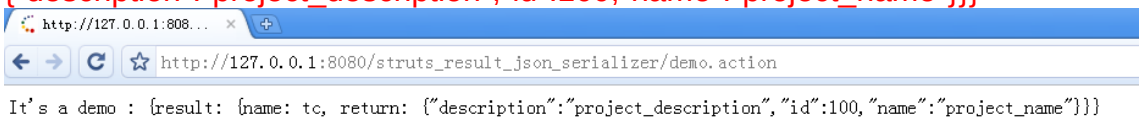
```
<struts>
    <!--  Overwrite Convention -->
    <constant name="struts.convention.action.suffix"
value="Controller"/>
    <constant name="struts.convention.action.mapAllMatches"
value="true"/>
    <constant name="struts.convention.default.parent.package"
value="rest-default"/>

    <constant name="struts.convention.package.locators"
value="example"/>

   <constant name="struts.objectFactory"
value="org.apache.struts2.spring.StrutsSpringObjectFactory"/>


     <package name="default" namespace="/" extends="struts-
default">
        <default-action-ref name="index" />
         <action name="demo" class="mockAction">
                  <result name="input">/index.jsp</result>
            <result name="SUCCESS">/index.jsp</result>
        </action>
    </package>

</struts>
```

The configuration of web.xml :

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app version="2.5"
 xmlns="http://java.sun.com/xml/ns/javaee"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
 http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd">

   <listener>
    <listener-class>
        org.springframework.web.context.ContextLoaderListener
    </listener-class>
      </listener>

      <filter>
            <filter-name>struts</filter-name>
            <filter-
class>org.apache.struts2.dispatcher.FilterDispatcher</filter-
class>
      </filter>

      <context-param>
         <param-name>contextConfigLocation</param-name>
         <param-value>/WEB-INF/application.xml</param-value>
      </context-param>

      <filter-mapping>
         <filter-name>struts</filter-name>
```

```
        <url-pattern>/*</url-pattern>
      </filter-mapping>

  <welcome-file-list>
    <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>

</web-app>
```

DEMO Step :
1. Put all the files
tests_files/struts_result_json_serializer.war
into %JBOSS_HOME_PATH%/server/default/deploy/struts_result_json_serializer.war
2. run jboss.
3. input http://127.0.0.1:8080/struts_result_json_serializer/demo.action in your explorer to see the demo : the result should be :
It's a demo : {result: {name: tc, return: {"description":"project_description","id":100,"name":"project_name"}}}



4 you can change the java been set in application.xml to see different result.
5 stop jboss

4.3.2 *Basic Flow*

The basic flow is as follows:

- User asks for a resource and this is mapped to an action
- The action retrieves the data and places it in the ValueStack (actually the STRUTS framework does that)
- That data is the passed on to AJAXResult, which serializes the processing of the give action into JSON.
- This is then streamed to the JavaScript caller.

4.3.3 *API usage*

```
// create a CustomFormatAJAXResult instance
  CustomFormatAJAXResult instance = new CustomFormatAJAXResult();
  // create an invocation
  MockActionInvocation invocation = new MockActionInvocation();
  MockAbstractAction action = new MockAbstractAction();
  AggregateDataModel dataModel = new AggregateDataModel();

  // create a java bean object
  Project project = new Project();
  project.setId(100);
  project.setName("TC_Project");
  project.setDescription("Project description");

  dataModel.setData("result", project);
  dataModel.setAction("tc");
```

```
    action.setModel(dataModel);

    // create HttpServletResponse instance
    invocation.setAction(action);
    MockHttpServletResponse response = new
        MockHttpServletResponse();
    ServletActionContext.setResponse(response);

    // invoke the execute
    instance.execute(invocation);

// here is the json result
 {"result":
      {"name": "tc",
      "return":
            {"description":"Project description",
             "id":100,
             "name":"TC_Project"}
       }
    }
```

5. Future Enhancements

None.