Type Conversion

Routine type conversion in the framework is transparent. Generally, all you need to do is ensure that HTML inputs have names that can be used in OGNL expressions. (HTML inputs are form elements and other GET/POST parameters.)

2truenone

Built in Type Conversion Support

Type Conversion is implemented by XWork.

{snippet:id=javadoc|javadoc=true|url=com.opensymphony.xwork2.conversion.impl.XWorkBasicConverter}

- Enumerations
- BigDecimal and BigInteger

Relationship to Parameter Names

There is no need to capture form values using intermediate Strings and primitives. Instead, the framework can read from and write to properties of objects addressed via OGNL expressions and perform the appropriate type conversion for you.

Here are some tips for leveraging the framework's type conversion capabilities:

- Use OGNL expressions - the framework will automatically take care of creating the actual objects for you.
- Use JavaBeans! The framework can only create objects that obey the JavaBean specification, provide no-arg constructions and include getters and setters where appropriate.
- Remember that *person.name* will call **getPerson().setName()**. If the framework creates the Person object for you, it remember that a `setPerson` method must also exist.
- The framework will not instantiate an object if an instance already exists. The PrepareInterceptor or action's constructor can be used to create target objects before type conversion.
- For lists and maps, use index notation, such as *people[0].name* or *friends['patrick'].name*. Often these HTML form elements are being rendered inside a loop. For JSP Tags, use the iterator tag's status attribute. For FreeMarker Tags, use the special property ${foo_index}[].
- For multiple select boxes, it isn't possible to use index notation to name each individual item. Instead, name your element *people.name* and the framework will understand that it should create a new Person object for each selected item and set its name accordingly.

Creating a Type Converter

Create a type converter by extending StrutsTypeConverter. The Converter's role is to convert a String to an Object and an Object to a String.

public class MyConverter extends StrutsTypeConverter { public Object convertFromString(Map context, String[] values, Class toClass) { ..... } public String convertToString(Map context, Object o) { ..... } }
To allow Struts to recognize that a conversion error has occurred, the converter class needs to throw XWorkException or preferably TypeConversionException.

Applying a Type Converter to an Action

Create a file called 'ActionClassName-conversion.properties' in the same location of the classpath as the Action class itself resides.

Eg. if the action class name is MyAction, the action-level conversion properties file should be named 'MyAction-conversion.properties'. If the action's package is com.myapp.actions the conversion file should also be in the classpath at /com/myapp/actions/.

Within the conversion file, name the action's property and the Converter to apply to it:

syntax: <propertyName> = <converterClassName> point = com.acme.PointConverter person.phoneNumber = com.acme.PhoneNumberConverter

Type conversion can also be specified via Annotations within the action.

Applying a Type Converter to a bean or model

When getting or setting the property of a bean, the framework will look for "classname-conversion.properties" in the same location of the **classpath** as the target bean. This is the same mechanism as used for actions.

Example: A custom converter is required for the Amount property of a Measurement bean. The Measurement class cannot be modified as its located within one of the application's dependencies. The action using Measurement implements ModelDriven<Measurement> so it cannot apply converters to the properties directly.
Solution: The conversion file needs to be in the same location of the classpath as Measurement. Create a directory in your source or resources tree matching the package of Measurement and place the converters file there.
eg. for com.acme.measurements.Measurement, create a file in the application source/resources at /com/acme/measurements/Measurement-conversion.properties:

none # syntax: <propertyName>=<converterClassName> amount=com.acme.converters.MyCustomBigDecimalConverter

Applying a Type Converter for an application

Application-wide converters can be specified in a file called xwork-conversion.properties located in the root of the classpath.

syntax: <type> = <converterClassName> java.math.BigDecimal = com.acme.MyBigDecimalConverter

A Simple Example

{snippet:id=javadoc|javadoc=true|url=com.opensymphony.xwork2.conversion.impl.XWorkConverter} {snippet:id=i18n-note|javadoc=true|url=com.opensymphony.xwork2.conversion.impl.XWorkConverter}

The framework ships with a base helper class that simplifies converting to and from Strings, `org.apache.struts2.util.StrutsTypeConverter`. The helper class makes it easy to write type converters that handle converting objects to Strings as well as from Strings.

From the JavaDocs:

{snippet:id=javadoc|javadoc=true|url=org.apache.struts2.util.StrutsTypeConverter}

Advanced Type Conversion

The framework also handles advanced type conversion cases, like null property handling and converting values in Maps and Collections, and type conversion error handling.

Null Property Handling

Null property handling will automatically create objects where null references are found.

{snippet:id=javadoc|javadoc=true|url=com.opensymphony.xwork2.conversion.impl.InstantiatingNullHandler} {snippet:id=example|javadoc=true|url=com.opensymphony.xwork2.conversion.impl.InstantiatingNullHandler}

Collection and Map Support

Collection and Map support provides intelligent null handling and type conversion for Java Collections.

The framework supports ways to discover the object type for elements in a collection. The discover is made via an *ObjectTypeDeterminer*. A default implementation is provided with the framework. The Javadocs explain how Map and Collection support is discovered in the `DefaultObjectTypeDeterminer`.

{snippet:id=javadoc|javadoc=true|url=com.opensymphony.xwork2.conversion.impl.DefaultObjectTypeDeterminer}

Additionally, you can create your own custom `ObjectTypeDeterminer` by implementing the `ObjectTypeDeterminer` interface. There is also an optional ObjectTypeDeterminer that utilizes Java 5 generics. See the Annotations page for more information.

Indexing a collection by a property of that collection

It is also possible to obtain a unique element of a collection by passing the value of a given property of that element. By default, the property of the element of the collection is determined in *Class*-conversion.properties using `KeyProperty_xxx=yyy`, where xxx is the property of the bean *Class* that returns the collection and yyy is the property of the collection element that we want to index on.

For an example, see the following two classes:

MyAction.javasolid /** * @return a Collection of Foo objects */ public Collection getFooCollection() { return foo; } Foo.javasolid /** * @return a unique identifier */ public Long getId() { return id; }

To enable type conversion, put the instruction `KeyProperty_fooCollection=id` in the `MyAction-conversion.properties` file. This technique allows use of the idiom `fooCollection(someIdValue)` to obtain the Foo object with value `someIdValue` in the Set `fooCollection`. For example, `fooCollection(22)` would return the Foo object in the `fooCollection` Collection whose `id` property value was 22.

This technique is useful, because it ties a collection element directly to its unique identifier. You are not forced to use an index. You can edit the elements of a collection associated to a bean without any additional coding. For example, parameter name `fooCollection(22).name` and value `Phil` would set name the Foo Object in the `fooCollection` Collection whose `id` property value was 22 to be Phil.

The framework automatically converts the type of the parameter sent in to the type of the key property using type conversion.

Unlike Map and List element properties, if `fooCollection(22)` does not exist, it will not be created. If you would like it created, use the notation `fooCollection.makeNew[index]` where *index* is an integer 0, 1, and so on. Thus, parameter value pairs `fooCollection.makeNew[0]=Phil` and `fooCollection.makeNew[1]=John` would add two new Foo Objects to `fooCollection --` one with name property value `Phil` and the other with name property value `John`. However, in the case of a Set, the `equals` and `hashCode` methods should be defined such that they don't only include the `id` property. Otherwise, one element of the null `id` properties Foos to be removed from the Set.

An advanced example for indexed Lists and Maps

Here is the model bean used within the list. The KeyProperty for this bean is the `id` attribute.

MyBean.javasolid public class MyBean implements Serializable { private Long id; private String name; public Long getId() { return id; } public void setId (Long id) { this.id = id; } public String getName() { return name; } public void setName(String name) { this.name = name; } public String toString() { return "MyBean{" + "id=" + id + ", name='" + name + '\'' + '}'; } }

The Action has a `beanList` attribute initialized with an empty ArrayList.

MyBeanAction.javasolid public class MyBeanAction implements Action { private List beanList = new ArrayList(); private Map beanMap = new HashMap(); public List getBeanList() { return beanList; } public void setBeanList(List beanList) { this.beanList = beanList; } public Map getBeanMap() { return beanMap; } public void setBeanMap(Map beanMap) { this.beanMap = beanMap; } public String execute() throws Exception { return SUCCESS; } }

These `conversion.properties` tell the TypeConverter to use MyBean instances as elements of the List.

MyBeanAction-conversion.propertiessolid KeyProperty_beanList=id Element_beanList=MyBean CreateIfNull_beanList=true

- When submitting this via a form, the `id` value is used as KeyProperty for the MyBean instances in the beanList.
- Notice the () notation! Do not use [] notation, which is for Maps only!
- The value for name will be set to the MyBean instance with this special id.
- The List does not have null values added for unavailable id values. This approach avoids the risk of OutOfMemoryErrors!

MyBeanAction.jspsolid <s:iterator value="beanList" id="bean"> <stextfield name="beanList(%{bean.id}).name" /> </s:iterator>

Type Conversion Error Handling

Type conversion error handling provides a simple way to distinguish between an input *validation* problem and an input *type conversion* problem.

{snippet:id=error-reporting|javadoc=true|url=com.opensymphony.xwork2.conversion.impl.XWorkConverter}

There are two ways the error reporting can occur:

1. Globally, using the Conversion Error Interceptor
2. On a per-field basis, using the conversion validator

By default, the conversion interceptor is included in `struts-default.xml` in the default stack. To keep conversion errors from reporting globally, change the interceptor stack, and add additional validation rules.

Common Problems

Null and Blank Values

Some properties cannot be set to null. Primitives like boolean and int cannot be null. If your action needs to or will accept null or blank values, use the object equivalents Boolean and Integer. Similarly, a blank string "" cannot be set on a primitive. At the time of writing, a blank string also cannot be set on a BigDecimal or BigInteger. Use server-side validation to prevent invalid values from being set on your properties (or handle the conversion errors appropriately).

Interfaces

The framework cannot instantiate an object if it can't determine an appropriate implementation. It recognizes well-known collection interfaces (List, Set, Map, etc) but cannot instantiate MyCustomInterface when all it sees is the interface. In this case, instantiate the target implementation first (eg. in a prepare method) or substitute in an implementation.

Generics and Erasure

The framework will inspect generics to determine the appropriate type for collections and array elements. However, in some cases Erasure can result in base types that cannot be converted (typically Object or Enum).

The following is an example of this problem:

public abstract class Measurement<T extends Enum> public void setUnits(T enumValue) {...} } public class Area extends Measurement<UnitsOfArea> { @Override public void setUnits(UnitsOfArea enumValue){...} }

Although to the developer the area.setUnits(enumValue) method only accepts a UnitsOfArea enumeration, due to erasure the signature of this method is actually setUnits(java.lang.Enum). The framework does not know that the parameter is a UnitsOfArea and when it attempts to instantiate the Enum an exception is thrown (java.lang.IllegalArgumentException: java.lang.Enum is not an enum type).

Next: Interceptors