6

DTML avancé : gestion des erreurs et arborescences

Vous combattez et détruisez toutes les erreurs,
mais que mettez-vous à leur place ?
— Marie du Deffand

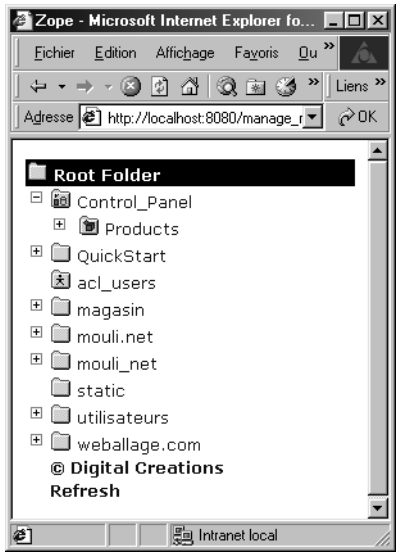
Le langage DTML propose divers mécanismes intégrés pour automatiser la présentation et la gestion de sites web, y compris des mécanismes de gestion d'erreurs. Nous verrons comment présenter une arborescence de contenu – à la manière de l'explorateur de fichiers Windows –, par quels mécanismes Zope gère les erreurs. Enfin, nous nous pencherons sur un format de texte particulier, le format `.stx` qui, grâce à des conventions de mise en forme simples, permet de rendre la structure d'un contenu sans passer par des balises HTML.

Arborescences

Nous avons vu, lors de l'étude de l'environnement de développement de Zope, que son interface était elle-même écrite en Zope. Les concepteurs ont donc fait en sorte que les programmateurs puissent se servir des outils utilisés dans cette interface.

Il en est ainsi de l'arborescence du site : la partie située à gauche dans les pages d'administration.

Figure 6-1
Arborescence d'une page d'administration



Création de l'arborescence d'un site

Cette page montrée sur la figure 6-1 est réalisée en DTML et est représentée en code HTML pur, sans que l'on ait à y entrer du code JavaScript. On peut aisément l'imiter sur son propre site avec la balise `<dtml-tree>` :

1. Dans la racine de Zope, on crée une `DTML Method` appelée `arbre` et contenant le code suivant :

```
<dtml-tree>
  <dtml-var id>
</dtml-tree>
```

2. En ouvrant ensuite notre navigateur à l'URL *http://localhost:8080/arbre*, on peut admirer le résultat (voir figure 6-2).

Impressionnant ! Trois lignes de code pour parvenir à un tel résultat… Voici un extrait du code HTML généré automatiquement grâce à la balise `<dtml-tree>` :

```
<TABLE CELLSPACING="0">
<TR>
<TD WIDTH="16" VALIGN="TOP" NOWRAP><A NAME="AAAAAAAAAAY="
➥HREF="arbre?tree-c=eJyLVneEAO9bdROFJH6krXosAIALB9E#AAAAAAAAAAY=">
➥<IMG SRC="/p_/mi" ALT="-" BORDER=0></A></TD>
<TD COLSPAN="2" VALIGN="TOP" ALIGN="LEFT">  Control_Panel
</TD>
</TR>
```

Figure 6-2
Rendu d'une méthode `arbre` *de trois lignes…*


```
<TR>
<TD WIDTH="16" NOWRAP></TD>
<TD WIDTH="16" VALIGN="TOP" NOWRAP><A NAME="AAAAAAAAAAg=" HREF="arbre?
➥tree-e=eJyLVneEAO9bdROFJH4kGj/dVjOWABlPC5k#AAAAAAAAAAg=">
➥<IMG SRC="/p_/pl" ALT="+" BORDER=0></A></TD>
<TD VALIGN="TOP" ALIGN="LEFT">  Products
</TD>
</TR>
```

(...)

```
<TR>
<TD WIDTH="16" NOWRAP></TD><TD WIDTH="16" NOWRAP></TD>
<TD VALIGN="TOP" ALIGN="LEFT">  articles
</TD>
</TR>
<TR>
<TD WIDTH="16" NOWRAP></TD><TD WIDTH="16" NOWRAP></TD>
<TD VALIGN="TOP" ALIGN="LEFT">  boutique
</TD>
</TR>
```

(...)

```
<TR>
<TD WIDTH="16" VALIGN="TOP" NOWRAP><A NAME="AAAAAAAAEkk=" HREF="arbre?
➥tree-e=eJyLVneEAO9bdROFON81O9tWPRYAgTOIEQ#AAAAAAAAEkk=">
➥<IMG SRC="/p_/pl" ALT="+" BORDER=0></A></TD>
```

```
<TD COLSPAN="2" VALIGN="TOP" ALIGN="LEFT">  mouli_net
</TD>
</TR>
<TR>
<TD WIDTH="16" NOWRAP></TD>
<TD COLSPAN="2" VALIGN="TOP" ALIGN="LEFT">  static
</TD>
</TR>
```

(...)

```
<TR>
<TD WIDTH="16" VALIGN="TOP" NOWRAP><A NAME="AAAAAAAAECA=" HREF="arbre?
➥tree-e=eJyLVneEAO9bdROFON/V2dFWPRYAf8OHvw#AAAAAAAAECA=">
➥<IMG SRC="/p_/pl" ALT="+" BORDER=0></A></TD>
<TD COLSPAN="2" VALIGN="TOP" ALIGN="LEFT">  weballage.com
</TD>
</TR>
</TABLE>
```

Il s'agit bien de code HTML strict, le développement des branches étant pris en charge par des liens au travers de variables appelées `tree-e` (« e » pour *expand*, « développer ») et `tree-c` (« c » pour *collapse*, « réduire »). Ces variables sont renseignées avec l'état de l'arbre d'une manière codée ; le programmeur n'a pas à s'en soucier – c'est le travail de Zope.

Le résultat n'est cependant pas tout à fait identique au menu de l'environnement de développement de Zope. Pour le lecteur curieux, voici reproduite la partie du code DTML qui gère ce menu (d'après le fichier DTML situé dans le fichier $*ZOPE/lib/python/App/dtml/menu.dtml*).

```
<dtml-tree nowrap=1>
  <dtml-if icon>
  <a href="<dtml-var tree-item-url fmt=url-quote>/manage_workspace"
    target="manage_main"><img src="&dtml-BASEPATH1;/&dtml-icon;" border="0"
    title="Click to open this item" alt="&dtml-meta_type;" /></a>
  </dtml-if>
  <a href="<dtml-var tree-item-url fmt=url-quote>/manage_workspace"
     target="manage_main">&dtml-id;</a>
</dtml-tree>
```

Et la page obtenue (voir figure 6-3).

Nous allons maintenant étudier le fonctionnement de la balise `<dtml-tree>` et examiner quelles options sont à notre disposition pour personnaliser son fonctionnement.

Figure 6-3
La méthode `arbre` *d'après le fichier* `menu.dtml`

Contenu de la balise `<dtml-tree>`

La balise `<dtml-tree>` a été principalement conçue pour faciliter le parcours d'arborescences Zope. C'est pourquoi la plupart des attributs disponibles, de par leur comportement et leur valeur par défaut, sont fortement liés à la structure des `Folder` de Zope.

> **Remarque**
> *Nous verrons dans l'étude de cas consacrée à LDAP qu'il est possible, au prix de quelques efforts, de parcourir autre chose que des objets* `Folder`.

La balise `<dtml-tree>` fonctionne d'une manière semblable à `<dtml-in>` : il s'agit d'une balise DTML de parcours. Autrement dit, chaque objet rencontré provoque un passage dans le corps de la balise (le code situé entre `<dtml-tree>` et `</dtml-tree>`). Par exemple, voici un code inutile mais démonstratif :

```
<dtml-tree>
  Hello, World !
</dtml-tree>
```

Le texte `Hello, World !` est affiché pour chaque branche.

Tout comme la balise `<dtml-in>`, la balise `<dtml-tree>` empile (c'est-à-dire place au sommet de l'espace de noms) l'objet parcouru à chaque itération[1]. Il est donc possible d'accéder directement aux attributs de l'objet concerné. Voilà pourquoi, nous pouvons écrire :

```
<dtml-tree>
  <dtml-var id>
</dtml-tree>
```

Nous sommes assurés que l'objet courant se trouve au sommet de la pile de l'espace de noms et que l'attribut `id` renvoyé sera bien celui de l'objet que nous souhaitons. Dans le code de `menu.dtml`, l'attribut `icon` de chaque objet est utilisé ; il s'agit d'une chaîne correspondant à l'URL de l'icône du type d'objet concerné.

Tout comme `<dtml-in>`, la balise `<dtml-tree>` permet l'utilisation de variables particulières dans le corps de la boucle :

Nom	Description
`tree-item-expanded`	*Variable vraie si l'élément courant est développé (c'est-à-dire si ses sous-éléments sont visibles).*
`tree-item-url`	*URL de l'élément courant.*
`tree-root-url`	*URL du document dans lequel l'arbre apparaît.*
`tree-level`	*Profondeur de l'élément courant dans l'arbre (les éléments figurant en haut de l'arbre ayant une profondeur de 0).*
`tree-colspan`	*Nombre de niveaux affichés dans l'arbre (par exemple, si aucun élément de l'arbre n'est développé, un seul niveau apparaît et* `tree-colspan` *vaut 1 ; si un élément est développé, le nombre de niveaux affichés par l'arbre est 2 ; si un élément développé est lui-même développé, le nombre de niveaux affichés vaut 3, et ainsi de suite).*
`tree-state`	*État de l'arbre (codé).*

La variable `tree-state` n'est quasiment jamais utilisée. `tree-colspan` est souvent utilisée, comme son nom l'indique, pour aligner les éléments dans un tableau. Les autres variables sont plus explicites.

Variables contrôlant l'état de l'arbre

Le programmeur dispose de certaines variables pour modifier l'état de l'arbre. Voici comment, par exemple, afficher deux liens permettant de développer ou de réduire l'ensemble de l'arbre :

```
<dtml-tree>
  <dtml-var id>
```

1. Voir la section « Espace de noms » au chapitre 4.

```
</dtml-tree>

<a href="<dtml-var URL>?expand_all=1">Tout développer</a><br>
<a href="<dtml-var URL>?collapse_all=1">Tout réduire</a><br>
```

Remarque

Compte tenu des contraintes de sécurité imposées par Zope, le développement complet de l'arbre est interdit. La balise `<dtml-tree>` *propose un attribut* `skip_unauthorized` *fonctionnant comme pour la balise* `<dtml-in>` *afin de contourner ce problème, mais cet attribut n'est hélas d'aucune efficacité dans notre cas. Le développement complet de l'arbre est donc réservé aux parcours d'objets définis par l'utilisateur, comme nous le verrons plus bas.*

L'état de l'arbre est stocké sous forme de cookie sur le poste client : voilà pourquoi l'état de l'arborescence de notre site perdure si l'on ferme puis si l'on ouvre à nouveau la fenêtre. Ce cookie porte le nom `tree-s` (« s » pour *state*).

Remarque

Il est en théorie possible d'utiliser directement les variables `tree-c`, `tree-e` *et* `tree-s` *pour altérer l'état de l'arbre par programmation, mais ceci sort du cadre du présent ouvrage. L'usage direct de ces variables est par ailleurs déconseillé.*

Attributs de la balise `<dtml-tree>`

La balise `<dtml-tree>` peut prendre un grand nombre d'attributs. En voici la liste et la description.

Attributs de génération d'arbre

Attributs `name` *et* `expr`

Les attributs `name` et `expr` sont implicites, comme nous l'avons déjà vu. Le choix de l'un ou de l'autre est indiqué par l'usage ou non de guillemets.

Dans la balise `<dtml-tree>`, ces attributs désignent la racine des objets à parcourir. S'ils ne sont pas mentionnés, c'est l'objet courant qui est parcouru par défaut.

Pour forcer le parcours à partir d'un autre `Folder`, on en spécifie l'`id` de la manière suivante :

```
<dtml-tree magasin>
  <dtml-var id>
</dtml-tree>
```

Le parcours ne s'effectue alors que pour les dossiers contenus dans le dossier `magasin`.

Attention

Dans nos exemples précédents, nous avons placé la balise `<dtml-tree>` *dans une* `DTML Method` *plutôt que dans un* `DTML Document` *car les attributs implicites* `name/expr` *désignent par défaut l'objet courant. Or, un* `DTML Document` *est un objet à part entière, à la différence d'une* `DTML Method`*, qui s'applique à un objet. Utiliser* `<dtml-tree>` *sans attribut dans un* `DTML Document` *revient à parcourir l'objet courant, c'est-à-dire… le* `DTML Document` *! Si nous avions utilisé un* `DTML Document`*, il aurait fallu pour que le parcours soit valide préciser* `<dtml-tree expr="PARENTS[0]">` *afin d'indiquer que le parcours devait se faire à partir du dossier conteneur et non du document lui-même.*

Attribut `branches` *et* `branches_expr`

Les attributs `branches` et `branches_expr` indiquent respectivement à la balise `<dtml-tree>` une méthode ou une expression permettant, à partir de l'objet courant, d'obtenir une liste de sous-objets.

Attention

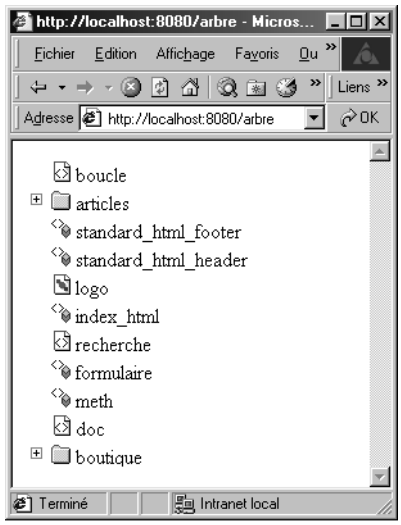
Ici, les guillemets s'appliquent dans les deux cas : les valeurs de `branches` *ainsi que celles de* `branches_expr` *doivent être encadrées par des guillemets.*

Par défaut, `branches` vaut `tpValues`. `tpValues` est une méthode spéciale qui ne tient compte que des objets de type conteneur (comme `Folder`, `Control Panel`, etc.). Si l'on souhaite obtenir la liste de tous les objets, et pas seulement celle des conteneurs, il faut spécifier explicitement `objectValues` :

```
<dtml-tree magasin branches="objectValues">
  <img src="<dtml-var icon>"> <dtml-var id>
</dtml-tree>
```

Cette fois, tous les objets sont affichés :

Figure 6-4
Arbre incluant tous les types d'objets



Attribut `id`

Dans la balise `<dtml-tree>`, l'attribut `id` indique sur quel attribut il faut s'appuyer pour identifier d'une manière *unique* chaque objet dans l'arbre. Par défaut, il s'agit de l'attribut `tpId`, un attribut présent sur la plupart des objets Zope, et non de l'attribut `id` car ce dernier n'est un discriminant unique qu'au sein d'un dossier donné. Pour l'illustrer, voyez l'exemple de hiérarchie suivant :

```
|-- DossierA
|-- DossierB
      |-- DossierA
```

L'attribut `id` ne suffit pas ici à distinguer les deux dossiers portant le même identificateur `DossierA`. L'arbre généré poserait problème (on ne saurait pas si on développe `DossierA` ou `DossierB/DossierA`). C'est la raison pour laquelle Zope fournit un attribut `tpId` qui permet d'identifier d'une manière unique les objets. Il s'agit d'une mécanique interne aux arbres.

Remarque

La valeur de `tpId` *est utilisée par la balise* `<dtml-tree>` *pour calculer et coder l'état de l'arbre.*

Il n'y a pas lieu de spécifier d'attribut autre que `tpId` lors du parcours d'objets Zope, mais il est indispensable de le faire pour parcourir des objets autres que des objets Zope, comme nous le verrons dans la troisième étude de cas.

Attribut `url`

L'attribut `url` permet au programmeur de spécifier le contenu de la variable `tree-item-url` pour chaque élément. Il doit ici aussi s'agir d'une méthode ou d'un attribut présent sur chaque objet parcouru.

Par défaut, il s'agit de l'attribut `tpUrl` qui renvoie l'URL de l'objet parcouru, relative à l'objet rendu. Il est parfois utile de remplacer cette valeur par `absolute_url`, qui renvoie l'URL absolue (et pas relative) de l'objet parcouru.

Observons la différence entre l'utilisation de `tpUrl` et celle de `absolute_url` :

```
<dtml-tree>
  <img src="<dtml-var icon>"> <dtml-var tree-item-url>
</dtml-tree>
```

(voir figure 6-5)

```
<dtml-tree url="absolute_url">
  <img src="<dtml-var icon>"> <dtml-var tree-item-url>
</dtml-tree>
```

(voir figure 6-5)

Figure 6-5
Utilisation de l'attribut `url` *par défaut,* `tpUrl`

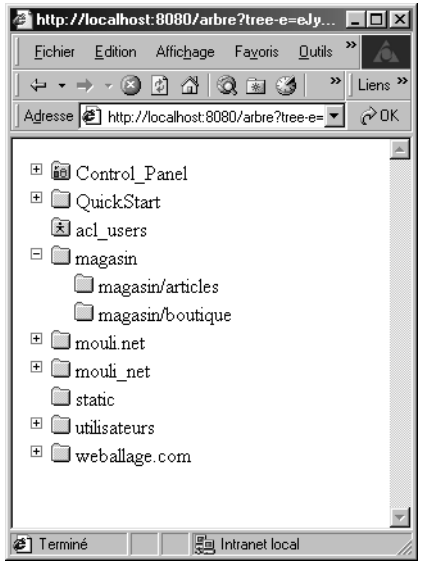
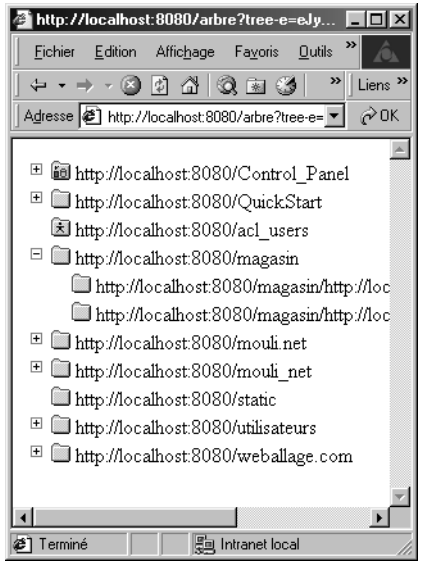


Figure 6-6
Utilisation de la valeur `absolute_url` *pour l'attribut* `url`



Tout comme l'attribut `id`, l'attribut `url` présente surtout un intérêt lors du parcours d'autres types d'objets que des objets Zope.

Attributs de contrôle de l'apparence

Attribut `nowrap`

L'attribut `nowrap` indique si le texte d'un élément parcouru peut être scindé en plusieurs lignes ou tronqué s'il est trop grand. L'attribut `nowrap` peut prendre les valeurs « `0` » ou « `1` », `0` indiquant que le texte doit être scindé, et `1` qu'il doit être tronqué.

Attributs `header`, `footer` *et* `leaves`

Les attributs `header`, `footer` et `leaves` permettent respectivement d'inclure un document lors du début du parcours, de la fin du parcours ou lors du parcours d'un élément « feuille », c'est-à-dire qui n'a pas de sous-éléments.

Attributs de contrôle du comportement

Attribut `sort`

L'attribut `sort` permet de trier les objets au vu d'un de leurs attributs, lors de l'affichage de l'arbre. Par défaut, les objets ne sont pas triés.

Voici par exemple comment les trier par titre :

```
<dtml-tree sort="title">
  <img src="<dtml-var icon>"> <dtml-var id>
</dtml-tree>
```

Attention

Tout comme pour la balise `<dtml-in>`*, l'attribut spécifié par* `sort` *doit exister pour que le parcours fonctionne. Par ailleurs, dans le cas de l'attribut* `title`*, les résultats peuvent paraître surprenants (les éléments n'étant pas réellement triés) parce que cet attribut a une valeur vide par défaut pour beaucoup d'objets Zope.*

Attribut `assume_children`

Dans certains cas, le calcul de la liste des enfants d'un élément peut prendre du temps (c'est le cas, par exemple, pour le parcours d'une base LDAP).

Or, pour que la balise `<dtml-tree>` détermine si un élément est une feuille (c'est-à-dire qu'il n'a pas de sous-éléments) ou une branche (avec des sous-éléments), elle est obligée de calculer la liste de ses sous-éléments.

L'attribut `assume_children`, s'il vaut 1, contraint la balise à ne pas faire cette vérification et à considérer que tout élément peut avoir des sous-éléments. Par conséquent, la case « + » sera présente sur tous les éléments affichés. Ce n'est que lorsqu'on essayera de développer une branche qui n'a pas de sous-éléments que Zope fera disparaître le « + ».

Cet attribut est surtout utile lorsque le parcours des objets lui-même demande du temps. Par défaut, il est désactivé.

Attribut `single`

L'attribut `single`, s'il vaut « 1 », indique que seule une branche de l'arbre peut être développée à la fois : lorsque l'utilisateur clique sur une branche non-développée, les branches déjà développées sont réduites.

Par défaut, l'attribut `single` vaut « 0 ».

> **Remarque**
>
> *Lors de la manipulation des attributs de la balise `<dtml-tree>`, il se peut que vous obteniez des résultats surprenants (plusieurs branches se développant ou se réduisant en même temps, par exemple). Ces effets sont dus aux cookies gérés par cette balise, qui ne sont pas réinitialisés à chaque parcours de la page : ainsi, par chance, ces désagréments ne se produisent-ils que sur la machine sur laquelle le programmeur travaille. Une fois le site publié, tout fonctionnera correctement sur les navigateurs vierges de cookies.*

Attribut `skip_unauthorized`

L'attribut `skip_unauthorized` fonctionne comme pour la balise `<dtml-in>`. Paradoxalement, la présence de l'attribut `skip_unauthorized` ne permet pas de développer toute l'arborescence d'un site Zope ainsi que nous avons pu le souligner plus haut (cette remarque est vraie pour la version 2.3.0 de Zope). Gageons que ce bogue sera corrigé dans les versions ultérieures.

Créer ses propres arbres

Nous verrons en étude de cas un exemple typique de l'utilisation de la balise `<dtml-tree>` dans un tout autre contexte que la manipulation d'objets Zope.

Nous n'allons pas présenter ici d'exemple de parcours d'autres objets que des objets Zope, car cela implique la connaissance de la programmation orientée objet en Python. Nous en verrons un exemple pratique dans l'étude de cas consacrée à LDAP.

En décrivant la façon dont la balise parcourt les objets, nous avons constaté qu'un objet devait comporter les attributs ou méthodes suivants :

- un attribut permettant d'identifier l'objet d'une manière unique ;
- une méthode renvoyant la liste des sous-objets d'un objet ;
- éventuellement, un attribut permettant d'associer une URL à un objet.

Il faut, enfin, qu'une méthode renvoie un objet qui puisse servir de racine à l'arbre.

Ces considérations seront utiles au programmeur Zope qui souhaite parcourir ses propres objets.

Gestion des erreurs

Au cours de notre apprentissage du langage DTML, nous avons rencontré beaucoup de cas pouvant produire une erreur (par exemple, quand on essaie d'accéder à une URL incorrecte, quand on utilise une variable qui n'est pas définie dans du code DTML…). Zope propose quelques outils pour mieux gérer ces états d'erreur.

Messages d'erreur

Lorsqu'il rencontre une erreur, Zope affiche un document appelé `standard_error_message`, dont un exemple se trouve à la racine du site lors de son installation. Il est possible de modifier ce document pour lui donner une apparence moins « zopienne » et plus adaptée à son intégration dans la charte graphique d'une entreprise.

On peut bien entendu définir plusieurs documents `standard_error_message` qui seront alors, dans le meilleur des cas, générés par acquisition (« acquis ») lorsqu'une erreur se produit.

Surcharge de `standard_error_message`

Il est possible de réécrire complètement le document `standard_error_message` ou de le redéfinir dans une partie du site. Dans ce document, trois variables particulières sont définies :

- `error_type` contient le type de l'erreur ;
- `error_value` contient la valeur ayant provoqué l'erreur `error_type` ;
- error_message, s'il est défini, contient un message d'erreur plus explicite (en anglais) ;
- `error_tb` contient la « traceback » (voir ci-après).

Il est possible d'utiliser ces trois variables pour afficher des messages d'erreur personnalisés.

> **Attention**
>
> *La variable* `error_message` *n'est pas définie pour toutes les erreurs. Il faut donc toujours s'assurer qu'elle existe avant de l'afficher.*

D'une manière générale, la page `standard_error_message` doit faire le moins possible appel à du code DTML, sous peine de provoquer des erreurs en cascade. Zope se tire d'une telle situation en affichant un message d'erreur par défaut, mais il est alors impossible de localiser l'erreur d'origine.

> **Conseil**
>
> *Mieux vaut n'écrire que du HTML dans cette page et n'utiliser que les trois variables* `error_type`, `error_value` *et* `error_message`.

Message acquis ou message par défaut

Globalement, deux types d'erreurs peuvent survenir dans Zope :

- les erreurs d'exécution qui surviennent lorsque Zope rend un objet (variable non définie, erreur de type…) ;
- les erreurs qui se produisent lorsqu'un utilisateur ou un objet tente d'accéder à une ressource qui n'est pas valide ou protégée (surtout si une erreur de sécurité est déclenchée ou si le document `standard_error_message` contient lui-même une erreur).

Dans le premier cas, Zope n'a aucun mal à « remonter » les objets grâce à l'acquisition, afin de retrouver un document `standard_error_message` et l'afficher (l'acquisition fonctionnant ici parfaitement).

Dans le second cas, Zope est incapable de déterminer quel `standard_error_message` il faut afficher : les concepteurs de Zope ont donc élaboré un document par défaut qui s'affiche lorsque Zope ne peut pas déterminer le document qu'il faut afficher. Il n'est pas possible de modifier ce document.

> **Remarque**
> *Cette limitation de Zope permet tout de même au programmeur d'attraper les fameuses « erreurs 404 », c'est-à-dire les URL incorrectes.*

Voici l'exemple d'un message d'erreur qui apparaît lorsque l'utilisateur entre un mot de passe incorrect pour accéder à une ressource protégée (voir figure 6-7).

La page commence par un petit texte expliquant ce qui s'est passé (« Zope has encountered an error while publishing this resource. »), puis indique le type de l'erreur (« Unauthorized ») et la valeur de l'erreur (ici, « You are not authorized to access *zbook*. »).

Traceback

Lorsque Zope est démarré avec l'option `–D` (c'est l'option par défaut lors de l'installation), un extrait de code, appelé `traceback`, est rajouté à la fin de tous les messages d'erreur (acquis ou pas). Voici ce que cela donne avec le précédent message :

```
Traceback (innermost last):
  File C:\Program Files\WebSite\lib\python\ZPublisher\Publish.py, line 222,
  ➥in publish_module
  File C:\Program Files\WebSite\lib\python\ZPublisher\Publish.py, line 187,
  ➥in publish
  File C:\Program Files\WebSite\lib\python\ZPublisher\Publish.py, line 171,
  ➥in publish
  File C:\Program Files\WebSite\lib\python\ZPublisher\mapply.py, line 160,
  ➥in mapply
    (Object: test)
```

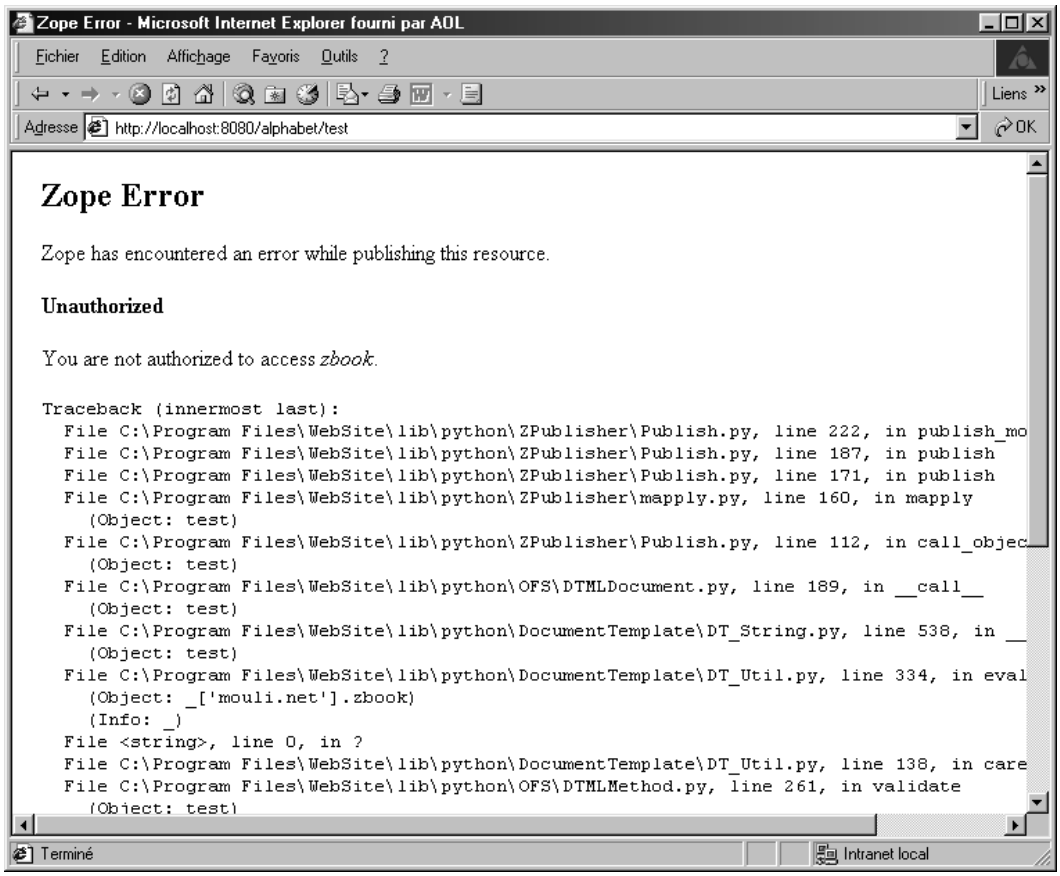


Figure 6-7

Message d'erreur par défaut de Zope

```
File C:\Program Files\WebSite\lib\python\ZPublisher\Publish.py, line 112,
➥in call_object
  (Object: test)
File C:\Program Files\WebSite\lib\python\OFS\DTMLDocument.py, line 189,
➥in __call__
  (Object: test)
File C:\Program Files\WebSite\lib\python\DocumentTemplate\DT_String.py,
➥line 538, in __call__
  (Object: test)
File C:\Program Files\WebSite\lib\python\DocumentTemplate\DT_Util.py, line 334,
➥in eval
```

```
    (Object: _['mouli.net'].zbook)
    (Info: _)
  File <string>, line 0, in ?
  File C:\Program Files\WebSite\lib\python\DocumentTemplate\DT_Util.py, line 138,
  ➥in careful_getattr
  File C:\Program Files\WebSite\lib\python\OFS\DTMLMethod.py, line 261, in validate
    (Object: test)
  File C:\Program Files\WebSite\lib\python\AccessControl\SecurityManager.py,
  ➥line 144, in validate
  File C:\Program Files\WebSite\lib\python\AccessControl\ZopeSecurityPolicy.py,
  ➥line 225, in validate
Unauthorized: (see above)
```

Cette information est très utile pour le programmeur, car elle lui permet de savoir plus ou moins précisément où se situe l'erreur dans son code. Cet extrait de code correspond à la pile des appels de méthodes au moment où l'erreur s'est produite. Elle se lit de haut en bas, et les informations sont présentées par ordre chronologique.

Les informations les plus utiles qui y figurent sont les mentions `(Object: xxx)`, où `xxx` représente l'id d'un objet Zope : elles permettent de localiser précisément quel objet a provoqué une faute (dans notre cas, le dernier objet accédé étant `test`, il y a de fortes chances que ce soit lui qui ait provoqué l'erreur).

Cet historique ne permet pas de déboguer du code DTML (le numéro de la ligne fautive n'y est même jamais indiqué pour du code DTML), mais, au moins, de cerner globalement l'erreur.

Remarque

Il est indispensable, dans un site en production, de désactiver l'affichage de l'historique : pour cela, il suffit d'enlever l'option `–D` *au démarrage de Zope (voir le script* `start` *situé dans le répertoire d'installation de Zope).*

En fait, quand on procède ainsi, l'historique n'est pas supprimé, mais simplement placé en commentaires HTML (entre `<!--` *et* `-->`*), pour permettre au programmeur d'y accéder en utilisant l'option « Voir le code source de la page » de son navigateur.*

Avertir le webmaster

À titre d'exemple, voici un `standard_error_message` qui envoie automatiquement un e-mail au développeur, contenant les informations dont ce dernier a besoin pour analyser le problème :

```
<dtml-var standard_html_header>

<dtml-if error_message>
 <dtml-var error_message>
<dtml-else>

<TABLE BORDER="0" WIDTH="100%">
```

```
<TR VALIGN="TOP">
<TD>
  <H2>Erreur du serveur</H2>

  <P>
  <STRONG>Error Type: <dtml-var error_type></STRONG><BR>
  <STRONG>Error Value: <dtml-var error_value></STRONG><BR>
  </P>

</TD></TR>
</TABLE>

<dtml-sendmail smtphost="smtp.ma-boutique-de-jouets.com">
To: errormaster@ma-boutique-de-jouets.com
From: Zope Error<errormaster@ma-boutique-de-jouets.com>
Subject: Zope Error on <dtml-var SERVER_NAME>
Content-Type: text/html;
   charset="iso-8859-1"
Content-Transfer-Encoding: quoted-printable
<HTML><BODY>
An error occured on <dtml-var SERVER_URL> :<BR><BR>
<B>URL:</B> <dtml-var URL><BR>
<B>Date:</B> <dtml-var ZopeTime><BR>
<B>Error Type:</B> <dtml-var error_type><BR>
<B>Error Value:</B> <dtml-var error_value><BR>
<BR>
<HR>
TraceBack:<BR>
<BR>
<PRE>
<dtml-var error_tb>
</PRE>
<HR>
REQUEST:<BR>
<dtml-var REQUEST>
</BODY></HTML>
</dtml-sendmail>

</dtml-if>

<dtml-var standard_html_footer>
```

Gestion des exceptions

Zope permet au programmeur d'intercepter les erreurs qui pourraient se produire dans du code DTML. Il lui est ainsi possible de traiter des cas où il sait qu'une erreur peut se produire.

Considérons le cas où l'utilisateur doit remplir un formulaire contenant un champ `nom`. Dans le code de traitement du formulaire, le programmeur exécutera sans doute l'instruction `<dtml-var nom>`, qui peut produire une erreur si le champ `nom` n'est pas rempli. Pour y parer, le programmeur pourrait commencer son formulaire par un `<dtml-unless>` pour traiter ce cas.

Mais la gestion des exceptions permet de se concentrer sur le code applicatif et de déléguer la gestion des erreurs à d'autres parties du code : cela rend le code plus lisible et plus aisé à maintenir.

Les balises `<dtml-try>`, `<dtml-except>` et `<dtml-finally>`

À titre de démonstration, créez un `DTML Document` appelé `test`, contenant simplement la ligne suivante :

```
<dtml-var nom>
```

Lorsqu'on accède à ce `DTML Document`, on obtient immanquablement une erreur (`KeyError` / `nom`). Les balises `<dtml-try>`, `<dtml-except>` et `<dtml-finally>` permettent de traiter les erreurs. On peut utiliser soit `<dtml-finally>`, soit `<dtml-except>`, mais pas les deux en même temps. Nous allons voir comment ces balises fonctionnent.

Le code situé entre `<dtml-try>` et `<dtml-except>` est exécuté. Si aucune erreur ne se produit, le code situé entre `<dtml-except>` et `</dtml-try>` est ignoré, et l'exécution du code DTML se poursuit normalement après la balise `</dtml-try>`.

Si une erreur survient entre `<dtml-try>` et `<dtml-except>`, l'exécution du code DTML est immédiatement suspendue, et le code situé entre `<dtml-except>` et `</dtml-try>` est exécuté. Puis, l'exécution reprend après la balise `</dtml-try>`.

Voici, par exemple, comment on peut indiquer à l'utilisateur qu'il n'a pas saisi sa variable :

```
<dtml-try>
  <dtml-var nom>
<dtml-except>
  Vous n'avez pas saisi votre nom !
</dtml-try>
```

La balise `<dtml-finally>` fonctionne d'une manière à peu près similaire mais, même si aucune erreur ne se produit, le code compris entre `<dtml-finally>` et `</dtml-try>` est exécuté. C'est l'endroit idéal pour inclure du code permettant par exemple d'afficher un bouton Aide. Ne rien mettre entre `<dtml-finally>` et `</dtml-try>` revient à ignorer l'erreur.

Bien entendu, il est possible d'utiliser des instructions DTML au sein des blocs `<dtml-try>`, et même d'imbriquer les blocs `<dtml-try>`.

Spécifier le type d'erreur à traiter

Il est parfois pratique de traiter des erreurs spécifiques parmi celles qui se produisent. Par exemple, dans notre cas, nous souhaiterons intercepter les erreurs de type `KeyError`, mais laisser Zope traiter normalement les autres. Il est possible, avec la balise `<dtml-except>`, de spécifier le type d'erreur à traiter. Ainsi :

```
<dtml-try>
  <dtml-var nom>
<dtml-except KeyError>
  Vous n'avez pas saisi votre nom !
</dtml-try>
```

Il est possible de traiter de cette manière plusieurs types d'erreur, en rajoutant à chaque fois une balise `<dtml-except>` :

```
<dtml-try>
  (instructions DTML)
<dtml-except KeyError>
  Il manque une donnée !
<dtml-except TypeError>
  Erreur de type !
<dtml-except>
  Autre erreur
</dtml-try>
```

Dans cet exemple, la balise `<dtml-except>` qui ne spécifie pas de type permet d'intercepter toutes les autres erreurs ; elle doit être placée en dernier. Enfin, la balise `<dtml-else>` peut être utilisée après les balises `<dtml-except>`, pour traiter le cas où aucune erreur n'est rencontrée.

Déclencher ses propres erreurs : `<dtml-raise>`

Parallèlement à la gestion des erreurs par les blocs `<dtml-try>`, le programmeur a la possibilité d'ordonner lui-même le déclenchement d'une erreur au moyen de la balise `<dtml-raise>`. Ici encore, si l'erreur survient dans un bloc de gestion des exceptions, elle y est traitée ; sinon, un message d'erreur est affiché par Zope, en application des règles que nous venons de préciser.

La balise `<dtml-raise>` requiert un attribut `type`, spécifiant le type d'erreur à déclencher. Le contenu de la balise indique, pour sa part, la valeur de l'erreur. Par exemple, pour simuler une erreur de type `KeyError` sur la variable `nom`, on peut écrire :

```
<dtml-raise type="KeyError">
  nom
</dtml-raise>
```

L'attribut `type` est implicite ; on pourrait aussi écrire :

```
<dtml-raise KeyError>
  nom
</dtml-raise>
```

Les messages d'erreur sont des chaînes de caractères (sans espaces ni caractères spéciaux), définies par l'utilisateur. Il est également possible de spécifier des erreurs du standard HTTP. Par exemple, le code suivant ordonne une redirection du navigateur vers le site officiel de Zope :

```
<dtml-raise redirect>
  http://www.zope.org
</dtml-raise>
```

Dans ce cas, le code d'erreur HTTP approprié est renvoyé au navigateur.

Remarque

Ce code est strictement équivalent à `<dtml-call "RESPONSE.redirect('http://www.zope.org')">`.

Dans cette optique, un type d'erreur mérite une attention particulière : il s'agit de l'erreur `Unauthorized`. Lorsqu'elle est déclenchée, elle provoque l'affichage par le navigateur de la boîte de dialogue qui demande les nom et mot de passe de l'utilisateur. C'est le moyen idéal pour proposer à un utilisateur de se connecter sous un autre nom (et c'est la méthode utilisée par Zope dans la partie supérieure des pages de l'environnement de développement).

```
<dtml-raise Unauthorized>


</dtml-raise>
```

Attention

Cette méthode ne permet pas de déconnecter réellement l'utilisateur courant : il suffit de cliquer sur le bouton Annuler de la boîte de dialogue demandant le mot de passe, puis de cliquer sur Précédent dans le navigateur pour retrouver les droits précédemment acquis.

Pour se déconnecter réellement, il faut entrer un nom d'utilisateur et un mot de passe non valides : dans ce cas, et dans ce cas seulement, le navigateur efface les informations d'authentification qu'il avait précédemment stockées.

Le format .stx (*structured text*)

Pour clore ce chapitre, nous allons étudier le structured text.

Le structured text est un type de formatage de texte, défini par Digital Creations, avec lequel on peut aisément convertir en HTML un texte qui est correctement structuré. Zope supporte nativement et très simplement le structured text. Très facile à apprendre, il permet à des utilisateurs qui ne connaissent pas HTML d'écrire des documents relativement complexes (incluant niveaux de titres, paragraphes, gras, souligné, italique, liens et tableaux) sans faire appel à la moindre balise HTML.

Conversion du structured text en HTML

En général, un document écrit en structured text sera contenu, dans Zope, dans un DTML Document (ce n'est pas une obligation : n'importe quel objet ou n'importe quelle variable peut contenir du structured text). Nous allons commencer par une petite démonstration :

1. Dans un Folder vide, créer une DTML Method appelée index_html et contenant simplement le code suivant :

```
<dtml-var Document fmt="structured-text">
```

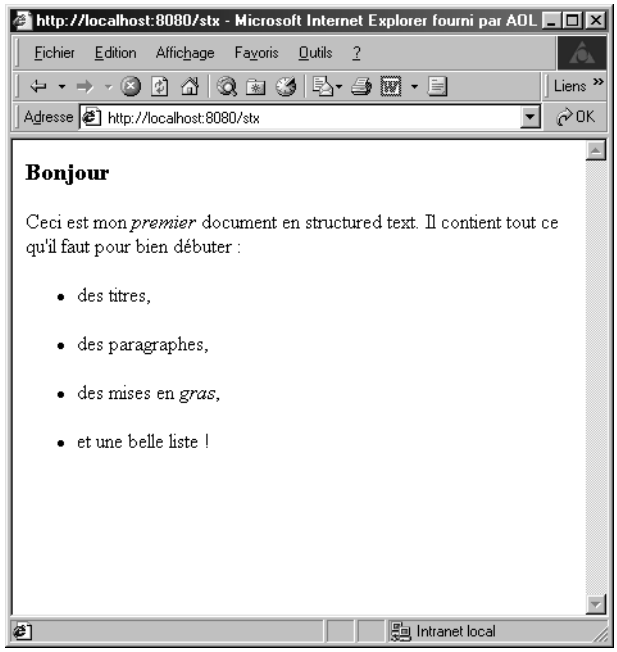
2. Créer un DTML Document appelé Document et contenant le texte suivant (en respectant les sauts de lignes et les espaces) :

```
Bonjour

  Ceci est mon *premier* document en structured text.
  Il contient tout ce qu'il faut pour bien débuter :

    * des titres,

    * des paragraphes,

    * des mises en *gras*,

    * et une belle liste !
```

3. Affichez index_html dans un navigateur. La page HTML présentée à la figure 6-8 s'affiche.

Ainsi, pour convertir un objet ou une variable contenant du structured text en HTML, il suffit d'utiliser l'attribut fmt="structured-text" dans la balise <dtml-var> correspondante – et c'est tout.

Figure 6-8
Un premier document en `structured text`



Syntaxe du structured-text

La syntaxe du `structured text` obéit à des règles très simples. Chaque paragraphe doit être séparé du précédent par une ligne vide, et chaque niveau d'indentation définit un niveau de titre.

Les documents sont formatés d'après les règles suivantes :

Mise en forme	Structured text	HTML
Paragraphes	`texte`	`<P>texte</P>`
	`texte`	`<P>texte</P>`
Titres (sur plusieurs niveaux)	`titre`	`<H1>titre</H1>`
	`    contenu (sur plusieurs lignes)`	`<P>contenu</P>`
Titres (sur plusieurs niveaux)	`titre`	`<H1>tite</H1>`
	`    sous-titre`	`<H2>sous-titre</H2>`
	`    contenu (sur plusieurs lignes)`	`<P>contenu</P>`

Mise en forme	Structured text	HTML

Listes non-numérotées (sur plusieurs niveaux)	`* élément`
	`ou`
	`- élément`
	`ou`
	`oª élément`
		`<ul>`
		`  <li>élément</li>`
		`  ...`
		`</ul>`

Listes numérotées (sur plusieurs niveaux)	`1 élément`
	
	`2 élément`
	
	`3 élément`
		`<ol>`
		`  <li>élément</li>`
		`  ...`
		`</ol>`

Listes numérotées (sur plusieurs niveaux)	`1.1. élément`
	`1.2. élément`
	`ou`
	`1.a. élément`
	`1.b. élément`
		`<ol>`
		`  <li>élément</li>`
		`  ...`
		`</ol>`

Descriptions	`élément -- description`	`<dl><dt>Élément</dt><dd>` `<p>Description</p>`

Exemples (nécessaire pour éviter que des exemples de code ne soient interprétés)	`description::`
	
	`  exemple`
	`ou`
	`"exemple"`
		`<p>description:</p>`
		
		`<pre>exemple</pre>`

Emphase	`*texte*`	`<EM>texte</EM>`
Emphase forte	`**texte**`	`<STRONG>texte</STRONG>`
Soulignement	`_texte_`	`<U>texte</U>`

Liens	`"Lien":URL (absolue ou relative)`
	`ou`
	`"Lien", URL`
		`<a href="URL">Lien</a>`

Liens internes (marques)	`.. [r1] Texte (en début de ligne)`	`<a name="r1">[r1]</a> Texte`
Liens internes (renvois)	`Lien interne [r1]`	`Lien interne` `<a href="r1">[r1]</a>`

a. Il s'agit de la lettre « o » en minuscule.

Remarque

Bien que son apprentissage et son utilisation ne présentent pas de difficulté, le format `stx` *requiert beaucoup de méticulosité dans l'utilisation des espaces.*

Le `structured text`, à l'apprentissage relativement naturel, est donc le format de document de prédilection pour les utilisateurs qui ne connaissent pas forcément HTML.

En résumé…

Dans ce chapitre, nous avons vu les outils que propose le DMTL pour gérer :

- les arborescences HTML automatisées, au moyen de la balise `<dtml-tree>` ;
- l'envoi automatique d'e-mails en DTML, lors de la saisie de formulaires par exemple ;
- l'affichage d'une page d'erreur à l'utilisateur : `standard_error_message` ; les traitements que l'on peut y effectuer (envoi d'un e-mail au webmaster, par exemple) ;
- la manière de se protéger de certaines erreurs, de les ignorer ou de les déclencher (gestion des exceptions avec `<dtml-try>`, `<dtml-except>` et `<dtml-finally>`) ;
- le formatage de page simplifié, proposé par le `structured-text`.

Dans le prochain chapitre, nous allons ouvrir Zope au monde extérieur, grâce à son support natif de SQL.