Développement de tag spécifique

Les tags offrent des fonctionnalités supplémentaires au contributeur en interprétant dynamiquement du contenu lors de l'affichage de la page. Cela peut être l'affichage de listes de fiches, la diffusion d'informations liées à l'utilisateur, l'ajout de liens de contribution, ... Ce guide présente comment créer (ou surcharger) un tag spécifique dans les toolbox du produit.

La mise en place d'un tag est découpée en deux parties :

- Composant de saisie du tag dans la toolbox (bouton et formulaire de saisie)
- Interprétation du tag afin de rendre le contenu spécifique dans la page

Dans la procédure à suivre, nous prendrons l'exemple d'un tag pour lequel le contenu inséré aura la forme suivante : [traitement_spec;CODE=code;CODE_RUBRIQUE=12346;LANGUE=0]

Saisie du tag dans la toolbox

Déclaration d'un TagType

Ce bean décrit un composant de saisie tag aussi bien en termes d'affichage dans l'interface de saisie qu'en termes de gestion (saisie, modification, insertion...).

Pour notre exemple, nous le déclarons de la manière suivante :

application_context.xml

```
<bean class="com.kosmos.toolbox.bean.TagType">
    <property name="id" value="monTag"/>
    <property name="category" value="tag"/>
    <property name="labelKey" value="KTAG.TRAITEMENT_SPEC"/>
    <property name="inputFragment"
value="./templates/tag_traitement_spec.jsp"/>
    <property name="jsScript"
value="/adminsite/scripts/contents/ktag/tag_traitement_spec.js"/>
    <property name="tagPattern"
value="[traitement_spec;CODE=#(CODE);CODE_RUBRIQUE=#(CODE_RUBRIQUE);LANG
UE=#(LANGUE)]"/>
    <property name="icons">
        <bean class="org.apache.commons.lang3.tuple.ImmutablePair">
            <constructor-arg index="0"
value="/adminsite/styles/img/24x24/ckeditor/ktag/traitement_spec.png"/>
            <constructor-arg index="1"
value="/adminsite/styles/img/24x24/ckeditor/ktag/highlight/traitement_sp
ec.png"/>
        </bean>
    </property>
    <property name="permissions">
        <list>
            <ref bean="monTagPermission"/>
        </list>
    </property>
</bean>
```

En détail :

- **id** : l'identifiant du tag (champ libre)
- **category** : doit être positionnée à la valeur "tag"
- **labelKey** : libellé affiché (on renseigne la clé définie dans les fichiers de langue)
- **inputFragment** : le fragment de JSP à inclure pour la saisie et l'édition du tag. Il est recommandé de placer ce fichier dans le répertoire "WEB-INF/jsp/toolbox/ktag/templates/" au sein de votre projet.
- **jsScript** : le fichier javascript à utiliser pour la gestion de la saisie/modification et insertion du tag. Il est recommandé de placer ce fichier dans le répertoire "adminsite/scripts/contents/ktag/" au sein de votre projet.

- **tagPattern** : le format du contenu à insérer. Il s'agit de notre tag de départ mais dont les valeurs ont été "variabilisées". Pour plus d'informations sur cette syntaxe, consulter la page Présentation de l'API Kportal CKEditor
- **icons** (optionnel) : une paire d'icônes représentant l'état normal et l'état sélectionné. Si cette propriété n'est pas renseignée, l'icône par défaut est affichée.
- **permissions** (optionnel) : liste des permissions

Déclaration des permissions

Les éléments TagType disposent d'une liste de permissions. Si l'utilisateur possède au moins une des permissions rattachées au TagType, ce dernier sera visible dans l'interface.

Une permission est déclarée de la façon suivante :

application_context.xml

```
<bean id="monTagPermission"
class="com.kportal.core.autorisation.Permission">
    <property name="idExtension" value="core"/>
    <property name="id" value="TECH"/>
    <property name="code" value="mtp"/>
    <property name="libelle" value="Insertion de mon tag"/>
    <property name="type" value="CONTRIBUTION"/>
</bean>
```

Si la permission n'est pas déclarée de manière indépendante, elle ne remontera pas dans l'interface de saisie des rôles.

Création de la JSP de saisie

Formulaire de saisie

Cette JSP va contenir les champs nécessaires à la saisie du tag.

Il est recommandé de respecter les règles suivantes :

- Nommer les champs en accord avec les placeholders du tagPattern pour faciliter la compréhension
- Préciser les éléments obligatoires (attribut "required")
- Identifier les champs intervenants dans les calculs JavaScript à l'aide de classes spécifiques (js-ma_classe)

La JSP décrite pour notre plugin prend donc la forme suivante :

tag_traitement_spec.jsp

```
<%@ page import="java.util.Map" %>
<%@ page import="com.jsbsoft.jtf.core.LangueUtil" %>
<%@ page import="com.kosmos.toolbox.service.impl.ServiceKTag" %>
<%@ page import="com.kportal.core.config.MessageHelper" %>
<%@ page import="com.univ.objetspartages.util.LabelUtils" %>
<%@ page import="org.apache.commons.lang3.StringUtils" %>
<jsp:useBean id="infoBean" class="com.jsbsoft.jtf.core.InfoBean"
scope="request" />
<jsp:useBean id="viewDescriptor"
class="com.kosmos.toolbox.model.TagViewDescriptor" scope="request" />
<p>
 <label for="CODE"><%=
MessageHelper.getCoreMessage("KTAG.TRAITEMENT_SPEC.CODE") %></label>
    <input type="text" id="CODE" name="CODE" value="<%=
StringUtils.defaultString(viewDescriptor.getData().get("CODE") %>"
required />
</p>
<div>
    <span class="label colonne "><%=
MessageHelper.getCoreMessage("BO_RUBRIQUE") %></span>
    <div id="kMonoSelectCODE_RUBRIQUE" class="kmonoselect ui-buttonset"
data-value="<%=
StringUtils.defaultString(viewDescriptor.getData().get("LIBELLE_CODE_RUB
RIQUE") %>" data-placeholder=""
data-editaction="/adminsite/tree/tree.jsp?JSTREEBEAN=rubriquesJsTree&amp
;DISPLAY=full&amp;SELECTED={0}&amp;CODE={1}"
data-popintitle="LOCALE_BO.popin.title.rubrique.mono"
data-popinwidth="530" data-popinvalidate="false" data-title="">
        <input type="hidden" name="#FORMAT_CODE_RUBRIQUE"
value="1;1;0;0;LIB=Rubrique;0">
        <input type="hidden" id="CODE_RUBRIQUE" name="CODE_RUBRIQUE"
value="<%=
StringUtils.defaultString(viewDescriptor.getData().get("CODE_RUBRIQUE")
%>">
        <input type="hidden" name="LIBELLE_CODE_RUBRIQUE" value="<%=
StringUtils.defaultString(viewDescriptor.getData().get("LIBELLE_CODE_RUB
RIQUE") %>">
    </div>
</div>
<p>
 <label for="LANGUE"><%=
MessageHelper.getCoreMessage("KTAG.TRAITEMENT_SPEC.LANGUE") %></label>
    <select id="LANGUE" name="LANGUE" required>
  <option value="0" <%=
"0".equals(viewDescriptor.getData().get("LANGUE")) ? "selected" :
StringUtils.EMPTY %>>Français</option>
  <option value="1" <%=
"1".equals(viewDescriptor.getData().get("LANGUE")) ? "selected" :
StringUtils.EMPTY %>>Anglais</option>
 </select>
</p>
```

Modification d'un tag

Le bean "viewDescriptor" présent dans la JSP décrite ci-dessus, est construit en amont par le service de gestion des tags et permet de gérer la modification.

Dans ce bean, la Map "data" contient l'ensemble des valeurs qu'il a été possible de récupérer en confrontant le "tagPattern" avec la valeur du tag.

Certaines valeurs ne peuvent pas être retrouvées directement dans le tag. Dans notre JSP par exemple, la valeur du champ "LIBELLE_CODE_RUBRIQUE" ne peut pas être retrouvée par la construction générique puisqu'elle n'est pas présente dans le tagPattern ou dans la valeur du tag. L'API du core expose donc une classe abstraite (AbstractTagDataProvider) permettant de palier aux manques de cette construction générique (cf. paragraphe suivant).

Création d'un DataProvider

Les DataProvider permettent d'alimenter la page JSP de saisie. Ils doivent être implémentés de manière spécifique pour chaque tag. Il ne peut ainsi y avoir qu'un seul DataProvider par tag.

La classe créée pour notre exemple sera donc écrite comme suit :

MonTagDataProvider.java

```
package com.kosmos.toolbox.provider.impl.link;

import java.util.Map;

import com.kosmos.toolbox.bean.TagValue;
import com.kosmos.toolbox.provider.impl.AbstractTagDataProvider;
import com.univ.objetspartages.util.InfosRubriquesUtils;

public class MonTagDataProvider extends AbstractTagDataProvider {

    @Override
    public void populateData(final Map<String, String> data, TagValue
tagValue) {
        final String label =
InfosRubriquesUtils.getIntitule(data.get("CODE_RUBRIQUE"));
        data.put("LIBELLE_CODE_RUBRIQUE", label);
    }
}
```

Les données passées sont issues de la construction générique. A ce stade, on dispose donc de toutes les informations que le processus générique a réussi à récupérer. L'objet TagValue représente le tag en cours de modification. Les données sont volontairement restreintes au type "String" : la JSP ne doit pas avoir de calcul complexe à effectuer.

Une fois le DataProvider créé, il faut le rattacher au TagType correspondant. La configuration Spring devient :

application_context.xml

```
<bean class="com.kosmos.toolbox.bean.TagType">
    <property name="id" value="monTag"/>
    <property name="category" value="tag"/>
    <property name="labelKey" value="KTAG.TRAITEMENT_SPEC"/>
    <property name="inputFragment"
value="./templates/tag_traitement_spec.jsp"/>
    <property name="jsScript"
value="/adminsite/scripts/contents/ktag/tag_traitement_spec.js"/>
    <property name="tagPattern"
value="[traitement_spec;CODE=#(CODE);CODE_RUBRIQUE=#(CODE_RUBRIQUE);LANG
UE=#(LANGUE)]"/>
    <property name="icons">
        <bean class="org.apache.commons.lang3.tuple.ImmutablePair">
            <constructor-arg index="0"
value="/adminsite/styles/img/24x24/ckeditor/ktag/traitement_spec.png"/>
            <constructor-arg index="1"
value="/adminsite/styles/img/24x24/ckeditor/ktag/highlight/traitement_sp
ec.png"/>
        </bean>
    </property>
</bean>

<bean class="com.kosmos.toolbox.provider.impl.tag.MonTagDataProvider">
    <property name="forId" value="monTag"/>
</bean>
```

Création du fichier javascript

Ce fichier va permettre de spécifier tous les comportements JS utilisés lors de la saisie du tag, en particulier le renvoi de la valeur du tag dans le champ toolbox associé.

Il est recommandé d'encapsuler vos comportements au sein d'une fonction anonyme.

La récupération des informations et la construction de la valeur à insérer dans la toolbox doivent être implémentées spécifiquement.

Pour notre tag, le fichier JS se présente comme suit :

tag_traitement_spec.js

```
(function($) {
    'use strict';

    var $editionPanel = $('[data-panel="monTag"]');

    function buildAndSendValue() {
        var $code = $('[name="CODE"]', $editionPanel),
            $codeRubrique = $('[name="CODE_RUBRIQUE"]', $editionPanel),
            $langue = $('[name="LANGUE"]', $editionPanel);
        // Construction et envoi du message
        if($code.val() && $codeRubrique.val() && $langue.val()) {
            // Construction de la valeur du tag
            var message = { 'tag': 'monTag', 'value':
'[traitement_spec;CODE=' + $code.val() + 'CODE_RUBRIQUE=' +
$codeRubrique.val() + ';LANGUE=' + $langue.val() + ']' };
            COMMONS_MESSAGES.postMessageToParent(message);
        } else {
            // Tous les champs obligatoires ne sont pas présents. La
valeur est invalide alors on envoi le message 'erase'
            COMMONS_MESSAGES.postMessageToParent('erase');
        }
    }

    $('input[name], select[name]').on('change', buildAndSendValue);

})(jQuery.noConflict());
```

La chose la plus importante à remarquer dans ce code est l'utilisation de "COMMON_MESSAGES" : cet utilitaire nous permet de remonter les informations au plugin "ktag" de manière simple. L'utilisation de cette fonction permet donc la récupération pour insertion au niveau du plugin, mais permet aussi de gérer l'état du bouton "OK" du dialogue d'insertion d'un tag :

- Si le message contient bien un champ 'tag' et un champ 'value', le bouton s'active
- Dans tous les autres cas, le bouton redevient inactif. Par convention, on envoi le message 'erase'

Interprétation du tag et restitution

Pour détecter et interpréter le tag, la configuration s'effectue au niveau de Spring via la classe « PluginTagToolbox ».

MonContext.xml

```
<bean
class="com.kportal.extension.module.plugin.toolbox.DefaultPluginTag">
    <property name="identifiantTag" value="[traitement_spec;" />
    <property name="extracteur">
        <bean
class="com.kportal.tag.extracteur.impl.DefautExtracteurTag" />
    </property>
    <property name="interpreteur">
        <bean
class="com.kosmos.toolbox.tag.interpreteur.impl.InterpreteurTagSpecifiqu
e">
   <property name="pluginTag" ref="identifiantTag"/>
   <property name="pathJsp"
value="/WEB-INF/jsp/common/fo/tags/tagSpecifique.jsp"/> </property>
  </bean>
  <property name="contexteTag">
     <list>
   <util:constant
static-field="com.kportal.tag.util.ContexteTagUtil.DEFAUT" />
  </list>
 </property>
    <property name="baliseOuvrante" value="[traitement_spec;" />
    <property name="baliseFermante" value="]" />
</bean>
```

Paramètres :

- identifiantTag : identifiant du bean
- extracteur : classe chargée de repérer le tag dans une chaîne de caractères
- interpreteur : classe spécifique chargée de générer le contenu attendu à partir du tag et de ses options. L'interpréteur peut aussi déclarer une JSP afin de gérer l'affichage du TAG en la déclarant dans la propriété pathJsp (optionnel).
- baliseOuvrante : chaine de caractères indiquant le point de départ du tag
- baliseFermante : chaine de caractères indiquant la fin du tag à interpréter
- contexteTag : contexte d'exécution de l'interpréteur (l'utilisation du contexte par défaut signifie que le tag sera tout le temps interprété, mais il est possible de préciser un autre contexteTag pour que le tag ne soit interprété que pour un contexte, ex : mobile, newsletter, spécifique)

Extracteur

L'extracteur vérifie juste la présence du tag dans le texte : présence de la balise ouvrante et de la balise fermante. La valeur à mettre est « DefautExtracteurTag ».

Interpréteur

L'interpréteur est propre à chaque tag, il permet de récupérer les paramètres du tag et de retourner à la JSP la valeur attendue.

Deux méthodes sont à implémenter :

- interpreterTag qui retourne la chaîne interprétée en fonction de la valeur fournie en entrée
- getReferenceTag qui calcule les références entre les fiches

InterpreteurTagSpecitfique.java

```
package com.kosmos.toolbox.tag.interpreteur.impl;

import org.apache.commons.lang3.StringUtils;
import com.kportal.tag.interpreteur.impl.AbstractInterpreteurTag;

public class InterpreteurTagSpecitfique  extends AbstractInterpreteurTag
{

    @Override
    public String interpreterTag(final String texteAInterpreter, final
String baliseOuvrante, final String baliseFermante) throws Exception {
        String textAffiche = StringUtils.EMPTY; //TODO : Traitement
Spécifique à implémenter
        return textAffiche;
    }

    @Override
    public String getReferenceTag(String texteAInterpreter, final String
baliseOuvrante, final String baliseFermante) {
        String refFiche = StringUtils.EMPTY;//TODO : Traitement
Spécifique à implémenter.
        return refFiche;
    }
}
```