nuxeo | Professional Open Source ECM

Nuxeo 5 Tutorial

Documents and Schemas

Last Modification	2007-01-30
Project Code	NUXTUT
Document ID	DOCS
Copyright	

Versioning

Version	Date	Participant(s)	Comments
0.1	2007-01-22	• Florent Guillaume	First draft
0.2	2007-01-30	• Florent Guillaume	Added actions

Contents

1 Documents

1.1 Basic concepts

In Nuxeo 5, a fundamental entity is the **document**. A file, a note, a vacation request, an expense report, can all be though of as documents. Objects that contain documents, like a folder or a workspace, are also themselves documents.

Any given document has a **document type**. The document type is specified at creation time, and does not change during the lifetime of the document. When referring to the document type, a short string is often used, for instance “Note” or “Folder”.

A document type is a grouping of several **schemas**. A schema represents the names and structure (types) of a set of fields in a document. For instance, a commonly-used schema is the *Dublin Core* schema, which specifies a standard set of fields used for document metadata like *title*, *description*, *modification date*.

1.2 Adding a document type

To add a new document type, we will start by creating a schema that the document type will use. The schema is defined in a `.xsd` file, and registered by a contribution to a standard extension point.

The document type will be registered through a contribution to another standard extension point, and will specify which schemas it uses. We will also register the document type as a high-level type used by the EJB and rendering layers.

> “Core” document types and “ECM” component types should not be confused. The former live in the core Nuxeo packages, the latter belong to the high level components, apart from their definition, most of the uses of document types refer to the “ECM” kind.

1.2.1 Adding a schema

First, we will create a schema in the **`resources/schemas/sample.xsd`** file:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://project.nuxeo.org/sample/schemas/sample/">
  <xs:element name="sample1" type="xs:string"/>
  <xs:element name="sample2" type="xs:string"/>
</xs:schema>
```

This schema defines two things:

- an XML namespace that will be associated with the schema,
- two elements and their type.

The two elements are `sample1` and `sample2`. They are both of type `xs:string`, which is a standard type defined by the XMLSchema specification.

This schema has to be referenced by a configuration file so that the system knows it has to be read. The configuration file will be placed in **OSGI-INF/core-types-contrib.xml** (the name is just a convention):

```
<?xml version="1.0"?>
<component name="org.nuxeo.project.sample.coreTypes">
  <extension target="org.nuxeo.ecm.core.schema.TypeService" point="schema">
    <schema name="sample" src="schemas/sample.xsd"/>
  </extension>
</component>
```

We name our schema **sample**, and the `.xsd` schema was placed under `resources/` which means that at runtime, it will be available directly from the Jar, therefore we reference it through the path **schemas/sample.xsd**. The schema is registered through an extension point of the Nuxeo component `org.nuxeo.ecm.core.schema.TypeService` named `schema`. Our own extension component is given a name, `org.nuxeo.project.sample.coreTypes`, which is not very important as we only contribute to existing extension points and don't define new ones.

Finally, we tell the system that the `OSGI-INF/core-types-contrib.xml` file has to be read, by mentioning it in the `Nuxeo-Component` part of the **META-INF/MANIFEST.MF** file of our bundle:

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 1
Bundle-Name: NXSample project
Bundle-SymbolicName: org.nuxeo.project.sample;singleton:=true
Bundle-Version: 1.0.0
Bundle-Vendor: Nuxeo
Require-Bundle: org.nuxeo.runtime,
 org.nuxeo.ecm.core.api,
 org.nuxeo.ecm.core
Provide-Package: org.nuxeo.project.sample
Nuxeo-Component: OSGI-INF/core-types-contrib.xml
```

1.2.2 Adding a core document type

By itself, the schemas is not very useful. We will associate it with a new “core” document type that we will be creating. In the same **OSGI-INF/core-types-contrib.xml** as above, we add the following:

```
<?xml version="1.0"?>
<component name="org.nuxeo.project.sample.coreTypes">
  <extension target="org.nuxeo.ecm.core.schema.TypeService" point="schema">
    <schema name="sample" src="schemas/sample.xsd" prefix="smp"/>
  </extension>
  <extension target="org.nuxeo.ecm.core.schema.TypeService" point="doctype">
    <doctype name="Sample" extends="Document">
      <schema name="common"/>
```

```
      <schema name="dublincore"/>
      <schema name="sample"/>
    </doctype>
  </extension>
</component>
```

The document type is defined through a contribution to an other extension point, `doctypes`, of the same extension component as before, `org.nuxeo.ecm.core.schema.TypeService`. We specify that our document type, **`Sample`**, will be an extension of the standard system type `Document`, and that it will be composed of three schemas, two standard ones and our specific one.

The Dublin Core schema that we use already contains standard metadata fields, like a title, a description, the modification date, the document contributors, etc. Adding it to our document type ensures that a minimum of functionality will be present.

1.2.3 Adding an ECM document type

After the "core" document type, we will need to create the "ECM" document type. This is done through a contribution to another extension point, that we will place in the **`OSGI-INF/ecm-types-contrib.xml`**, the basic structure of this file is:

```
<?xml version="1.0"?>
<component name="org.nuxeo.project.sample.ecm.types">
  <require>org.nuxeo.ecm.types.TypeService</require>
  <extension target="org.nuxeo.ecm.types.TypeService" point="types">
    <type id="Sample" coretype="Sample">
      ...
    </type>
  </extension>
</component>
```

The document id is usually the same as the underlying core type, but this is not mandatory. The `type` element will contain all the information for this type

This extension file is added to **`META-INF/MANIFEST.MF`** so that it will be taken into account by the deployment mechanism:

```
Nuxeo-Component: OSGI-INF/core-types-contrib.xml,
 OSGI-INF/ecm-types-contrib.xml
```

Inside the **`type`** element, we will provide various information.

1.2.3.1 UI label and icon

```
<label>Sample document</label>
<icon>/icons/file.gif</icon>
```

The label and icon are used by the user interface, for instance in the creation page when a list of possible types is displayed. The icon is also used whenever a list of document wants to associate an icon with each.

1.2.3.2 Default view

```
<default-view>view_documents</default-view>
```

The default view specifies the name of the Facelet to use for this document if nothing is specified. This corresponds to a file that lives in the webapp, in this case `view_documents.xhtml` is a standard view defined in the base Nuxeo bundle, that takes care of displaying available tabs, and the document body according to the currently selected type. Changing it is not advised unless extremely nonstandard rendering is needed.

1.2.3.3 Layout

```
<layout>
  <widget jsfcomponent="h:inputText"
    schemaname="dublincore" fieldname="title"
    required="true" />
  <widget jsfcomponent="h:inputTextarea"
    schemaname="dublincore" fieldname="description" />
  <widget jsfcomponent="h:inputText"
    schemaname="sample" fieldname="sample1" />
  <widget jsfcomponent="h:inputText"
    schemaname="sample" fieldname="sample2" />
</layout>
```

This section defines a series of widgets, that describe what the standard layout of this document type will be. A layout is a series of widgets, which make the association between the field of a schema with a JSF component.

The layout is used by the standard Nuxeo modification and summary views, to automatically display the document according to the layout rules. Note that the layout system is still young and is subject to minor changes in the future.

Here we define four widgets, displayed as simple input fields or as a text area.

1.2.3.4 Containment rules

```
<type id="Folder" coretype="Folder">
  <subtypes>
    <type>Sample</type>
  </subtypes>
</type>
<type id="Workspace" coretype="Workspace">
  <subtypes>
    <type>Sample</type>
  </subtypes>
</type>
```

This contributes a rule to an already existing type, `Folder`. The **`subtypes`** element specifies which types can be created inside the type in which the element is embedded.

Here, we specify that our Sample type can be created in a `Folder` and a `Workspace`.

1.2.3.5 Summary

The final **`OSGI-INF/ecm-types-contrib.xml`** looks like:

```
<?xml version="1.0"?>
<component name="org.nuxeo.project.sample.ecm.types">
  <require>org.nuxeo.ecm.types.TypeService</require>
  <extension target="org.nuxeo.ecm.types.TypeService" point="types">
    <type id="Sample" coretype="Sample">
      <label>Sample document</label>
      <icon>/icons/file.gif</icon>
      <default-view>view_documents</default-view>
      <layout>
        <widget jsfcomponent="h:inputText"
          schemaname="dublincore" fieldname="title"
          required="true" />
        <widget jsfcomponent="h:inputTextarea"
          schemaname="dublincore" fieldname="description" />
        <widget jsfcomponent="h:inputText"
          schemaname="sample" fieldname="sample1" />
        <widget jsfcomponent="h:inputText"
          schemaname="sample" fieldname="sample2" />
      </layout>
    </type>
    <type id="Folder" coretype="Folder">
      <subtypes>
        <type>Sample</type>
      </subtypes>
    </type>
    <type id="Workspace" coretype="Workspace">
      <subtypes>
        <type>Sample</type>
      </subtypes>
    </type>
  </extension>
</component>
```

1.3 Associating actions to a document

Various **actions** can be associated to a document. Actions are organized in categories and are used in various context, the main one being the “tab” category usually displayed a the top of the main area to provide various contextual views.

Actions are registered through extensions, and are therefore made available through a new file which we will place in **`OSGI-INF/actions-contrib.xml`**. This file is referenced in **`META-INF/MANIFEST.MF`**:

```
Nuxeo-Component: OSGI-INF/core-types-contrib.xml,
 OSGI-INF/ecm-types-contrib.xml,
 OSGI-INF/actions-contrib.xml
```

1.3.1 Defining an action

The extension file **OSGI-INF/actions-contrib.xml** has the following structure:

```
<?xml version="1.0"?>
<component name="org.nuxeo.project.sample.actions">

  <require>org.nuxeo.ecm.platform.ec.actions</require>

  <extension target="org.nuxeo.ecm.actions.ActionService" point="actions">
    <action id="tab_sample_view"
      link="/incl/tabs/sample_view.xhtml" enabled="true"
      label="action.sample.view" icon="/icons/file.gif" order="9">
      <category>VIEW_ACTION_LIST</category>
    </action>
  </extension>
</component>
```

The **action** element has a number of attributes:

- **id** is a unique id that can be used to reference it, as we will see below for filters,
- **link** is a reference to an XHTML file holding the JSF fragment that will be used to display this action when it is selected,
- **enabled** is a simple boolean flag that can be used to quickly disable an action,
- **label** is used to associate a human-readable title to the action, **icon** is used to associate an icon; both label and icon reference files in the webapp, which are therefore in the web/ folder,
- **order** specifies a relative order to the other actions in the category.

The **category** element specifies to which category the action is to be associated. In the case of the standard document tabs, this category is **VIEW_ACTION_LIST**.

1.3.2 Action JSF

The **incl/tabs/sample_view.xhtml** file is a JSF fragment, its basic structure is shown below with an example rendering two fields in edit mode and allowing the changes to be saved:

```
<div xmlns="http://www.w3.org/1999/xhtml"
  xmlns:f="http://java.sun.com/jsf/core"
  xmlns:h="http://java.sun.com/jsf/html"
  xmlns:c="http://java.sun.com/jstl/core"
  xmlns:t="http://myfaces.apache.org/tomahawk">
```

```
<h3><h:outputText value="#{messages['label.sample.view']}"/></h3>

<h:form>
  <h:panelGrid columns="2" styleClass="dataInput"
    columnClasses="labelColumn, fieldColumn">

    <h:outputText value="#{messages['label.sample.sample1']}"/>
    <h:inputText value="#{currentItem.sample.sample1}"
      class="dataInputText" id="sample1" required="true">
      <f:validateLength minimum="5"/>
    </h:inputText>
    <h:outputText value=""/>
    <h:message styleClass="errorMessage" for="sample1"/>

    <h:outputText value="#{messages['label.sample.sample2']}"/>
    <h:inputText value="#{currentItem.sample.sample2}"
      class="dataInputText" id="sample2"/>
    <h:outputText value=""/>
    <h:message styleClass="errorMessage" for="sample2"/>

    <h:outputText value=""/>
    <h:commandButton type="submit"
      value="#{messages['command.save']}" styleClass="button"
      action="#{documentActions.updateCurrentDocument}"/>
  </h:panelGrid>
</h:form>
</div>
```

This example shows various features standard of JSF, and how the fields of our document are referenced in the JSF Expression Language (EL):

- **#{currentItem.sample.sample1}** retrieves the value of the **sample1** field in the **sample** schema of the document. **currentItem** is a Seam component holding the current document.
- **#{documentActions.updateCurrentDocument}** is a JSF action tied to the Nuxeo 5 framework whose purpose is to process the fields modified in the form and to save them in the current document.
- The **h:message** tag will display any error message encountered during the JSF validation.

1.3.3 Translations

In the JSF file, the EL expression **#{messages['label.sample.view']}** translates **label.sample.view** by looking it up in the standard component **messages**. This component is managed by Seam and holds all the translations that have been set up.

Translations are created in the **OSGI-INF/l10n/message_LL.properties**. LL is replaced by the appropriate language code, for instance for French we'll use **OSGI-INF/l10n/message_fr.properties**:

```
action.sample.view=Voir Sample
label.sample.view=Visualisation Sample
label.sample.sample1=Sample 1
label.sample.sample2=Sample 2
```

Note that the **command.save** message used in the JSF file is a standard one provided by the Nuxeo 5 framework.

1.3.4 Action filters

Some rules are needed to decide when to display an action, as it's rarely the case that an action must be shown everywhere. To express these rules, we use **filters**.

A filter is either true or false in a given context. If the filter is true then any action associated to that filter will be displayed, otherwise it will not. The filter holds a set of rules that will decide its outcome given the context. The rules can be based on the **type** of the current document, but also on any **facet** it has, or even on arbitrary **condition**s.

A filter can be put directly inside the **action** element of the **actions** extension point:

```
<extension target="org.nuxeo.ecm.actions.ActionService" point="actions">
  <action ...>
    ...
    <filter id="sample_filter">
      <rule grant="true">
        <type>Sample</type>
      </rule>
    </filter>
  </action>
</extension>
```

In this case, the **sample_filter** filter will be true when the type is **Sample**, and therefore when we display a document of that type. Because the filter is put inside our action, it is automatically associated with it.

A filter can also be expressed in a separate **filters** extension point:

```
<extension target="org.nuxeo.ecm.actions.ActionService" point="filters">
  <filter id="view">
    <rule grant="true">
      <type>Sample</type>
    </rule>
  </filter>
  <filter id="rights">
    <rule grant="true">
      <type>Sample</type>
    </rule>
  </filter>
</extension>
```

If any filter has already been defined in the framework with the same **id**, then the filter is *extended*. In the example above, we extend two standard filters, **view** and **rights**, to add a rule to them making them true in the case of our **Sample** type. This means that any standard tab already associated with these filters will take into account our new document type, in this case the standard summary and rights management tabs will be displayed.

1.3.5 Summary

The final file is:

```
<?xml version="1.0"?>
<component name="org.nuxeo.project.sample.actions">

  <require>org.nuxeo.ecm.platform.ec.actions</require>

  <extension target="org.nuxeo.ecm.actions.ActionService" point="actions">
    <action id="tab_sample_view"
      link="/incl/tabs/sample_view.xhtml" enabled="true"
      label="action.sample.view" icon="/icons/file.gif" order="9">
      <category>VIEW_ACTION_LIST</category>
      <filter id="sample_filter">
        <rule grant="true">
          <type>Sample</type>
        </rule>
      </filter>
    </action>
  </extension>

  <extension target="org.nuxeo.ecm.actions.ActionService" point="filters">
    <filter id="view">
      <rule grant="true">
        <type>Sample</type>
      </rule>
    </filter>
    <filter id="rights">
      <rule grant="true">
        <type>Sample</type>
      </rule>
    </filter>
  </extension>

</component>
```

2 Writing a view

2.1 Package structure

2.2 Writing a tab view

3 Writing an Action Listener

3.1 Package structure

3.2 Basic EJB