CRUD Demo I

Struts 2 CRUD Example

Due to the export of the wiki to static pages, attachements are not visible. You can find the .war with the source here

Introduction

Welcome to the Struts 2 (S2) CRUD Example. This example was created to be as simple as possible and as such, it does not use all of the advanced (integration) features such as Spring IoC, Hibernate Open-session-in-view, OS Sitemesh, annotations, etc .. For these and other examples, please refer to the Struts 2 Guides.

Struts 2

Struts 2 is a traditional MVC2 action-based framework (such as Struts 1, Stripes, Simple, ..) as opposed to the newer event-based frameworks (such as JSF, Wicket, Rife, ..). Struts 2 uses XWork under the hood, a command-pattern based framework that handles conversion, validation, interception, and a lot more. Struts 2 is based on WebWork, which was originally started as an effort to overcome some problems with Apache Struts 1.

.war layout

The .zip file you can download on this site can be dropped in your servlet container (rename it to .war then) and contains the source code under the WEB-INF directory. The layout is also kept as simple as possible:

- struts-crud -- [css] (contains the stylesheets) -- [WEB-INF] ---- [classes] (contains the compiled src files) ---- [lib] (contains the dependencies) ---- [jsp] (contains the view pages) ---- [src] (contains the source files) ---- web.xml (our webapplication descriptor) -- index.html (simple redirect page)

Configuration files

WEB-INF/web.xml

The webapplication's descriptor file contains one filter and its mapping. By default, the filter is mapped to /*, meaning all requests will be intercepted, but only those ending with a specific suffix (.action, by default) and certain special paths (for static files) will be processed and handled by Struts 2.

xml <web-app> <display-name>Struts 2 CRUD Demo</display-name> <filter> <filter-name>struts</filter-name> <filter-class>org.apache.struts2.dispatcher.FilterDispatcher</filter-class> </filter> <filter-mapping> <filter-name>struts</filter-name> <url-pattern>/*</url-pattern> </filter-mapping> </web-app>

See web.xml for further details.

WEB-INF/classes/struts.xml

struts.xml contains the configuration for XWork: actions, validation, interceptors and results are defined in there.
To understand these terms, we'll need to take a look at Struts 2's (and XWork's) architecture. A basic request goes a bit like this:
A request is submitted, and our ActionMapper will try to determine the correct Action to execute. To find it, it will look up the registered names in your struts.xml. If one is found, before executing the Action, it will loop through a defined stack of interceptors.

{float:right|width=200px} {tip:title=Developer Info} [Learn More About Interceptors|Interceptors] [Learn About the Parameters Interceptor|Parameters Interceptor] {tip} {float}

Interceptors are a very important part of S2 - they will be invoked before and after your action is executed and the view is rendered, and as such, they are perfect for validation, authentication, open-session-in-view patterns, catching exceptions, setting or altering parameters, hiding complex operations, and more. S2 provides a number of prebuilt stacks with a ranging number of features, but nothing keeps you from defining your own interceptor stack with custom interceptors.

One of the most practical interceptors is the 'params' interceptor. It will translate your request parameters to set them on your action. Thus, if your action had a setName(String), and one of your request parameters is called 'name', then WW will set the name for you. Not so special, you say ? Ok, how about setId(Long id) ? This will work just fine as long as your id parameter value can be converted to a Long. Still not special enough ? How about submitting a parameter named empolyee.id ? As you might have guessed, this will invoke a getEmployee().setId(Long id). S2 handles all common objects (Integer, String, Date, .. and Arrays, Lists, Maps, ..) And for those who just can't get enough, you can always add your own convertors for those (or your own) very complex objects (think social security number).

An important aspect of a framework is validation. Adding validation can be a slow and complex process - not to mention the user feedback when something goes wrong. As we think you shouldn't reinvent the wheel, so S2 has a loosely coupled validation framework which you'll see this in action when you try to insert or update an Employee.

Now, when an Action is executed, the result will be used to control the flow - these simple Strings ("success", "error", "input", ..) will be used to invoke a certain Result - this Result can be a dispatcher to a JSP file, render a Freemarker template, generate a chart, output xml, you name it. And, it's totally independend from your Action. Note when validation fails, the result will be by default "input".

Now, as soon as the result is rendered/dispatched/executed/.. WW will loop through the interceptors again in reverse order - which is perfect for cleaning up resources, logging, timing, .. etc.

Let us take a more detailed look at our struts.xml:

xml <struts> <!-- Configuration for the default package. --> <package name="default" extends="struts-default"> <!-- Default interceptor stack. --> <default-interceptor-ref name="paramsPrepareParamsStack"/> <action name="index" class="com.aurifa.struts2.tutorial.action.EmployeeAction" method="list"> <result name="success">/WEB-INF/jsp/employees.jsp</result> <!-- we don't need the full stack here --> <interceptor-ref name="basicStack"/> </action> <action name="crud" class="com.aurifa.struts2.tutorial.action.EmployeeAction" method="input"> <result name="success" type="redirect-action">index< /result> <result name="input">/WEB-INF/jsp/employeeForm.jsp</result> <result name="error">/WEB-INF/jsp/error.jsp</result> </action> </package> < /struts>

There are four major elements covered in this example: package, interceptor, action, and result.

Packages are a way to group actions, results, result types, interceptors, and interceptor-stacks into a logical configuration unit. Conceptually, packages are similiar to objects in that they can be extended and have individual parts that can be overridden by "sub" packages. See Package Configuration for further details.

Interceptors allow you to define code to be executed before and/or after the execution of an Action method. For now, we will configure our example to use the {{paramsPrepareParamsStack }} interceptor stack; you can ignore this for now. See Interceptor Configuration for further details.

The action mappings are the basic "unit-of-work" in the framework. Essentially, the action maps an identifier to a handler class. When a request matches the action's name, the framework uses the mapping to determine how to process the request.

We define two actions in our example using the same Action class, EmployeeAction. One is registered with the name 'index', and will be used for the index page, while the other one, 'crud' will be used to execute the various create/read/update/delete actions. You also note they list different method attributes: one will execute, by default, the list method while our crud action goes with the input method. See Action Configuration for further details.

Results define what happens after an action executes. Each Action execution returns a String result which is used to select from the list of configured result elements.

For our index action, we don't require any input, we will assume nothing goes wrong, so we only list the success result (Note: you can register global results as well). This result uses, under the hood, the default DispatcherResult result-type to dispatch the request to an employees.jsp file. Also note the fact that since we didn't need the full stack (no validation, fileupload, preparing, or other funky things), we prefer to use the basic interceptor stack for this request.

It gets more interesting for our crud Action: besides the success result, we also specify the input result (which dispatches to our input form) and the error result (which is returned when an exception is thrown during the Action execution - for example a database exception). You can also see we specified a different result-type for our success result, in case the 'redirect-action'. This is nothing more than a fancy redirect which will append the chosen S2 suffix, so we could have also used the redirect result-type, with index.action as its text. But this approach is slightly better since it is suffix agnostic (you can switch suffixes very easily in S2, so that is why we always advise to program suffix agnostic and let the framework handle it for you).

See Result Configuration for further details.

WEB-INF/classes/struts.properties

This file only contains two lines, and is used to set Struts 2 specific settings (such as which IoC container to use, which fileuploader, what templates, etc ..). You don't really need it, but for i18n reasons, we use it to register our resource bundle 'guest.properties'. Note that you can register properties file on different levels in Struts 2.

struts.custom.i18n.resources=guest

More information on struts.properties

It is recommended that configuration be handled with your struts.xml file: see Constant Configuration for details.

WEB-INF/classes/guest.properties

This guest.properties file will contain the keys and values to internationalize your webapplication in a straightforward way. Rather than hardcoding Welcome ! in your page, you should use specify a key (eg. 'welcome_msg') with a certain value ('Welcome !'). By adding new resource bundles, you can override the key and specify a different value (eg. 'Bienvenue !' for a french Locale).

The code

Since 90% of the code is identical to the other CRUD examples displayed on this site, we'll only analyze the Action class, EmployeeAction.
As always, first things first - the class definition:

... import com.opensymphony.xwork2.ActionSupport; import com.opensymphony.xwork2.Preparable; ... public class EmployeeAction extends ActionSupport implements Preparable { ...

Now, first there is the extending of the ActionSupport class - although you don't have to extend it, it provides a lot of useful extras, so you are encouraged to extend and override parts of it, but you don't have to. The interface we are implementing is a bit more interesting. The Preparable interface only defines one method, public void prepare(). By implementing this interface in your Action, you tell the prepare interceptor to call this method on your Action - so that makes it perfect to, for example, retrieve objects from a database, which is what we're 'faking' here by requesting an Employee object from the EmployeeService.

There are quite some different interfaces that you can implement that will be used by interceptors (SessionAware, ServletRequestAware, ..), but you need to be sure the interceptors are listed in your interceptor stack, or the interceptor won't be executed on your action.

Besides the prepare() method we just explained, our Action also contains the doInput(), doSave(), doList() and doDelete() methods. Remember how we specified the method attribute in the struts.xml file ? Well, these are the methods that are going to be executed. Technically, you could have defined input(), save(), list() and delete() as well, but S2 will automagically find the correct method. Originally, this was because often a default() method is declared - but since default is a reserved keyword, you have to make that doDefault(). You don't have to care about that, but just so you know.

Let's explore the doList method, which is the default method we specified in our struts.xml for our index action.

public String doList() { employees = empService.getAllEmployees(); return SUCCESS; }

Surprisingly simple, no? Simply fill in the employees object, and return "success" (defined as a final static variable SUCCESS). That's it? That's it. The only thing you need now to show the list in your view layer is a simple getter for employees in your Action. The reason for this is another Struts 2 feature: the ValueStack.

The ValueStack: it's magic, baby!

The ValueStack is like a normal stack. You can put objects on it, remove them from it, and query it. And even more, you can query it with expressions! It's the heart of Struts 2, and allows easy access to a wide range of objects from nearly any place in the framework: interceptors, results, tags, ... you name it.

Your action is placed on top of the stack when it is executed.

Once it's on the valueStack, you could query for it 'employees' - a special glue language called OGNL will then transform that request to 'top. getEmployees()' - and since top is the top of your stack, and your Action is located just there, it will invoke the getEmployees() method on your Action, and return the result.

More information on the Struts 2 architecture, Interceptors, OGNL

First view: the employee listing

S2 defines a lot of tags that will dramatically speed up your development. They can alter or query the value stack, render input forms, javascript widgets, iterate over collection, and so on. On top of that, they have different themes, which add even more functionality/layout by the switch of a parameter. Themes are out of the scope of this example, but you should definitely check them out; see Themes and Templates.

xml <%@taglib prefix="s" uri="/struts-tags" %> <head> <link href="<s:url value='/css/main.css'/>" rel="stylesheet" type="text/css"/> <title><s:text name=" label.employees"/></title> </head>

The `s:url` tag allows you to build urls for actions, possibly with parameters. This saves you from having to type them out manually, remember (or change) what action suffix you're using, or include the web-app context. See Url for further details.

The `s:text` tag will look up keys in resource bundles in the valueStack (depending on the locale). So adding a new language would be a breeze. In these cases we build an url '/css/main' and we display the value of a key 'label.employees' from our guest.properties resource bundle. See Text for further details.

xml <s:url id="url" action="crud!input" /> <a href="<s:property value="#url"/>">Add New Employee

Here we use `s:url` to generate a URL and assign it to an ID so we can refer to it later.

The `s:property` tag provides access to objects on the OGNL value stack. Our object, `url`, may be accessed using the # character: non-root objects on the value stack need the # character. See OGNL for further details.

The ActionMapper

The S2 ActionMapper doesn't just map names to Actions - it can also change the method, or even action, by using special requests. The weird value we encountered above, crud!input, tells the ActionMapper to invoke the input()/doInput() method of the Action known as crud. So for example, you could have several slighly different methods, and rather than having to register each of them in the struts.xml file, you can use the ! notation to specify which method to execute.

You may also use Wildcard Mappings to run specific methods rather than use the ! character.

Another thing is the fact that you can override which action/method to invoke based on a special name:action parameter, which we'll use later on in our employeeForm.jsp to make a nice cancel button.

More information on the ActionMapper

More tags: s:iterator, s:if/else, and s:property

Take a look at the following lines from our employees.jsp:

xml <s:iterator value="employees" status="status"> <tr class="<s:if test="#status.even">even</s:if><s:else>odd</s:else>">

The `s:iterator` tag does pretty much what you expect from it: it loops through an array/list/collection, and pushes each object it encounters on the stack on each iteration. It also has a helper attribute called `status`, which generates a status object that keep track of the index, tells you if you're in an even or odd row, and so on, while you're inside the `s:iterator` tag. See iterator for further details.

As you can see, we use that same status object in our next tag, the `s:if` and its companion, the `s:else` tag. The test attribute of `s:if` allows you to query the valueStack with an OGNL expression, which is exactly what we do here: see if the status object from our `s:iterator` tag we put on the valueStack, returns true when the method isEven() is invoked. The corresponding `s:else` method is then executed if the test method return false.

Unlike JSTL's `c:choose`/`c:when`/`c:otherwise` tags you do not nest `s:if`/`s:else` inside a "parent" tag.

Finally, there's the `s:property` tag, which is used for displaying objects/expressions from the ValueStack. Examine this:

xml <s:property value="age" />

This might be a little confusing at first, but it's very simple. This actually translates to top.getAge(), meaning we'll execute a getAge() on the top object on our ValueStack. Which happens to be .. the Action? Nope. The employees List? Closer. The current Employee object in the employee list? Bingo.

Why? Because we told you what the `s:iterator` tag did: it places each object on the top of the stack. Since the employees List contains Employee objects, an Employee object gets 'on top' of the stack. That's why you can simply use "age" for `s:property`'s value attribute and have the employee's age printed out.

Adding & Editing an employee

Easy enough: return "input" from your Action, and register the result in your struts.xml with a dispatcher to employeeForm.jsp.

public String doInput() { return INPUT; }

Nothing to it. So let's take a look at how we are going to edit an Employee.

Editting, or better, preparing for editing, comes in many flavours. Fact is, you often need 'extra' objects, like a list of possible departments in our case, to be set up before rendering the insert or edit page. In S2, there are quite some ways to accomplish this, but two approaches are recommended:

- Using a `prepare()` method and interceptor to setup any additional objects you need
- Using an `s:action` tag to use another action to create the objects for you - for example, a DepartmentAction could return a list of Department objects.
Here we'll show you both, but in the example application you'll only find the first one (might change, that's why we're adding it here).

The prepare approach

A little rehearsal: the prepare interceptor, when listed in your interceptor stack, will call the prepare method on our Action. We use this prepare method to set up our deparments, so the list will be available whenever our crud action is called.

We also use this to retrieve an employee bean whenever an id is set - which is precisely how we make the difference between inserting and editing (and we use a similar method in the doSave() method in our EmployeeAction). So, visiting the crud!input action without an employee.id parameter to retrieve the employee, will result in an empty form, while passing the parameter will result in an Employee object to be retrieved for editing.

See Prepare interceptor for further details.

The s:action approach

The other commonly used approach is to use the s:action tag. The s:action tag allows you to execute (additional) S2 actions, which makes them perfect to generate objects for input forms, such as select boxes. You could create for example, a Department action, which does nothing more than listing an doList() method to set up a list of departments, and getter for it, called getDepartments().

Register the DepartmentAction in your struts.xml as 'department', and you can simply call it in your page, and use the id approach we used in the s:url tag to store the result on the stack under a custom name called allDepartments:

xml <s:action name="department!list" id="allDepartments"/>

Now, this, by itself, does not do much. We'll show you later when we talk about the employeeForm page how to use it as an alternative approach.

More information on the s:action tag

Forms made easy

The employeeForm.jsp page is really concise. We told you about the themes that S2 uses under the cover, right? Well, those are going to be responsable for rendering our form, complete with labels for names and errors, input fields, and so on.

But first, we find another useful tag: `s:set`.

xml <s:if test="employee==null || employee.employeeId == null"> <s:set name="title" value="%{'Add new employee'}"/> </s:if> <s:else> <s:set name="title" value="%{'Update employee'}"/> </s:else>

The set tag allows you to store certain objects on the stack (as well as their scope - request/session/page/...), which is what we're going to do here because out of sheer laziness (and performance reasons) we refuse to do the same if/else more than once. As you can guess, it relies on the same principle as the id attribute of the `s:url` tag we saw earlier, meaning I can access it with `#title` on the ValueStack.

More information on the s:if tag, s:else tag

The actual form

xml <s:form action="crud!save" method="post"> <s:textfield name="employee.firstName" value="%{employee.firstName}" label="%{getText('label.firstName')}" size="40"/> <s:textfield name="employee.lastName" value="%{employee.lastName}" label="%{getText('label.lastName')}" size="40"/> <s:textfield name="employee.age" value="%{employee.age}" label="%{getText('label.age')}" size="20"/> <s:select name="employee.department.departmentId" value="%{employee.department.departmentId}" list="departments" listKey="departmentId" listValue="name"/> <s:hidden name="employee.employeeId" value="%{employee.employeeId}"/> <s:submit value="%{getText('button.label.submit')}"/> <s:submit value="%{getText('button.label.cancel')}" name="redirect-action:index"/> </s:form>

Wow - a lot of code at once. Let's dissect it tag by tag. It may seem complicated, but as you'll see, it's actually really easy.

The `s:form` tag generates a standard html form, while the first `s:textfield` will generate an <input type="text" .. /> - but wait, it does more. It transforms your first tag from this:

xml <s:textfield name="employee.firstName" value="%{employee.firstName}" label="%{getText('label.firstName')}" size="40"/>

Into this:

xml <tr> <td class="tdLabel"> <label for="crud!save_employee_firstName" class="label">First Name:</label> </td> <td> <input type="text" name="employee.firstName" size="40" value="" id="crud!save_employee_firstName"/> </td> </tr>

Let's analyze that `s:textfield` tag in greater detail. First thing we encounter is the name attribute, which is similar to the HTML input tag's name attribute. So it is by itself, not very special. However, as we've seen above, this allows S2 to call the getEmployee().setFirstName() method - and even, if necessary, create the Employee object for you.

No more tens of property setters in your action - just one setter for a good ol' POJO (Plain Old Java Object), and you can call its setters right away (Note: for those who are concerned about mailicious injections, you can limit what can be set on your POJO)!

The value attribute uses a special %{..} notation, to indicate the value is not just a string 'employee.firstName', but in fact an expression that should be looked up on the ValueStack. This will make OGNL analyze the content, and look up on the ValueStack to see if it can find a method named getEmployee(), which returns a POJO which has a getFirstName() method on it.

Now, when such an expression returns null (since our Employee pojo is NOT initialised - we are doing an insert here, remember ?), S2 creates the Employee object for you, and its getFirstName() returns, of course, null. So, we're actually cheating here, because it will allow us to reuse that same form when we are going to edit an Employee. In that case, our getEmployee() would return an initialised object, so its getFirstName() would return a value, and thus display it in our input field ! Great - reusal of forms is always nice (of course, you can always use two seperate forms, but you don't need to).

Ok, so %{..} indicates an expression on the ValueStack, and returns a blank if a null is returned, or the toString() value otherwise. The OGNL analyzer is pretty powerful, so you can do things like value="my_special_valentine_is_%{girlfriend.name}", %{100 * loan.tax}, or even %{new int[100]} and %{new java.util.Date()}.

Now, we saw earlier that we could use the `s:text` tag to retrieve values from resource bundles for i18n reasons. Now that begs the question, how do we use those same values in our tags? Let's say we want to i18n'ize our textfield label. Something like this:

xml <s:textfield name="employee.firstName" value="%{employee.firstName}" label="<s:text name="label.firstName"/>" size="40"/>

Ugh. No, that's ugly, and it wouldn't work either. The solution is much cleaner and simpler: use another expression! If you were to check the extra methods provided by making our EmployeeAction extends ActionSupport, you would see a method called getText(String key). This method will look up a value in the resource bundle by its key, which is exactly what we need. So, the label would become something like this: %{getText('our_key')}. Makes sense? Thought so.

Meet the next tag, s:select, which renders a select box using an iteratable collection:

xml <s:select name="employee.department.departmentId" value="%{employee.department.departmentId}" list="departments" listKey="departmentId" listValue="name"/>
There currently is a small bug in Struts 2.0.5 that does not select the correct value by default. You can solve this by using value="%{employee.department.departmentId.toString()}".

We already covered the name attribute, so let's look at the list attribute: departments. Hmm, this might seems strange at first. Where does it come from? Well, perhaps it makes much more sense to see this: list="%{departments}". Yes, it is in fact an expression on the ValueStack that will query our action for the departments we've setup before ! Then, why are we missing the %{..} ? The answer is twofold: you can still write %{departments}, and it would work as you expected. But you have to realize, that whatever you are going to iterate is going to be an expression! So, being pretty lazy and with all this Ruby-on-rails 'minimalistic' code, we decided we might as well save you a couple of RSI-related finger moves and let you discard the %{..}.

Another quick intermezzo: you can use expressions to make your own list in an expression - which is perfect for small yes/no and male/female/eunuch selections - like this:

xml <s:select name="gender" list="%{#{'male':'Male', 'female':'Female'}}" />

By the way, did you remember the `s:action` tag we used as an alternative to the prepare method to fill up the select box? We can now use the action we executed and stored on the ValueStack by referencing it by its id:

xml <s:select name="gender" list="%{#allDepartments.departments}" />

So, instead of getting the departments from the current action, we used `s:action` to execute a different action that retrieves the department List. This allows you to split up, and reuse Actions (Note: you can go even further and program your own components with built-in actions, but that's out of scope).

Let's get back to the original plan.

Now, the listKey and listValue attributes tell `s:select` what it should use as keys and values in our select box - and what do you know, those (hidden) expressions are going to be invoked on each object it encounters in your list - in this case, getDepartmentId() and getName().

Finally, the value attribute will tell S2 where to place the typical 'selected' attribute in our generated options, and it will print it as soon as the value expression equals the key expression. Thus, in our case, as soon as your employee.getDepartment().getDepartmentId() equals the getDepartmentId() expression on one of the Department objects in our departments List. Since we don't have an employee we're editing, the expression would return null, so no selected attribute would be printed.

Finally, the last two tags, the submission tags. Submission tags, yes. One for submitting the form, and another one to cancel it. The first submit button, submits the form to the form's action attribute, in case crud!save.

xml <s:submit value="%{getText('button.label.submit')}"/> <s:submit value="%{getText('button.label.cancel')}" name="redirect-action:index"/>

The second one is more interesting though. It also submits the form to the same crud!save action as the first one, but lists a special name attribute. This name attribute will cause the ActionMapper to intercept the call, and in this case, redirect it to another action. No more fiddling with javascript to have forms with multiple submit buttons - it's all done for you, without any javascript or the troubles that come with it.

More information about Themes and Templates, Form Tags, and ActionMapper.

Validation

Alright, so you've set up the form for adding and updating employees. This is the point where you normally start sweating, you hands start shaking and you feel slightly dizzy. Have no fear, S2 is here!

S2 uses XWork's validation framework internally. It allows you to validate Models and Actions using a set of specialised (field)validators, grouped together in an xml file named YourModel-validation.xml or ActionName-validation (and, in the case of the alias, ActionName-alias-validation.xml).

Since we only want to validate the crud action, we create a file EmployeeAction-crud-validation.xml and place it in our classpath (mostly next to our compiled Action class).

xml <validators> <field name="employee.firstName"> <field-validator type="requiredstring"> <message key="errors.required.firstname"/> </field-validator> </field> <field name="employee.lastName"> <field-validator type="requiredstring"> <message key="errors.required.lastname"/> </field-validator> </field> <field name="employee.age"> <field-validator type="required" short-circuit="true"> <message key="errors.required.age"/> </field-validator> <field-validator type="int"> <param name="min">18</param> <param name="max">65</param> <message key="errors.required.age.limit"/> </field-validator> </field> < /validators>

A very important reminder: validation is once again done by an interceptor, so it should be in your stack. Even more important, validators 'query your Action /Model' and NOT your request! Keep this in mind at all times.

With that out of the way, let's analyze this snippet: there are two types of validators: field validators and general validators. We start by analyzing the field employee.firstName - so this means: we will analyze the result from the invocation of getEmployee().getFirstName() on our Action, not the request parameter named employee.firstName!

Field validators will not validate input fields - they are named field validators because they will automatically mark the input field in your form with the validation error, whereas normal validators would create actionErrors.

First we apply a `requiredstring` validator to the getEmployee().getFirstName() return value. There are quite a few validators, ranging from requiredstring, required, intrange, emailaddress, url, .. etc. Writing your own validator is not hard, and it can be reused easily. Here we use the requiredstring, which does 2 checks:

- check the availabilty of the string (!= null)
- check the length of the string to be > 0

Why the stringlength greater than 0? HTML forms will submit an empty string (**not** a null!) for an empty text input, so simply testing for null would fail if we required at least one character to be submitted.

The employee.lastName validation is exactly the same as the firstName. The only difference is in the validation of the age property.

We use in fact 2 validators: a 'required' one, which will make sure that our age is in fact set, and not null, and another one that will check the range to make sure our employee is between 18 and 65 years old. If course, if the first validator fails, we shouldn't continue processing, so that's why we can specify the short-circuit attribute. If a validator short-cicuits the validation, validation is failed and skips the current (field) validator.

The "int" validator takes 2 optional parameters: min and max, who can be set by providing two simple param tags (most items in S2 can be configured that way). This way, we ensure validation to be loosely coupled with our code and do not require re-compilation.

Of course, you should also be able to i18n'ize your error messages, so that's why we providing a <message key="my_key"/> - if you prefer otherwise, you can always add a hardcoded text as a child element of the message tags.

It does not stop there. The text you pass, be it hardcoded or a looked-up value, can in fact contain OGNL expressions as well !

Take a look at the value we get back from the 'errors.required.age.limit'-key:

errors.required.age.limit=Please provide an age between ${min} and ${max}.

And guess what: min and max are indeed the two parameters we just set before ! Using OGNL expressions you can retrieve whatever you want from your validator/action/context, giving you really nice error messages (we like nice error messages). In fact, you can not only use these expressions in your error messages, but you can even set the min and max parameters dynamically. Different types of employees could have different age requirements - OGNL and polymorphy to the rescue !

More information about Validation

Conclusion

This 'quick intro' turned out a bit longer than I anticipated. We haven't even barely touched the possibilities with Struts 2 - different templates, IoC (Pico, Spring), annotations, REST-ful action mappings, components, Hibernate integration, ajax support, pdf/xml/rss/.. generation, groovy, ... More info on 3rd party integration.

When you look at the various frameworks out there, they all let you create a certain type of (web) application really, really quickly. It's only when you want to add new/different/complex things, that you discover the limitations of the framework, often meaning serious hacking to get it working. Struts 2 is different. It is no out-of-the-box framework, where you just click a few buttons to generate a blog/cms/product catalog, but it's a framework in the true spirit

of the word. Its architecture and design is so flexible that we yet have to discover where we cannot use it for, and as such it's a great overall framework that should belong in the backpack of any serious Java (web)application programmer.