

PLONE

- Gestion de contenu
- Intégration graphique
- Paramétrage
- Framework de développement



Ce support de formation est créé et mis à jour par Pilot Systems
sous licence Creative Commons 2.0 France



Paternité
Pas d'utilisation commerciale
Pas de modification



Pilot Systems
9 rue Desargues, 75011 Paris - Tél : +33 (0)1 44 53 05 55
info@pilotsystems.net

Table des matières

| | |
|--|-----------|
| 1. Introduction au document..... | 1 |
| 1.1. Version actuelle de ce document..... | 1 |
| 1.2. A propos de ce document..... | 1 |
| 1.3. Contributions et crédits..... | 2 |
| 1.4. La licence Creative Commons..... | 3 |
| 2. Présentation de Plone..... | 4 |
| 2.1. Structure d'un site..... | 6 |
| 2.2. Types de contenu..... | 8 |
| 2.3. Workflows..... | 9 |
| 2.4. Introduction au catalogue..... | 9 |
| 2.5. Présentation du mécanisme de skins..... | 10 |
| 3. Page Template dans Plone..... | 12 |
| 3.1. Rappels/Introduction à TAL et METAL..... | 14 |
| 3.2. Considérations d'internationalisation..... | 19 |
| 3.3. Création d'un formulaire Plone 2..... | 23 |
| 3.4. Présentation du gabarit et des macros standards..... | 28 |
| 3.5. Structure d'une page..... | 30 |
| 4. Développement de nouveaux mécanismes..... | 32 |
| 4.1. Requêtes sur le catalogue..... | 32 |
| 4.2. Requêtes sur un SGBDR..... | 35 |
| 4.3. Création d'un portlet..... | 38 |
| 4.4. Définition d'une action de site..... | 40 |
| 4.5. Gestion des utilisateurs..... | 41 |
| 4.6. Dossiers automatiques (smart folders)..... | 43 |
| 5. Création d'un nouveau type de contenu..... | 45 |
| 5.1. Présentation d'Archetypes..... | 45 |
| 5.2. Définition d'un type et de ses vues..... | 46 |
| 5.3. Cas d'utilisation : RichDocument..... | 49 |
| 6. Personnalisation de workflow..... | 55 |
| 6.1. Création d'un nouvel objet workflow, définition des états et transitions..... | 55 |
| 6.2. Actions automatiques aux transitions..... | 57 |

1. Introduction au document

Version actuelle de ce document

A propos de ce document

Contributions et crédits

La licence Creative Commons

1.1. Version actuelle de ce document

Version 0.92 du 22 septembre 2006

1.2. A propos de ce document



Ce document est une photographie de notre système de travail collaboratif Wiki At Work.

L'ensemble des pages de ce document est accessible par l'URL <https://projets.pilotsystems.net>. Si vous n'êtes pas un utilisateur déclaré dans le système, vous pouvez en faire la demande à l'adresse <mailto:waw@pilotsystems.net>.

Wiki At Work repose sur un serveur d'application Zope.

Un wiki est un système de gestion de contenu de site Web qui rend les pages Web librement et également modifiables par tous les visiteurs autorisés. Les wikis sont utilisés pour faciliter l'écriture collaborative de documents avec un minimum de contrainte. Le wiki a été inventé par Ward Cunningham en 1995, pour une section d'un site sur la programmation informatique qu'il a appelée WikiWikiWeb. Le mot « wiki » vient du terme hawaïen wiki wiki, qui signifie « rapide » ou « informel ». Au milieu des années 2000, les wikis ont atteint un bon niveau de maturité et sont associés au Web 2.0. Créée en 2001, l'encyclopédie Wikipédia est devenue le wiki le plus visité au monde. *Définition Wikipedia*

Précisions sur les liens

L'ensemble des liens internes au document pointe sur l'infrastructure sécurisée Wiki At Work. Par conséquent, il vous faudra disposer d'un utilisateur déclaré dans le système pour accéder à ces liens.

1.3. Contributions et crédits

Vous souhaitez contribuer ?

- corriger des erreurs
- participer à la structure de ce document

Pour contribuer, vous pouvez nous contacter par email à l'adresse suivante : <mailto:plone@pilotsystems.net>.

Ont déjà participé à ce document

- vous ?
- l'équipe Pilot Systems

1.4. La licence Creative Commons



Paternité - Pas d'Utilisation Commerciale - Pas de Modification 2.0 France

Vous êtes libres de reproduire, distribuer et communiquer cette création au public selon les conditions suivantes :



Paternité. Vous devez citer le nom de l'auteur original.



Pas d'Utilisation Commerciale. Vous n'avez pas le droit d'utiliser cette création à des fins commerciales.



Pas de Modification. Vous n'avez pas le droit de modifier, de transformer ou d'adapter cette création.

- A chaque réutilisation ou distribution, vous devez faire apparaître clairement aux autres les conditions contractuelles de mise à disposition de cette création.

- Chacune de ces conditions peut être levée si vous obtenez l'autorisation du titulaire des droits.

Ce qui précède n'affecte en rien vos droits en tant qu'utilisateur (exceptions au droit d'auteur : copies réservées à l'usage privé du copiste, courtes citations, parodie...)

Ceci est le Résumé Explicatif du Code Juridique (la version intégrale du contrat*).
Avertissement disclaimer**

* <http://creativecommons.org/licenses/by-nc-nd/2.0/fr/legalcode>

** <http://creativecommons.org/licenses/disclaimer-popup?lang=fr>

2. Présentation de Plone

Qu'est ce que Plone ?

Définition

Plone est un système de gestion de contenu Web Open Source publié selon les termes de la GNU GPL. Il est construit au-dessus du serveur d'application Zope et de son extension CMF (Content Management Framework).

Un système de gestion de contenu (en anglais Content Management System ou CMS) est une famille de logiciels de conception et de mises à jour dynamiques de sites web possédant en particulier les fonctionnalités suivantes :

- permettre à plusieurs individus de travailler sur un même document
- fournir une chaîne de publication (workflow) offrant par exemple la possibilité de publier (mettre en ligne) des documents
- séparer les opérations de gestion de la forme et du contenu
- structurer le contenu (utilisation de FAQ, de document, de blog, forum, etc.)
- administration via une interface web

Le serveur d'applications Zope

Zope est un serveur d'application web, orienté objet, libre et écrit dans le langage de programmation Python (avec quelques parties en C pour des raisons de performances).

Il peut être entièrement géré à partir d'une interface Web (la ZMI, Zope Management Interface).

Zope publie des objets Python enregistrés dans une base de données objet, la ZODB.

Des types d'objets basiques, tels que des documents, des images, des patrons (templates) de page, sont à la disposition des utilisateurs pour être créés et gérés via l'internet. Des types d'objets spécialisés, tels que les wikis, les blogs, les galeries de photos, sont disponibles.

La structure de Zope est hiérarchique et composée d'objets contenant eux-mêmes des objets contenant eux-mêmes.. à l'infini.

Ces objets se caractérisent par un état défini comme l'ensemble des valeurs de ses attributs et par un comportement décrit sous la forme de méthodes qui lui sont applicables.

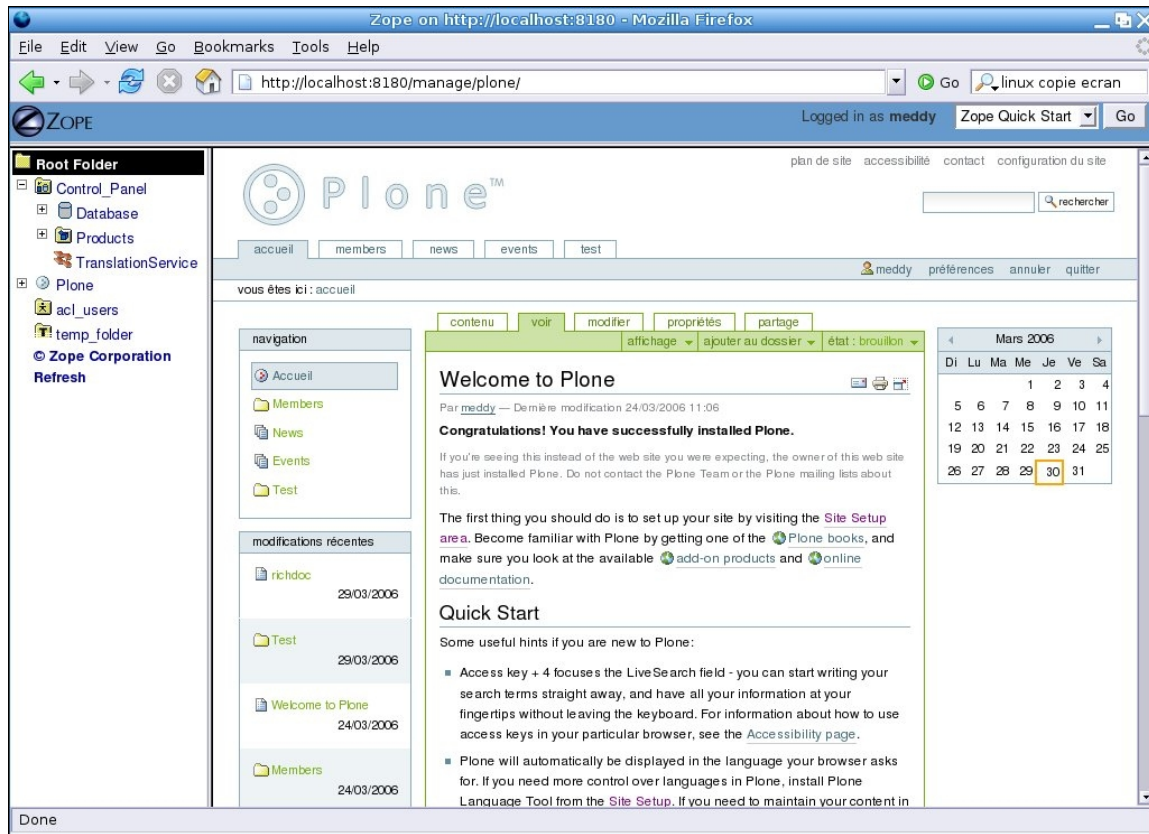
Chaque objet est identifié par un identifiant (id) unique qui est utilisé pour l'indexation en ZODB.

CMF (Content Management Framework)

CMF est une plateforme de gestion de contenu fournissant aux développeurs les outils pour construire des CMSs complexes (comme Plone par exemple).

CMF repose sur Zope et Plone est la couche supérieure de CMF.

A quoi ressemble Plone ?



Ici Plone est affiché via la ZMI (Zope Manager Interface)

2.1. Structure d'un site

Présentation

Un site Plone peut-être décomposé ainsi :

- un ensemble de skins
- un moteur de workflow
- un ensemble de types de contenus
- un ensemble de contenus

Plone est basé sur une architecture modulaire.

CMF a introduit le concept des outils (ou tools). Il en fournit d'ailleurs un grand nombre. Plone s'appuie sur ces outils et en a introduit d'autres pour simplifier certains concepts.

Voici une liste non exhaustive des outils existant sous Plone.

Portal_syndication

La syndication, c'est-à-dire le fait d'exporter le contenu dans le format RSS, est contrôlée par l'outil `portal_syndication`. Elle est utile lorsque différents sites souhaitent échanger leurs contenus.

Portal_url

L'outil `portal_url` est un outil mis à disposition des développeurs. Il permet de connaître l'url de l'instance Plone et des objets contenus. Il ne dispose pas d'interface utilisateur.

Portal_catalog

L'outil `portal_catalog` est la version CMF du Zcatalog prête à l'emploi. Le vocabulaire standard est utilisé ; les index et les méta données ont été définis pour les types de CMFDefault.

Pour rappel, les index sont les champs stockés dans le ZCatalog pour l'indexation et les méta données sont les données qui peuvent être utilisées lors de l'affichage du résultat.

Portal_discussion

L'outil `portal_discussion` est responsable de la gestion des commentaires que l'on peut associer à un document. Il définit une action *reply* qui est soumise à une condition : l'outil doit autoriser les commentaires. Cette autorisation est définie dans les types définis dans `portal_types` dans le champ *allow Discussion*.

Portal_calendar

Cet outil permet de paramétrer le calendrier fourni avec Plone. Par défaut, ce dernier n'affiche que les objets de type *news item*, mais il est possible d'afficher les différents types d'objets. Il suffit d'utiliser l'onglet *properties* pour sélectionner les objets qui doivent apparaître dans le calendrier.

Portal_membership

L'outil `portal_membership` permet de contrôler l'authentification. Il permet également de contrôler la création des répertoires pour les membres. De plus, il définit les actions utilisateurs. Les actions sont détaillées dans la section *portal_actions*. L'outil *portal_registration* complète cet outil en gérant l'enregistrement des utilisateurs.

Portal_memberdata

L'outil `portal_memberdata` est responsable des données propres à l'utilisateur. L'onglet *properties* permet de visualiser les propriétés disponibles pour les utilisateurs. Enfin, l'onglet *content* permet de connaître le nombre d'utilisateurs de Plone.

Portal_metadata

Ce composant permet de gérer les méta données des types présents dans Plone.

Portal_undo

L'outil `portal_undo` permet de gérer la fonctionnalité *undo* de la ZODB. Cette fonctionnalité permet d'annuler des modifications. Elle est basée sur un système d'historique. L'outil définit deux actions :

- *undo* : cette action globale permet de lister les undo disponibles. Pour visualiser les undo disponibles, il suffit de visualiser l'url `http://server/site/undo_form`.
- *quick_undo* : cette action permet aux membres d'effectuer un *undo* sur un objet.

Les actions sont détaillées dans la section `portal_actions`.

Portal_properties

Cet outil permet de gérer des feuilles de propriétés pour l'instance Plone. Il en existe deux :

- *navtree_properties* : cette feuille permet de contrôler le slot de navigation. Il est ainsi possible de supprimer certains types de l'affichage, de modifier l'ordre de cet affichage etc..
- *site_properties* : cette feuille contient les propriétés du site. Elles permettent de définir la langue par défaut, le format de la date, le charset par défaut, les éditeurs WYSIWIG, et l'autorisation de création de métadonnées.

2.2. Types de contenu

Qu'est ce que c'est ?

Concrètement, un type de contenu correspond à un certain schéma de données, c'est-à-dire un ensemble de champs à remplir dans un formulaire.

L'intérêt de spécifier des types de contenu spécifiques est de bien catégoriser le contenu du site, de pouvoir effectuer des recherches précises sur chacun des champs qui composent un type de contenu.

Plone fournit en standard les types de contenu suivants :

- Dossier (Folder) : peut contenir n'importe lequel des types suivants
- Document (Page) : texte simple, HTML ou STX (Structured Text)
- Événement (Event) : définit un événement
- Fichier (File) : contenu externe
- Lien (Link) : hyper lien autonome
- Image (Image): tout type graphique standard (JPEG, GIF, PNG)
- Actualité (News Item) : document daté
- Dossier automatique (Smart Folder) : dossier virtuel qui effectue une recherche sur critères

Grâce au produit (add-on) Archetypes, il est possible de facilement définir d'autres types de contenu (voir plus loin dans cette documentation).

En pratique ça donne quoi ?



On peut voir ici que l'ajout d'un nouveau type de contenu se fait aisément via un menu *ajouter au dossier* (*add to folder*). Dans cet exemple, le type *article* n'est pas un type de contenu de base. C'est un type qui a été ajouté par l'intermédiaire d'un produit (Le produit **RichDocument** dont nous aurons l'occasion de reparler).

2.3. Workflows

Définition

Wikipédia : [...] le workflow décrit le circuit de validation, les tâches à accomplir entre les différents acteurs d'un processus, les délais, les modes de validation, et fournit à chacun des acteurs les informations nécessaires pour la réalisation de sa tâche. Pour un processus de publication en ligne par exemple, il s'agit de la modélisation des tâches de l'ensemble de la chaîne éditoriale.

Les workflows dans Plone

Plone intègre un moteur de workflow basé sur des états. Un exemple de ce type de workflow est un circuit de publication d'un document dans un intranet :

1. Un membre de l'intranet rédige un document et le donne à valider à un responsable.

Pendant qu'il rédige ce document, le membre peut choisir de ne pas le rendre visible aux autres utilisateurs du site, afin de pouvoir le modifier sans que personne ne puisse prendre en compte des informations non définitives qui pourraient y être contenues.

2. Le responsable, lorsqu'il se connecte au site, voit apparaître ce document dans une liste de documents à valider.

Il peut publier le document s'il le juge acceptable, ou le rejeter (s'il ne correspond pas à la ligne éditoriale, par exemple, ou si, plus généralement, il doit être corrigé) pour que le membre le modifie.

Le moteur de workflow peut servir à définir des workflows simples ou complexes, et permet de restreindre les actions ("demander la publication", "publier", "rejeter", etc.) possibles pour chaque état ("privé", "en cours de validation", "public").

L'accès aux contenus peut être restreint d'après leur état : par exemple, seul l'auteur d'un contenu pourra accéder à un document qu'il a créé et déclaré privé.

De même, dans un site internet, seuls les contenus "validés" seraient accessibles à l'internaute non authentifié.

2.4. Introduction au catalogue

Le ZCatalog fournit de puissants moyens d'indexation et de recherche de contenu, à partir la ZMI. Un ZCatalog est un objet Zope pouvant être placé à l'intérieur d'un dossier, géré via le web, et pouvant être étendu de plusieurs façons.

Voici entre autres les propriétés du ZCatalog :

- Les recherches sont rapides et consomment peu de mémoire grâce aux structures de données utilisées.
- Les recherches sont puissantes. Le ZCatalog supporte les booléens, les synonymes et les caractères jokers.
- L'indexation est flexible.
- L'utilisation peut se faire en dehors de Zope. Le ZCatalog peut être utilisé dans n'importe quel script Python.
- Si quelque chose se passe mal après l'indexation d'un contenu, l'index est restauré à son état précédent. De plus un ZCatalog peut être altéré en privé ce qui signifie que personne d'autre ne peut

voir les changements effectués sur l'index.

- Une recherche renvoyant un très grand nombre de concordances ne retournera pas un grand ensemble de résultats. Seule une partie des résultats sera renvoyée.

2.5. Présentation du mécanisme de skins

Introduction aux skins

C'est le rôle des skins que d'habiller le site; chaque skin est un ensemble cohérent de briques fonctionnelles élémentaires appelées des layers (couches) ou sous-skins.

Il peut y avoir, pour un site donné, plusieurs skins et, dans ce cas, l'utilisateur peut choisir celui qui lui convient.

Il revient à l'administrateur de choisir celui par défaut depuis les *Propriétés* de */Plone/portal_skins*.

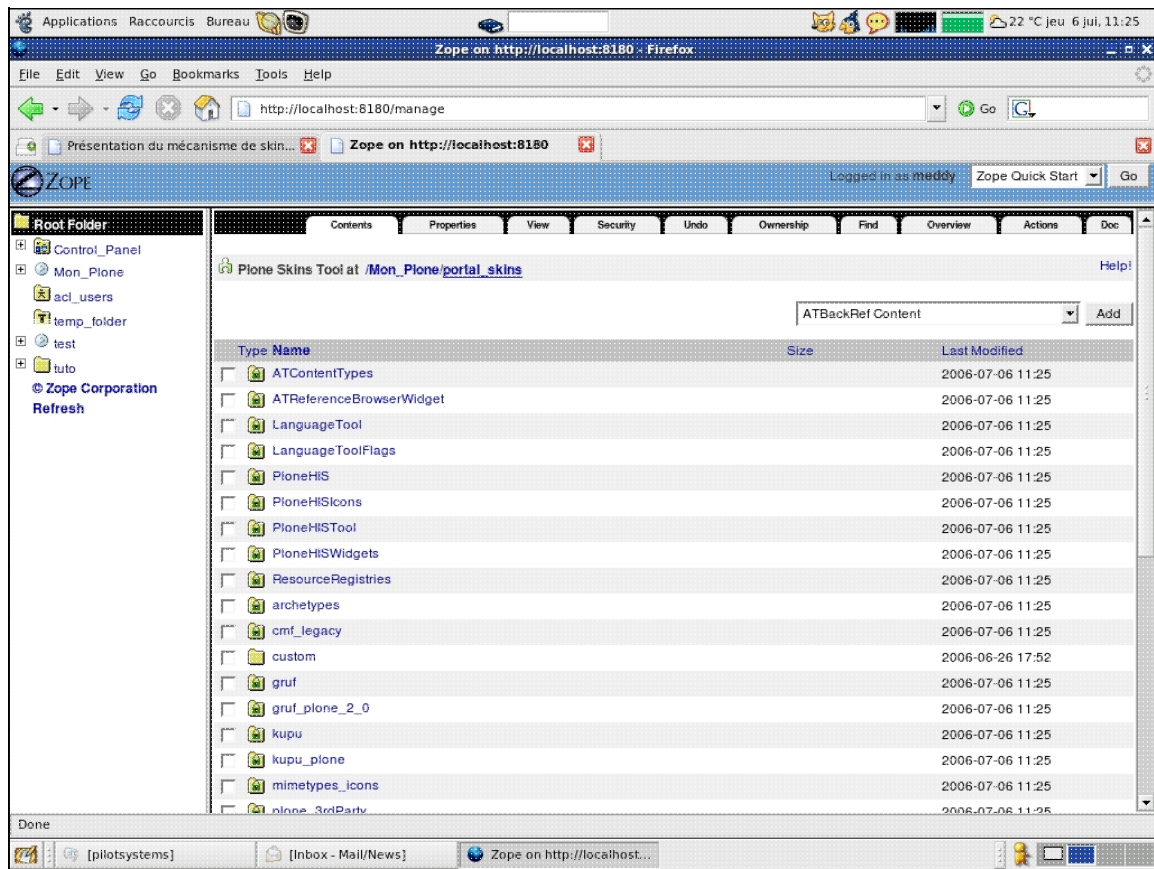
Dans ces mêmes *Propriétés*, l'administrateur peut créer de nouveaux skins en détaillant les layers qui les composent. Les layers génériques de Plone sont *plone_content*, *plone_styles*, *plone_scripts* et *plone_templates* complétés par le dossier contenant les images du site *plone_images*.

Il existe deux types de layers : les objets standards de type *File System Directory View* et ceux de type *Folder* créés pour personnaliser le site tels que *Custom*.

L'ordre dans lequel sont classées les layers est important : images, scripts et templates de *Custom* sont prioritaires sur les objets de mêmes noms mais contenus dans les layers placés plus bas dans la liste.

Altération d'un skin

- Ajout d'un nouveau dossier dans le chemin de recherche du skin.
- Personnalisation du layer dans ce nouveau dossier
- La recherche prend en compte le premier layer trouvé : le layer personnalisé
- Nom de dossier pour les personnalisations : *Custom*



3. Page Template dans Plone

En quoi ça consiste ?

Lorsqu'on veut adapter un site Plone à ses propres besoins, il est très utile de savoir modifier les scripts qui le composent. Ceux-ci sont principalement de deux types :

- Les scripts python
- Les Zope Page Templates (ZPT).

ZPT est un outil de génération de pages web qui s'appuie sur le langage TAL (Template Attribute Language).

C'est ce système qui est utilisé dans Plone pour créer des pages au format HTML.

ZPT présente l'avantage de mieux séparer traitement et présentation que ne le fait DTML (Document Template Markup Language). DTML est plus ancien que ZPT et tend à devenir obsolète. Il ne sera pas abordé dans ce support, à part dans la partie SGBD où il est encore utilisé.

Le langage TAL est simple et réduit à quelques expressions. Un avantage majeur est que le code est conforme XML-XHTML.

Concrètement, ZPT est constitué de 4 composants :

- XML, langage de description de données.
- TAL, Template Attribute Language
- TALEX, TAL Expression Syntax
- METAL , Macro Expansion for TAL

TAL et METAL permettent de définir des attributs pour XML tandis que TALEX permet de définir des valeurs d'attributs TAL.

METAL permet de définir un bloc de code (macro) qui pourra être appelé ailleurs dans la même page ou depuis d'autres pages. On retrouve un peu le même principe que le *define* de PHP.

Quelques exemples

TAL :

Le code suivant :

```
<p tal:content="exemple"></p>
```

signifie: affiche la valeur de la variable "exemple" entre les tags <p> et </p>

L'expression entre guillemets doit obéir à la syntaxe TALES. Ici elle est très simple, c'est juste un nom de variable.

TAL s'occupe avant tout de la présentation, il est donc (volontairement) très limité.

il ne comporte que 8 commandes : *define*, *condition*, *repeat*, *content*, *replace*, *attributes*, *omit-tag*, *on-error*

TALES :

Le code suivant :

```
<tal:content="string:Hello World"></p>
```

signifie: affiche *Hello World* entre les tags <p> et </p>

Ici, *string* est un type d'expression TALES. Il en existe 6 : *string*, *path*, *python*, *recall*, *not*, *exists*

METAL :

Le code suivant :

```
<metal:header define-macro="header">
Titre, image etc.. constant dans tout le site
</metal:header>
```

signifie: définir une macro appelée *header* contenant le haut d'une page.

Le code suivant :

```
<metal:header use-macro="here/macro_header/macros/header">
texte
</metal:header>
Le contenu de la page...
```

signifie: remplacer *texte* par le contenu de la macro *header*.

3.1. Rappels/Introduction à TAL et METAL

Le langage TAL

C'est le langage qu'on utilise le plus lorsqu'on écrit des ZPT. Ce langage vient se greffer dans les tags XML (ou HTML) sous la forme d'attributs.

Ses attributs sont de la forme : *tal:nom="valeur"*

TALES est le langage de définition des valeurs d'attributs TAL.

Exemple :

```
<div tal:content="here/objectIds" />
```

tal:content : attribut TAL

here/objectIds : expression TALES

Les Commandes TAL

Voici les différentes commandes TAL et leurs fonctions respectives :

- *define* définit une variable
- *condition* soumet le tag et ce qu'il contient à une condition
- *repeat* répète le tag et ce qu'il contient
- *content* affiche l'expression (en conservant le tag html)
- *replace* affiche l'expression (en remplaçant le tag html)
- *attributes* permet de redéfinir un attribut html (p.ex href dans un lien)
- *omit-tag* omet le tag si une condition est vérifiée
- *on-error* appelle une expression lorsqu'une erreur se produit

Il est heureusement possible d'inclure plusieurs commandes dans un même tag.

L'ordre dans lequel vous écrivez les attributs TAL dans un même tag n'affecte pas l'ordre dans lequel ils sont exécutés. En effet, quelque soit l'ordre de vos déclarations, ils seront **toujours** effectués dans cet ordre :

1. *define*
2. *condition*
3. *repeat*
4. *content/replace*
5. *attributes*

Exemples TAL

Remplacement :

Attribut : *tal:replace*

Exemple :

```
<span tal:replace="here/Creator">
  Premier nom
</span>
```

Affiche :

John Smith

Omission conditionnelle :

Attribut : *tal:condition*

Exemple :

```
<a href="edit" tal:condition="not:isAnon">
  Lien vers le formulaire d'édition
</a>
```

Affiche :

Affiche le lien *Lien vers le formulaire d'édition* seulement dans le cas où l'utilisateur n'est pas anonyme.

Edition d'attributs:

Attribut : *tal:attributes*

Exemple :

```
<a tal:attributes="href portal/absolute_url;
  title portal/Description">
  Lien vers la racine du portail
</a>
```

donne :

```
<a href="http://..." title="...">Lien vers la racine du portail</a>
```

Répétition :

Attribut : *tal:repeat*

Exemple :

```
<li tal:repeat="elem python:[1,2,3]">
Numéro &lt;span tal:replace="elem" />
</li>
```

Affiche :

```
<li>Numéro 1</li>
<li>Numéro 2</li>
<li>Numéro 3</li>
```

Exemples TAL courants

Création d'un lien vers un objet :

```
<a tal:attributes="href obj/absolute_url; title obj/Description">
<img tal:attributes="src obj/getIcon" />
<span tal:replace="obj/pretty_title_or_id" />
</a>
```

Insertion d'une image-objet :

```
<img tal:replace="structure obj/tag" />
```

Les expressions TALES

Il existe 3 types différents d'expression TALES pour les ZPT.

Path Expressions :

path: préfixe un chemin (path) vers la propriété d'un objet Zope sous la forme: */objet/propriété*.

Ainsi, *path:unepage/title* est un chemin vers la propriété *title* de l'objet *unepage*.

En interprétant cette expression, Zope va remplacer *unepage/title* par la valeur *title* de *unepage*.

Remarque: *path* est le type TALES par défaut, ce préfixe peut donc être omis.

Il existe un certain nombre de noms prédéfinis pour les ZPT et qui permettent d'appeler différents objets. Citons entre autres :

- *here* correspond au contexte d'acquisition. Exemple: *here/title* affiche le titre de la page.
- *context* est équivalent à *here*.
- *template* correspond à l'objet ZPT lui-même.
- *request* correspond à l'objet requête Zserver. On l'utilise pour récupérer des variables passées dans la requête. Exemple *request/param*
- *user* correspond à l'utilisateur authentifié.
- *root* correspond à l'objet racine de Zope.

String Expressions :

Ce type préfixe une chaîne de caractères. Exemple: *string:hello world*

Il est possible d'inclure des variables en ajoutant un \$ devant la variable.

Exemple: *string:cette page a pour titre \${here/title}*

Python Expressions :

python: évalue l'expression python et en retourne la valeur.

Exemple: *python:here.objet.methode()*

On peut également l'utiliser pour accéder aux propriétés d'un objet ou afficher des chaînes de caractères.

Exemples:

- *python:page.title* est équivalent à *path:page/title*
- *python:'Hello '+nom* est équivalent à *string:Hello \${nom}* (où nom est type string).

Exemples TALEs courants

Ces 3 exemples affichent la même chose :

```
<span tal:content="here/title_and_id"/>
<span tal:content="string:${here/title} (${here/id})"/>
<span tal:content="python:%s (%s) %(here.Title(),here.getId())"/>
```

Variables de Plone

On peut accéder aux variables définies par Plone via la ZMI :

```
portal_skins > plone_templates > global_defines
```

Elles permettent d'obtenir un accès rapide aux informations et aux outils.

Quelques variables Plone:

- *isAnon*
- *member*
- *portal*
- *utool*
- *object_title*
- *here_url*
- *workflow_actions*

Exemple d'utilisation des variables

Cet exemple présente l'utilisation des variables dans un document :

```
<html>
<body>
  Voici le template
  <span tal:replace="template/title_or_id"/>
  Votre navigateur est :
  <pre tal:content="request/HTTP_USER_AGENT" />
```

3.2. Considérations d'internationalisation

Présentation

L'internationalisation (i18n) est un des principes clés de Plone. i18n permet de traduire son site plone en plusieurs langues en insérant des attributs TAL ou METAL.

Il est possible de traduire à la fois les labels des différents types de contenu mais aussi leurs champs, ainsi que les formulaires d'édition.

Il existe 5 attributs spécifiques pour i18n qui sont :

- *i18n:translate* pour le contenu
- *i18n:attributes* pour les attributs
- *i18n:name* pour les traduction paramétrées
- *i18n:domain* pour définir le domaine de traduction
- *i18n:data* pour tout ce qui n'est pas une chaîne de caractères (exemple un format de date).

Il en existe un 6ème, *i18n:source* mais qui n'est pas utilisé dans Plone.

Exemple

Considérons le template suivant :

```
<html>
<body>
<p>Welcome to Plone.</p>

<p>There have been over
<span tal:content="here/download_count">100,000</span>
downloads of Plone.
</p>
<p>Please visit
<a href="about">About Plone</a>
for more information.
</p>
</html>
```

Il y a ici plusieurs éléments à traduire:

- "Welcome to Plone" qui peut-être directement traduit dans un autre langage.
- Le texte "Plone Icon" de l'attribut `alt` du tag `img`.
- La phrase "There have been over 100,000 downloads of Plone.". Ici, on veut que le nombre soit toujours calculé dynamiquement. De plus, en fonction de la langue, la structure de la phrase peut changer. En effet, le nombre peut être placé en début, en fin ou à un autre endroit de la phrase.
- La phrase "Please visit About Plone for more information.". Ici, on veut que "About Plone" reste un lien, et qu'il puisse être traduit à part. De plus, comme précédemment, la position du lien dans la phrase peut-être différent selon les langages.

Cas 1 : "Welcome to Plone"

Dans ce cas, on veut signifier à l'équipe de traduction que la traduction à effectuer est une traduction simple, sans texte à traiter spécialement ou qui doit être calculé dynamiquement.

Pour faire cela, on utilise l'attribut *i18n:translate* avec la valeur "XXX" à l'intérieur du tag <p>

Exemple :

```
<p i18n:translate="XXX">Welcome to Plone.</p>
```

L'équipe de traduction choisira un identifiant de message qui remplacera "XXX" pour identifier cette expression, et qui deviendra la nouvelle valeur de l'attribut *i18n:translate*.

Cas 2 : "Le texte alt de l'image"

Dans ce cas, on a besoin de faire savoir à l'équipe de traduction que c'est la valeur d'un attribut et non le contenu du tag qui doit être traduit.

Pour faire ça, on utilise l'attribut *i18n:attributes*.

Cet attribut liste tous les attributs du tag qui contiennent du texte devant être traduit. S'il y a plus d'un attribut, vous pouvez les séparer avec un espace.

Exemple :

```

```

Exemple avec 2 attributs :

```

```

Cas 3 : "Le contenu dynamique"

La partie qui nous intéresse ici est :

```
<p>There have been over
<span tal:content="here/download_count">100,000</span>
downloads of Plone.
</p>
</pre>
```

Ici on va utiliser l'attribut *i18n:name*. Cela nous donne un moyen d'identifier la partie d'un long morceau qui doit être traduite.

Par exemple, on veut que les traducteurs voient notre texte comme cela:

There have been \${count} downloads of Plone.

Ainsi ils peuvent la traduire en déplaçant le compteur n'importe où dans la phrase, en fonction des besoins.

En utilisant *i18n:name*, on donne un nom à la partie calculée dynamiquement :

```

<pre>
<p i18n:translate="">There have been over
<span tal:content="here/download_count "
i18n:name="count">100,000</span>
downloads of Plone.
</p>
</pre>

```

On remarque que *name* n'a pas besoin d'avoir de relation avec l'expression TAL qui le calcule. De plus, *name* doit être unique à l'intérieur d'un tag.

Un exemple avec plusieurs *i18n:name* :

```

<p i18n:translate="">My name is
<span tal:content="here/first" i18n:name="first">first</span>
<span tal:content="here/last" i18n:name="last">last</span>
</p>
</pre>

```

Cas 4 : "Combinaison des concepts"

Pour ce cas , on va utiliser un mélange des idées vues dans les cas précédents.

On a :

```

<p>Please visit
<a href="about">About Plone</a>
for more information.
</p>

```

Nous voulons que les traducteurs voient :

Please visit $\{about-plone\}$ for more information.

Mais, au contraire du contenu dynamique du cas précédent, on veut que la phrase interne *About Plone*, soit aussi traduite.

Pour ce faire, on insère un attribut *i18n:name* autour du tag, et on insère un attribut *i18n:translate* à l'intérieur même du tag. Ainsi, les traducteurs verront que ce tag doit être traduit.

On a donc :

```
<p i18n:translate="">Please visit
<span i18n:name="about-plone">
<a href="about" i18n:translate="">
About Plone</a>
</span>
for more information.
</p>
```

Dans le cas où on a un attribut qui doit être traduit dans le tag link (par exemple *"title"*) on procède ainsi :

```
<p i18n:translate="">Please visit
<span i18n:name="about-plone">
<a href="about" i18n:translate=""
i18n:attributes="title" title="Go to About Page">
About Plone</a>
</span>
for more information.
</p>
</pre>
```

3.3. Création d'un formulaire Plone 2

Problématique

En Zope classique, l'action d'un formulaire charge un script python ou un autre formulaire.

Ce mode de fonctionnement peut poser problème dans les cas suivants :

- lorsqu'il y a plusieurs boutons d'actions
- lorsqu'il y a une erreur au traitement du formulaire
- lorsqu'une validation des champs est nécessaire

En effet, comment revenir au formulaire initial ? Comment gérer différentes actions en fonction du bouton ?

Le produit CMFFormController

CMFFormController est un produit d'extension Zope pour le CMF.

CMFFormController aide les développeurs en simplifiant le processus de validation des formulaires.

il fournit les types suivants :

- Controller Page Template
- Controller Python Script
- Controller Validator
- Controller State

Formulaires

Voici le code d'un formulaire basique utilisant CMFFormController :

```
<form tal:define="errors options/state/getErrors"
      tal:attributes="action string:${here/absolute_url}/${template/id};"
      method="post">
  <input type="hidden" name="form.submitted" value="1" />
  <p tal:define="err errors/foo|nothing" tal:condition="err"
      tal:content="err" />
  <input type="text"
        name="foo"
        tal:define="val request/foo|nothing"
        tal:attributes="value val" />
  <input type="submit" name="submit" value="submit" />
</form>
```

On remarque les choses suivantes :

- Le formulaire est soumis à lui-même. Tous les formulaires doivent se comporter ainsi.
- Une variable cachée (hidden) spéciale est présente : *form.submitted*. La page template vérifie l'objet REQUEST pour *form.submitted* pour voir si le formulaire a été soumis ou si on vient d'y accéder via un lien par exemple. Les formulaires doivent toujours contenir cette variable *form.submitted*.
- Au début du code du formulaire, la variable *errors* est définie. Elle permet entre autre au validateur d'effectuer une action particulière en fonction du type de l'erreur.

Avant d'utiliser ce formulaire il est nécessaire de spécifier les validateurs utilisés pour vérifier les valeurs du formulaire. Il est également nécessaire de spécifier l'action effectuée après la validation.

Validateurs

Il existe 2 manière pour spécifier des validateurs de formulaire :

1. Dans le système de fichiers.
2. Directement en ZMI

Spécifier les validateurs dans le système de fichiers

On peut spécifier des validateurs dans le système de fichiers en utilisant un fichier de propriétés possédant l'extension *.metadata*.

Pour créer un fichier *.metadata*, il suffit de créer un fichier possédant le même nom que la page template en question et de rajouter l'extension *.metadata*.

Si on a un Controller Page Template appelé *document_edit_form.cpt* alors les propriétés de ce fichier devront être stockées dans le fichier *document_edit_form.cpt.metadata*.

Dans le fichier *.metadata* la section concernant les validateurs aura la forme suivante :

```
[validators]
validators = validate_script1, validate_script2
```

Les scripts de validation *validate_script1* et *validate_script2* seront appelés dans cet ordre.

Si on souhaite que différents validateurs soient appelés en fonction du contexte du formulaire, on peut procéder ainsi :

```
[validators]
validators = validate_script1
validators.Document = validate_script2
```

Dans cet exemple, si le contexte est un objet de type *Document*, *validate_script2* sera appelé. Dans tout autre cas, c'est *validate_script1* qui sera appelé.

Supposons maintenant qu'on ait 2 boutons différents dans le formulaire, et qu'on veuille des séquences de validation différentes en fonction du bouton pressé. On peut procéder de la manière suivante :

```
<input type="submit"
  name="form.button.button1"
  value="First Button" />
<input type="submit"
  name="form.button.button2"
  value="Second Button" />
```

On spécifie ensuite les validateurs pour *button1* et *button2* dans le fichier *.metadata* :

```
[validators]
validators..button1 = validate_script1, validate_script3
validators..button2 = validate_script2, validate_script4
```

Spécifier les validateurs en ZMI

En ZMI, Un *Controller Page Template* possède entre autre l'onglet *Validation*.

L'onglet *Validation* affiche tous les validateurs pour la page template en question.

Add a New Form / Script Validator Override

Template test_cpt

Context type

Button

Validators

On peut spécifier ici des validateurs avec le même niveau de spécialisation que précédemment via un formulaire web.

Les informations concernant les validateurs de tous les formulaires sont stockées dans l'outil *portal_form_controller* de Plone.

L'outil *portal_form_controller* possède des méthodes qu'on peut utiliser pour spécifier les validateurs d'un Controller Page Template donné :

```
portal_form_controller.addFormValidators(id,  
                                         context_type,  
                                         button,  
                                         validators)
```

Ici, *id* est l'Id du Controller Page Template, *context_type* est le nom de la classe pour la classe de l'objet context, *button* est le nom du bouton pressé et *validators* est une liste de chaînes de caractères.

Si on souhaite qu'un validateur s'applique à toutes les classes, on met la variable *context_type* à *None*. Idem pour les boutons.

Actions

La suite de validateurs qui est exécutée retourne un statut dans l'objet *state*. Le statut par défaut est *success*, c'est à dire que si aucun validateur n'est exécuté, le statut sera *success*. Si un validateur rencontre une erreur, le statut sera *failure*.

Il faut maintenant spécifier ce qui doit être fait lorsqu'un statut donné est renvoyé.

Comme pour les validateurs, il existe 2 moyens pour spécifier des actions de formulaire :

1. Dans le système de fichiers.
2. Directement en ZMI

Spécifier les actions dans le système de fichiers

Les actions sont stockées dans le même fichier *.metadata* que les validateurs. La syntaxe de la section actions est la suivante :

```
[actions]
action.success = traverse_to:string:script1
```

Ici, lorsque le formulaire est soumis et que les scripts de validation ont retournés le statut *success*, *traverse_to_action* est appelé avec l'argument *string:script1*. En d'autres termes, si les données du formulaire sont valides, le script *script1* sera exécuté.

Il aurait aussi été possible de procéder de la façon suivante :

```
action.success = redirect_to:string:http://my_url_here
```

Dans ce cas, le navigateur redirigera la page vers *http://my_url_here*.

Il existe 4 types de transitions :

- *traverse_to*
- *traverse_to_action*
- *redirect_to*
- *redirect_to_action*

Les 2 premières permettent un transfert de la variable d'état, les 2 suivantes une redirection par le navigateur mais avec perte de la variable d'état.

L'action par défaut du statut *failure* est de recharger le formulaire. Le formulaire aura alors accès à tous les messages d'erreurs via l'objet *state*.

Supposons que l'on veuille différentes actions en fonction du contexte du formulaire. On peut procéder ainsi :

```
[actions]
action.success = traverse_to:string:script1
action.success.Document = traverse_to:string:document_script
```

Ici, si le contexte est un objet de type *Document*, *document_script* sera exécuté dès que la validation aura le statut *success*. Dans tous les autres cas, *script1* sera exécuté.

Supposons maintenant que l'on ait 2 boutons différents dans notre formulaire et que l'on souhaite que des actions différentes aient lieu en fonction du bouton pressé. On procède ainsi :

```
<input type="submit"
  name="form.button.button1"
  value="First Button" />
<input type="submit"
  name="form.button.button2"
  value="Second Button" />
```

Ensuite, on spécifie les actions pour *button1* et *button2* :

```
[actions]
action.success..button1 = traverse_to:string:script1
action.success..button2 = traverse_to:string:script2
```

Spécifier les actions en ZMI

En ZMI, Un *Controller Page Template* possède entre autre l'onglet *Actions*.

L'onglet *Actions* affiche tous les validateurs pour la page template en question.

On peut spécifier ici des actions avec le même niveau de spécialisation que précédemment via un formulaire web.

Les informations concernant les actions de tous les formulaires sont stockées dans l'outil *portal_form_controller* de Plone.

Comme pour les validations, l'outil *portal_form_controller* possède des méthodes permettant de spécifier des actions :

```
portal_form_controller.addAction(id,  
                                status,  
                                context_type,  
                                button,  
                                action_type,  
                                args)
```

action_type est le type d'action qui aura lieu et *args* est une chaîne de caractères (typiquement, une expression TALES) qui sera passée en argument de l'action.

3.4. Présentation du gabarit et des macros standards

L'outil `portal_skins`

La partie graphique d'un site Plone est gérée par l'outil `portal_skins`. Ce dernier permet aux utilisateurs du site de choisir son aspect graphique.

Cet outil permet de séparer comme il se doit, l'apparence et le contenu du site.

Le paramétrage de l'outil `portal_skins` est accessible via la ZMI.

L'onglet Contents

Cet onglet permet d'accéder aux différents répertoires contenant les éléments graphiques du site.

On trouve en particulier :

- `plone_images` : les images et les icônes Plone
- `plone_templates` : les zpt décrivant l'architecture des pages de Plone
- `plone_scripts` : Les scripts Python utilisés dans les ZPT
- `plone_styles` : les feuilles de styles de Plone
- `plone_content` : les pages ZPT permettant la gestion des différents types
- `plone_forms` : les différents formulaires de Plone
- `plone_ecmascript` : les scripts javascripts
- `plone_3rdParty` : les autres skins de produits externes
- `plone_wysiwyg` : le modèle pour l'éditeur WYSIWYG (what you see is what you get)
- `custom` : ce répertoire est extrêmement utilisé lors de l'adaptation d'un modèle. Il est placé comme premier répertoire lors de l'acquisition. Ainsi, si un logo nommé `logo.png` ne vous satisfait pas, il suffit de placer dans ce répertoire un autre logo avec le même nom.

L'onglet Properties

Cet onglet permet de définir les modèles disponibles aux utilisateurs. Les modèles sont définis par un ensemble de répertoires fournissant un modèle d'acquisition.

Quand une page Plone recherche un objet, elle réalise l'acquisition normalement (c'est à dire dans le répertoire courant puis dans le répertoire père) puis recherche dans les chemins proposés dans le modèle.

De plus, cet onglet permet de définir un certain nombre de paramètres du site comme :

- le skin par défaut (*Default skin*)
- le nom du cookie chargé de l'aspect du site (*Request variable name*)
- la possibilité de choisir un skin (*Skin flexibility*)
- la durée de vie du cookie (*Skin persistence*)

enfin il est possible de créer ou supprimer un ou plusieurs skins, un skin étant composé d'un ensemble de chemins d'acquisition.

Le répertoire *plone_styles*

Les feuilles de styles (CSS) sont une partie essentielle dans la définition d'un modèle Plone.

Le répertoire *plone_styles* contient différents répertoires correspondant aux différents modèles fournis avec Plone.

Ces répertoires contiennent tous un objet appelé *stylesheet_properties*. Cet objet stocke un certain nombre de variables et leurs valeurs associées.

Ces variables sont utilisées par la feuille de style de Plone : *plone.css*

Il est ainsi aisé aux administrateurs de définir facilement de nouvelles feuilles de styles.

L'objet *stylesheet_properties*

Il définit les variables utilisées par la feuille de style de Plone.

Pour modifier les valeurs de la feuille de style, il faut utiliser le bouton *customize* de cet objet.

Cette action a pour effet de créer dans le répertoire custom un répertoire portant le même nom, et d'y recopier ses valeurs dans les propriétés du répertoire.

Pour définir une valeur, il suffit de sélectionner le répertoire puis l'onglet *Properties* et enfin de spécifier les valeurs.

3.5. Structure d'une page

Présentation

Nous allons aborder ici la structure générale des pages d'un site plone.

Visuellement, lorsqu'on affiche une page on distingue facilement 4 parties :

1. la partie du haut (l'entête ou header)
2. les côtés
3. la partie centrale
4. la zone du bas (le pied de page ou footer)

L'entête

L'entête contient la plupart du temps un logo, un petit formulaire de recherche, des renseignements utiles et des onglets de navigation.

En cliquant sur le logo, on retourne toujours sur la page d'accueil du site.

Pour faire une recherche, il suffit d'entrer des mots à rechercher puis de valider. Le résultat de votre recherche sera alors immédiatement visible dans la partie centrale.

Les onglets de navigation permettent de se diriger directement dans les différentes sections du site. (Présentation, Accueil, etc.)

Les barres latérales

Elles contiennent plusieurs boîtes à outils (appelées portlets)

Ce qui est visible dans les menus de gauche et de droite dépend bien évidemment du site et de sa vocation. Tout est bien sûr configurable

Par exemple on peut y trouver :

- un calendrier
- un portlet de navigation
- un portlet affichant les modifications récentes

La partie centrale

C'est la partie qui affiche le contenu du document que vous visualisez.

C'est ici aussi qu'apparaîtront les informations de certains dossiers où l'utilisateur a le droit de contribuer. Dans ce cas précis, de nouveaux onglets apparaissent : de nouvelles actions sont possibles (il faut bien sûr être authentifié pour y avoir accès).

Ces onglets donnent accès à certaines fonctionnalités telles que changer l'état d'un document ou le modifier.

En fonction du document et des autorisations qui sont attribuées à l'utilisateur, certaines fonctionnalités directement liées à ce document (les actions rapides) apparaissent en haut à droite de la partie centrale.

De gauche à droite, ces actions permettent de :

- afficher le flux RSS correspondant
- envoyer à quelqu'un le lien de la page par e-mail
- imprimer la page
- basculer en mode plein écran

Le pied de page

Il est principalement esthétique. En fonction du type de site, il est pratique d'y trouver des liens pour retourner en haut de la page ou consulter le plan du site.

4. Développement de nouveaux mécanismes

Plone permet d'effectuer un grand nombre de configurations, de modifications et d'ajouts de fonctionnalités et ce, dans pratiquement tous les domaines de la gestion de contenu.

Plone peut par exemple s'interfacier facilement avec un grand nombre de SGBDR (systèmes de gestion de base de données).

Plone permet de rechercher facilement l'ensemble des contenus d'un site, et de configurer finement ces recherches. Pour cela, le catalogue est à disposition, ainsi que les smart folders (dossiers automatiques).

L'apparence d'un site est également hautement configurable, notamment grâce au concept de portlets. Il est possible de créer de nouveaux portlets et de les disposer comme bon nous semble.

Il est également possible, entre autres choses, de définir de nouvelles actions de site et bien sûr de gérer finement les utilisateurs du site.

4.1. Requêtes sur le catalogue

Introduction

Toutes les informations du catalogue se trouvent dans l'outil *portal_catalog* qui est une version étendue de l'outil ZCatalog de Zope.

Le catalogue de Plone fournit les 3 fonctionnalités suivantes :

- créer des index de contenu
- fournir des méta-données sur le contenu de l'index
- fournir une interface de recherche

Le catalogue est intimement lié aux différents contenus de votre site Plone. Ainsi, les objets Zope et les outils ne sont pas placés dans le catalogue.

Les index

Les index se trouvent sous l'onglet *indexes* du *portal_catalog*.

Les index peuvent avoir les types suivants :

- *DateIndex* : Pour indexer les dates et faire des recherches basées sur les dates et les heures.
- *DateIndexRange* : Plus puissant que *DateIndex*, permet de faire des recherches sur 2 dates en même temps.
- *FieldIndex* : Permet de traiter automatiquement les résultats et autorise la recherche de ce que l'index peut contenir.
- *KeywordIndex* : Prend plusieurs mots-clés et les sépare en différents mots. Retournera un résultat si au moins 1 mot-clé correspond à la requête.
- *PathIndex* : Pour indexer le chemin d'un objet.
- *TextIndex* : Pour indexer du texte. Plus ancien que *ZCTextIndex*.
- *TopicIndex* : Pour créer des résultats prédéfinis. Utile dans le cas où une requête est souvent appelée.

- *ZCTextIndex* : Plus récent que *TextIndex*, ce type fournit une recherche efficace sur des morceaux entiers de texte.

Par défaut, un certain nombre d'index sont présents dans Plone. en voici quelques-uns :

| Nom | Type | Description |
|-----------------------------|---------------------|--|
| <i>Creator</i> | <i>FieldIndex</i> | Le nom d'utilisateur de la personne qui a créé l'objet |
| <i>AllowedRolesAndUsers</i> | <i>KeywordIndex</i> | Les personnes pouvant voir le contenu |
| <i>review_state</i> | <i>FieldIndex</i> | L'état d'un objet dans le workflow |
| <i>Title</i> | <i>TextIndex</i> | Le titre d'un objet |

Les Méta-données

Lorsque le catalogue renvoie un résultat, il ne renvoie pas directement un objet mais les méta-données contenues dans le catalogue.

Ces méta-données sont en fait une série de champs existants pour chaque valeur d'un objet.

Voici quelques méta-données par défaut :

| Nom | Description |
|---------------------|--|
| <i>CreationDate</i> | La date de création d'un objet |
| <i>Creator</i> | Le nom d'utilisateur de la personne ayant créé l'objet |
| <i>ExpiresDate</i> | La date d'expiration de l'objet |
| <i>end</i> | Pour les objets de type <i>event</i> , la date de fin de l'événement |
| <i>getId</i> | L'id de l'objet |

Réindexer le contenu d'un site

Il est parfois nécessaire de réindexer l'ensemble du contenu d'un site, après avoir installé un nouveau produit par exemple.

Pour ce faire, dans *portal_catalog*, cliquez sur *Advanced* puis sur *Update Catalog*.

Attention cette opération peut prendre beaucoup de temps et utiliser beaucoup de ressources.

Recherches dans le catalogue : exemples

Le meilleur moyen pour faire des recherches est d'utiliser un script python. Cependant, il est également possible de le faire à partir templates.

On utilisera la méthode *searchResults* sur l'objet *portal_catalog*.

- premier exemple :

```
context.portal_catalog.searchResults( review_state = "published",
                                     SearchableText = "Plone",
                                     sort_order="Date"
                                   )
```

Cette recherche renvoie l'intersection des résultats de chaque index. Elle affichera donc les items publiés mentionnant *Plone*.

Remarque : *sort_order* est un paramètre réservé conditionnant l'ordre d'affichage. Il existe un certain nombre de paramètres réservés en fonction du type de l'index.

- deuxième exemple :

```
results = context.portal_catalog.searchResults( Type = "Image" )
```

Ce code permet de rechercher toutes les images d'un site.

- troisième exemple :

```
from DateTime import DateTime
start = DateTime('2006/04/01' )
end = DateTime('2006/04/12' )
results = context.portal_catalog.searchResults(
    Type = "News Item",
    CreationDate = { "query": [ start , end ] , "range" : "minmax" }
)
```

Cet exemple recherche tous les événements du mois de décembre.

Remarque : *minmax* est un paramètre réservé. Il permet de préciser que les valeurs de *CreationDate* doivent se trouver entre *start* et *end*.

4.2. Requêtes sur un SGBDR

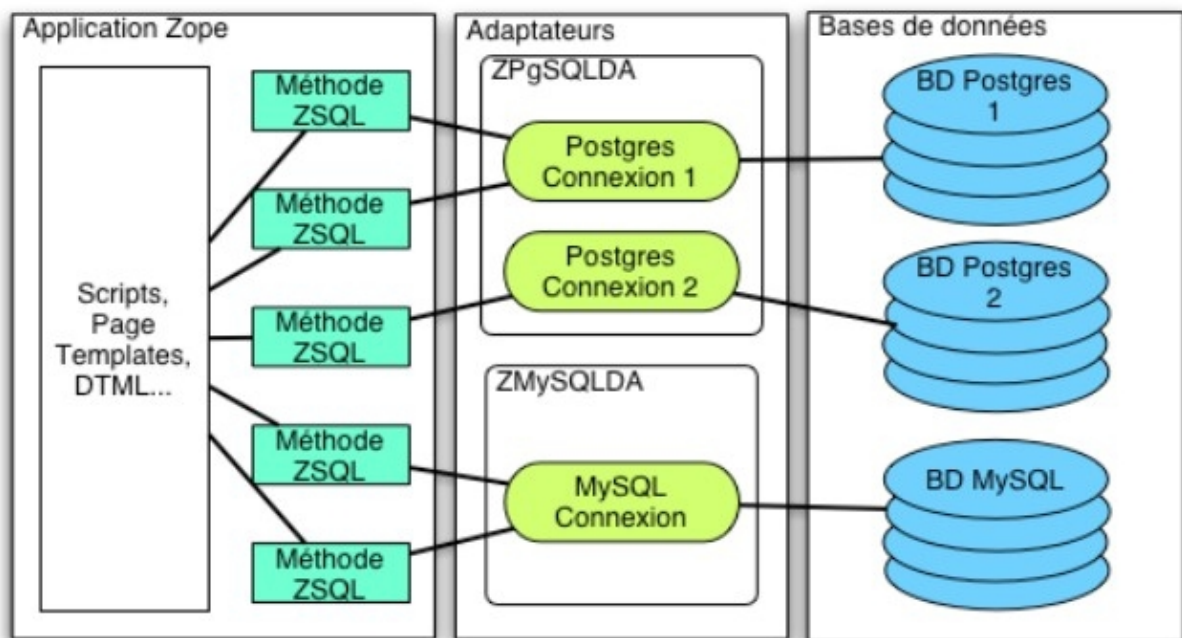
Introduction

Par défaut, Plone stocke ses données dans la base de données de Zope, la ZODB (Zope Object DataBase), une base de données orientée-objet particulièrement adaptée aux applications de gestion de contenu.

Cependant, dans certains cas, il est préférable ou nécessaire de stocker les données sur un système de fichiers ou dans une base de données relationnelle.

La plupart des grands SGBDR du marché sont connectables à Zope et par conséquent à Plone. Ces connecteurs sont généralement libres (MySQL, PostgreSQL, même Oracle), d'autres sont payants, en adéquation avec la politique de l'éditeur concerné (Microsoft SQL Server et Access).

Concepts généraux



Les adaptateurs sont des interfaces : ils permettent d'abstraire une implémentation spécifique. Ce sont eux qui font la liaison entre l'application Zope et les différentes bases de données.

Les méthodes ZSQL fournissent les moyens de traiter les programmes SQL en tant qu'objets qui peuvent être utilisés pour publier des données provenant de bases de données et créer des applications Zope avancées.

Les méthodes ZSQL sont appelées dans l'application Zope via des templates ou des scripts.

Les adaptateurs fournissent des objets de type de connexions qui permettent un accès à une base de données particulière dans Zope. Ce sont les connexions qui gèrent les transactions. Un adaptateur est un produit Zope. Il se place donc dans le répertoire *Products* de l'instance.

Il peut y avoir plusieurs connexions par instance de Zope. Elles sont configurables en ZMI et utilisées via les méthodes ZSQL.

Les scripts accèdent aux connexions via les méthodes ZSQL.

Il existe des adaptateurs pour la plupart des SGBDR courants :

- ODBC (mxODBC, ODBCDA)
- MySQL (ZMySQLDA)
- PostgreSQL (ZPgSQLDA)
- Sybase (ZSybaseDA)
- Oracle (ZOracleDA)
- SQL Server via ODBC

Les méthodes SQL sont un type universel compatible avec tous les types de bases de données.

En pratique, une requête ZSQL est composée de code DTML et de code SQL. Les constructions dynamiques sont possibles. Les résultats d'une requête sont renvoyés sous forme d'une séquence python. Ils sont exploitables avec une boucle *for .. in* ou avec l'attribut *tal:repeat* par exemple.

Exemples

Création d'une table :

```
create table contacts (  
  nom varchar,  
  prenom varchar,  
  mail varchar,  
  numero integer  
)
```

Ajout de valeurs dans cette table via un formulaire :

```
<html metal:use-macro="here/main_template/macros/master">  
<div metal:fill-slot="main">  
<form action="result_html">  
Nom:<input type="text" name="nom"/>  
Prenom:<input type="text" name="prenom"/>  
Mail:<input type="text" name="mail"/>  
Numero:<input type="text"  
      name="numero:int"/>  
<input type="submit"/>  
</form>  
</div></html>  
</pre>
```

Le même ajout de valeurs via une méthode ZSQL :

```
insert into contacts  
(nom, prenom, mail, numero)  
values  
(<dtml-sqlvar nom type="string">,  
<dtml-sqlvar prenom type="string">,  
<dtml-sqlvar mail type="string">,  
<dtml-sqlvar numero type="int"> )
```

Lors de l'édition de cette méthode en ZMI, il est nécessaire de spécifier les arguments *nom prenom mail numero* dans le champ correspondant.

Page de traitement d'ajout de valeur :

```
<html metal:use-macro="here/main_template/macros/master">
<div metal:fill-slot="main">
<tal:block tal:define="empty python:
    here.addNewRecord(nom=request['nom'],
                      prenom=request['prenom'],
                      mail=request['mail'],
                      numero=request['numero'])"
    tal:on-error="string:Erreur survenue.">
Les données ont été ajoutées.
</tal:block>
</div>
</html>
```

Formulaire de recherche :

```
<html metal:use-macro="here/main_template/macros/master">
<div metal:fill-slot="main">
<form action="search_html">
Nom à chercher:
<input type="text" name="nom"/>
<input type="submit"/>
</form>
</div>
</html>
```

Ceci est un formulaire de recherche par nom.

Requête de recherche par nom :

```
select * from contacts
where nom=<dtml-sqlvar nom type="string">
```

Là encore, spécifier *nom* en argument dans la ZMI.

Résultat de la recherche par nom :

```
<html metal:use-macro="here/main_template/macros/master">
<div metal:fill-slot="main">
Résultat :
<table>
<tr tal:repeat="item
python:here.searchByName(nom=request['nom'])">
    <td tal:content="item/nom"/>
    <td tal:content="item/prenom"/>
    <td tal:content="item/mail"/>
    <td tal:content="item/numero"/>
</tr>
</table>
</div>
</html>
```

4.3. Création d'un portlet

Introduction

La création et l'ajout de portlet sont des actions relativement simples.

Pour créer un portlet, il est nécessaire d'utiliser un template avec une macro (METAL) à l'intérieur.

Le code de base pour un template de portlet est le suivant :

```
<div metal:define-macro="portlet">
  <dl class="portlet">
    <dt class="portletHeader">Titre du portlet</dt>
    <dd class="portletItem">
      <!-- Le code se trouve ici -->
    </dd>
  </dl>
</div>
```

Pour ajouter un portlet, en ZMI, allez à la racine de votre site Plone puis cliquez sur *properties*.

En fonction de l'endroit où vous voulez que votre portlet soit affiché, vous ajouterez *here/le_nom_de_votre_portlet/macros/portlet* dans *right_slots* ou *left_slots*.

Remarque : l'ordre des déclarations des portlets correspond à l'ordre de leur affichage.

Exemple

Voici le code complet d'un portlet de recherche :

```
<html xmlns:tal="http://xml.zope.org/namespaces/tal"
      xmlns:metal="http://xml.zope.org/namespaces/metal"
      i18n:domain="plone">

<body>

<div metal:define-macro="portlet" tal:omit-tag="">

<dl class="portlet" id="portlet-search">

<dt class="portletHeader">Recherche</dt>

<dd class="portletItem">

<fieldset style="text-align:right; border: 0;">

<form name="searchform"
      action="search"
      tal:attributes="action string:${portal_url}/search" >
<label for="searchGadget" class="hiddenStructure"
      i18n:translate="text_search">Search</label>
<input id="searchGadget"
      name="SearchableText"
      type="text"
      size="20"
      value=""
```

```
alt="Search"
title="Search"
accesskey="accesskeys-search"
i18n:attributes="alt accesskey title"
tal:attributes="value request/SearchableText|nothing;
tabindex tabindex/next" class="visibility:visible" />
<input class="searchButton"
type="submit"
value="Search"
accesskey="accesskeys-search"
tal:attributes="tabindex tabindex/next"
i18n:attributes="value accesskey" style="margin-top: 0.6em"/>
</form>
</fieldset>
<dd>
</dl>
</div>
</body>
```

4.4. Définition d'une action de site

Qu'est ce que c'est ?

Les actions désignent des liens (des url donc) déclenchant une action.

Il existe différentes catégories d'actions :

- Les actions globales : elles sont définies quelque soit le contexte.
- Les actions locales : elles permettent d'associer des actions à des types de contenus ou des workflows.

D'où proviennent ces actions ?

Le `portal_actions` gère non seulement ses propres actions mais aussi celles des autres objets possédant des actions et figurant dans son onglet Actions providers.

Le `portal_action` est le produit responsable de gérer toutes les actions d'un site CMF/Plone. C'est lui par exemple qui est responsable des onglets du portail (welcome, search, news etc.) ou des boutons du `folder_contents` qui permettent de copier, coller, supprimer des objets d'un dossier, et tant d'autres !

Décomposition d'une action

Une action se décompose en 7 champs :

1. Name : son libellé
2. Id : son identifiant
3. Action : L'expression générant notre lien
4. Condition : une condition qui, si elle se vérifie fera que notre lien s'affiche (en python)
5. Permission : l'user pour lequel on affichera l'action devra avoir cette permission. Sinon l'action ne sera pas affichée.
6. Category : un identifiant commun à nos actions qui nous permettra de les retrouver parmi toutes les actions.
7. Visible : Permet de masquer/afficher temporairement ou définitivement une action.

Les types d'actions :

1. Onglets de site : `portal_tabs`
2. Actions de site : `site_actions`
3. Boutons de dossier : `folder_buttons`
4. Actions de document : `document_actions`
5. Onglets de contenu : `object_tabs`
6. Actions de dossier et de contenu : `folder / object`
7. Transitions de workflow : `workflow`

4.5. Gestion des utilisateurs

Pour gérer les utilisateurs et leurs permissions sur les objets, Plone utilise une combinaison de rôles, rôles locaux et workflows basés sur Zope.

Permissions et Rôles

Les permissions permettent de spécifier ce qu'un utilisateur peut faire et ne peut pas faire dans un contexte donné.

On peut se les représenter en tant que barrières permettant (ou pas), d'accéder à des méthodes, des scripts, des templates, ou des transitions de workflow.

Les permissions les plus courantes sont :

- *View*
- *Modify portal content*
- *Access contents information*
- *List folder contents*

Les permissions sont reliées aux rôles des utilisateurs et non aux utilisateurs eux-mêmes. Ainsi, dans un dossier quelconque, toute personne ayant le rôle *RandomRole* possède la permission *Modify portal content*.

Les rôles par défaut sont :

- *Member*
- *Manager*
- *Reviewer*

Il existe également des rôles "automatiques" :

- *Owner*
- *Anonymous*

Le rôle *Anonymous* est donné aux utilisateurs n'étant pas loggés sur le site.

Owner s'applique uniquement lorsque l'utilisateur est le possesseur (owner) d'un contenu.

En ZMI, presque tous les objets possèdent un onglet *Security*.

C'est ici qu'on voit les permissions associées aux rôles.

La case *Acquire permission settings*, si elle est cochée, indique que le rôle donnant cette permission est le même que celui de l'objet parent.

On utilise le formulaire en bas de la page de l'onglet *Security* pour ajouter et supprimer des rôles.

On peut assigner des rôles aux utilisateurs manuellement en utilisant le dossier *acl_users* placé à la racine de l'instance de Plone (pas celui de la racine de la ZMI).

Les groupes

Plone ajoute le concept de groupes d'utilisateurs au modèle de sécurité de base de Zope.

L'utilisation de groupes est un moyen efficace permettant de gérer les rôles (et donc les permissions) pour plusieurs utilisateurs simultanément.

Les groupes sont gérés directement dans plone dans la partie *users and groups administration* dans *plone setup*.

On peut donner un ou plusieurs rôles à un groupe si nécessaire. Dans ce cas, tous les utilisateurs appartenant à ce groupe hériteront automatiquement du ou des rôles de celui-ci.

Les rôles locaux et le partage

Il est souvent utile de donner à un utilisateur ou à un groupe des permissions spécifiques, uniquement sur une partie précise du site.

Dans Plone, l'onglet *sharing* (partage), permet de donner aux utilisateurs différentes permissions dans différentes zones.

La sélection des rôles s'acquiert "par le bas". Cela signifie que si un utilisateur a le rôle de *Manager*, sur le dossier */un_dossier* alors il l'a aussi pour */un_dossier/documents/un_document*.

Workflows

Dans la plupart des cas, l'utilisation de workflows est le meilleur moyen de gérer les permissions sur le contenu.

On utilise pour cela l'outil *portal_workflow*.

cf. le chapitre sur les workflows

4.6. Dossiers automatiques (smart folders)

Présentation

Les smart folders sont une nouveauté de Plone 2.1. Ils permettent d'accéder directement au résultat d'une recherche sauvegardée. Cette recherche est mise à jour automatiquement.

Il est possible de définir les critères de recherche que l'on souhaite, et de spécifier la façon dont les résultats seront affichés.

Ainsi, grâce à l'utilisation d'un smart folder, il est par exemple possible d'afficher en un simple clic l'ensemble des images ajoutées par l'auteur *Toto* classées par date d'ajouts avec 10 résultats par pages.

Création d'un smart folder

Un smart folder s'ajoute simplement comme tout autre type de contenu standard de Plone.

Il possède les champs suivants :

- *Titre*
- *Description*
- *Limite des résultats de la recherche* : Si la case est cochée alors seuls les X premiers éléments seront affichés où X correspond au champ *Nombre d'éléments*
- *Nombre d'éléments* : Le nombre d'éléments affichés par page.
- *Utiliser une vue personnalisée* : Si la case est cochée, l'affichage des résultats sera sous forme d'un tableau avec les champs sélectionnés dans *Champs de la vue personnalisée*. Sinon la vue classique de Plone est utilisée.
- *Champs de la vue personnalisée* : permet de spécifier les champs à afficher si l'option *Utiliser une vue personnalisée* est activée.

Les critères de recherche

Une fois le smart folder ajouté, il faut spécifier les critères de recherche.

Pour cela, on clique sur l'onglet *critères*.

voir modifier propriétés critères sous-dossiers partage

actions affichage ajouter un(e) dossier automatique état : brouillon

Critères pour test_smart_folder

| champs | détails des critères |
|--|--|
| <input type="checkbox"/> Type Un type d'élément (ex. Événement) | Un critère types de contenu Un critère types de contenu Valeur ■ Un des types enregistrés du portail. Image Gros dossier Lien Actualité Document |
| | nom de l'opérateur ■ Opérateur utilisé pour assembler les tests sur chaque valeur. <input type="radio"/> et <input checked="" type="radio"/> ou |

enregistrer supprimer annuler

Ajouter de nouveaux critères de recherches

| Champs nom | Critères |
|--|--|
| Liste des champs disponibles Date de création | Les critères ne correspondent pas Date relative |
| <input type="button" value="add"/> | |

Définir l'ordre de trie

| Champs nom | Inversé |
|---|--|
| Liste des champs disponibles Pas d'ordre de trie | Inverser l'ordre d'affichage <input type="checkbox"/> |
| <input type="button" value="enregistrer"/> | |

Il suffit ensuite de sélectionner les critères de recherche en ajoutant des champs (titre, créateur, date de création, lieu, description, type, état ..).

Certains champs possèdent plusieurs valeurs possibles. Par exemple le champ type a comme valeurs possibles : document, fichier, image, dossier, lien...

Enfin, il est possible de spécifier l'opérateur booléen (et/ou) permettant d'assembler les tests sur chaque valeur.

5. Création d'un nouveau type de contenu

Principe

Plone fournit de nombreux types de contenu standards :

- *Image*
- *File*
- *Link*
- *Folder*
- *Smart Folders*
- *Event*
- *news*

Mais parfois, cela n'est pas suffisant ou ne convient pas parfaitement à ce que l'on souhaite faire. Il est alors possible de créer de nouveaux types de contenus en partant de zéro, ou en héritant de fonctionnalités de types existant.

Par exemple, si je souhaite disposer d'un forum sur mon site, je peux l'implémenter moi-même en créant les types suivants :

- Un type *Forum* qui aura plus ou moins les mêmes propriétés qu'un dossier et qui pourra contenir des objets de type *Thread*.
- Un type *Thread* correspondant à une discussion et qui contiendra donc des objets de type *Post*.
- Un type *Post* correspondant à un message sur le forum.

5.1. Présentation d'Archetypes

Préliminaire : Les produits Zope

Un produit Zope (Product) est un ensemble de :

- Modules python
- Ressources (images, objets zope)
- Documentations

Il est plutôt avantageux d'utiliser un produit dans les cas suivants :

- Lorsqu'on a un problème récurrent d'un projet à un autre
- Lorsqu'on veut faire un paquetage d'un site (pour une livraison par exemple)
- Lorsqu'on veut implémenter des solutions techniques complexes telles que :
 - ◆ Un site multilingue
 - ◆ Des algorithmes complexes
 - ◆ Des types d'objets personnalisés (le cas qui nous intéresse ici)

Les Caractéristiques principales des Produits sont :

- Le partage de briques logicielles
- La réutilisabilité des briques

- La maintenabilité
- La sécurité

Exemple : Un produit minimal

L'objectif de cet exemple : définir un type d'objet qui affiche un message à sa publication.

Il est donc nécessaire d'avoir les 2 éléments suivants :

- Un script python chargé d'afficher le message
- Une vue de présentation en ZPT permettant cet affichage.

```
class ZhelloWorld:
```

```
def getMessage(self): return "Hello World !"
```

Les produits se placent dans le sous-répertoire *Products* de l'installation Zope ou de l'instance.

Ici, on aura donc un répertoire *MyMinimalProduct* avec les fichiers :

- *ZHelloWorld.py* : la définition de la classe ZhelloWorld.
- *skins/helloworld/helloworld_view.pt* : le template chargé de l'affichage.

Le produit Archetypes

Archetypes est un produit Zope qui simplifie la création de nouveaux types de contenu.

Un type de contenu peut être par exemple un document, un évènement, une image, ou n'importe quel autre objet pouvant être ajouté dans un site Plone.

Archetypes a l'avantage de nécessiter de la part du programmeur moins de déclarations et d'initialisations.

Archetypes possède des vues par défaut : édition, vue...

Il est possible également grâce à Archetypes de générer automatiquement des types directement à partir de diagrammes UML (Unified Modeling Language).

5.2. Définition d'un type et de ses vues

Le Schéma d'un type

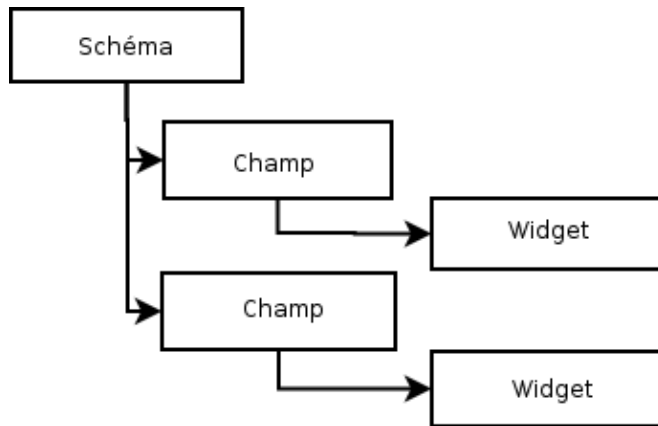
Dans Archetypes, un schéma peut se définir comme une structure d'objets. Cette structure spécifie quels attributs existent pour un type de contenu donné.

Un schéma contient de nombreux champs (fields), chaque champ étant un attribut : titre, une date d'expiration, une description, un document attaché.. Un champ possède un type : string, text, integer,...

Chaque champ possède un widget. Un widget est une représentation visuelle d'un champ dans l'interface utilisateur.

Par exemple, un champ string peut être affiché sous forme de choix dans une liste déroulante ou dans une liste de boutons radios. En définissant un widget pour ce champ, on peut définir exactement l'affichage du widget.

Le schéma suivant représente les relations entre un schéma, les champs, et les widgets associés :



Dans Archetypes, le schéma de base est *BaseSchema*. Il contient les 2 éléments que tout type de contenu possède dans Plone:

- Un titre
- Un ID

Les schémas peuvent être composés avec l'opérateur +.

Exemple :

```
schema = BaseSchema + Schema((
    StringField('message',
        widget = StringWidget(label = 'Message',
            description = 'Introduire un message ici '),
        ), ))
```

Les Champs d'un schéma

Un champ est l'association d'un type et de propriétés.

Voici quelques propriétés d'un champ :

- *name* : obligatoire, un nom unique pour le champ.
- *widget* : le widget utilisé pour afficher le champ.
- *required* : True ou False. Spécifie si au moins une valeur d'un champ est requise.
- *searchable* : True ou False. Spécifie si le champ est utilisable dans les recherches.
- *default, default_method* : Spécifie les valeurs par défaut d'un champ
- *vocabulary* : Une liste de valeurs à choisir. ex: ["rouge", "vert", "bleu"]
- *enforceVocabulary* : True ou False. Seules sont acceptées les valeurs spécifiées dans vocabulary.

Les Widgets

Un widget contient les informations concernant la manière dont l'objet sera représenté visuellement.

- *StringWidget* : Une simple boîte en HTML permettant de rentrer une chaîne de caractères.
- *DecimalWidget* : idem
- *IntegerWidget* : idem
- *ReferenceWidget* : Une sélection d'éléments en HTML parmi une liste de références possibles.
- *ComputedWidget* : Retourne la valeur calculée en HTML
- *TextAreaWidget, RichWidget* : Une zone de texte permettant l'upload du contenu dans de nombreux formats.
- *BooleanWidget* : Affiche 2 cases à cocher pour les valeurs possibles.
- *SelectionWidget, MultiSelectionWidget, KeywordWidget* : Des widgets de sélection.
- *FileWidget, ImageWidget* : Un élément permettant d'uploader un fichier.
- *LabelWidget, IdWidget, PasswordWidget*

Exemples :

```
StringField('fruit' vocabulary = ["Apple", "Orange", "Mango", "Banana"],
widget = SelectionWidget(label= "Favourite Fruit")
)
```

```
ImageField('portrait',
    widget = ImageWidget(label= "My picture")
)
```

```
TextField('body',
    searchable = 1,
    primary = 1,
    default_output_types = 'text/html',
    allowable_content_types = ('text/plain',
                               'text/structured',
                               'text/restructured',
                               'text/html' ),
    widget = RichWidget(label = 'Body' ),
)
```

Validation

Il peut s'avérer très utile d'effectuer plusieurs validations afin de tester que les valeurs du type de contenu soient correctes.

Pour faire cela, il suffit d'ajouter un paramètre de validation au champ.

Par exemple, si on veut vérifier qu'un champ *IntegerField* est vraiment un integer on peut procéder ainsi :

```
from Products.Archetypes.public import IntegerField
from Products.Archetypes.public import IntegerWidget

age = IntegerField('age', validators = ("isInt"), widget=IntegerWidget(label="Your age"))
```

Ci-dessous, quelques validations disponibles :

- *isDecimal* : S'assure que la chaîne de caractères est un décimal, incluant valeurs positives, négatives et exponentielles.
- *isInt* : S'assure qu'il s'agit bien d'un entier
- *isPrintable* : S'assure qu'il s'agit bien d'une lettre ou d'un nombre
- *isURL* : S'assure que l'entrée commence par http://, https:// ou ftp://
- *isEmail* : S'assure de la conformité de la syntaxe standard e-mail

Remarque : il est également possible de créer ses propres validations.

5.3. Cas d'utilisation : RichDocument

Introduction

RichDocument, par rapport à un type *page/document* standard de Plone, apporte les fonctionnalités suivantes :

- possibilité d'ajouter des images ou des fichiers attachés directement à l'intérieur du document, à partir de l'onglet *edit*.
- génération automatique de miniatures (thumbnails) à partir d'images uploadées.

Structure du produit

RichDocument étend la classe *ATDocument* de *ATContentTypes*. Le contenu du produit RichDocument est le suivant :

```
__init__.py
config.py

interfaces/
interfaces/richdocument.py<br>
content/
content/__init__.py
content/attachments.py
content/richdocument.py<br>
widgets/
widgets/__init__.py
widgets/attachments.py
widgets/images.py<br>
Extensions/Install.py
Extensions/utils.py<br>
skins/
skins/RichDocument/
skins/attachment_widgets/
```

Remarque : la plupart des produits ont la même structure.

__init__.py

Ce fichier est exécuté lorsque Zope charge le produit RichDocument. Il contient le code qui initialisera le produit. Il enregistre également *FileSystemDirectoryViews* du dossier *skins*. Cela signifie qu'une fois RichDocument installé, *portal_skins/RichDocument* fera référence au dossier *skins/RichDocument* par exemple.

config.py

Ce fichier contient les constantes de configuration, incluant le nom du produit et les permissions d'ajout de contenu à utiliser.

interfaces/ folder

Ce dossier contient les interfaces définies pour ce produit. L'interface `IRichDocument` étend l'interface `IDocument` d'`ATContentTypes`. Déclarer des interfaces n'est pas strictement nécessaire.

content/ folder

Ce dossier contient les types de contenu en question. Le fichier `richdocument.py` contient le type `RichDocument` tandis que `attachements.py` contient les types `ImageAttachment` et `FileAttachment`. Le script `__init.py__` charge ces fichiers de telle manière qu'on puisse importer `RichDocument*` en écrivant :

```
from Products.RichDocument import RichDocument
```

widgets/ folder

Ce dossier contient les widgets qu'utilise `RichDocument` dans un module initialisé par `__init.py__` de la même façon que `content/__init.py__`. `RichDocument` utilise deux widgets assez complexes qui utilisent des actions de contrôle de formulaire (form controller actions) pour supporter l'upload et la gestion d'images et de fichiers attachés.

Extensions/ folder

Ce dossier contient les méthodes externes d'installation. Le fichier `Install.py` est lu par l'outil `QuickInstaller` dans Plone. Ses méthodes `install()` et `uninstall()` sont exécutées quand le produit est installé ou désinstallé. Le fichier `utils.py` contient d'autres méthodes additionnelles appelées à partir de la méthode principale `install()`.

skins/ folder

Le processus d'installation standard s'assure que les dossiers `skins/RichDocument` et `skins/attachment_widget` sont enregistrés avec `portal_skins`.

Etendre ATContentTypes

La plupart des choses importantes se trouvent dans le dossier *content/folder*.

Ci-dessous, on peut voir le contenu du fichier *content/richtdocument* :

En premier lieu, le schéma du document est défini. On copie d'abord le schéma d'*ATDocument*. Faire une copie est important car en cas de modifications ultérieures sur le schéma, il est possible de modifier par inadvertance le schéma standard *ATDocument*.

Après la copie, on peut ajouter nos propres champs :

```
RichDocumentSchema = ATDocument.schema.copy() + Schema((

    BooleanField('displayImages',
        default=False,
        languageIndependent=1,
        widget=ImagesManagerWidget(
            description="If selected, a list of uploaded images will be "
                "presented at the bottom of the document to allow "
                "them to be easily downloaded.",
            description_msgid='RichDocument_help_displayImages',
            i18n_domain='RichDocument',
            label="" "Display images download box" "",
            label_msgid='RichDocument_label_displayImages',
        ),
    ),

    BooleanField('displayAttachments',
        default=True,
        languageIndependent=1,
        widget=AttachmentsManagerWidget(
            description="If selected, a list of uploaded attachments will be "
                "presented at the bottom of the document to allow "
                "them to be easily downloaded",
            description_msgid='RichDocument_help_displayAttachments',
            i18n_domain='RichDocument',
            label="" "Display attachments download box" "",
            label_msgid='RichDocument_label_displayAttachments',
        ),
    ),

),)
```

On utilise ici *BooleanFields* avec les widgets *ImagesManagerWidget* et *AttachementsManagerWidget* (situés dans le dossier *widgets/*). Le champ booléen est utilisé pour déterminer si une boîte de download pour les images et les fichiers attachés sera affichée en bas de la vue de la template.

Après la définition du schéma, on appelle la méthode *finalizeATCTSchema* :

```
finalizeATCTSchema(RichDocumentSchema)
```

Maintenant que le schéma est en place, définir la classe de contenu est aisé. On remarque qu'elle hérite à la fois de *OrderedBaseFolder* et de *ATDocument* :

```
class RichDocument(OrderedBaseFolder, ATDocument):
    """
    A document which may contain directly uploaded images and attachments
    """
```

```
# Standard content type setup portal_type = meta_type = RichDocument archetype_name = Rich
document content_icon = RichDocument.gif schema = RichDocumentSchema typeDescription= A
document which can contain rich text, images and attachments typeDescMsgId =
RichDocument_description_edit
```

Pour s'assurer qu'on soit capable d'ajouter des images et des fichiers attachés aux documents RichDocument, il est nécessaire de prévenir Archetypes que ce sont des types de contenu autorisés.

Ci-dessous, la définition de *ImageAttachement* et *FileAttachement*:

```
allowed_content_types = [ImageAttachement, 'FileAttachement']
```

Maintenant la définition de la vue par défaut et des vues supplémentaires :

```
default_view = richdocument_view
immediate_view = richdocument_view
suppl_views = (richdocument_view_preview, richdocument_view_float)
```

Il est nécessaire de dire à Zope quelles interfaces on implémente. On utilise celle d'*ATDocument* mais aussi *INonStructuralFolder* de Plone et *IRichDocument* définie dans *interfaces/richdocument.py* :

```
__implements__ = ATDocument.__implements__ + (IRichDocument, INonStructuralFolder,)
```

Les actions dans Plone permettent de définir des liens. Les onglets verts (view, edit...) correspondent à des actions sur un type de contenu. Pour s'assurer que l'on a les mêmes actions que le type document/page standard, on fait:

```
actions = ATDocument.actions
```

Finalement, on enregistre le type avec Archetypes:

```
registerType(RichDocument)
```

Etendre les autres types

Pour s'assurer qu'on puisse contrôler le workflow ainsi que les autres aspects des types images et fichiers attachés indépendamment des types Image et Fichier standards de Plone, on fournit les types ImageAttachment et FileAttachment qui étendent leurs équivalents ATContentTypes.

Ces types sont définis dans le fichier *content/atachements.py* :

```
class FileAttachment(ATFile):
    """A file attachment"""
    portal_type = meta_type = 'FileAttachment'
    archetype_name = 'File attachment'
    content_icon = 'file_icon.gif'
    typeDescription= 'A file attached to a document'
    typeDescMsgId = 'FileAttachment_description_edit'
    global_allow = 0

    default_view = 'fileattachment_view'
    immediate_view = 'fileattachment_view'
    suppl_views = ()

    __implements__ = ATFile.__implements__
    actions = ATFile.actions

registerType(FileAttachment)

class ImageAttachment(ATImage):
    """An image attachment"""
    portal_type = meta_type = 'ImageAttachment'
    archetype_name = 'Image attachment'
    content_icon = 'image_icon.gif'
    typeDescription= 'An image attached to a document'
    typeDescMsgId = 'ImageAttachment_description_edit'
    global_allow = 0

    default_view = 'imageattachment_view'
    immediate_view = 'imageattachment_view'
    suppl_views = ()

    __implements__ = ATImage.__implements__
    actions = ATImage.actions

registerType(ImageAttachment)
```

On désactive la variable *global_allow* pour s'assurer qu'ils puissent être ajoutés uniquement à l'intérieur d'un RichDocument.

Utilisation de classes combinées (Mix-in classes)

En fait, il n'est pas nécessaire d'étendre un type entier. ATContentTypes est divisé en classes combinées que l'on peut utiliser dans nos propres types, fournissant les réglages et les méthodes pour différentes classes de types de contenu. Elles sont définies dans le fichier *ATContentTypes/content/base.py* :

ATCTMixin

Fournit les réglages FTI (Factory Type Information) standards et les méthodes de base pour tous les types *ATContentTypes*.

ATCTContent

Une classe de base pour tous les contenus *ATContentTypes*. Elle combine *ATCTMixin* et *BaseContent* d'Archetypes.

ATCTFileContent

Spécialisation d'*ATCTContent* qui peut se comporter comme une classe de base pour les fichiers pouvant être téléchargés directement par le navigateur sans en afficher le contenu dans Plone.

ATCTFolder

Spécialisation d'*ATCTContent* pour les types de contenus devant se comporter comme un dossier Plone.

ATCTOrderedFolder

Alternative à *ATCFolder* qui autorise l'ordonnancement manuel de contenus. *ATFolder* utilise cette classe de base.

ATCTBTreeFolder

Alternative à *ATCFolder* qui autorise les dossiers à contenir un très grand nombre d'objets (des milliers). Les dossiers *Btree* sont plus efficaces, mais en contrepartie ils perdent certaines fonctionnalités.

6. Personnalisation de workflow

La gestion de workflow est un des points forts de Plone. Il est possible de configurer entièrement le workflow pour qu'il soit adapté au flux de publication souhaité.

L'outil *portal_workflow* de Plone permet de tout configurer finement.

6.1. Création d'un nouvel objet workflow, définition des états et transitions

La personnalisation ou l'ajout/suppression de workflows se fait en ZMI via *portal_workflow*.

Un workflow est en fait un objet Zope.

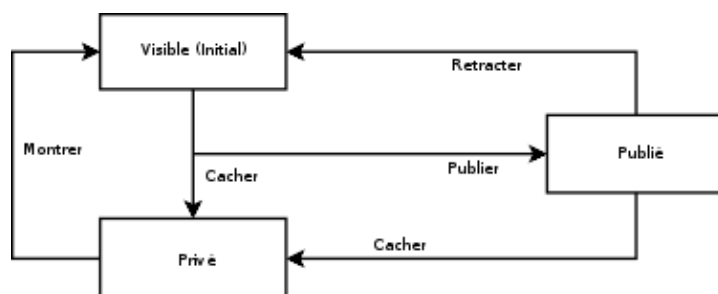
Les workflows standards sont:

- *folder_workflow* pour les dossiers
- *plone_workflow* pour les contenus

Un workflow est constitué d'états et de transitions.

état : Un état possède un intitulé et des permissions associées.

transition : Une transition possède un intitulé et une condition. Une transition définit ce qui permet de passer d'un état à un autre.



Ici, on a 3 états et leurs transitions associées représentées par des flèches.

Définition des états

Pour accéder à un workflow particulier, dans *portal_workflow* on utilise l'onglet *Contents*.

Puis on clique sur le nom du workflow souhaité pour le configurer.

L'onglet *States* liste les états présents pour ce workflow.

Un état représente un objet à un point particulier du temps.

Chaque état a un identifiant unique.

Sous le nom de l'état, on peut voir la liste des transitions possibles.

On peut également définir quel sera l'état initial.

Définition des transitions

Les transitions représentent les changements possibles pour un objet.

L'onglet *Transitions* liste les transitions du workflow.

Voici les différents champs d'une transition :

- *Title* : le titre de la transition
- *Description* : une description détaillée pour la transition
- *Destination state* : L'état cible pour cette transition.
- *Trigger type* : Indique la façon dont la transition sera effectuée. *Automatic* signifie qu'elle le sera dès qu'elle passera dans cet état. *Initiated by user action* (le plus couramment utilisé) signifie qu'un utilisateur a provoqué la transition en cliquant sur un lien.
- *Script(before)* : Exécute le script avant que la transition ait lieu.
- *Script(after)* : Exécute le script après que la transition ait lieu.
- *Guard* : La sécurité pour cet état.
- *Display in actions box* : La manière dont la transition sera affichée dans Plone.

Permission associées

L'onglet *Permissions* liste les permissions gérées par le workflow.

Pour ajouter une permission, on la sélectionne dans le menu déroulant puis on clique sur *Add*.

Attention : Après toute modification/ajout/suppression, il est nécessaire de cliquer sur *Update security settings* à partir de l'onglet *Contents*. Cela aura pour effet de répercuter les changements sur tous les objets du site.

6.2. Actions automatiques aux transitions

Intérêt

Les actions automatiques peuvent être très utiles dans les workflows.

Par exemple, on peut souhaiter envoyer un mail à un utilisateur dès qu'un de ses contenus est publié. Dans ce cas, on appellera le script d'envoi de mail après la transition.

En fonction de ce qui est souhaité, il est possible d'appeler un script avant ou après une transition.

Méthode

Pour ajouter une transition automatique il est nécessaire d'ajouter au préalable un ou plusieurs scripts via l'onglet *Scripts*.

Ensuite, dans l'onglet *Transitions*, il faut choisir l'état voulu en cliquant sur son nom.

Enfin, dans l'un des 2 menus déroulants *Script (before)* ou *Script (after)* il suffit de sélectionner le script qui convient.