


Space Details

Key:	XW
Name:	XWork
Description:	
Creator (Creation Date):	plightbo (Apr 18, 2004)
Last Modifier (Mod. Date):	plightbo (Apr 25, 2004)

Available Pages

- XWork 
 - Documentation
 - Basics
 - Components
 - Configuration
 - Logging
 - Core Concepts
 - Dependencies
 - Interceptors
 - DefaultWorkflowInterceptor
 - PrepareInterceptor
 - ValidationInterceptor
 - Introduction
 - Localization
 - Ognl
 - Release Notes - 1.0.1
 - Release Notes - 1.0.2
 - Type Conversion
 - Null Property Access
 - Type Conversion Error Handling
 - Type Conversion In Collections
 - Upgrading from 1.0
 - Upgrading from 1.0.1
 - Validation Framework
 - Building a Validator
 - Generic Object Validation
 - Sample Validation Rules
 - Standard Validators
 - Press Releases
 - 1.0.1 Press Release
 - 1.0.2 Press Release
 - About

Configuration

This page last changed on Jun 17, 2004 by [unkyaku](#).

- [xwork.xml](#)
- [Logging](#)

xwork.xml

XWork is configured through the use of a file named xwork.xml in the root of the classpath which conforms to the DTD. This file defines the action, interceptor, and result configurations and mappings. The following is a sample xwork.xml file; it's a condensed version of the one used in the XWork test cases. It's a long example, but it exhibits all the major configuration possibilities:

```
<!DOCTYPE xwork PUBLIC "-//OpenSymphony Group//XWork
1.0//EN" "http://www.opensymphony.com/xwork/xwork-1.0.dtd"><xwork><package
name="default"><result-types><result-type name="chain"
class="com.opensymphony.xwork.ActionChainResult"/></result-types><interceptors><interceptor
name="timer"
class="com.opensymphony.xwork.interceptor.TimerInterceptor"/><interceptor
name="logger"
class="com.opensymphony.xwork.interceptor.LoggingInterceptor"/><interceptor
name="chain"
class="com.opensymphony.xwork.interceptor.ChainingInterceptor"/><interceptor
name="params"
class="com.opensymphony.xwork.interceptor.ParametersInterceptor"/><interceptor
name="static-params"
class="com.opensymphony.xwork.interceptor.StaticParametersInterceptor"/><interceptor
name="test" class="com.opensymphony.xwork.TestInterceptor"><param
name="foo">expectedFoo</param></interceptor><interceptor-stack
name="defaultStack"><interceptor-ref name="static-params"/><interceptor-ref
name="params"/></interceptor-stack><interceptor-stack
name="debugStack"><interceptor-ref name="timer"/><interceptor-ref
name="logger"/></interceptor-stack></interceptors><global-results><result
name="login" type="chain"><param
name="actionName">login</param></result></global-results><action name="Foo"
class="com.opensymphony.xwork.SimpleAction"><param name="foo">17</param><param
name="bar">23</param><result name="success" type="chain"><param
name="actionName">Bar</param></result><interceptor-ref
name="debugStack"/><interceptor-ref name="defaultStack"/></action><action name="Bar"
class="com.opensymphony.xwork.SimpleAction"><param name="foo">17</param><param
name="bar">23</param></action><action name="TestInterceptorParam"
class="com.opensymphony.xwork.SimpleAction"><interceptor-ref name="test"><param
name="expectedFoo">expectedFoo</param></interceptor-ref></action><action
name="TestInterceptorParamOverride"
class="com.opensymphony.xwork.SimpleAction"><interceptor-ref name="test"><param
name="foo">foo123</param><param
name="expectedFoo">foo123</param></interceptor-ref></action></package><package
name="bar" extends="default" namespace="/foo/bar"><interceptors><interceptor-stack
name="barDefaultStack"><interceptor-ref name="debugStack"/><interceptor-ref
name="defaultStack"/></interceptor-stack></interceptors><action name="Bar"
class="com.opensymphony.xwork.SimpleAction"><interceptor-ref
name="barDefaultStack"/></action><action name="TestInterceptorParamInheritance"
class="com.opensymphony.xwork.SimpleAction"><interceptor-ref name="test"><param
```

```

name="expectedFoo">expectedFoo</param></interceptor-ref></action></package><package
name="abstractPackage" namespace="/abstract" abstract="true"><action name="test"
class="com.opensymphony.xwork.SimpleAction"/></package><package
name="nonAbstractPackage" extends="abstractPackage"
namespace="/nonAbstract"/><package name="multipleInheritance"
extends="default,abstractPackage,bar" namespace="multipleInheritance"><action
name="testMultipleInheritance" class="com.opensymphony.xwork.SimpleAction"><result
name="success" type="chain"><param
name="actionName">foo</param></result><interceptor-ref
name="barDefaultStack"/></action></package><include file="includeTest.xml"/></xwork>

```

xwork.xml elements

Package

The package element has one required attribute, "name", which acts as the key to later reference this package. The "extends" attribute is optional and allows one package to inherit the configuration of one or more previous packages including all interceptor, interceptor-stack, and action configurations. Note that the configuration file is processed sequentially down the document, so the package referenced by an "extends" should be defined above the package which extends it. The "abstract" optional attribute allows you to make a package abstract, which will allow you to extend from it without the action configurations defined in the abstract package actually being available at runtime. ([see above for example](#))

Attribute	Required	Description
name	yes	key to for other packages to reference
extends	no	inherits package behavior of the package it extends
namespace	no	see #Namespace
abstract	no	declares package to be abstract (no action configurations required in package)

Namespace

The optional namespace attribute warrants its own discussion section. The namespace attribute allows you to segregate action configurations into namespaces, so that you may use the same action alias in more than one namespace with different classes, parameters, etc. This is in contrast to Webwork 1.x, where all action names and aliases were global and could not be re-used in an application. The default namespace, which is "" (an empty string) is used as a "catch-all" namespace, so if an action configuration is not found in a specified namespace, the default namespace will also be searched. This allows you to have global action configurations outside of the "extends" hierarchy, as well as to allow the previous Webwork 1.x behavior by not specifying namespaces. It is also intended that the namespace functionality can be used for security, for instance by having the path before the action name be used as the namespace by the Webwork 2.0 ServletDispatcher, thus allowing the use of J2EE declarative security on paths to be easily implemented and maintained. ([see above for example](#))

Result-Type

Result types define classes and map them to names to be referred in the action configuration results. This serves as a shorthand name-value pair for these classes.

Interceptors

Interceptors also serve as a name-value pairing for referring to specific Interceptor classes by a shorthand name where we use interceptor-ref elements, such as in Interceptor stacks and action configurations.

Interceptor-Stack

The interceptor-stack element allows you to assign a name for later referencing via interceptor-ref to an ordered list of interceptors. This allows you to create standard groups of interceptors to be used in a certain order. The interceptor stack name/value pairs are stored in the same map as the interceptor definitions, so anywhere you use an interceptor-ref to refer to an interceptor you may also refer to an interceptor stack.

Interceptor Parameterization

Interceptors are instantiated and stored at the ActionConfig level, i.e. there will be one instance per action configured in xwork.xml. These Interceptor instances may be

parameterized at either the declaration level, where you map an Interceptor class to a name to be referenced later, like so:

```
<interceptor name="test" class="com.opensymphony.xwork.TestInterceptor">
  <param name="foo">expectedFoo</param>
</interceptor>
```

or at the interceptor-ref level, either inside an interceptor-stack or in an action declaration, like so:

```
<interceptor-ref name="test">
  <param name="expectedFoo">expectedFoo</param>
</interceptor-ref>
```

Although it is allowed by the DTD, parameters applied to interceptor-refs which refer to interceptor-stack elements will NOT be applied, and will cause a warning message.

Global-results

The global results allows you to define result mappings which will be used as defaults for all action configurations and will be automatically inherited by all action configurations in this package and all packages which extend this package.

Action

The action element allows the mapping of names to action classes. These names are used, along with the namespace described above, to retrieve the action config from the configuration. The action may also be parameterized by using the param elements, as above, which will set parameters into the Action at execution (with the help of the StaticParametersInterceptor).

The action may also have one or more results mapped to string return codes, such as "success", "error", or "input", which are the default 3 return states for actions, although ANY return string may be used and mapped to a result. The result, which is mapped to a result class by the "type" attribute which refers to the result-type name defined above, may also be parameterized by using the param element.

There is one shortcut when defining results, as a lot of the time a result will have only

one parameter (which is often a location to redirect to (when using XWork in the web sense)).

Here is the long form of a result with a single location parameter:

```
<result name="test">
  <param name="location">foo.jsp</param>
</result>
```

and this is the 'shortcut' form:

```
<result name="test">foo.jsp</result>
```

Note that this shortcut **only** works when there is a single parameter for the result and the result has defined what its default parameter is.

The action may also define one or more interceptor refs to either interceptors or interceptor stacks to be applied before and after the action execution (see the section on Interceptors below). The ordering of these interceptor refs determines the order in which they will be applied.

Includes - using multiple configuration files

The xwork.xml configuration file may be broken up into several files, each with its own set of package declarations, by using the `<include>` element zero or more times at the bottom of your xwork.xml file, like so:

```
<include file="includeTest.xml"/>
```

These files will be processed, in order, in the same manner as the main xwork.xml, thus may have all of the same structures, including the `<include>` element. Although it is, therefore, possible to have packages in configuration files listed later in an include element extend packages defined in an earlier included file, it is not recommended. It should be considered best practice, in the case of using multiple configuration files, to declare default packages meant to be extended in the xwork.xml and to have no dependencies between sub-configuration files.

Configuration Providers

XWork configuration is handled through classes which implement the `ConfigurationProvider` interface. The default implementation is the `XmlConfigurationProvider` class. You can either create a new provider by implementing the `ConfigurationProvider` interface or you can extend the `XmlConfigurationProvider` class. The `XmlConfigurationProvider` class includes a protected method called `getInputStream()` which is called to acquire the configuration `InputStream` which is expected to be an XML data stream. The default implementation looks for a file called `xwork.xml` in the class path but by overriding the `getInputStream()` method you can pull configuration data from any source.

Custom configuration providers must be registered with the `ConfigurationManager` before they will be used to load configuration data. If no custom configuration providers are registered then the default configuration provider is used. If any custom configuration providers are registered then the default configuration provider will no longer be used (although you could add a new instance of it yourself to include it in the list of providers which is searched). To add a configuration provider just call the `ConfigurationManager.addConfigurationProvider()` method with the custom configuration provider as the argument.

- Overview
- [PrepareInterceptor](#)
- [ValidationInterceptor](#)
- [DefaultWorkflowInterceptor](#)

Overview

Interceptors allow you to define code to be executed before and/or after the execution of an action. They are defined outside the action class, yet have access to the action and the action execution environment at runtime, allowing you to encapsulate cross-cutting code and provide separation of concerns.

Interceptors are given the `ActionInvocation` object at runtime, and may do whatever processing needed, then forward processing to the rest of the `ActionInvocation`, which will either call the next `Interceptor` or the `Action`, if there are no more `Interceptors`, and do whatever post-processing needed.

Interceptors may also decide to short-circuit processing and return whatever result string desired WITHOUT forwarding processing, thus keeping the `Action` from executing. This ability should be used with caution, however, as any data loading or processing expected to be done by the `Action` will not happen.

Here is the `invoke()` method from `ActionInvocation`, which calls the `Interceptors` and the `Action`:

```
publicString invoke() throws Exception {
    if (executed) {
        thrownew IllegalStateException("Action has already executed");
    }

    if (interceptors.hasNext()) {
        Interceptor interceptor = (Interceptor) interceptors.next();
        result = interceptor.intercept(this);
    } else {
        result = action.execute();
        executed = true;
    }

    return result;
}
```

It may not be immediately apparent how the rest of the `Interceptors` and the `Action`

come to be called from the code snippet. For this we need to look at the Interceptor implementation in **AroundInterceptor**:

```
publicString intercept(ActionInvocation invocation) throws Exception {  
    before(invocation);  
  
    result = invocation.invoke();  
    after(invocation);  
  
    return result;  
}
```

Here we can see that the Interceptor calls back into the `ActionInvocation.invoke()` to tell the `ActionInvocation` to continue down the chain and eventually executes the Action. It is here that the Interceptor can decide not to forward to the rest of the Interceptors and the Action, and choose instead to return a return code.

It is also important to know what the `AroundInterceptor` is doing when you extend it to implement your own Interceptors.

The `AroundInterceptor` defines a base class for interceptor implementations. It delegates calls to subclasses, which must implement the abstract methods `before()` and `after()`. The `before()` call is first called, then the rest of the `ActionInvocation` is called and the `String` result is saved (and is available to the Interceptor implementation during the `after()` method). Finally, the `after()` method is called and the result is returned.



Note that all Interceptor implementations must be threadsafe.

Utility Interceptors

The `TimerInterceptor` and `LoggingInterceptor` are provided as simple examples and utilities.

- The **LoggingInterceptor** simply logs before and after executing the rest of the `ActionInvocation`.
- The **TimerInterceptor** times the execution of the remainder of the `ActionInvocation`.

The `TimerInterceptor` does not extend `AroundInterceptor` because it needs to keep some state (the start time) from before the rest of the execution. Interceptors must be stateless, so it is impossible to save this in an instance field. It is a good rule of thumb to say that if your interceptor needs to maintain information from the beginning to the end of the interceptor call, it should

implement **Interceptor** directly, not subclass **AroundInterceptor**. Here is the code for **intercept()** in **TimerInterceptor**:

```
public String intercept(ActionInvocation dispatcher) throws Exception {
    long startTime = System.currentTimeMillis();
    String result = dispatcher.invoke();
    long executionTime = System.currentTimeMillis() - startTime;
    log.info("Processed action " + dispatcher.getProxy().getActionName() + " in " + executionTime + "ms.");

    return result;
}
```

It is important to remember to call **invoke()** on the **ActionInvocation** if you directly implement **Interceptor**, otherwise the rest of the **Interceptors** and the **Action** will not be executed.

Parameter Interceptors - populating your Action

The **StaticParametersInterceptor** and **ParametersInterceptor** populate your **Action** fields during the **ActionInvocation** execution.

- The **StaticParametersInterceptor** applies the parameters defined in the **Action** configuration with the `<param>` elements.
- The **ParametersInterceptor** populates the **Action** with the parameters passed in as part of the request.

The **StaticParametersInterceptor** should be applied before the **ParametersInterceptor** so that the static parameters may be set as the defaults and overridden by the request parameters.

ModelDrivenInterceptor - choosing your model

Normally, the **StaticParameterInterceptor** and the **ParametersInterceptor** apply themselves directly to the **Action**. Using the **ModelDrivenInterceptor**, you can specify an alternate object to have the parameters applied to instead.

Consider the following **Action**:

```
public class AddContactAction implements Action {
```

```

privateString name;
privateString addr;
privateString city;

public void setName(String name) { this.name = name ; }
public void setAddr(String addr) { this.addr = addr ; }
public void setCity(String city) { this.city = city ; }

publicString execute() {
    Contact contact = new Contact();
    contact.setName(name);
    contact.setAddr(addr);
    contact.setCity(city);

    // save contact information here
}
}

```

We can see that our action will be populated with name, addr, and city parameters if they are passed in. In the execute we copy these values to a contact object and save the contact.

Here's the ModelDriven interface:

```

publicinterface ModelDriven {
    publicObject getModel();
}

```

Let's apply the ModelDriven interface to Action above:

```

public class AddContactAction implements Action, ModelDriven {
    private Contact contact = new Contact();

    publicObject getModel() { returnthis.contact ; }

    public void execute() {
        // save the contact information
    }
}

```

Now the ParametersInterceptor and the StaticParametersInterceptor will be applied directly to our Contact so when execute gets called, this.contact will already be populated with all the information we need. Neat, huh?

Behavior similar to model driven can be achieved just using the parameter interceptor. For example, rather than implementing ModelDriven, we could have written:

```

public class AddContactAction implements Action {
    private Contact contact = new Contact();
}

```

```
public Contact getContact { returnthis.contact ; }

public void execute() {
    // save the contact information
}
}
```

The difference between this Action and the previous ModelDriven action is twofold:

- Using the ModelDriven action, we can reference our parameters name, addr, and city. Also, the Model (Contact) will be pushed onto the ValueStack so we'll have Contact and AddContactAction on the value stack
- When not using the ModelDriven action, we need to reference our parameters as contact.name, contact.addr, and contact.city.

One potential drawback when using ModelDriven actions is that if you need to access some parameters in order to load the model for the ModelDriven action, you will need to call the ParametersInterceptor and/or the StaticParametersInterceptor twice (before and after the ModelDrivenInterceptor). The first time sets all parameters on the Action, the second time sets all parameters on the model.

ChainingInterceptor

The **ChainingInterceptor** populates the Action it is being applied to with the results of the previous action. When actions are chained together, the action being chained FROM is on the ValueStack when the next action is called. This means that when the next ActionProxy is executed, the action that is being chained TO will be placed onto the valuestack, but the old action will also be there, just down one level. This interceptor works by traversing the ValueStack to find the parameters of any objects there and sets them onto the final action.

XWork Core Concepts

XWork is based on a number of core concepts that helps to explain how the framework works. The core concepts can be broken down into two parts: Architecture Concepts and Terminology.

Architecture Concepts

- Explain Command Driven Architecture (in general)
- Explain the implementation in XWork

Terminology

Actions

Actions are classes that get invoked in response to a request, execute some code and return a Result. Actions implement at a minimum a single method, `execute()`, that defines the entry point called by the framework. This method allows developers to define a unit of work that will be executed each time the Action is called.

ActionContext

The `ActionContext` provides access to the execution environment in the form of named objects during an Action invocation. A new `ActionContext` is created for each invocation allowing developers to access/modify these properties in a thread safe manner. The `ActionContext` makes a number of properties available that are typically set to appropriate values by the framework. In `WebWork 2` for example, the `ActionContext` session map wraps an underlying `HttpSession` object. This allows access to environment specific properties without tying the core framework to a specific execution environment. For more information, see `ActionContext` in [Basics](#).

Interceptors

In XWork, Interceptors are objects that dynamically intercept Action invocations. They provide the developer with the opportunity to define code that can be executed before and/or after the execution of an action. They also have the ability to prevent an action from executing. Interceptors provide developers a way to encapsulate common functionality in a re-usable form that can be applied to one or more Actions. See [Interceptors](#) for further details.

Stacks

To handle the case where developers want to apply more than a single Interceptor to an Action, Stacks have been introduced. Stacks are an ordered list of Interceptors and/or other Stacks that get applied when an Action is invoked. Stacks centralize the declaration of Interceptors and provide a convenient way to configure multiple actions.

Results

Results are string constants that Actions return to indicate the status of an Action execution. A standard set of Results are defined by default: error, input, login, none and success. Developers are, of course, free to create their own Results to indicate more application specific cases.

Result Types

Result Types are classes that determine what happens after an Action executes and a Result is returned. Developers are free to create their own Result Types according to the needs of their application or environment. In WebWork 2 for example, Servlet and Velocity Result Types have been created to handle rendering views in web applications.

Packages

Packages are a way to group Actions, Results, Result Types, Interceptors and Stacks into a logical unit that shares a common configuration. Packages are similar to objects in that they can be extended and have individual parts overridden by "sub" packages.

ValueStack

The ValueStack is a stack implementation built on top of an OGNL core. The OGNL expression language can be used to traverse the stack and retrieve the desired object. The OGNL expression language provides a number of additional features including: automatic type conversion, method invocation and object comparisons. For more information, see the [OGNL Website](#).

Components

XWork provides the ComponentManager interface (and a corresponding implementation in the DefaultComponentManager class) to apply a design pattern known as **Inversion of Control** (or IoC for short). In a nutshell, the IoC pattern allows a parent object (in this case XWork's ComponentManager instance) to control a client object (usually an action, but it could be any object that implements the appropriate *enabler*). See [Components](#) for further details.

Introduction

This page last changed on Jun 03, 2004 by [plightbo](#).

XWork is a generic command pattern framework.

The Purpose:

- To create a generic, reusable, and extensible command pattern framework not tied to any particular usage.

Features:

- Flexible and customizable configuration based on a simple Configuration interface
- Core command pattern framework which can be customized and extended through the use of interceptors to fit any request / response environment
- Built in type conversion and action property validation using Ognl
- Powerful validation framework based on runtime attributes and a validation interceptor

How does XWork relate to Webwork

Webwork 2.0+ is built on top of XWork and provides web-specific features that allow you to quickly build web applications using XWork's command pattern and interceptor framework.

Upgrading from 1.0

This page last changed on Jun 03, 2004 by [plightbo](#).








Upgrading to XWork 1.0.1 from 1.0 involves very little work. All you need to do is copy over the new xwork-1.0.1.jar in replace of xwork-1.0.jar and make sure that the new [Dependencies](#) are all in place.













XWork 1.0.1

Key Changes

- Introduction of an ObjectFactory that provides for easy integration to libraries like Spring and Pico
- Added actionMessages support – just like errorMessages but not counted as an error
- Major performance improvements with Ognl 2.6.5

Changelog

OpenSymphony JIRA (19 issues)		
T	Key	Summary
	XW-188	Correct logging level in DefaultActionInvocation.invokeAction(.
	XW-183	NPE thrown when trying to set a sub-property of a property that doesnt exist
	XW-182	NPE thrown by LocalizedTextUtil.findText
	XW-167	XWorkBasicConverter conversion from Date to String is not localized
	XW-166	[Patch] Improve support for ModelDriven interface
	XW-165	[PATCH] VisitorFieldValidator not setting fieldError correctly
	XW-164	[PATCH] TypeConverter

		should check class hierarchy for conversion properties
	XW-163	[PATCH] Added some javadocs to Interceptors
	XW-162	TypeConverter created by ObjectFactory
	XW-161	Replaceable ObjectFactory for creating framework objects to allow easier integration with IoC containers
	XW-160	Add infinite recursion detection to the ChainingInterceptor
	XW-159	XWork build is broken
	XW-158	Email and URL Validators adding error messages for empty fields
	XW-157	Further Optimize Validator Lookup in ActionValidatorManager
	XW-156	add actionMessages support
	XW-155	NPE thrown when invalid method is looked up
	XW-117	Additional methods to determine if errors using JSTL
	XW-56	Add localization support to XWorkConverter
	XW-23	Investigate using runtime attributes to configure interceptors

XWork

This page last changed on Jun 22, 2004 by [plightbo](#).

XWork is a generic command pattern framework. It was split out of WebWork 1.x and forms the core of [WW:WebWork](#) 2.0. It features:

- Flexible and customizable configuration based on a simple Configuration interface, allowing you to use XML , programmatic, or even product-integrated configuration
- Core command pattern framework which can be customized and extended through the use of interceptors to fit any request / response environment
- Built in type conversion and action property validation using [OGNL](#)
- Powerful validation framework based on runtime attributes and a validation interceptor

Useful links:

- [Documentation](#)
 - [API Javadocs](#)
- [Press Releases](#)
- [Download Binaries](#)
- [CVS](#)

In addition to the documentation, it might be useful to see [Rickard Oberg's thoughts](#) on the future directions of XWork, especially as it relates to Portal development

XWork-Optional

XWork optional is a repository where optional modules for XWork may be created. Just about anyone can get developers access to this repository and is encouraged to use this area as a staging zone for developing projects based upon XWork. Currently a mail dispatcher project is there which uses a mail folder to dispatch actions for processing the emails. For download instructions code go to <https://xwork-optional.dev.java.net/servlets/ProjectSource>

Type Conversion Error Handling

This page last changed on Jun 03, 2004 by [plightbo](#).

Type conversion errors are handled by the `XWorkConverter` whenever any `Exception` is thrown by a converter during converting a value. Type conversion errors are added to a `Map` stored in the `ActionContext` which is available via `ActionContext.getContext().getConversionErrors()`. This map is a map of field name to values which will allow you to access the original value which failed conversion.

There are 2 ways of automatically populating field errors with the type conversion errors into the field errors of the `Action`. The first will populate all of the field errors from the conversion errors and is implemented as an `Interceptor`. There are actually 2 `Interceptors`, one in `XWork` and one in `WebWork` which extends the one in `XWork`. They differ in the implementation of the method

```
protectedboolean shouldAddError(String propertyName, Object value)
```

The `XWork` version always returns `true`, whereas the `WebWork` `Interceptor` returns `false` for values of `null`, `""`, and `{""}`, preventing type conversion exceptions for these common empty values from propagating to the field errors of the `Action`. The `WebWork` version of this `Interceptor` has been added to the `webwork-defaults.xml` and to the `defaultStack` defined therein.

If you choose not to use this `Interceptor`, you can choose to enable this on a per-field basis by using the `Validation` framework with the new field validator added for this, defined in the `validators.xml` file like this:

```
<validator name="conversion"  
class="com.opensymphony.xwork.validator.validators.ConversionErrorFieldValidator"/>
```

This validator will look up the conversion errors and, if it finds a conversion error for the field it is applied to, it will add the appropriate error message to the `Action`.

Both of these methods use the

```
XWorkConverter.getConversionErrorMessage(propertyName, stack)
```

which looks up the type conversion error message for the property name as was done previously, by looking for a message keyed by `"invalid.fieldvalue.propertyName"` and using a default value if it is not found.

Building a Validator

This page last changed on Jun 03, 2004 by [unkyaku](#).

Validators implement the **com.opensymphony.xwork.validator.Validator** interface

```
public interface Validator {
    void setDefaultMessage(String message);

    String getDefaultMessage();

    String getMessage(Object object);

    void setMessageKey(String key);

    String getMessageKey();

    /**
     * This method will be called before validate with a non-null ValidatorContext.
     * @param validatorContext
     */
    void setValidatorContext(ValidatorContext validatorContext);

    ValidatorContext getValidatorContext();

    /**
     * The validation implementation must guarantee that setValidatorContext
     * will be called with a non-null ValidatorContext before validate is called.
     * @param object
     * @throws ValidationException
     */
    void validate(Object object) throws ValidationException;
}
```

FieldValidators implement **com.opensymphony.xwork.validator.FieldValidator**, which extends Validator:

```
public interface FieldValidator extends Validator {

    /**
     * Sets the field name to validate with this FieldValidator
     * @param fieldName
     */
    void setFieldName(String fieldName);

    /**
     * @return the field name to be validated
     */
    String getFieldName();
}
```

If you want to be able to use the "short-circuit" attribute, you should also implement **com.opensymphony.xwork.validator.ShortCircuitableValidator**.

Validators and FieldValidators can extend base classes **com.opensymphony.xwork.validator.validators.ValidatorSupport** and **com.opensymphony.xwork.validator.validators.FieldValidatorSupport** to get the base message and short-circuiting behavior, and will only need to implement `validate(Action action)`.

The Support classes provide the functionality to use the message key and default message to get the localized message body and the parsing of the message body to provide for parameterized messages. Implementations of the Validator Interface which do not extend the Support base classes should provide this functionality as well for consistency.

The **ValidatorContext** set into the Validator is an interface which extends both **ValidationAware** and **LocaleAware** and is used for looking up message texts and setting errors. When validators are called from the ValidationInterceptor, a `DelegatingValidatorContext` is created which delegates these calls to the Action if it implements these interfaces. If the Action does not implement `LocaleAware`, a `LocaleAwareSupport` instance is created which uses the Action's class to look up resource bundle texts, if available. If the action does not implement `ValidationAware`, an implementation which simply logs the validation errors is created and delegated to. When calling the validation framework from outside the ValidationInterceptor, any `ValidatorContext` implementation can be passed in.

Validator classes may define any number of properties using the usual `getX()` `setX()` naming convention and have those properties set using `<param name="x">foo</param>` elements below the `<validator>` element. The values of these properties may then be used in the `validate()` method to parameterize the validation. Validators which extend the Support classes may also use the

```
Object getFieldValue(String name, Action action)
```

method to get the field value of a named property from an Action.

Generic Object Validation

This page last changed on Jun 03, 2004 by [unkyaku](#).

XWork's validation framework is not just limited to Actions. It can be used to validate virtually any object. Once you've set up your validator config file (**validators.xml**) and your validation rules (**ClassName-validation.xml**), all it takes are a couple lines of code:

```
String context = null;
ValidatorContext context = new DelegatingValidatorContext(objectToValidate);
ActionValidatorManager.validate(objectToValidate, null, context);
```

This will cause any errors to be logged (where it gets logged to depends on how commons-logging is configured).

Ideally, you would either implement your own ValidatorContext to handle how error messages are logged and evaluated, or have your objects that are to be evaluated implement ValidationAware, TextProvider and/or LocaleProvider.

Matthew Payne has a JUnit demo of this at <http://www.sutternow.com/index.do?post=51fe0c00fc17aec500fc33f6bb8e006e>.

Actions

Actions are the basic unit of execution...

The Action Interface

The basic interface which all XWork actions must implement. It provides several standard return values like SUCCESS and INPUT, and only contains one method:

```
publicString execute() throws Exception;
```

In general, Actions should simply extend ActionSupport, which provides a default implementation for the most common actions.

ActionContext

The ActionContext provides access to the execution environment in the form of named objects during an Action invocation. A new ActionContext is created for each invocation allowing developers to access/modify these properties in a thread safe manner. The ActionContext makes a number of properties available that are typically set to appropriate values by the framework. In WebWork 2 for example, the ActionContext session map wraps an underlying HttpSession object. This allows access to environment specific properties without tying the core framework to a specific execution environment.

The ActionContext is acquired through the static ActionContext.getContext() method. The ActionContext is a thread local variable and thus the properties of the ActionContext will be relative to the current request thread. The ActionContext has several methods for commonly used properties as well as get() and set() methods which can be used for application specific properties.

Logging

This page last changed on Jun 05, 2004 by [unkyaku](#).

Logging in XWork is handled by [Commons-Logging](#). If Log4J is present in the classpath, logging tasks will be passed through to Log4J.

The logger names are the class names. The pattern used is:

```
Log log = LogFactory.getLog(ThisClass.class);
```

For details on configuring commons-logging, see <http://jakarta.apache.org/commons/logging/guide.html#Configuration>.

Null Property Access

This page last changed on Jun 24, 2004 by [unkyaku](#).

Null property access is a unique feature to XWork that allows object graphs to be created at runtime, saving you the headache of having to pre-initialize them.

Simple object graphs

The feature is quite simple: **only** during the ParametersInterceptor (for WebWork this would be when http parameters are applied to an action), if an expression being set, such as "document.title", results in a NullPointerException, XWork will attempt to create the null object and try again. So in this case, if "document" is null, WebWork will construct a new Document object so that setting the title will succeed.

This is very useful because it reduces the amount of flattening (and unflattening) you are required to do in your action classes when displaying and setting data from web pages. Rather, you can usually represent the complete object graph by just naming your input fields with well thought-out names (like "document.title").

Collections, Lists and Maps

XWork extends this feature even further by offering special support for Collections, Lists and Maps. If you are providing input for one of these interfaces, XWork can be told what type of objects they will hold and automatically populate them accordingly. What this means is that if you refer to "children[0].name", XWork will automatically create a new List to hold the children and also add an empty Child object to that list, so that setting a name on the expression "children[0].name" will work correctly. The same goes for maps.

For this to work you must tell the converters what the objects in the List or Map will be. You do this by specifying "Collection_**property** = com.acme.CollectionItem" in the conversion properties file. So if you have an action that has a "children" property that should be filled with Child objects, YourAction-conversion.properties should contain:

```
Collection_children = come.acme.Child
```

For the purposes of conversion, WebWork considers Collections and Lists to be the same. If the "children" property was declared to be a Collection, a List would be created and used.

Custom Objects

Advanced users who need to instantiate custom objects will want to extend **com.opensymphony.xwork.ObjectFactory** and override the `buildBean(Class)` method. The default implementation is rather trivial:

```
public Object buildBean(Class clazz) throws Exception {  
    return clazz.newInstance();  
}
```

Once you have your own custom `ObjectFactory`, you'll need to let `XWork` know to use it. You do this by calling `ObjectFactory.setObjectFactory(yourObjFactory)`.

Type Conversion

This page last changed on Jul 21, 2004 by [unkyaku](#).

- Overview
- [Null Property Access](#)
- [Type Conversion Error Handling](#)

Type conversion allows you to easily convert objects from one class to another. Whenever XWork attempts to get or set a property on an Action or object and the Class of the value to be get/set is not what is expected, XWork will try to convert it to the correct class.

A common use case is when submitting forms on the web, where we are usually converting to and from String classes. A good example is date conversion. Let's suppose we have the following class:

```
public class Foo implements Action {
    private Date date;
    public void setDate(Date date) { this.date = date ; }
    public String execute() {
        // do work here
    }
}
```

XWork's type converter would allow us to pass in the String "07/08/2003" and have it be automatically converted to a Date object that's been set to July 8, 2003.

XWork already supports basic type conversion, but you can also use your own type converters. To define a custom type converter, you'll need to perform the following steps:

- Create your custom type converter object by implementing `ognl.TypeConverter` or extending `ognl.DefaultTypeConverter`
- Define your conversion rules in a file named **className-conversion.properties** or add the appropriate entry to **xwork-conversion.properties**

Defining Conversion Rules

Conversion rules can be defined at both the class level as well as the application level.

Class-specific conversion rules

To define a class-specific conversion rules, create a file named

className-conversion.properties. The syntax for these files are:

```
property = full.class.name.of.converter  
Collection_property = full.class.name  
Map_property = full.class.name
```

This file defines the type converter to use for properties of an Action or object. If a property is a Collection (List only) or Map, you can also specify the type of elements in the Collection/Map. Please note that there are some issues to consider when working with [type conversion in collections](#).

When looking for a type converter, XWork will search up the class hierarchy and directly implemented for conversion rules. Given the following class hierarchy:

```
interface Animal;  
class AnimalImpl implements Animal;  
interface Quadraped extends Animal;  
class QuadrapedImpl extends AnimalImpl implements Quadraped;
```

XWork will look for a conversion rule in the following files in the following order:

```
QuadrapedImpl-conversion.properties  
Quadraped-conversion.properties  
AnimalImpl-conversion.properties  
Animal-conversion.properties
```

Application-wide conversion rules

To define application-wide conversion rules, create a file named **xwork-conversion.properties** and place it in the root classpath. The syntax for this file is:

```
full.class.name = full.class.name.of.converter
```

Whenever XWork sees any classes listed in this file, it will use the specified converter to convert values to that class.

When looking up a default converter, XWork will search up the class hierarchy to find the appropriate converter. Given the class hierarchy above, XWork will look for a conversion rule for the class in the following order:

```
QuadrapedImpl  
AnimalImpl
```

An Example

To use the Contact example (where Contact is a persistent object), let's say we have the following Action:

```
public class AddContactAction implements Action {
    private Contact contact1;
    private Contact contact2;
    public void setContact1(Contact contact) { this.contact1 = contact; }
    public void setContact2(Contact contact) { this.contact2 = contact; }

    publicString execute() { ... }
}
```

What we're expecting from the UI is for contact to be "1", the primary key of the contact. We want the type converter to convert the string "1" to the Contact with a contactId of 1. Likewise, we'd like the converter to be able to reverse the operation. When displayed, a Contact object should print out its contactId as a String.

The first step is to create our custom TypeConverter:

```
public class ContactConverter extends ognl.DefaultTypeConverter {
    publicObject convertValue(Map ognlContext, Object value, Class toType) {
        if( toType == String.class ) {
            Contact contact = (Contact)value;
            returnnewString(contact.getContactId());
        } elseif( toType == Contact.class ) {
            Integer id = newInteger((String)value);
            Session session = ... // get a Hibernate Session
            Contact contact = (Contact)session.load(Contact.class, id);
            return contact;
        }
        returnnull;
    }
}
```

The next part is to bind our ContactConverter to the previous AddContactAction. I'll bind the ContactConverter to the AddContactAction by creating a file called AddContactAction-conversion.properties that's in the same directory as the AddContactAction class.

I would then populate the properties file as follows:

```
contact1 = com.acme.ContactConverter
contact2 = com.acme.ContactConverter
```

Now, when XWork attempts to populate your object from parameters, you'll be given the actual instances of Contact from your database.

Having said all that, I can't really recommend doing database lookups here as a best practice. In fact, I'd say it's not such a good idea, but it does illustrate type converters well 😊 Any exception thrown here will be handled as described in [Type Conversion Error Handling](#).

Overview

[XWork](#) provides the `ComponentManager` interface (and a corresponding implementation in the `DefaultComponentManager` class) to allow a design pattern known as **Inversion of Control** (or **IoC** for short) to be applied to your actions or other arbitrary objects. In a nutshell, the IoC pattern allows a parent object (in this case XWork's `ComponentManager` instance) to control a client object (usually an action, but it could be any object that implements the appropriate *enabler*).

You may also want to look at [WW:Components](#) to see how WW2 uses XWork's IoC architecture.

Why IoC?

So why is IoC useful? It means that you can develop components (generally services of some sort) in a top-down fashion, without the need to build a registry class that the client must then call to obtain the component instance.

Traditionally when implementing services you are probably used to following steps similar to these:

1. Write the component (eg an `ExchangeRateService`)
2. Write the client class (eg an XWork action)
3. Write a registry class that holds the component object (eg `Registry`)
4. Write code that gives the component object to the registry (eg `Registry.registerService(new MyExchangeRateService())`)
5. Use the registry to obtain the service from your client class (eg `ExchangeRateService ers = Registry.getExchangeRateService()`)
6. Make calls to the component from the client class (eg `String baseCurrencyCode = ers.getBaseCurrency()`)

Using IoC, the process is reduced to the following:

1. Write the component class (eg an `ExchangeRateService`)
2. Register the component class with XWork (eg `componentManager.addEnabler(MyExchangeRateService, ExchangeRateAware)`)
3. Write the client class, making sure it implements the enabling interface (eg an XWork action that implements `ExchangeRateAware`)

4. Access the component instance directly from your client action (eg `String baseCurrencyCode = ers.getBaseCurrency()`)

XWork takes care of passing components through to enabled action classes or other components.

Some other benefits that IoC can provide include:

- A component describes itself. When you instantiate a component, you can easily determine what dependencies it requires without looking at the source or using trial and error.
- Dependencies can be discovered easily using reflection. This has many benefits ranging from diagram generation to runtime optimization (by determining in advance which components will be needed to fulfill a request and preparing them asynchronously, for example).
- Avoids the super-uber-mega-factory pattern where all the components of the app are held together by a single class that is directly tied back to other domain specific classes, making it hard to 'just use that one class'.
- Adheres to Law of Demeter. Some people think this is silly, but in practise I've found it works much better. Each class is coupled to only what it actually uses (and it should never use too much) and no more. This encourages smaller responsibility specific classes which leads to cleaner design.
- Allows context to be isolated and explicitly passed around. `ThreadLocals` may be ok in a web-app, but they aren't well suited for high concurrency async applications (such as message driven applications).

Configuration - xwork.xml

The `ComponentInterceptor` class is used to apply the IoC pattern to XWork actions (ie, to supply components to actions). The `ComponentInterceptor` should be declared in the `<interceptors>` block of `xwork.xml` as follows:

```
<interceptor name="component"
    class="com.opensymphony.xwork.interceptor.component.ComponentInterceptor"/>
```

You should ensure that any actions that are to be supplied with components have this interceptor applied. (See [XW:Interceptors](#) for information on how to apply interceptors to actions.)

If you want to apply IoC to objects other than actions or other components, you will need to use the `ComponentManager` object directly.

Writing Component Classes

The actual component class can be virtually anything you like. The only constraints on it are that it must be a concrete class with a default constructor so that XWork can create instances of it as required. Optionally, a component may implement the Initializable and/or Disposable interfaces so it will receive lifecycle events just after it is created or before it is destroyed. Simply:

```
public class MyComponent implements Initializable, Disposable {
    public void init () {
        //do initialization here
    }

    public void dispose() {
        //do any clean up necessary before garbage collection of this component
    }
}
```

Component Dependencies

One feature that is not immediately obvious is that it is possible for components to depend on other components. For example if the ExchangeRateService described above depended on a Configuration component, XWork will pass the Configuration component through to the ExchangeRateService instance after ExchangeRateService is instantiated. Note that XWork automatically takes care of initializing the components in the correct order, so if A is an action or component that depends on B and C, and B depends on C and if A, B, and C have not been previously instantiated, the ComponentManager will in the following order:

1. Instantiate C and call it's init() method if it implements Initializable.
2. Instantiate B, then using the enabler method, set C to be used by B
3. Call B's init() method, if it implements Initializable.
4. Set B using B's enabler method to be used by A.

And so on and so forth. Of course, if there are instances of B or C that would be reused in this case, those instances would be passed using the enabler method rather than a new instance.

Writing Enablers

An enabler should consist of just a single method that accepts a single parameter. The parameter class should either be the component class that is to be enabled, or one of the component's superclasses. XWork does not care what the name of the enabler's method is.

Here is an example of what the ExchangeRateAware enabler might look like:

```
public interface ExchangeRateAware {  
    public void setExchangeRateService(ExchangeRateService exchangeRateService);  
}
```

Note that typically an enabler would be an interface, however there is nothing to prevent you from using a class instead if you so choose.

Writing "Enabler-aware" Actions

All an action needs to do is implement the relevant enabler interface. XWork will then call the action's enabler method just prior to the action's execution. As a simple example:

```
public class MyAction extends ActionSupport implements ExchangeRateAware {  
    ExchangeRateService ers;  
  
    public void setExchangeRateService(ExchangeRateService exchangeRateService) {  
        ers = exchangeRateService;  
    }  
  
    public String execute() throws Exception {  
        System.out.println("The base currency is " + ers.getBaseCurrency());  
    }  
}
```

If you have an object that is not an action or another component, you must explicitly tell XWork to supply any enabled components to your object by calling `componentManager.initializeObject(enabledObject);`

Using an external reference resolver

You can also use an external reference resolver in XWork, i.e., references that will be resolved not by XWork itself. One such example is using an external resolver to integrate XWork with the [Spring Framework](#)

You just need to write an external reference resolver and then tell XWork to use it in the package declaration:

```
<package
  name="default"
  externalReferenceResolver="com.atlassian.xwork.ext.SpringServletContextReferenceResolver">
```

Now, to use external references you do something like this:

```
<external-ref name="foo">Foo</external-ref>
```

Where the name attribute is the setter method name and Foo is the reference to lookup.

For more details and sample code about this integration, take a look at the javadocs to the `com.opensymphony.xwork.config.ExternalReferenceResolver` class (unfortunately unavailable online) and at [XW-122](#)

-Chris

DefaultWorkflowInterceptor

This page last changed on Jun 26, 2004 by [unkyaku](#).

```
<interceptor name="workflow"
class="com.opensymphony.xwork.interceptor.DefaultWorkflowInterceptor"/>
```

This interceptor implements the workflow that was found in ActionSupport in WebWork 1.x. The following workflow steps are applied before the rest of the Interceptors and the Action are executed, and may short-circuit their execution:

1. If the Action implements **com.opensymphony.xwork.Validateable**, the **validate()** method is called on it, to allow the Action to execute any validation logic coded into it.
2. If the Action implements **com.opensymphony.xwork.ValidationAware** the **hasErrors()** method is called to check if the Action has any registered error messages (either Action-level or field-level). If the ValidationAware Action has any errors, **Action.INPUT** ("input") is returned without executing the rest of the Interceptors or the Action.
3. If the execution did not short-circuit in the step above, the rest of the Interceptors and the Action are executed by calling **invocation.invoke()**

ValidationInterceptor

This page last changed on Jun 26, 2004 by [unkyaku](#).

```
<interceptor name="validation"  
class="com.opensymphony.xwork.validator.ValidationInterceptor"/>
```

This interceptor validates the Action it is applied to. See the documentation on the [Validation Framework](#) for details.

PrepareInterceptor

This page last changed on Jun 27, 2004 by [unkyaku](#).

```
<interceptor name="prepare"  
class="com.opensymphony.xwork.interceptor.PrepareInterceptor" />
```

This interceptor watches for Actions that implement **com.opensymphony.xwork.Preparable** and calls the **prepare()** method on it.

Any action can indicate that it supports localization by implementing `com.opensymphony.xwork.TextProvider`. To access a localized message, simply use one of the various `getText()` method calls.

The default implementation for this is `com.opensymphony.xwork.TextProviderSupport`, which in turn relies on `com.opensymphony.xwork.util.LocalizedTextUtil`. Any Action that extends `com.opensymphony.xwork.ActionSupport` will automatically gain localization support via `TextProviderSupport`.

In this implementation, when you attempt to look up a message, it attempts to do the following:

- Look for the message in the Action's class hierarchy.
 - Look for the message in a resource bundle for the class
 - If not found, look for the message in a resource bundle for any interface implemented by the class
 - If not found, get the super-class and repeat from the first sub-step unless the super-class is `Object`
- If not found and the Action is a ModelDriven Action, then look for the message in the model's class hierarchy (repeat sub-steps listed above).
- If not found, look for the message in a child property. This is determined by evaluating the message key as an OGNL expression. For example, if the key is `user.address.state`, then it will attempt to see if "user" can be resolved into an object. If so, repeat the entire process from the beginning with the object's class and `address.state` as the message key.
- If not found, look for the message in the Action's package hierarchy.
- If still not found, look for the message in the default resource bundles.

Default Resource Bundles.

It is possible to register default resource bundles with XWork via `LocalizedTextUtil.addDefaultResourceBundle()`.

Message lookup in the default resource bundles is done in reverse order of their

registration (i.e. the first resource bundle registered is the last to be searched).

By default, one default resource bundle name is registered with LocalizedTextUtil – "com/opensymphony/xwork/xwork-messages" – which is bundled with the XWork jar file to provide system-level message texts.

Example

Given a ModelDriven Action called BarnAction where getModel() returns a Horse object, and the Horse object has the following class structure:

```
interface acme.test.Animal;  
class acme.test.AnimalImpl implements Animal;  
interface acme.test.Quadrapped extends Animal;  
class acme.test.QuadrappedImpl extends Animal implements Quadrapped;  
class acme.test.Horse extends QuadrappedImpl;
```

Then the localization system will attempt to look up the message in the following resource bundles in this order:

```
acme.test.BarnAction.properties  
acme.test.Horse.properties  
acme.test.QuadrappedImpl.properties  
acme.test.Quadrapped.properties  
acme.test.AnimalImpl.properties  
acme.test.Animal.properties  
acme.test.package.properties  
acme.package.properties
```

Message Key Interpolation

When looking for the message, if the key indexes a collection (e.g. user.phone[0]) and a message for that specific key cannot be found, the general form will also be looked up (i.e. user.phone[*]).

Message Interpolation

If a message is found, it will also be interpolated. Anything within `${...}` will be treated as an OGNL expression and evaluated as such.

Standard Validators

This page last changed on Jul 16, 2004 by [unkyaku](#).

These are the standard validators that come with XWork:

- [#RequiredFieldValidator](#)
- [#RequiredStringValidator](#)
- [#StringLengthFieldValidator](#)
- [#EmailValidator](#)
- [#URLValidator](#)
- [#IntRangeFieldValidator](#)
- [#DateRangeFieldValidator](#)
- [#ConversionErrorFieldValidator](#)
- [#ExpressionValidator](#)
- [#FieldExpressionValidator](#)
- [#VisitorFieldValidator](#)

All the example XML validation rule snippets below have been put together into one big example [here](#).

RequiredFieldValidator

This validator checks that a field is non-null.

Example:

```
<validators><field name="user"><field-validator type="required"><message>You must
enter a value for user.</message></field-validator></field></validators>
```

RequiredStringValidator

This validator checks that a String field is not empty (i.e. non-null with a length > 0).

Parameter	Required	Default	Notes
trim	no	true	Boolean property. Determines whether the String is trimmed before performing the length check.

Example:

```
<validators><field name="userName"><field-validator
type="requiredstring"><message>You must enter an
username.</message></field-validator></field></validators>
```

StringLengthFieldValidator

This validator checks that a String field is of the right length. It assumes that the field is a String.

Parameter	Required	Default	Notes
trim	no	true	Boolean property. Determines whether the String is trimmed before performing the length check.
minLength	no		Integer property. The minimum length the String must be.
maxLength	no		Integer property. The maximum length the String can be.

If neither *minLength* nor *maxLength* is set, nothing will be done.

Example:

```
<validators><field name="userName"><field-validator type="stringlength"><param
name="minLength">3</param><param name="maxLength">10</param><message>Username must
```

```
be between ${minLength} and ${maxLength} characters  
long.</message></field-validator></field></validators>
```

EmailValidator

This validator checks that a field is a valid e-mail address if it contains a non-empty String.

Example:

```
<validators><field name="email"><field-validator type="email"><message>You must  
enter a valid email address.</message></field-validator></field></validators>
```

URLValidator

This validator checks that a field is a valid URL.

Example:

```
<validators><field name="homepage"><field-validator type="url"><message>You must  
enter a valid URL.</message></field-validator></field></validators>
```

IntRangeFieldValidator

This validator checks that a numeric field has a value within a specified range.

Parameter	Required	Default	Notes
min	no		Integer property. The minimum the number must be.
max	no		Integer property. The maximum number can be.

If neither *min* nor *max* is set, nothing will be done.

Example:

```
<validators><field name="age"><field-validator type="int"><param
name="min">0</param><param name="max">100</param><message>Not a valid
age!</message></field-validator></field></validators>
```

DateRangeFieldValidator

This validator checks that a date field has a value within a specified range.

Parameter	Required	Default	Notes
min	no		Date property. The minimum the date must be.
max	no		Date property. The maximum date can be.

If neither *min* nor *max* is set, nothing will be done.

Example:

```
<validators><field name="startDate"><field-validator type="date"><param
name="min">12/22/2002</param><param name="max">12/25/2002</param><message>The date
must be between 12-22-2002 and
12-25-2002.</message></field-validator></field></validators>
```

ConversionErrorFieldValidator

This validator checks if there are any conversion errors for a field and applies them if they exist. See [Type Conversion Error Handling](#) for details.

Example:

```
<validators><field name="startDate"><field-validator
type="conversion"><message>Could not convert input to a valid
date.</message></field-validator></field></validators>
```

ExpressionValidator

This validator uses an [OGNL](#) expression to perform its validation. The error message will be added to the **action** if the expression returns false when it is evaluated against the value stack.

Parameter	Required	Default	Notes
expression	yes		An OGNL expression that returns a boolean value.

Example:

```
<validator type="expression"><param name="expression">foo > bar</param><message default="Foo must be greater than Bar. Foo = ${foo}, Bar = ${bar}."/></validator>
```

FieldExpressionValidator

This validator uses an [OGNL](#) expression to perform its validation. The error message will be added to the **field** if the expression returns false when it is evaluated against the value stack.

Parameter	Required	Default	Notes
expression	yes		An OGNL expression that returns a boolean value.

Example:


```
<validators><field name="homepage"><field-validator type="fieldexpression"><param name="expression">homepage.indexOf('opensymphony.com') == -1</param><message>Please provide an OpenSymphony website</message></field-validator></field></validators>
```

VisitorFieldValidator

The validator allows you to forward validation to object properties of your action using the objects own validation files. This allows you to use the ModelDriven development pattern and manage your validations for your models in one place, where they belong, next to your model classes. The VisitorFieldValidator can handle either simple Object properties, Collections of Objects, or Arrays.

The error message for the VisitorFieldValidator will be appended in front of validation messages added by the validations for the Object message.

Parameter	Required	Default	Notes
context	no	action alias	Determines the context to use for validating the Object property. If not defined, the context of the Action validation is propagated to the Object property validation. In the case of Action validation, this context is the Action alias.
appendPrefix	no	true	Determines whether the field name of this field validator should be prepended to the field name of the visited field to determine the full field name when an error occurs. For example, suppose that the bean being validated has a "name" property. If <i>appendPrefix</i> is true, then the field error will be stored under the field "bean.name". If <i>appendPrefix</i> is false, then the field error will be stored under the field

			<p>"name".</p> <p> If you are using the VisitorFieldValidator to validate the model from a ModelDriven Action, you should set <i>appendPrefix</i> to false unless you are using "model.name" to reference the properties on your model.</p>
--	--	--	--

Example:

```
<validators><field name="user"><field-validator type="visitor"><param
name="context">anotherContext</param><message>user:
</message></field-validator></field></validators>
```

Here we see the *context* being overridden in the validator mapping, so the action alias context will not be propagated.

ModelDriven example:

```
<validators><field name="model"><field-validator type="visitor"><param
name="appendPrefix">false</param><message /></field-validator></field></validators>
```

This will use the model's validation rules and any errors messages will be applied directly (nothing is prefixed because of the empty message).

- Overview
- [Standard Validators](#) - details on validators bundled with XWork
- [Building a Validator](#)
- [Generic Object Validation](#)

Overview

The validation framework in XWork is designed to help you apply simple validation rules to your Actions before they are executed.

At its core, the framework takes just about any object and a String context name for which to validate that object. This allows you to have different validation rules for the same class in different contexts. You can define default validation rules in the class-level validation file (**ClassName-validation.xml**), and then define validation rules which are added on top of these for a specific context (**ClassName-contextName-validation.xml**). The validation rules are applied in the order they are listed in the validation files and error messages are saved into the Object (if it implements ValidationAware).

In the case of Action validation, which is what most XWork users will be doing, the class name is the Action class name and the context is the Action alias.

Registering Validators

Validation rules are handled by validators, which must be registered with the ValidatorFactory. This may either be done programmatically, using the registerValidator(String name, Class clazz) static method of the ValidatorFactory, or by adding a file named **validators.xml** to the root of the classpath that contains this information. The syntax for **validators.xml** is:

```
<validators><validator name="required"
    class="com.opensymphony.xwork.validator.validators.RequiredFieldValidator"/><validator
name="requiredstring"
    class="com.opensymphony.xwork.validator.validators.RequiredStringValidator"/><validator
name="stringlength"
    class="com.opensymphony.xwork.validator.validators.StringLengthFieldValidator"/><validator
name="int"
    class="com.opensymphony.xwork.validator.validators.IntRangeFieldValidator"/><validator
name="date"
    class="com.opensymphony.xwork.validator.validators.DateRangeFieldValidator"/></validators>
```

```

name="expression"
    class="com.opensymphony.xwork.validator.validators.ExpressionValidator"/><validator
name="fieldexpression"
    class="com.opensymphony.xwork.validator.validators.FieldExpressionValidator"/><validator
name="email"
    class="com.opensymphony.xwork.validator.validators.EmailValidator"/><validator
name="url"
    class="com.opensymphony.xwork.validator.validators.URLValidator"/><validator
name="visitor"
    class="com.opensymphony.xwork.validator.validators.VisitorFieldValidator"/><validator
name="conversion"
    class="com.opensymphony.xwork.validator.validators.ConversionErrorFieldValidator"/></val

```

Turning on Validation

The process of applying validation rules to an Action before it is executed is handled by the ValidationInterceptor. As such, all that is required to enable validation for an Action is to declare the ValidationInterceptor in your XWork config file and add an interceptor-ref to it for your Action. See [XW:Configuration](#) for details on how to construct your XWork config file. Here's a simple example:

```

<xwork><package name="example"><interceptors><interceptor name="validator"
    class="com.opensymphony.xwork.validator.ValidationInterceptor"/></interceptors><action
name="Foo" class="com.opensymphony.xwork.SimpleAction"><param
name="foo">17</param><param name="bar">23</param><result name="success"
type="chain"><param name="actionName">Bar</param></result><interceptor-ref
name="validator"/></action></package></xwork>

```

Bear in mind that the ValidationInterceptor only performs validation. The Action will still be executed even if there are any validation errors.

Defining Validation Rules

To define validation rules for an Action, create a file named **ActionName-validation.xml** in the same package as the Action. You may also create alias-specific validation rules which add to the default validation rules defined in **ActionName-validation.xml** by creating another file in the same directory named **ActionName-aliasName-validation.xml**. In both cases, **ActionName** is the name of the Action class, and **aliasName** is the name of the Action alias defined in the **xwork.xml** configuration for the Action.

The framework will also search up the inheritance tree of the Action to find validation rules for directly implemented interfaces and parent classes of the Action. This is particularly powerful when combined with ModelDriven Actions and the

[VisitorFieldValidator](#). Here's an example of how validation rules are discovered. Given the following class structure:

- interface Animal;
- interface Quadraped extends Animal;
- class AnimalImpl implements Animal;
- class QuadrapedImpl extends AnimalImpl implements Quadraped;
- class Dog extends QuadrapedImpl;

The framework method will look for the following config files if Dog is to be validated:

1. Animal
2. Animal-aliasname
3. AnimalImpl
4. AnimalImpl-aliasname
5. Quadraped
6. Quadraped-aliasname
7. QuadrapedImpl
8. QuadrapedImpl-aliasname
9. Dog
10. Dog-aliasname

While this process is similar to what the [XW:Localization](#) framework does when finding messages, there are some subtle differences. The most important difference is that validation rules are discovered from the parent downwards.

Syntax for Validation Rules

Here is a sample config file containing validation rules for SimpleAction from the Xwork test cases:

```
<!DOCTYPE validators PUBLIC "-//OpenSymphony Group//XWork Validator
1.0.2//EN" "http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd"><validators><field
name="bar"><field-validator type="required"><message>You must enter a value for
bar.</message></field-validator><field-validator type="int"><param
name="min">6</param><param name="max">10</param><message>bar must be between ${min}
and ${max}, current value is ${bar}.</message></field-validator></field><field
name="date"><field-validator type="date"><param name="min">12/22/2002</param><param
name="max">12/25/2002</param><message>The date must be between 12-22-2002 and
12-25-2002.</message></field-validator></field><field name="foo"><field-validator
type="int"><param name="min">0</param><param name="max">100</param><message
key="foo.range">Could not find
foo.range!</message></field-validator></field><validator type="expression"><param
name="expression">foo > bar</param><message>Foo must be greater than Bar. Foo =
${foo}, Bar = ${bar}.</message></validator></validators>
```

All `<validator>` (and `<field-validator>`) elements must have a *type* attribute, which refers to a name of a Validator registered with the ValidatorFactory as described above. These elements may also have `<param>` elements with name and value attributes to set arbitrary parameters into the Validator instance.


All `<validator>` (and `<field-validator>`) elements must also define one **message** subelement, which defines the message that should be added to the Action if the validator fails. By default, the message will be that contained in the body of the message tag. The **message** element also has one optional attribute, *key*, which specifies a message key to look up in the Action's ResourceBundles using `getText()` from `LocaleAware` if the Action implements that interface (as `ActionSupport` does). This provides for Localized messages based on the Locale of the user making the request (or whatever Locale you've set into the `LocaleAware` Action). When a *key* is specified, any text contained in the body of the message tag becomes the default message if a message cannot be found for the given key.

If the validator fails, the validator is pushed onto the ValueStack and the message – either the default or the locale-specific one if the *key* attribute is defined (and such a message exists) – is parsed for `${...}` sections which are replaced with the evaluated value of the string between the `${` and `}`. This allows you to parameterize your messages with values from the validator, the Action, or both. Here is an example of a parameterized message:

```
bar must be between ${min} and ${max}, current value is ${bar}.
```

This will pull the min and max parameters from the `IntRangeFieldValidator` and the value of bar from the Action.

A more complete example of the validation rules can be found [here](#).

 Since validation rules are in an XML file, you must make sure you escape special characters. For example, notice that in the expression validator rule above we use ">" instead of ">". Consult a resource on XML for the full list of characters that must be escaped. The most commonly used characters that must be escaped are: & (use `&`), > (use `>`), and < (use `<`).

Validator vs. Field-Validator

The `<field-validator>` elements are basically the same as the `<validator>` elements

except that they inherit the *fieldName* attribute from its enclosing `<field>` element. FieldValidators will have their *fieldName* automatically filled with the value of the parent `<field>` element's *fieldName* attribute. The reason for this structure is to clearly group the validators for a particular field under one element, and because the *fieldName* attribute would otherwise always have to be set for all field validators.

That said, it's perfectly legal to only use validator elements without the field elements and set the *fieldName* attribute for each of them. The following are effectively equal:

```
<field name="bar"><field-validator type="required"><message>You must enter a value
for bar.</message></field-validator></field><validator type="required"><param
name="fieldName">bar</param><message>You must enter a value for
bar.</message></validator>
```

Short-circuiting Validators

Beginning with XWork 1.0.1 (bundled with WebWork 2.1), it is possible to short-circuit a stack of validators. Here is another sample config file containing validation rules from the Xwork test cases:

```
<!DOCTYPE validators PUBLIC "-//OpenSymphony Group//XWork Validator
1.0.2//EN" "http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd"><validators><field
name="email"><field-validator type="required" short-circuit="true"><message>You must
enter a value for email.</message></field-validator><field-validator type="email"
short-circuit="true"><message>Not a valid
e-mail.</message></field-validator></field><field name="email2"><field-validator
type="required"><message>You must enter a value for
email2.</message></field-validator><field-validator type="email"><message>Not a
valid e-mail2.</message></field-validator></field><validator
type="expression"><param
name="expression">email.equals(email2)</param><message>Email not the same as
email2</message></validator><validator type="expression" short-circuit="true"><param
name="expression">email.startsWith('mark')</param><message>Email does not start with
mark</message></validator></validators>
```

Notice that some of the `<field-validator>` and `<validator>` elements have the *short-circuit* attribute set to true. Since validators are evaluated in the order they are declared, a validator with this attribute set to true will prevent the evaluation of subsequent validators if the validator fails, where failure is determined by the addition of an error (action or field error depending on the type of validator) to the ValidationContext of the object being validated.

A field-validator that gets short-circuited will only prevent other field-validators for the same field from being evaluated. A non field-validator that gets short-circuited will completely break out of the validation stack – no other validators will be evaluated.

As mentioned above, the framework will also search up the inheritance tree of the

action to find default validations for interfaces and parent classes of the Action. If you are using the *short-circuit* attribute and relying on default validators higher up in the inheritance tree, make sure you don't accidentally short-circuit things higher in the tree that you really want!



Upgrade Alert: The *short-circuit* attribute was added to the DTD in version 1.0.2. Your validation files must use version 1.0.2 of the `xwork-validation.dtd` to validate properly.

Sample Validation Rules

This page last changed on Jul 16, 2004 by [unkyaku](#).

In this example, we have an Action that attempts to fill the values of a User object. The first validation file would be for a simple Action that had a "user" property. The second validation file is for the exact same Action if it had implemented the ModelDriven interface and exposed the User object via **getModel()** instead of **getUser()**. Both of them uses the VisitorFieldValidator to use the User object's own validation rules to validate the User object, which can be found in the third validation file.

Notice that the *short-circuit* attribute of the **field-validator** element is used in several places to prevent subsequent validators from running. This is useful in preventing multiple error messages when one will do. For example, if the value provided for the User's startDate cannot be parsed as a date, display the appropriate message and don't even bother checking that it is in the correct range.

SampleAction-validation.xml:

```
<validators><field name="user"><field-validator type="required"
short-circuit="true"><message>You must provide a
user.</message></field-validator><field-validator type="visitor"><param
name="context">anotherContext</param><message>user:
</message></field-validator></field></validators>
```

SampleModelDrivenAction-validation.xml:

```
<validators><field name="model"><field-validator type="visitor"><param
name="appendPrefix">false</param><message /></field-validator></field></validators>
```

User-validation.xml:

```
<validators><field name="userName"><field-validator type="requiredstring"
short-circuit="true"><message>You must enter an
username.</message></field-validator><field-validator type="stringlength"><param
name="minLength">3</param><param name="maxLength">10</param><message>Username must
be between ${minLength} and ${maxLength} characters
long.</message></field-validator></field><field name="email"><field-validator
type="email"><message>You must enter a valid email
address.</message></field-validator></field><field name="homepage"><field-validator
type="url" short-circuit="true"><message>You must enter a valid
URL.</message></field-validator><field-validator type="fieldexpression"><param
name="expression">homepage.indexOf('opensymphony.com') == -1</param><message>Please
provide an OpenSymphony website</message></field-validator></field><field
name="age"><field-validator type="int"><param name="min">0</param><param
name="max">100</param><message>Not a valid
age!</message></field-validator></field><field name="startDate"><field-validator
type="conversion" short-circuit="true"><validator type="startDate"><message>Could
```



```
not convert input to a valid date.</message></field-validator><field-validator  
type="date"><param name="min">12/22/2002</param><param  
name="max">12/25/2002</param><message>The date must be between 12-22-2002 and  
12-25-2002.</message></field-validator></field></validators>
```

OGNL is the Object Graph Navigation Language - see <http://www ognl.org> for the full documentation of OGNL. In this document we will only show the additional language features that are provided on top of the base OGNL EL.

XWork-specific Language Features

The ValueStack

The biggest addition that XWork provides on top of OGNL is the support for the ValueStack. While OGNL operates under the assumption there is only one "root", XWork's ValueStack concept requires there be many "roots".

For example, suppose we are using standard OGNL (not using XWork) and there are two objects in the OgnlContext map: "foo" -> foo and "bar" -> bar and that the foo object is also configured to be the single **root** object. The following code illustrates how OGNL deals with these three situations:

```
#foo.blah // returns foo.getBlah()
#bar.blah // returns bar.getBlah()
blah      // returns foo.getBlah() because foo is the root
```

What this means is that OGNL allows many objects in the context, but unless the object you are trying to access is the root, it must be prepended with a namespaces such as @bar. XWork, however, is a little different...

In XWork, the entire ValueStack is the root object in the context. But rather than having your expressions get the object you want from the stack and then get properties from that (ie: peek().blah), XWork has a special OGNL PropertyAccessor that will automatically look at the all entries in the stack (from the top down) until it finds an object with the property you are looking for.

For example, suppose the stack contains two objects: Animal and Person. Both objects have a "name" property, Animal has a "species" property, and Person has a "salary" property. Animal is on the top of the stack, and Person is below it. The follow code fragments help you get an idea of what is going on here:

```
species    // call to animal.getSpecies()
salary     // call to person.getSalary()
name       // call to animal.getName() because animal is on the top
```

In the last example, there was a tie and so the animal's name was returned. Usually this is the desired effect, but sometimes you want the property of a lower-level object. To do this, XWork has added support for indexes on the ValueStack. All you have to do is:

```
[0].name    // call to animal.getName()
[1].name    // call to person.getName()
```

Note that the ValueStack is essentially a List. Calling [1] on the stack returns a sub-stack beginning with the element at index 1. It's only when you call methods on the stack that your actual objects will be called. Said another way, let's say I have a ValueStack that consists of a model and an action ([model, action]). Here's how the following OGNL expressions would resolve:

```
[0]          // a CompoundRoot object that contains our stack, [model, action]
[1]          // another CompoundRoot that contains only [action]
[0].toString() // calls toString() on the first object in the ValueStack
               // (excluding the CompoundRoot) that supports the toString() method
[1].foo       // call getFoo() on the first object in the ValueStack starting from action
               // (excluding the CompoundRoot) that supports a getFoo() method
```

Accessing static properties

OGNL supports accessing static properties as well as static methods. As the OGNL docs point out, you can explicitly call statics by doing the following:

```
@some.package.ClassName@FOO_PROPERTY
@some.package.ClassName@someMethod()
```

However, XWork allows you to avoid having to specify the full package name and call static properties and methods of your action classes using the "vs" (short for ValueStack) prefix:

```
@vs@FOO_PROPERTY
@vs@someMethod()

@vs1@FOO_PROPERTY
@vs1@someMethod()
```

```
@vs2@BAR_PROPERTY  
@vs2@someOtherMethod()
```

The important thing to note here is that if the class name you specify is just "vs", the class for the object on the top of the stack is used. If you specify a number after the "vs" string, an object's class deeper in the stack is used instead.

The *top* keyword

XWork also adds a new keyword – **top** – that can be used to access to first object in the ValueStack.

Type Conversion In Collections

This page last changed on Jul 21, 2004 by [unkyaku](#).

There is a limit to how much type conversion XWork can do automatically when it works with collections.

Internally, XWork uses XWorkList and XWorkMap. They basically extend ArrayList and HashMap respectively to support type conversion. Anything that gets inserted into them are automatically converted if they're not already the required type.

In situations where the collection is null, XWork will create either an XWorkList or an XWorkMap and set it back on to the target (see [Null Property Access](#) for details), and everything works beautifully. However, if the collection is not null, and the collections returned are neither an XWorkList nor an XWorkMap, there's no way type conversion can be performed. In this case, the you want type conversion to work, you will have to seriously consider using XWorkList or XWorkMap instead of ArrayList or HashMap.

Example

Imagine a User object that has a Collection of e-mail addresses. Now, suppose the input is a String array ["foo@bar.com"], and the target is "emails[1]". What we'd really like XWork to do is convert that String array into a String before inserting it into the collection. The first thing we'd do is create a User-conversion.properties with the line:

```
Collection_emails = java.lang.String
```

If User.getEmails() returns null, then XWork will create a new XWorkList and call User.setEmails() with it. It will then add an empty string at position [0] and insert "foo@bar.com" at position [1] (the array is automatically converted into a String).

If User.getEmails() returns something other than XWorkList, when XWork tries to set emails[1] to ["foo@bar.com"], no type conversion will occur. In addition, if the array is empty, ArrayList will throw an IndexOutOfBoundsException.

Overview

- [Introduction](#)
 - What is Xwork?
 - How does Webwork 2.0 relate to Xwork?

XWork Versions

- Current Release
 - Release Notes
 - [Release Notes - 1.0.1](#)
 - [Release Notes - 1.0.2](#)
 - [Dependencies](#)
- Upgrading from previous versions
 - [Upgrading from 1.0](#)
 - [Upgrading from 1.0.1](#)

Reference Guide

- [Core Concepts](#): Terminology and an introduction to XWork
- [Basics](#): Actions, ActionSupport, ActionContext, and action lifecycles
- [Configuration](#): xwork.xml
- [Ognl](#)
- [Localization](#)
- [Type Conversion](#)
- [Interceptors](#)
- [Validation Framework](#)
- [Components](#): Inversion of Control

Documentation Tasks Remaining

- Merge XWork specific docs from WebWork. In particular:
 - config docs
- Description for all interceptors

- Beef up [Basics](#)
- Make sure we document ActionChaining somewhere

Dependencies

This page last changed on Aug 04, 2004 by [plightbo](#).

Dependencies are always an important issue for me. I like to know *exactly* what dependencies any module has.

Here are the dependencies for [XWork](#), split into runtime and compile time dependencies.

- Runtime dependencies are stored in CVS in the **lib/core** directory.
- Compile time dependencies are stored in CVS in the **lib/build** directory.

You can see all the dependencies as well as their version numbers by looking at the **libraries.txt** file located in each directory.

Upgrading from 1.0.1

This page last changed on Aug 04, 2004 by [plightbo](#).





Upgrading to XWork 1.0.2 from 1.0.1 involves very little work. All you need to do is copy over the new xwork-1.0.2.jar in replace of xwork-1.0.1.jar and make sure that the new [Dependencies](#) are all in place.

XWork 1.0.2

Key Changes

- Added an **xwork-default.xml** configuration file that can be used as a standard starting place for non-web related XWork deploys (such as in Quartz)
- Localized text for collections can now be referenced in the form **someArray[*].someField**, where * provides a "catch-all" for any index element
- Text retrieved using the default TextProvider now looks attempts to look for messages in the property files of child objects (if applicable) if none can be found.
- **Field-level validation can be short-circuited**, meaning that you can stop the validation chain right away if you wish. See [Validation Framework](#) for details.
- Simple **type conversion, such as int and float, now use the Locale**. This will cause the correct use of "," and "." depending on the locale.
- Much improved handling for type conversion of elements in collections.

Changelog

OpenSymphony JIRA (15 issues)		
T	Key	Summary
	XW-210	Make default type conversion message a localized text that can be overridden
	XW-205	missing xwork 1.0.2 dtd in jar and website and typo in ValidationInterceptor
	XW-204	TextProvider.getText() should look in child property files
	XW-203	Add "trim" parameter to string validators

	XW-202	Integer and Float conversion dont work in CVS HEAD
	XW-200	i18n broken when the name of the text to find starts with a property exposed by the action
	XW-195	Add interface XWorkStatics which contains XWork-related constants from WebWorkStatics
	XW-194	Patch to help LocalizedTextUtil deal with messages for indexed fields (collections)
	XW-193	InstantiatingNullHandler and Typeconversion fails
	XW-192	Create a version 1.0.2 of the XWork validation DTD with short circuit
	XW-191	Type conversion improvement.
	XW-190	Provide a xwork-default.xml.
	XW-189	Improve ActionValidationManager's short circuit behaviour
	XW-179	Optimise OgnlUtil.copy method
	XW-172	XWorkBasicConverter doesn't care about the current locale