DojoTooltipsForClientValidation

NOTE: This is outdated information that applies only to Tapestry 4.

Andrea Chiumenti has patched the client side validation script file so that an input field in exception status reports its related exceptions with a dojo tooltip. When the validation is performed a second time all the exception tooltip widgets are disposed and recreated as necessary.

Here's the code for validation.js (copied from https://issues.apache.org/jira/browse/TAPESTRY-1246)

```
dojo.provide("tapestry.form.validation");

dojo.require("dojo.validate.check");
dojo.require("dojo.html.style");
dojo.require("dojo.widget.*");
dojo.require("tapestry.widget.AlertDialog");
dojo.require("dojo.widget.Tooltip");
dojo.require("dojo.collections.ArrayList");


tapestry.form.validation={
//exceptionWidgets: [], //new dojo.collections.ArrayList(),
missingClass:"fieldMissing", // default css class that will be applied to fields missing a value
invalidClass:"fieldInvalid", // default css class applied to fields with invalid data

dialogName:"tapestry:AlertDialog",

/**
* Main entry point for running form validation. The
* props object passed in contains a number of fields that
* are managed by tapestry.form:
*
* props = {
* validateForm:[true|false] // whether to run validation at all
* profiles:[profile1, profile2] // set of dojo.validate.check() style profiles
* // that may have been registered with form
* }
*
* The individual profiles will contain any of the data described by the dojo documentation
* for dojo.validate.check(). In addition to that, each profile will also have a corresponding
* string message to display if the specified condition has been met. For example, if you have
* specified that a select field named "select1" was required your profile would look something
* like:
*
* profile = {
* "required":["select1"], // normal dojo.validate.check data
* "select1":{ // tapestry field/error type specific data
* "required":"You must select a value for select1."
* }
* }
*
* It is intended for you to call dojo.validate.check(form, profile) for each profile
* stored in the "profiles" field, as well as deciding how to display errors / warnings.
*
* @return Boolean indicating if form submission should continue. If false the form
* will ~not~ be submitted.
*/
validateForm:function(form, props){
if (typeof form == "undefined") {return false;}
if (typeof props == "undefined") {return true;} // form exists but no profile? just submit I guess..
if (!props.validateForm) {return true;}

try {
this.clearValidationDecorations(form, props);
for (var i=0; i < props.profiles.length; i++) {
this._clearExceptionWidgets(props.profiles[i]);
var results=dojo.validate.check(form, props.profiles[i]);
if (!this.processResults(form, results, props.profiles[i])) {
this.summarizeErrors(form, results, props.profiles[i]);
return false;
```

```
}
}
} catch (e) {
// since so many dynamic function calls may happen in here it's best that we
// catch all of them and log them or else peoples forms might still get submitted
// and they'd never be able to figure out what was wrong
dojo.log.exception("Error validating", e, true);
return false;
}

return true;
},

/**
* Called for each registered profile on a form after
* dojo.validate.check() has been called. This function is
* expected to do UI related notifications of fields in error.
*
* @param form The form that was validated.
* @param results The result of calling dojo.validate.check(form,profile)
* @param profile The original profile used to validate form, also holds
* validation error messages to be used for each field.
*
* @return Boolean, if false form should not be submitted and all validation
* should be stopped. If true validation will continue and eventually
* form will be submitted.
*/
processResults:function(form, results, profile){
if (results.isSuccessful()) { return true; }

var formValid=true;
if (results.hasMissing()) {
var missing=results.getMissing();
for (var i=0; i < missing.length; i++) {
this.handleMissingField(missing[i], profile);
}

formValid=false;
}

if (results.hasInvalid()) {
var invalid=results.getInvalid();
for (var i=0; i < invalid.length; i++) {
this.handleInvalidField(invalid[i], profile);
}

formValid=false;
}

return formValid; // if got past successful everything is invalid
},

/**
* Default field decorator for missing fields.
*
* @param field The field element that was missing data.
* @param profile The form validation profile.
*/
handleMissingField:function(field, profile){
field=dojo.byId(field);
if (dj_undef("type", field)) {return;}
dojo.html.removeClass(field, this.invalidClass);

if (!dojo.html.hasClass(field, this.missingClass)){
dojo.html.prependClass(field, this.missingClass);
}
},

/**
* Default field decorator for invalid fields.
*
```

```
* @param field The field element that had invalid data.
* @param profile The form validation profile.
*/
handleInvalidField:function(field, profile){
field=dojo.byId(field);
if (dj_undef("type", field)) {return;}
dojo.html.removeClass(field, this.missingClass);

if (!dojo.html.hasClass(field, this.invalidClass)){
dojo.html.prependClass(field, this.invalidClass);
}
},

/**
* Clears out previous css classes set on fields
* in error.
*/
clearValidationDecorations:function(form, props){
for (var i=0; i< form.elements.length; i++) {
if (dj_undef("type", form.elements[i]) || typeof form.elements[i].type == "undefined"
|| form.elements[i].type == "submit"
|| form.elements[i].type == "hidden") { continue; }

dojo.html.removeClass(form.elements[i], this.missingClass);
dojo.html.removeClass(form.elements[i], this.invalidClass);
}
},

/**
* Optionally allows an alert dialog/dhtml dialog/etc to
* be displayed to user to alert them to the invalid state
* of their form if validation errors have occurred.
*
* @param form The form being validated.
* @param results Returned value of dojo.validate.check(form, profile)
* @param profile Validation profile definition
*/
summarizeErrors:function(form, results, profile){
var merrs=[];
var ierrs=[];
var fieldErrs=[];

tapestry.form.currentFocus=null;
//this._clearExceptionWidgets();
if (results.hasMissing()){
var fields=results.getMissing();
for (var i=0; i<fields.length; i++){
fieldErrs=[];
if(i==0 && !tapestry.form.currentFocus){
tapestry.form.currentFocus=fields[i];
}

if (profile[fields[i]] && profile[fields[i]]["required"]){
if (dojo.lang.isArray(profile[fields[i]]["required"])) {
for (var z=0; z < profile[fields[i]]["required"].length; z++) {
merrs.push(profile[fields[i]]["required"][z]);
fieldErrs.push(profile[fields[i]]["required"][z]);
}
} else {
merrs.push(profile[fields[i]]["required"]);
fieldErrs.push(profile[fields[i]]["required"]);
}
//alert(fields[i]);
this._buildExceptionTooltipWidget(profile, fields[i], fieldErrs, null);
}
}
}
if (results.hasInvalid()){
var fields=results.getInvalid();
for (var i=0; i<fields.length; i++){
fieldErrs=[];
```

```
if(i==0 && !tapestry.form.currentFocus){
tapestry.form.currentFocus=fields[i];
}
if (profile[fields[i]] && profile[fields[i]]["constraints"]){
if (dojo.lang.isArray(profile[fields[i]]["constraints"])) {
for (var z=0; z < profile[fields[i]]["constraints"].length; z++) {
ierrs.push(profile[fields[i]]["constraints"][z]);
fieldErrs.push(profile[fields[i]]["constraints"][z]);
}
} else {
ierrs.push(profile[fields[i]]["constraints"]);
fieldErrs.push(profile[fields[i]]["constraints"]);
}
//alert(fields[i]);
this._buildExceptionTooltipWidget(profile, fields[i], null, fieldErrs);
}
}
}

var msg="";
if (merrs.length > 0) {
msg+='<ul class="missingList">';
for (var i=0; i<merrs.length;i++) {
msg+="<li>"+merrs[i]+"</li>";
}
msg+="</ul>";
}
if (ierrs.length > 0) {
msg+='<ul class="invalidList">';
for (var i=0; i<ierrs.length;i++) {
msg+="<li>"+ierrs[i]+"</li>";
}
msg+="</ul>";
}

var ad=dojo.widget.byId("validationDialog");
if (ad) {
ad.setMessage(msg);
ad.show();
return;
}

var node=document.createElement("span");
document.body.appendChild(node);
var dialog=dojo.widget.createWidget(this.dialogName,
{
widgetId:"validationDialog",
message:msg
}, node);
dialog.show();
},

/**
* Clears all exception tooltip widgets
*
*/
_clearExceptionWidgets: function(profile) {
if (!profile.exceptionWidgets) {
profile.exceptionWidgets = new dojo.collections.ArrayList();
}
var iter = profile.exceptionWidgets.getIterator();

while (widget = iter.get()) {
try {
widget.destroy()
} catch (e) {
dojo.log.exception("Error destroying widget.", e, true);
}

}
profile.exceptionWidgets.clear();
```

```
},
/**
* Creates the field exception tooltip if necessary.
*
* @param fieldId The id of the field that has to expose an exception tooltip
* @return Returns The exception tooltip dom element.
*/
_createExceptionTooltip: function(fieldId) {

var id = fieldId + "-err";

var el = dojo.byId(id);
if (!el) {
el = document.createElement("div");
el.id=id;
el.className="exceptionTooltip";
dojo.dom.insertBefore(el, dojo.byId(fieldId));
}
el.innerHTML = "";
return el;
},
/**
* Fills the content of the exception tooltip with exception messages
* if messages is not empty
*
* @param exceptions The exception message array
* @param listType May be 'missingList' or 'invalidList', default is 'missingList'.
*
* @return Returns the innerHTML that needs to be added to the tooltip dom element
*/
_buildExceptionTooltip: function(exceptions, listType) {
var msg ="";

if ((exceptions)&&(exceptions.length > 0)) {
if (!listType) {
listType = "missingList";
}
msg+='<ul class="'+listType+'">';
for (var i=0; i<exceptions.length;i++) {
msg+="<li>"+exceptions[i]+"</li>";
}
msg+="</ul>";
}
return msg;
},
/**
* Creates a tooltip exception widget.
*
* @param fieldId. The id of the field in exception status.
* @param missingListExceptions The array containing missing value exception messages
* @param invalidListExceptions The array containing invalid value exception messages
*/
_buildExceptionTooltipWidget: function(profile, fieldId, missingListExceptions, invalidListExceptions) {
if (!profile.exceptionWidgets) {
profile.exceptionWidgets = new dojo.collections.ArrayList();
}
var tooltipEl = this._createExceptionTooltip(fieldId);
tooltipEl.innerHTML = this._buildExceptionTooltip(missingListExceptions, "missingList");
tooltipEl.innerHTML += this._buildExceptionTooltip(invalidListExceptions, "invalidList");
var widget = dojo.widget.createWidget("Tooltip", {id:fieldId + "-tip", connectId:fieldId, toggle:"explode"},
tooltipEl);
profile.exceptionWidgets.add(widget);
},
/**
* Validates that the input value matches the given
* regexp pattern.
*
* @param value The string value to be evaluated.
* @param pattern The regexp pattern used to match against value.
*/
isValidPattern:function(value, pattern){
```

```
if (typeof value != "string" || typeof pattern != "string") { return false; }

var re = new RegExp(pattern);
return re.test(value);
},

isPalleteSelected:function(elem){
if (elem.length > 0) { return true; }
return false;
}
}
```