

# Zope 3 Cookbook

Tarek Ziadé - [ziade.tarek@gmail.com](mailto:ziade.tarek@gmail.com)



# Table des matières

Introduction	iii
1 Comprendre les interfaces	1
2 Ecrire des tests unitaires pour Python	7
3 Écrire des tests fonctionnels	13
4 Comprendre les adaptors	19
5 Comprendre les évènements	23
6 Écrire du reSTructuredText pour la documentation et les doc-tests	27
7 Trouver le bon rythme entre codage, documentation, tests fonctionnels et unitaires	31
8 Enregistrer une session utilisateur pour les tests	35
9 Organiser le code dans un paquet	39
10 Créer un flux RSS pour tous les conteneurs	45
11 Récupérer l'URL d'un objet	51
12 Mettre en place une file asynchrone d'envoi de mails	53



# Introduction

Ce livre contient des recettes pour Zope 3.2.1, issues du projet [zope-cookbook.org](http://zope-cookbook.org). Zope 3 continue son évolution mais la majorité de ces recettes restent d'actualité.

Ce livre a été écrit par Tarek Ziadé et est distribué en licence C-C Attribution-NonCommercial-NoDerivs 2.5

<http://creativecommons.org/licenses/by-nc-nd/2.5/>



# Chapitre 1

## Comprendre les interfaces

Les interfaces sont à la base de la programmation orienté composants, et à fortiori à la base de toutes les briques de Zope 3. Cette recette en présente le principe.

### Comprendre le mécanisme

Historiquement, les interfaces sont nées de la standardisation des fichiers d'en-tête du langage C. La séparation de la description et de l'implémentation permet de présenter les définitions du code, et de masquer les détails d'implémentation aux utilisateurs. L'évolution du code sous-jacent peut se faire dans ce cas indépendamment du reste du système.

Cette technique a ensuite été intégrée et standardisée avec quelques nuances dans la plupart des langages orientés objets modernes, comme Java via ses *protocols*, ou Delphi, sur un modèle proche de C++.

Python ne propose pas *encore* de système d'interfaces, malgré de nombreuses discussions dans la communauté sur le sujet ces cinq dernières années. Pour palier à ce manque, Zope a implémenté son propre système d'interfaces, qui a été ensuite adopté par d'autres frameworks comme *Twisted*.

Dans Zope, une interface définit un contrat, composé d'attributs et de méthodes. Une classe peut *implémenter* cette interface, en fournissant le code pour l'ensemble de cette signature. On dit que la classe *respecte* l'interface. Une même classe peut bien sûr implémenter plusieurs interfaces.

L'intérêt de cette mécanique prend tout son sens dans Zope, lorsque les outils d'interactions manipulent ces abstractions au lieu de manipuler les objets qui les implémentent. Les directives ZCML par exemple, peuvent associer des actions à des interfaces. Les objets publiés déclenchant ces actions s'ils implémentent les dites interfaces.

### Définir des interfaces

Zope fourni dans le paquet *zope.interface* un ensemble d'outils pour déclarer et manipuler des interfaces.

Elements de définition d'interfaces :

- *Interface* : interface de base ;

- *Attribute* : attribut de base ;
- *invariant* : contrainte appliquée aux objets implémentant une interface donnée.

## Interface

*Interface* est une classe un peu particulière, comparable à la classe de base *object* de Python. Elle doit être utilisée comme classe de base pour toute définition d'interface. Une interface est donc une classe, mais sans aucun code : seules les signatures de méthodes y sont définies.

L'interface *IDechire* :

```
>>> from zope.interface import Interface
>>> class IDechire(Interface):
...     def tout(langage):
...         """spécifie si langage fourni en entrée déchire"""
...     
```

Notes:

Dans les définitions d'interface, l'attribut *self* disparaît

Le corps des méthodes n'est composé que du docstring

Une fois définie, cette interface peut être implémentée par une classe, qui s'associe avec la directive *implements* .

La classe *Dechire* :

```
>>> from zope.interface import implements
>>> class Dechire(object):
...     implements(IDechire)
...     def tout(self, langage):
...         if langage == 'python':
...             return True
...         return False
...
>>> dechire = Dechire()
>>> dechire.tout('python')
True
>>> dechire.tout('java')
False
>>> dechire.tout('ruby')
False
```

## Attribute

Outre les méthodes, des attributs peuvent être associés à une interface, pour étendre la signature. *Attribute* est une classe qui s'instancie avec une simple chaîne de caractères.

L'interface *IDocument* :

```
>>> from zope.interface import Attribute
>>> class IDocument(Interface):
...     titre = Attribute('Titre du document')
...     description = Attribute('Description du document')
...     def recuperer():
...         """renvoi titre et description"""
... 
```

Les classes qui implémentent l'interface doivent, comme pour les méthodes, fournir les attributs.

La classe Document :

```
>>> class Document(object):
...     implements(IDocument)
...     def __init__(self, titre, description):
...         self.titre = titre
...         self.description = description
...     def recuperer(self):
...         return self.titre, self.description
... 
```

```
>>> mon = Document("Recette d'omelette", "Avec les secrets de Mireille")
>>> mon.titre
"Recette d'omelette"
```

## invariant

La fonction *invariant* permet d'associer une contrainte à une interface. Toutes les instances d'objet dont la classe implémente une interface sont alors passées à un objet *callable*, qui peut contrôler si l'objet est qualifié.

Si l'objet n'est pas qualifié, une erreur dérivée de *zope.interface.Invalid* doit être levée.

Prenons l'exemple d'une interface définissant des Livres, et ne s'appliquant que sur les livres dont le nombre de pages ne dépassent pas 250.

Contraintes sur ILivre :

```
>>> from zope.interface import Invalid
>>> class BibleError(Invalid):
...     def __repr__(self):
...         return "BibleError(%r)" % self.args
... 
```

```
>>> def taille_max(ob):
...     if ob.pages > 250:
...         raise BibleError(ob)
... 
```

```
>>> from zope.interface import invariant
>>> class ILivre(Interface):
...     pages = Attribute('nombre de pages')
...     invariant(taille_max)
...     def contenu(page):
...         """renvoi le contenu de la page donnée"""
```

```

...
>>> class Livre(object):
...     implements(ILivre)
...     def __init__(self, pages):
...         self.pages = pages
...     def contenu(self, page):
...         return "takalacheter"
...

```

*taille\_max* contrôle que l'objet de type *Livre* ne pèse pas plus de 250 pages. Dans le cas contraire, la fonction déclenche une erreur *BibleError*.

Les instances de *Livre* peuvent être ensuite validées par le biais de la méthode *validateInvariants* d'Interface.

Tests sur l'invariant :

```

>>> programmation_python = Livre(570)
>>> programmation_python.pages
570
>>> programmation_python.contenu(45)
'takalacheter'
>>> ILivre.validateInvariants(programmation_python)
Traceback (most recent call last):
...
BibleError: <...Livre object at ...>
>>> paris_matsh = Livre(46)
>>> ILivre.validateInvariants(paris_matsh)

```

## Manipuler les interfaces

Outre la fonction *implements* qui permet d'associer une classe à une interface, *zope.interface* fournit un accès similaire à un mapping.

Manipulations sur IDocument :

```

>>> list(IDocument)
['titre', 'recuperer', 'description']
>>> IDocument['recuperer']
<zope.interface.interface.Method object at ...>
>>> titre = IDocument['titre']
>>> titre
<zope.interface.interface.Attribute object at ...>
>>> titre.__name__
'titre'
>>> titre.__doc__
'Titre du document'

```

Un ensemble de fonctions utilitaires, et de méthodes associées à *Interface*, permet également de manipuler les interfaces.

Les plus utiles étant :

- *Interface.providedBy(object)* : renvoie vrai si *object* implémente l'interface.

- *implementedBy(class)* : renvoie la liste des interfaces implémentées par *class* .
- *directlyProvides(class, interface)* : permet de spécifier par code que *class* implémente directement *interface* , sans avoir à modifier le code de *class* pour y ajouter une directive *implements* .
- *directlyProvidedBy(class)* : renvoie un objet qui liste les interfaces associées à *class* par *directlyProvides* .

Exemples de manipulation d'interfaces :

```
>>> class ILeFait(Interface):
...     def bien():
...         """fait un calcul savant"""
...
>>> class JeLeFais(object):
...     implements(ILeFait)
...     def bien(self):
...         return 1 + 1
...
>>> ok_ca_va = JeLeFais()
>>> ok_ca_va.bien()
2
>>> ILeFait.providedBy(ok_ca_va)
True
>>> from zope.interface import directlyProvides
>>> from zope.interface import directlyProvidedBy
>>> class JauraisPuLeFaire(object):
...     def bien(self):
...         return 2 + 2
...
>>> directlyProvides(JauraisPuLeFaire, ILeFait)
>>> list(directlyProvidedBy(JauraisPuLeFaire))
[<InterfaceClass ....ILeFait>]
```

## Comprendre le rôle des interfaces dans Zope

La quasi totalité des directives ZCML se basent sur l'association entre une interface et un comportement. Ces comportements sont à leur tour souvent rattachés à une autre interface, qui cible le code capable d'exécuter la demande.

Le framework Zope, qui manipule des éléments concrets, comme les objets persistants ou les classes en charge de l'affichage, se charge de traduire les abstractions définies par les interfaces et de les appliquer au contexte d'exécution réel. Il invoque par le biais des fonctions vues dans cette recette, les bons objets aux bons moments. Cette décorrélation permet de faire évoluer indépendamment une portion de code de la manière dont elle s'utilise et s'articule avec d'autres portions dans un contexte donnée.



## Chapitre 2

# Ecrire des tests unitaires pour Python

Dans un de ces pamphlets sur les avantages de Python, Bruce Eckel oppose le typage statique fort à la programmation dirigée par les tests (TDD), en expliquant que les erreurs relevées avant l'exécution du programme par les langages compilés ne garantissent pas pour autant une meilleure qualité.

La programmation dirigée par les tests n'est pas simplement une solution de secours adoptée par les langages à typage dynamique comme Python, pour retrouver les erreurs qui ne le seraient qu'au moment de l'exécution : c'est aussi une philosophie de programmation, qui augmente drastiquement la productivité des développeurs et la qualité des programmes. Cette technique est aussi adoptée dans les environnements compilés, lorsque les développeurs prennent conscience des gains obtenus par cette approche.

Cette recette résume le principe du TDD et son utilisation avec Python.

## Comprendre le TDD

La programmation dirigée par les tests part du postulat que chaque portion de code écrite doit être accompagnée de tests, qui valident que les cas d'utilisation fonctionnent correctement.

Une fonction *moyenne()* par exemple, peut être testée avec une série de jeux d'essais.

*Fonction moyenne et ses tests :*

```
>>> def moyenne(*elements):
...     return sum(elements) / len(elements)
...
>>> assert moyenne(5, 9, 0, 7) == 5
>>> assert moyenne(5) == 5
>>> assert moyenne(9, 18, 0) == 9
```

Ces essais peuvent être ensuite regroupés dans une fonction, dédiée au test de *moyenne*.

test\_moyenne :

```
>>> def test_moyenne():
...     assert moyenne(5, 9, 0, 7) == 5
...     assert moyenne(5) == 5
...     assert moyenne(9, 18, 0) == 9
...
>>> test_moyenne()
```

Construire en parallèle la fonction de test et la fonction testée, a plusieurs avantages :

- Le développeur prend du recul par rapport au code qu'il est en train de mettre au point, et évacue 90% des erreurs et incohérences par ce simple regard d'*utilisateur* sur son code.
- Les tests, si ils ne sont pas faits en parallèle, sont rarement faits par la suite.
- La non-régression est assurée par cet exercice : ajouter du code, et son test, puis relancer l'ensemble des tests, permet de garantir que le code modifié fournit toujours les fonctionnalités précédemment disponibles.

Lorsqu'un bug est découvert, la fonction de test est enrichie du cas d'utilisation qui provoque l'erreur *avant* de la corriger. Une fois que le test a reproduit exactement l'erreur, le code est corrigé.

Dans notre exemple, l'erreur pourrait être une division par zéro, si une séquence vide est passée. La correction consiste à ajouter ce cas dans les tests, à vérifier que l'erreur se reproduit, puis à corriger le code.

Correction du bug de division par zéro :

```
>>> elements = []
>>> moyenne(*elements)
Traceback (most recent call last):
...
ZeroDivisionError: integer division or modulo by zero
>>> def test_moyenne():
...     assert moyenne(5, 9, 0, 7) == 5
...     assert moyenne(5) == 5
...     moyenne(9, 18, 0) == 9
...     elements = []
...     assert moyenne(*elements) == 0
...
>>> test_moyenne()
Traceback (most recent call last):
...
ZeroDivisionError: integer division or modulo by zero
>>> def moyenne(*elements):
...     if len(elements) == 0:
...         return 0
...     return sum(elements) / len(elements)
...
>>> test_moyenne()
```

Notes:

Les tests précédents sont bien sûr conservés et relancés, pour éviter toute régression et capitaliser les combinaisons.

Cette technique offre enfin un avantage important dans les projets où de nombreux développeurs interviennent : les tests représentent un filet de sécurité et une documentation supplémentaire pour la modification d'une portion de code non connue.

Présenter le principe du TDD dans une section de chapitre est très court, et la lecture de *Programmation Python (T.Ziadé / Eyrolles)*, qui s'attarde plus longuement sur le sujet, peut être un bon complément.

## Ecrire des tests unitaires avec les outils Python

Python offre dans sa bibliothèque standard des outils de conception de tests pour accélérer le travail : *unittest*, un portage vers Python de *JUnit*, et *doctest*, un système original de tests basés sur les docstrings.

### **unittest**

Le module *unittest* fournit des *helpers* pour écrire des tests, principalement sous la forme d'une classe inspirée du modèle *JUnit*.

Cette classe peut être utilisée comme classe de base pour regrouper des tests et former *une campagne de tests*, et offrir des amorces au début et à la fin des tests. Ces amorces, appelés *text fixture*, permettent de mettre en place, lorsque c'est nécessaire, un environnement de test.

Les tests de l'exemple précédent peuvent être écrits sous la forme d'une classe, sauvegardée dans un module Python. Chaque module de code *code.py* possède en général un module de test *test\_code.py* associé.

le fichier *test\_moyenne.py*

```
import unittest
# la fonction moyenne est dans le fichier
# moyenne.py
from moyenne import moyenne

class TestMoyenne(unittest.TestCase):

    def test_moyenne_simple(self):
        res = moyenne(5, 9, 0, 7)
        self.assertEqual(res, 5)

    def test_moyenne_un_element(self):
        self.assertEqual(moyenne(5), 5)

    def test_moyenne_vide(self):
        self.assertEqual(moyenne(*[]), 0)

if __name__ == '__main__':
    unittest.main()
```

Toutes les méthodes commençant par le mot *test* sont exécutés lorsque le script est appelé.

Lancement de *test\_moyenne.py* :

```
tziade@dabox:~/Desktop$ python test_moyenne.py
...
```

```
-----
Ran 3 tests in 0.000s
```

```
OK
```

Les modules de tests sont regroupés, pour chaque paquet Python, dans un sous-répertoire *tests*, qui peut être scanné par un script de lancements de tests, qui regroupe et exécute tous les modules dont le nom commence par *test*.

## Les doctests

Une autre technique pour la conception des tests est basée sur l'utilisation des commentaires de code, les *docstrings*. Des sessions de prompt, telles que l'on pourrait en écrire depuis l'invite Python, peuvent être insérées.

tests par *docstrings* pour *moyenne* :

```
>>> def moyenne(*elements):
...     """ fait une moyenne, et la fait bien
...     >>> moyenne(5, 9, 0, 7)
...     5
...     >>> moyenne(5)
...     5
...     >>> moyenne*[]
...     0
...     """
...     if len(elements) == 0:
...         return 0
...     return sum(elements) / len(elements)
... 
```

Cette écriture offre une lisibilité accrue des tests, qui apparaissent comme des petits essais de prompt. Cependant l'écriture des tests à proximité du code, présenté parfois comme un avantage par certains développeurs, a la facheuse tendance d'obfusquer la lisibilité du module : lorsque les tests s'allongent, le code est noyé dans des alternances de *docstrings*.

La parade est de séparer ces doctests dans un fichier texte, à la manière des tests unitaires classiques, pour bénéficier des avantages de cette écriture, sans rendre le code illisible.

*docstrings* séparés dans un fichier texte :

```
Moyenne
=====
```

```
Moyenne calcul la moyenne d'une séquence de valeurs::
```

```
>>> moyenne(5, 9, 0, 7)
5
>>> moyenne(5)
5
```

Moyenne est sympa, et ne se vautre pas lamentablement sur les séquences vides

```
>>> moyenne(*[])
0
```

Même si ce genre d'appel est effectué par les esprits les plus tordus.

Cette écriture a également l'avantage de pouvoir alterner explications et séquences de code. On parle de *documentation agile*.

Cette forme est la plus utilisée et la préférée des développeurs Zope 3, qui peuvent ainsi documenter leur code et le tester en même temps.

Les tests unitaires classiques sont réservés dans ce cas aux tests de plus bas niveau, qui nécessitent du code de préparation, et polluerait ainsi la documentation si ils étaient fait en doctests.



## Chapitre 3

# Écrire des tests fonctionnels

Les tests fonctionnels, ou "acceptance tests", sont complémentaires aux tests unitaires : ce sont des boîtes noires qui testent les fonctionnalités du système complet, conformément à chaque besoin énoncé.

Ces tests ne se concentrent que sur l'utilisation du système, comme le ferait un utilisateur lambda, pour s'assurer que les fonctionnalités attendues sont présentes et fonctionnent.

Ils deviennent également un témoin de qualité inégalable pour l'utilisateur final, qui peut parfois écrire ou étendre lui-même ces tests.

En terme de proportion, il est admis qu'un système devrait avoir approximativement 2/3 de tests unitaires et 1/3 de tests fonctionnels, ces derniers étant de plus haut niveau.

Pour les applications Web, ces tests nécessitent un environnement d'exécution particulier, qui imite au mieux un navigateur, et fourni des Apis pour effectuer des requêtes et étudier les réponses, comme le ferait Firefox ou Internet Explorer.

## Comprendre le fonctionnement

Un test fonctionnel par définition, ne se concentre que sur les fonctionnalités d'une application, sans se soucier des détails d'implémentation, contrairement aux tests unitaires. C'est le test pratiqué par l'utilisateur final, qui peut par exemple parcourir les menus de l'application, pour tester chacun d'entre eux. Il valide alors, muni d'une check-list, que chaque fonctionnalité attendue est bien présente et se comporte comme souhaité. Le système est alors "validé" si la check-list n'a aucun point manquant ou non conforme.

Les tests fonctionnels en Python et a fortiori avec Zope, se font par le biais de classes de tests dérivées des classe de tests unitaires, ou dans des doctests.

Entre tests unitaire et fonctionnel, seule l'interface entre le système et le testeur change : ce dernier doit se mettre dans les mêmes conditions que l'utilisateur.

## Faire des tests fonctionnels avec des classes

Zope fourni un module spécifique dans le paquet `zope.app.testing` appelé *functional*, qui fournit des classes dérivées de `unittest.TestCase`. La classe la plus utilisée est `BrowserTestCase`.

Elle fournit principalement les méthodes suivantes :

- `getRootFolder()` : renvoie le dossier racine de Zope. Utilisé pour manipuler le site avant d'analyser une publication.
- `publish(path, basic=None)` : renvoie le rendu d'un objet pour le chemin `path` sous la forme d'un objet `Response`. Si `basic` est fourni, c'est une chaîne contenant le login et le mot de passe, sous la forme `'login :password'`. La chaîne `'mgr :mgrpw'` peut être employée pour s'authentifier avec les droits manager.
- `checkForBrokenLinks(body, path, basic=None)` : vérifie que `body`, qui est une chaîne de caractère représentant le corps d'une réponse (récupéré par `Response.getBody()`), ne contient pas de liens cassés. `path` fourni le chemin relatif qui correspond à l'objet à l'origine de la réponse. Ce paramètre permet à la méthode de se repérer sur le site pour les références relatives.

L'objet `Response` est un objet renvoyé par le publisher et augmenté par les outils de tests de quelques méthodes facilitant la lecture. On retiendra surtout :

- `getBody()` : renvoie le corps de la réponse
- `getOutput()` : renvoie la réponse complète (corps et en-têtes)
- `getStatus()` : renvoie le statut
- `getHeaders()` : permet de récupérer les en-têtes

Dans l'exemple ci-dessous, un dossier est créé dans le dossier racine, et son rendu vérifié.

Classe de vérification du rendu d'un dossier :

```
>>> import transaction
>>> from zope.app.testing import functional
>>> from zope.app.folder import Folder
>>> class FolderTest(functional.BrowserTestCase):
...     def test_folder(self):
...         root = self.getRootFolder()
...
...         # create a Folder
...         root['cool_folder'] = Folder()
...         transaction.commit()
...
...         # get the root content through the publisher
...         response = self.publish('/cool_folder', basic='mgr:mgrpw')
...         self.assert_(' <title>Z3: cool_folder</title>' in
...                        response.getBody())
```

Notes:

Le code `transaction.commit()` permet de valider la modification effectuée sur le site, pour la rendre disponible lors de l'appel à `publish`.

Ce test est exécuté comme un test unitaire classique.

Lancement du test :

```
>>> import unittest
>>> suite = unittest.makeSuite(FolderTest)
>>> test_runner = unittest.TextTestRunner()
>>> test_runner.run(suite)
<unittest._TextTestResult run=1 errors=0 failures=0>
```

## Faire des tests fonctionnels par doctests

Les tests fonctionnels sont des tests de très haut niveau, et se rapprochent beaucoup des exemples de code qu'une documentation peut contenir. Il est possible en Python d'associer ce besoin documentaire avec les tests par le biais des doctests.

### Comprendre les doctests de Python

L'originalité de Python est de fournir un système de tests basé sur les docstrings : le code à tester est écrit dans des docstrings, sous la forme de séquences de prompt Python.

#### Exemple de doctest :

```
>>> def laVerite():
...     """
...     >>> laVerite()
...     'les doctests, ca claque'
...     """
...     return 'les doctests, ca claque'
```

Le module `doctest` de Python fournit des outils pour extraire et exécuter ces tests.

#### Exécution du doctest :

```
>>> import doctest
>>> doctest.run_docstring_examples(laVerite, globals(), verbose=True)
Finding tests in NoName
Trying:
    laVerite()
Expecting:
    'les doctests, ca claque'
ok
```

Cependant, les tests contenus dans les doctests de chaque module, classe et fonction parasitent la lisibilité du code. Pour éviter cette obfuscation, il est possible de les regrouper dans un fichier texte séparé du code, et "exécuter" ce fichier. Les exemples peuvent alors être agrémentés d'explications et suivre un déroulement logique. On parle alors de "documentation agile", ou "documentation exécutable".

Ces documents peuvent ensuite être appelés comme module de test.

#### Document agile :

```
Le module verite
=====
```

Ce document ne contient pas de tests fonctionnels mais plutôt un test unitaire.

Ce n'est pas bien grave.

Le module `verite` contient une fonction `laVerite()` qui peut être appelée pour afficher un message. Cette méthode ne sert pas à grand chose.

Exemple d'utilisation::

```
>>> from verite import laVerite
>>> laVerite()
'les doctests, ca claque'
```

### Comprendre les doctests fonctionnels dans Zope

Pour manipuler facilement Zope dans les doctests, le paquet `zope.testbrowser` fournit une classe `Browser` qui permet de simuler un navigateur et ses fonctionnalités. Par convention, cette classe est invoquée avec une URL qui commence par `http://localhost`, puis le chemin sur le serveur Zope.

Atteindre la page à la racine peut donc s'écrire ainsi :

```
>>> from zope.testbrowser.testing import Browser
>>> browser = Browser('http://localhost')
>>> browser.url
'http://localhost'
>>> browser.contents
'...<title>Z3: </title>...'
```

Notes:

La propriété `contents` permet de récupérer le contenu de la page sous forme d'une chaîne de caractères.

Les `'...'` (ou Ellipsis) ont une signification précise dans les doctests : ils servent à remplacer une portion de texte quelconque. Dans l'exemple ci dessus, ils permettent de s'assurer que `contents` contient bien la portion HTML indiquée.

`Browser`, outre `contents` et `url`, fournit les éléments principaux suivants :

- `open(url)` : permet d'ouvrir l'URL fournie.
- `isHtml` : le contenu est-il du HTML ?
- `title` : renvoie directement le titre de la page.
- `headers` : renvoie un objet Python `httplib.HTTPMessage` contenant les en-têtes.
- `getLink(text)` : récupère un objet `Link` pour le lien représenté par `text`. Ce lien peut ensuite être suivi par sa méthode `click()`, pour recharger la nouvelle page dans l'instance de `Browser`.

Des éléments supplémentaires sont proposés mais cet outil est présenté plus en détail dans d'autres recettes.

## Limitations des tests fonctionnels

Les tests fonctionnels Web ont une limitation inhérente au fonctionnement

des navigateurs : il est impossible de tester le javascript contenu dans les pages car chaque navigateur implémente son propre moteur. Il faut utiliser d'autres techniques, basées par exemple sur le système de test *Selenium*, qui lance les tests depuis un navigateur réel.



## Chapitre 4

# Comprendre les adapters

Lorsque les applications grossissent, un effet de bord commun à toutes les technologies, est le mélange progressif du code. Au gré des contextes d'échanges entre objets, des interdépendances se mettent en place et le code devient vite *un plat de spagethis* .

Les Adapters, qui sont une implémentation du *principe de substitution de Liskov*, apportent une solution souple et formalisée, pour lutter contre ce phénomène.

## Comprendre la mécanique

Les Adapters fournissent un mécanisme de coopération entre un objet et un contexte d'exécution particulier et se basent sur les interfaces. Ils permettent de rendre des objets compatibles avec une interface donnée en fournissant une couche d'encapsulation qui met en place cette compatibilité.

Ce mécanisme, que l'on retrouve dans les systèmes comme QueryAdapter de COM (Microsoft), permet de réunir des objets dans un contexte fonctionnel précis. On parle de *mécanisme de glue*.

Ce principe présente plusieurs avantages :

- Il permet de réunir des objets dans un contexte pour lesquels ils n'avaient pas été initialement prévus, sans avoir à les modifier ou les rendre interdépendants.
- Il offre un moyen souple de capitaliser des fonctionnalités génériques qui s'appliquent à plusieurs familles de classes.

Les Adapters sont une formalisation du *duck typing* et retrouvent leurs racines dans le *PEP 246* et dans des implémentations tierces, comme *PyProtocols*

## Définir un adapter : IClassDoc

Les Adapters se construisent avec une instance de classe et implémentent une interface particulière.

Prenons l'exemple de *IClassDoc* , une interface qui définit des méthodes pour récupérer une documentation formatée d'un objet de n'importe quel type de classe.

L'interface IClassDoc :

```
>>> from zope.interface import Interface
>>> class IClassDoc(Interface):
...     def getMethods():
...         """ renvoie un texte qui liste les méthodes de la classe """
...     def getAttributes():
...         """ renvoie un texte qui liste les attributs de la classe """
... 
```

Un adapter pour cette interface l'implémente, pour tout type d'objet.

Un adapter pour IClassDoc :

```
>>> from zope.interface import implements
>>> class ClassDoc(object):
...     implements(IClassDoc)
...     def __init__(self, context):
...         self.context = context
...     def getMethods(self):
...         methods = [(element, getattr(self.context, element).__doc__)
...                     for element in dir(self.context)
...                     if callable(getattr(self.context, element))]
...         for method, doc in methods:
...             print '%s: %s' % (method, doc)
...     def getAttributes(self):
...         attributes = [element
...                       for element in dir(self.context)
...                       if not callable(getattr(self.context, element))]
...         for attribute in attributes:
...             print attribute
... 
```

```
>>> objet = ['une', 'liste']
>>> doc = ClassDoc(objet)
>>> doc.getAttributes()
__doc__
>>> doc.getMethods()
__add__: x.__add__(y) <==> x+y
...
append: L.append(object) -- append object to end
count: L.count(value) -> integer -- return number of occurrences of value
extend: L.extend(iterable) -- extend list by appending elements from the iterable
index: L.index(value, [start, [stop]]) -> integer -- return first index of value
insert: L.insert(index, object) -- insert object before index
pop: L.pop([index]) -> item -- remove and return item at index (default last)
remove: L.remove(value) -- remove first occurrence of value
reverse: L.reverse() -- reverse *IN PLACE*
sort: L.sort(cmp=None, key=None, reverse=False) -- stable sort *IN PLACE*;
cmp(x, y) -> -1, 0, 1
```

## Enregistrer l'adapter pour IClassDoc dans le système

Zope 3 propose un système de registre pour les Adapters, qui centralise les classes Adapters pour les interfaces données. L'enregistrement de l'adapter se fait via une directive ZCML qui associe une classe à une interface. Cette classe est appelée *factory*.

L'adapter peut aussi, en fonction des besoins, adapter une Interface vers une autre.

Les paramètres de la directive *adapter* sont :

- *factory* : la ou les classes adapter ;
- *provides* : l'interface fournie par la ou les Adapters ;
- *for* : les classes et interfaces qui sont adaptées ;
- *permission* : une permission associée qui filtre l'accès à l'adapter ;
- *name* : le nom associé ;
- *trusted* : booléen, qui détermine si l'adapter est *digne de confiance*. Dans ce cas, les éventuels contrôles de sécurités sur les objets appelés par l'adapter sont retirés ;
- *locate* : Booléen, qui permet de rendre l'adapter local. (\*\*todo\*\*explain here)

Enregistrement de l'adapter pour IClassDoc :

```
>>> from zope.configuration import xmlconfig
>>> ignored = xmlconfig.string("""
...     <configure
...         xmlns='http://namespaces.zope.org/zope'>
...         <include package="zope.app.component" file="meta.zcml" />
...         <adapter
...             factory="recipe.ClassDoc"
...             for="*"
...             provides="recipe.IClassDoc"
...             />
...     </configure>
... """)
```

Notes:

La directive *include* n'est ici que pour faire fonctionner les tests, et n'a pas à être ajouté dans un environnement d'exécution normal.

## Utiliser l'adapter IClassDoc

l'adapter pour IClassDoc peut être invoqué par le biais de *zapi.getAdapter(object, interface=Interface, name="", context=None)* , qui prend les paramètres suivants :

- *object* : l'objet à adapter ;
- *interface* : l'interface souhaitée ;
- *name* : le nom de l'adapter, si spécifié ;
- *context* : le contexte, pour recherche les Adapters locaux ;

Utilisation de l'adapter :

```
>>> from zope.app import zapi
>>> la_coupe = ['est', 'pleine']
>>> la_coupe_doc = zapi.getAdapter(la_coupe, IClassDoc)
>>> la_coupe_doc.getMethods()
__add__: x.__add__(y) <==> x+y
...
append: L.append(object) -- append object to end
count: L.count(value) -> integer -- return number of occurrences of value ...
```

## Chapitre 5

# Comprendre les évènements

Lorsqu'une action doit être associée à un évènement particulier de l'application, et que cet évènement peut être généré par plusieurs composants, des interdépendances risquent de se mettre en place si les objets à l'origine de l'évènement sont directement associés au code responsable de l'action.

La programmation par évènements, présentée dans cette recette, résoud cette problématique par le biais du design pattern *Observer*.

## Comprendre le design pattern Observer

La solution la plus courante pour éviter une dépendance entre les acteurs qui provoquent un évènement donné et ceux en charge de l'exécution de code lorsque cet évènement se produit est d'utiliser le design pattern (DP) *Observer*.

Ce DP met en place un système de notification où chaque acteur s'enregistre en tant qu'observateur d'un évènement donné dans une classe dédiée à la gestion d'évènements. Les acteurs qui provoquent l'évènement préviennent cette classe intermédiaire qui à son tour prévient tous les observateurs.

### Gestionnaire d'évènement basique en Python

```
>>> class EventManager(object):
...     observers = {}
...     def subscribe(self, event_id, caller):
...         if event_id in self.observers:
...             if caller not in self.observers[event_id]:
...                 self.observers[event_id].append(caller)
...         else:
...             self.observers[event_id] = [caller]
...     def notify(self, event_id):
...         if event_id not in self.observers:
...             return
...         for caller in self.observers[event_id]:
...             caller()
...
>>> manager = EventManager()
```

```

>>> class Observer(object):
...     def __init__(self, id):
...         self.id = id
...         manager.subscribe('bougeotte', self)
...     def __call__(self):
...         print '%s: il a bougé !' % self.id
...
>>> class Bougeur(object):
...     def __call__(self):
...         print 'allez, je bouge'
...         manager.notify('bougeotte')
...         print "oui j'avoue"
...
>>> bougeur = Bougeur()
>>> camera1 = Observer('camera 1')
>>> camera2 = Observer('camera 2')
>>> bougeur()
allez, je bouge
camera 1: il a bougé !
camera 2: il a bougé !
oui j'avoue

```

Cette décorrélation permet d'implémenter des comportements sans avoir à rendre dépendantes les classes actrices. Dans Zope, ce mécanisme est beaucoup utilisé et de nombreux événements sont disponibles d'office, comme :

- La création d'un objet;
- la suppression ou le déplacement d'un objet;
- etc.

## Utiliser l'implémentation d'observer dans Zope

Le package *zope.component* implémente une directive *subscriber* qui permet d'associer une fonction à une interface représentant un événement. Lorsqu'une instance d'objet qui implémente cette interface est notifiée par le biais de la fonction globale *zope.event.notify*, la fonction est appelée.

Dans l'exemple ci-dessous, *groupie\_1* et *groupie\_2* sont appelés lorsque qu'un événement *ILeMessie* se produit.

Evenements Zope :

```

>>> from zope.interface import Interface, implements, Attribute
>>> class ILeMessie(Interface):
...     nom = Attribute('Nom du messie')
...     groupies = Attribute('Nombre de groupies')
...
>>> class LeMessie(object):
...     implements(ILeMessie)
...     def __init__(self, nom):
...         self.nom = nom
...         self.groupies = 0
...

```

```

>>> def groupie_1(event):
...     print 'Longue vie à toi, %s !' % event.nom
...     event.groupies += 1
...
>>> def groupie_2(event):
...     print 'Gloire à %s !' % event.nom
...     event.groupies += 1
...
>>> from zope.configuration import xmlconfig
>>> ignored = xmlconfig.string("""
...     <configure
...         xmlns='http://namespaces.zope.org/zope'>
...         <include package="zope.app.component" file="meta.zcml" />
...         <subscriber
...             handler="recipe.groupie_1"
...             for="recipe.ILeMessie"
...         />
...         <subscriber
...             handler="recipe.groupie_2"
...             for="recipe.ILeMessie"
...         />
...     </configure>
... """)
>>> jose = LeMessie('José')
>>> from zope.event import notify
>>> notify(jose)
Longue vie à toi, José !
Gloire à José !
>>> jose.groupies
2

```

La directive *subscriber* propose des configurations plus riches pour associer l'évènement à un adapter, utilisées dans le noyau de Zope. Cependant le mode de fonctionnement présenté ci-dessus couvre tous les besoins de gestion d'évènements d'une application Zope.

## Les évènements courants de Zope

Zope fourni dans *zope.app.event.interfaces*, *zope.app.component*, *zope.app.publication* et *zope.app.container*, une série d'interfaces-évènements, notifiées par le framework dans ses manipulations bas niveau, à savoir :

- IObjectEvent : interface de base. Déclenché pour tout évènement sur un objet ;
- IObjectCreatedEvent : un objet a été créé ;
- IObjectCopiedEvent : un objet a été copié ;
- IObjectModifiedEvent : un objet a été modifié ;
- IObjectAnnotationsModifiedEvent : Les annotations d'un objet ont été modifiées ;
- IObjectContentModifiedEvent : le contenu d'un objet a été modifié ;

- IRegistrationEvent : interface de base. Déclenché pour tout évènement sur les enregistrements ;
- IRegistrationActivatedEvent : un enregistrement a été activé.
- IRegistrationDeactivatedEvent : un enregistrement a été désactivé.
- IObjectMovedEvent : un objet a été déplacé dans un conteneur.
- IObjectAddedEvent : un objet a été ajouté dans un conteneur.
- IObjectRemovedEvent : un objet a été supprimé dans d'un conteneur.
- IContainerModifiedEvent : un réordonnancement, un ajout ou une suppression s'est produit dans un conteneur.
- IBeforeTraverseEvent : le *publisher* s'apprête à effectuer un traversal.
- IEndRequestEvent : le *publisher* a terminé le calcul d'une requête.

## Chapitre 6

# Écrire du reSTructuredText pour la documentation et les doctests

L'ensemble de la documentation des paquets de Zope est réalisée au format reSTructuredText, ou reST, qui est une évolution du StructuredText. Ce format fournit une structure simple, qui n'alourdit pas le texte, contrairement à LaTeX, et peut être lu directement. Des outils permettent ensuite de manipuler ce contenu pour le transformer en HTML, LaTeX, XML ou PDF.

Cette recette est une *cheatsheet* minimale, qui permet d'écrire des doctests destinés à des paquets Zope, une documentation plus complète pouvant être trouvée à cette URL : <http://docutils.sourceforge.net/rst.html>.

### Structurer le fichier

Pour organiser un fichier en section, sous-section, sous-sous-section, etc., il suffit de souligner le titre de la section avec un caractère dans l'ensemble :

```
= - ' : ' " ~ ^ _ * + # < >
```

A chaque section, si un nouveau caractère est utilisé, reST l'associe au niveau donné pour le reste du document.

Exemple de structure reST :

```
Titre  
=====
```

```
Ok, donc j'ai deux sections. Profitez-en.
```

```
Section 1  
~~~~~
```

Le titre est un frimeur. j'en parle dans une sous-section.

Sous-Section 1

-----

Je suis chargé par ma section de casser du sucre sur le dos de Titre, mais je n'en ferais rien, je vise une promotion.

Section 2

~~~~~

Calmons nous, le document est fini de toute manière...

## Insérer des exemples de code

Insérer du code se fait en l'indentant d'au minimum 1 caractère (en général 2 ou 4). Ce bloc doit être précédé du signe " : " et d'une ligne vide, puis suivi d'une ligne vide.

*Insertion de code :*

Le module Math

=====

Python dispose d'un module math, qui fournit une fonction pour les puissances::

```
>>> import math
>>> math.pow(56, 3)
175616.0
```

Il y a aussi pi::

```
>>> math.pi
3.1415926535897931
```

Notes:

Zope 3 indente de deux caractères, mais certains développeurs préfèrent une indentation sur 4 caractères, peut être pour éditer plus facilement ces fichiers dans leur éditeur de code Python.

## Insérer des liens et formater le texte

reST permet aussi d'afficher du texte en italique, en gras.

*Gras et italique :*

Recette

=====

Il n'y a absolument pas de **gras** dans les plats 'italiens'.

La notation italique est utilisée pour mettre en exergue un module ou tout élément de code dans le texte. La notation grasse est réservée à la mise en valeur de mots du texte.

Pour les liens, les deux types les plus courants sont les liens complets vers des ressources Web (URL), et les liens vers une autre partie du document.

Les URL peuvent être associés à un texte mis entre simple côtes, et suivit d'un espace souligné. Ce texte est ensuite répété en bas de document, précédé de deux points, un espace, puis un espace souligné, puis de l'URL précédé du symbole " :".

Exemple d'URL :

Ordonnance  
=====

Matin et soir : un tour sur 'zope-cookbook'\_  
Midi: Ne pas oublier la pilule.

.. \_'zope-cookbook': http://zope-cookbook.org

Notes:

La référence en bas de page disparaît bien sûr, lorsque le texte est par exemple généré en HTML ou PDF.

Pour définir des liens dans le document même, le même type de marquage est employé, sauf pour la section cible : la phrase répétée est suivie uniquement de " :".

Exemple de références dans le document :

Ordonnance  
=====

Matin et soir : un tour sur 'zope-cookbook'\_  
\_'pilule':  
Midi: Ne pas oublier la pilule.

\_'zope-cookbook':  
Zope-cookbook.org est un site très sympa. Mais avant d'y aller,  
avez vous pensé à votre 'pilule'\_ ?

## Adopter de bonnes pratiques

Voici une liste de bonnes pratiques en vrac :

- Vérifier à ce que le fichier reST soit toujours syntaxiquement valide, avec *rest2html* par exemple ;
- regrouper les liens en fin de fichier ;
- les fichiers reST constituent la documentation du paquet, ils doivent être particulièrement soignés ;
- Les en-têtes de modules, de classes et autres fonctions peuvent aussi adopter ce format.



## Chapitre 7

# Trouver le bon rythme entre codage, documentation, tests fonctionnels et unitaires

Coder, tester, coder, tester, documenter, coder...

Le rythme des activités de développement n'est pas évident à équilibrer pour un développeur novice. En fin de projet, tests et documentation passent souvent à la trappe, victimes du stress du développeur, qui préfère écourter ces étapes pour rendre son travail dans les temps, même si il sait pertinemment qu'il le payera par la suite. Ces périodes sont pourtant les plus importantes pour le code (remaniement d'apis, suppression de pans complets de code, etc..)

Il est possible de limiter ces phénomènes et gagner en efficacité, par de bonnes pratiques décrites dans cette recette. Elles s'appliquent évidemment à Zope, mais peuvent être utilisées dans n'importe quel projet Python.

### Comprendre les rôles du développeur

Le développeur, dans la quasi totalité des projets, a trois activités :

- Développer ;
- documenter ;
- tester ;

La première phase consiste bien sûr à créer le code. Un cycle de tests fonctionnels plus poussés peut suivre cette phase, puis un retour vers le développement est envisageable, pour des modifications, évolutions ou debugguages. Ces cycles de vie logiciels sont plus ou moins équivalents dans toutes les méthodologies existantes, comme le RUP (IBM Rational Unified Process). Ce sujet dépasse cependant le cadre de la recette.

## Créer le code

Pour concevoir une fonctionnalité, un développeur récupère un cahier des charges, composé de spécifications fonctionnelles, et parfois de spécifications techniques. En se basant sur ce document, la création du code s'effectue en trois étapes :

- Transcrire le cahier des charges en fichier texte (doctest) ;
- insérer des exemples de codes dans ce texte, qui répond aux besoins ;
- développer le code sous-jacent aux exemples.

### Transcrire le cahier des charges en doctest

Transcrire en doctest les besoins exprimés se fait en synthétisant au maximum le cahier des charges, et en décrivant la structure du paquet qui va être développé.

Prenons l'exemple d'un module en charge de récupérer des flux RSS pour les stocker dans des objets Python persistants, puis les rendre affichables dans l'application. Le développeur décide d'organiser son paquet en trois modules :

- *rssfetcher.py* : module en charge de la récupération des flux RSS ;
- *rssobject.py* : module en charge de la création et du stockage des données ;
- *rssview.py* : module en charge de l'affichage des données.

Il commence par écrire quatre fichiers doctests, qui expriment cette structure :

- *README.txt*
- *rssfetcher.txt*
- *rssobject.txt*
- *rssview.txt*

Le fichier *README.txt* décrit en quelques phrases l'objectif du paquet, et présente sa structure.

Fichier *README.txt*

```
rssreader
=====
```

Ce paquet permet de présenter des flux rss. Il est organisé en trois composants :

- *rssfetcher*: sait lire un flux RSS 1.0
- *rssobject*: stocke le flux et fourni des méthodes d'accès
- *rssview*: sait afficher un flux dans le navigateur

Chaque module est présenté à son tour dans un doctest.

Fichier *rssfetcher.txt*

```
rssfetcher
=====
```

Le module *rssfetcher* transforme un fichier XML au format RSS 1.0 vers une structure Python exploitable. Il est invoqué avec l'URL d'un flux et renvoi une liste contenant les entrées du flux.

A ce stade, le développeur n'a pas encore écrit de code, mais dispose d'une spécification technique et architecturale exploitable. Cette étape permet parfois de soulever des problématiques de logique des spécifications fournies.

## Insérer des exemples de codes

Sur la base de ces fichiers textes, le développeur peut commencer à préparer le développement des modules, en explicitant la face visible de ces derniers, c'est à dire les API qui seront utilisées dans d'autres modules.

Modification du fichier *rssfetcher.txt*

```
rssfetcher
=====
```

Le module `rssfetcher` transforme un fichier XML au format RSS 1.0 vers une structure Python exploitable. Il est invoqué avec l'URL d'un flux et renvoi une liste contenant les entrées du flux.

Il fourni une fonction `'fetchFeed(source)'` qui renvoie une structure Python en fonction d'une URL ou d'un fichier ::

```
>>> from rssfetcher import fetchFeed
>>> fetchFeed('tests/rss.xml')
[{'Subject': 'entry one', 'content': "le contenu est ici"}]
```

Notes:

Le paquet doit prévoir dans un sous répertoire un fichier échantillon *rss.xml*, contenant un flux utilisé pour les tests.

Ce principe est appliqué pour l'ensemble des modules du paquet, susceptibles de fournir des fonctionnalités.

## Développer les fonctionnalités

Le doctest précédent ne peut bien sûr fonctionner que si l'implémentation est réalisée. Le développeur conçoit donc le module *rssfetcher.py* jusqu'à ce que le test du doctest passe. Ce développement nécessitera certainement la création d'autres tests unitaires, qui seront conçus classiquement dans des classes dans le sous-répertoire de *tests*.

Lorsque tous les modules sont conçus, *README.txt* contient un exemple complet d'utilisation, qui simule généralement quelques directives ZCML et quelques échanges avec le publisher.

## Maintenir et faire évoluer le code

L'évolution du paquet, que ce soit dans le cadre d'une correction, ou d'une modification, doit se faire de la même manière : les fichiers textes sont revus en premiers, puis le code est modifié en conséquence.

Les corrections de bogues sont traités d'une manière particulière : un test unitaire qui reproduit le problème est ajouté dans les classes de tests, et le code invoqué corrigé. Ajouter ces tests dans les doctests dénature leur objectif secondaire : servir de documentation.

Cependant, si il est explicitement demandé que les bugfix soient documentés, et facilement lisibles, un doctest *bugfix.txt* peut être ajouté. Ce n'est pas le cas pour Zope 3 par exemple, car le couple tracker+svn fourni une bonne tracabilité des bugfix.

## Chapitre 8

# Enregistrer une session utilisateur pour les tests

L'écriture d'un test fonctionnel est souvent dictée par un scénario d'utilisation précis, ou "user story", où chaque étape est réalisée via l'interface homme-machine fournie. Dans notre cas, l'interface est bien sûr le navigateur.

Programmer une session Web complète par code est fastidieux.

Cette recette présente des techniques pour enregistrer des actions réalisées dans un vrai navigateur, et les récupérer pour accélérer l'écriture des tests fonctionnels.

### Connaître le paquet `zope.app.recorder`

Zope 3 fournit `zope.app.recorder`, qui permet d'ajouter un publisher particulier au fichier `zope.conf`, en charge d'enregistrer les manipulations de l'utilisateur. Le problème avec cet outil est que le résultat est composé des échanges HTTP bruts. Recycler ces résultats pour les transformer en données facilement exploitables avec, par exemple `zope.testbrowser`, peut être aussi fastidieux.

### Utiliser le paquet `zope.testrecorder`

Zope Corp a développé un petit outil, appelé `zope.testrecorder`, qui offre le même style de fonctionnalité, mais avec des sorties dans deux formats plus facilement recyclables :

- *Python Doctest*, directement exploitable avec `zope.testbrowser` ;
- *Selenium Test*, qui peut être utilisé avec `Selenium`.

### Installer

L'outil ne fait pas parti de Zope 3, et doit être récupéré séparément. Il ne dispose pas non plus d'un site Web, ou d'une distribution par archive. Il faut donc récupérer ce paquet directement sur le subversion de Zope.org, et le placer dans le répertoire `/lib/python` de l'instance de Zope.

Récupération de *zope.testrecorder* par subversion

```
tziade@dabox:~$ cd /home/zopes/zope3/lib/python/
tziade@dabox:/home/zopes/zope3/lib/python$ svn co svn://svn.zope.org/repos/main/zope.
A   testrecorder/html
A   testrecorder/html/recorder.js
[...]
A   testrecorder/testrecorder.py
Révision 65919 extraite.
```

Le paquet est ensuite présenté à Zope par le biais d'un fichier d'initialisation, appelé *testrecorder-configure.zcml*, placé dans le répertoire *etc/package-includes* de l'instance, avec le contenu ci-dessous.

Le fichier *testrecorder-configure.zcml*

```
<include package="testrecorder" />
```

Zope, en redémarrant, possède un nouveau dossier nommé *recorder*, et déclaré comme ressource pointant sur le répertoire *html* du paquet.

**Enregistrer une session**

Une fois installé, *testrecorder* est accessible via `/@@/recorder/index.html`. Cette page présente une bannière où le testeur peut saisir une URL et commencer à enregistrer une session. La page visitée apparaît sous la bannière et le testeur peut circuler sur le site.

Le bouton *Stop Recording* permet d'arrêter l'enregistrement.

**Récupérer les résultats**

Une fois la session effectuée, *testrecorder* propose deux sorties, qui apparaissent dans le navigateur. La sortie *Python Doctest* est la plus fréquemment utilisée, puisqu'elle permet de récupérer une portion de code Python pouvant être utilisée telle quelle dans un doctest.

Exemple de sortie de type *doctest*

```
=====
Doctest generated Sun Mar 12 2006 17:07:22 GMT+0100 (CET)
=====
```

Create the browser object we'll be using.

```
>>> from zope.testbrowser import Browser
>>> browser = Browser()
>>> browser.open('http://localhost/')
>>> browser.getLink('++etc++site').click()
>>> browser.getLink('[top]').click()
>>> browser.getLink('Folder').click()
>>> browser.getControl(name='new_value').value = 'MyFolder'
>>> browser.getControl('Apply').click()
>>> browser.getLink('MyFolder').click()
```

```
>>> browser.open('http://localhost/')
>>> browser.getLink('[top]').click()
>>> browser.getLink('Folder').click()
>>> browser.getControl(name='new_value').value = 'test'
>>> browser.getControl('Apply').click()
>>> browser.getLink('test').click()
```

Cette portion de doctest peut ensuite être modifiée pour ajouter des tests complémentaires, pour vérifier des éléments sur chacune des pages de la session.



## Chapitre 9

# Organiser le code dans un paquet

Un paquet est composé de fichiers Python et ZCML, parfois de Page Templates et de ressources, que ce soit des fichiers CSS, Javascript, ou des images.

Cette recette explique comment organiser tout ces éléments, pour rendre le paquet facile à maintenir.

### Comprendre ce qu'est un paquet

La définition d'un paquet en Python est : un répertoire contenant des modules et des sous-répertoire, le tout composant un espace de nom ajouté au chemin de l'interpréteur. Chaque répertoire composant le paquet doit contenir un fichier `__init__.py` pour être pris en compte.

Zope utilise les paquets pour définir son code d'extension. Ces paquets s'appelaient des *Produits* sous Zope 2, puis ont repris le nom plus pythonique de *paquet* sous Zope 3. Comme l'initialisation d'une extension se fait par la lecture des directives ZCML, Zope agit comme Python mais en recherchant pour sa part le fichier `configure.zcml` du paquet.

Aux yeux de Zope, le code peut donc être organisé de n'importe quelle manière dans le paquet, du moment qu'il est lié dans le fichier ZCML. Mais pour faciliter la maintenance et l'évolution, un minimum de structuration est nécessaire, basé sur quelques conventions.

### Définir une structure minimum

Le plus petit paquet possible est composé d'un fichier `__init__.py` vide, et d'un fichier `configure.zcml` contenant au minimum une directive `configure` vide.

Le plus petit fichier `configure.zcml` possible :

```
<configure\>
```

Ces deux fichiers sont réunis dans un dossier, qui est déclaré dans un fichier de configuration placé dans le dossier *INSTANCE/etc/package-includes*. Zope parcourt au démarrage *package-includes* pour connaître la liste des paquets à initialiser. Les fichiers de ce dossier doivent avoir un nom avec un suffixe *-configure.zcml*, et utilisent en général le nom du paquet. Le paquet *dummypackage* aura donc comme fichier de déclaration *dummypackage-configure.zcml*.

Le fichier *dummypackage-configure.zcml* :

```
<include package="dummypackage"/>
```

Au démarrage, Zope intégrera le premier paquet nommé *dummypackage* trouvé dans son chemin. La meilleure place pour conserver les paquets est *INSTANCE/lib/python* .

## Définir une structure plus avancée

Le paquet *dummypackage* va maintenant implémenter une fonctionnalité simple, qui se base sur une structure typique : un nouveau type d'objet qui conserve en interne le nombre de fois qu'il a été appelé par une vue. Le paquet a donc besoin de définir :

- Une déclaration d'interface ;
- une classe de contenu.
- une classe de vue ;

### Déclaration d'interface

Toutes les déclarations d'interface d'un paquet sont réunies dans un fichier unique nommé *interfaces.py*. Cette convention offre :

- Une manière unique de faire référence à des interfaces. Par exemple, l'attribut *for* d'une directive *browser :view* est toujours de la forme : *package.interfaces.IMyInterface*.
- Des facilités pour organiser l'héritage entre les interfaces.

L'interface de l'exemple se nomme *IDummyCounter* et est ajoutée au fichier *interfaces.py* .

*interfaces.py* :

```
from zope.interface import Interface

class IDummyCounter(Interface):

    def count():
        """ returns the counter value
        after it has been inc'ed """
```

### Classe de contenu

La classe de contenu gère le compteur, et dérive d'une classe persistante de base, qui permet son stockage dans la ZODB. Cette classe est sauvegardée dans un fichier portant le même nom.

*dummycounter.py* :

```
import persistent
from zope.interface import implements
from interfaces import IDummyCounter

class DummyCounter(persistent.Persistent):

    implements(IDummyCounter)

    def __init__(self, initial=0):
        self._count = initial

    def count(self):
        self._count += 1
        return self._count
```

*DummyCounter* est déjà fonctionnel.

Tests de *DummyCounter* dans un prompt :

```
>>> from dummypackage.dummycounter import DummyCounter
>>> my_counter = DummyCounter()
>>> my_counter.count()
1
>>> my_counter.count()
2
>>> my_counter.count()
3
```

## Classe de vue

Le code en charge de l'affichage des instances de *DummyCounter* est une simple classe dérivée de *BrowserView*, et appelée *DummyCounterView*. Comme les vues sont déclarées par le biais de directives *browser*, elles sont réunies dans un fichier *browser.py*

*browser.py* :

```
from zope.app.publisher.browser import BrowserView

class DummyCounterView(BrowserView):

    def getCount(self):
        return 'I have been seen %d times' % self.context.count()
```

*getCount* fourni le code spécifique aux besoins d'affichage.

Tests de *DummyCounterView* dans le prompt :

```
>>> from dummypackage.dummycounter import DummyCounter
>>> my_counter = DummyCounter()
>>> from dummypackage.browser import DummyCounterView
```

```
>>> my_counter_view = DummyCounterView(my_counter, None)
>>> my_counter_view.getCount()
'I have been seen 1 times'
>>> my_counter_view.getCount()
'I have been seen 2 times'
>>> my_counter_view.getCount()
'I have been seen 3 times'
```

L'ensemble est ficelé dans le fichier ZCML par le biais de quelques directives *browser*.

configure.zcml final :

```
<configure
  xmlns="http://namespaces.zope.org/zope"
  xmlns:browser="http://namespaces.zope.org/browser"
  i18n_domain="zope"
  >

  <class class=".dummycounter.DummyCounter">
    <require permission="zope.View"
      interface=".interfaces.IDummyCounter" />
  </class>

  <browser:page
    for=".interfaces.IDummyCounter"
    name="counter.html"
    permission="zope.View"
    class=".browser.DummyCounterView"
    attribute="getCount"
    />

  <browser:addform
    schema=".interfaces.IDummyCounter"
    label="Add dummy counter"
    content_factory=".dummycounter.DummyCounter"
    name="AddDummy.html"
    permission="zope.ManageContent"
    />

  <browser:addMenuItem
    class=".dummycounter.DummyCounter"
    title="Dummy Counter"
    permission="zope.ManageContent"
    view="AddDummy.html"
    />

</configure>
```

## Choisir entre structure par modules et dossiers

Pour l'instant, notre paquet est composé de cinq fichiers :

- `__init__.py` : déclare le dossier comme paquet Python
- `configure.zcml` : déclare à Zope les directives ZCML
- `interfaces.py` : groupe les interfaces
- `dummycounter.py` : implémente la classe de contenu
- `browser.py` : implémente la classe de vue

Cette structure est basée sur des modules, et convient bien à l'exemple présenté. Quand le paquet grossit, il est conseillé d'utiliser des sous-dossiers pour simplifier le contenu des fichiers. Ainsi *interfaces* et *browser* deviennent des dossiers. Dans ce cas, le dossier *browser* contient des modules pour les vues et définit son propre fichier *configure.zcml*, inclus dans le fichier ZCML principal. Le dossier *interfaces* définit des modules pour les interfaces.



## Chapitre 10

# Créer un flux RSS pour tous les conteneurs

Les flux RSS (Real Simple Syndication) sont devenus en quelques années un format incontournable pour la syndication de contenu web. Ces rendus XML, respectant un format précis (RSS 1.0, RSS 2.0 ou encore Atom), peuvent être lus par des programmes spécialisés, appelés agrégateurs.

Les agrégateurs, comme *Akregator* sous GNU/Linux, permettent de gérer plusieurs sources RSS et d'offrir un moyen rapide de collecter des informations de plusieurs sites web.

Sous Zope, ces informations sont généralement représentées par des documents regroupés dans un conteneur spécialisé. Plus globalement, tout conteneur est susceptible de présenter un flux RSS pour syndiquer son contenu dans ce format.

Cette recette explique comment concevoir un flux RSS générique, pouvant être utilisé avec tout conteneur.

## Comprendre le principe d'une syndication RSS

Syndiquer le contenu d'un dossier consiste à regrouper pour chaque élément :

- Un titre;
- un lien;
- une description;
- une date de publication

Ce regroupement est exprimé dans une grappe XML.

Exemple d'élément :

```
<item>
  <title>La news</title>
  <link>http://mes.news/la.news.html</link>
  <description>A lire, absolument</description>
  <pubDate>2006-04-28 11:16:51</pubDate>
</item>
```

L'ensemble des éléments est regroupé dans un *canal*, qui fournit aussi un lien, une description, et un titre. Ce contenu est encapsulé dans une balise *rss*.

Un flux rss complet :

```
<rss version="2.0">
  <channel>
    <title>Les news</title>
    <link>http://mes.news/rss</link>
    <description>Des news croustillantes</description>
    <item>
      <title>La news</title>
      <link>http://mes.news/la.news.html</link>
      <description>A lire, absolument</description>
      <pubDate>2006-04-28 11:16:51</pubDate>
    </item>
    <item>
      <title>La news 2</title>
      <link>http://mes.news/la.news.2.html</link>
      <description>Détails croustillants inside</description>
      <pubDate>2006-04-30 11:16:51</pubDate>
    </item>
  </channel>
</rss>
```

Le nombre d'entrée est en général limité, en fonction de la nature des informations servies. Par exemple, un site de nouvelles restreint le flux à dix ou vingt entrées, et présente seulement les plus récentes. Ce contenu est ensuite interprété par des applications clientes, qui collectent et conservent chaque entrée dans une base locale.

Des informations plus précises sur le format RSS peuvent être récupérées sur Wikipédia : [http://fr.wikipedia.org/wiki/Rich\\_Site\\_Summary](http://fr.wikipedia.org/wiki/Rich_Site_Summary).

## Comprendre la norme de métadonnées Dublin Core

Les informations requises pour construire un flux RSS se retrouvent toutes dans des métadonnées décrites dans une norme appelée *Dublin Core*, qui a pour objectif de standardiser les informations accessibles sur chaque document électronique, comme le titre ou la description.

L'ensemble des informations disponibles sont décrites sur le site de la *Dublin Core Metadata Initiative* : <http://dublincore.org/>

Zope implémente, comme la majeure partie des systèmes de publication, la norme Dublin Core, et permet d'accéder directement aux informations, sans avoir à connaître précisément chaque type de document présenté dans le flux.

## Utiliser IZopeDublinCore

Pour l'implémentation du Dublin Core, Zope se base sur les annotations et fournit un adaptateur, *IZopeDublinCore*, qui permet de donner un accès en écriture

et en lecture aux métadonnées pour tout objet respectant l'interface *IAnnotatable*.

Le rendu d'une page Zope se base par exemple sur la métadonnée *title*, stockée dans une annotation associée à l'objet, pour définir le titre de la page.

Modification du titre de la page racine via le Dublin Core :

```
>>> from zope.app.dublincore.zopedublincore import IZopeDublinCore
>>> from zope.testbrowser.testing import Browser
>>> dc = IZopeDublinCore(getRootFolder())
>>> dc.title = u'Mon site Zope'
>>> Browser('http://localhost').title
'Z3: Mon site Zope'
```

La création du flux RSS peut adopter cette stratégie pour récupérer via l'adapter les données à agréger dans le flux XML pour chaque objet.

## Concevoir une vue spécialisée

Le code nécessaire à la conception du flux peut être regroupé dans une vue pour conteneur, qui sait :

- renvoyer le titre, la description et le lien du canal, en lisant les métadonnées du conteneur ;
- qualifier pour le flux les éléments du conteneur compatibles avec l'adapter *IZopeDublinCore* ;
- renvoyer pour chaque élément qualifié les informations.

La vue *FolderRSSView* :

```
>>> from zope.app import zapi
>>> from zope.component import queryAdapter
>>> from zope.app.publisher.browser import BrowserView
>>> class FolderRSSView(BrowserView):
...     def _getItemInfos(self, item):
...         """ returns item metadatas displayed by the feed """
...         dublin_core = IZopeDublinCore(item)
...         link = zapi.absoluteURL(item, self.request)
...         return {'title': dublin_core.title,
...                 'link': link,
...                 'description': dublin_core.description,
...                 'pubDate': dublin_core.created}
...     def getItems(self, size=10):
...         """ returns a list of 'size' entries, ordered by item creation date,
...         """
...         def _itemHasDC(item):
...             if queryAdapter(item, IZopeDublinCore) is not None:
...                 return True
...             return IZopeDublinCore.providedBy(item)
...         items = [(IZopeDublinCore(item).created, item)
...                  for item in self.context.values()
...                  if _itemHasDC(item)]
...         items.sort()
```

```

...         items.reverse()
...         return [self._getItemInfos(item) for created, item in items[:size]]
...     def getChannelTitle(self):
...         """ return channel title """
...         return IZopeDublinCore(self.context).title
...     def getChannelLink(self):
...         """ return channel URL """
...         return '%s/@@rss' % zapi.absoluteURL(self.context, self.request)
...     def getChannelDescription(self):
...         """ return channel description """
...         return IZopeDublinCore(self.context).description
...

```

Cette vue peut ensuite être utilisée sur tout conteneur du site. Dans l'exemple ci-dessous, plusieurs éléments sont ajoutés à la racine.

*FolderRSSView en action :*

```

>>> from zope.app.folder import Folder
>>> from zope.app.file.image import Image
>>> from zope.publisher.browser import TestRequest
>>> root = getRootFolder()
>>> un = Folder()
>>> root['un'] = un
>>> IZopeDublinCore(un).title = u'un'
>>> deux = Image()
>>> root['deux'] = deux
>>> dc_deux = IZopeDublinCore(deux)
>>> dc_deux.title = u'deux'
>>> dc_deux.description = u'belle image de coucher de soleil'
>>> request = TestRequest()
>>> rss_root = FolderRSSView(root, request)
>>> rss_root.getChannelTitle()
u'Mon site Zope'
>>> rss_root.getItems()
[{'link': 'http://127.0.0.1/deux',
  'description': u'belle image de coucher de soleil', 'pubDate': None,
  'title': u'deux'},
 {'link': 'http://127.0.0.1/un',
  'description': u'', 'pubDate': None, 'title': u'un'}]

```

## Concevoir un template XML pour RSS 2.0

La vue peut être combinée à un template XML, pour construire le flux.

*Template XML :*

```

<?xml version="1.0"?>
<rss version="2.0"
  xmlns:tal="http://xml.zope.org/namespaces/tal">
  <channel>
    <title tal:content="view/getChannelTitle"/>

```

```

<link tal:content="view/getChannelLink"/>
<description tal:content="view/getChannelDescription"/>
<item tal:repeat="item view/getItems">
  <title tal:content="item/title"/>
  <link tal:content="item/link"/>
  <description tal:content="item/description"/>
  <pubDate tal:content="item/pubDate"/>
</item>
</channel>
</rss>

```

## Publier la vue dans le ZCML

L'ensemble peut ensuite être exposé via une directive ZCML pour être utilisé sur le site.

*Mise en place de la vue @@rss :*

```

<configure
  xmlns='http://namespaces.zope.org/zope'
  xmlns:browser='http://namespaces.zope.org/browser'
  >
  <browser:page
    for="zope.app.folder.interfaces.IFolder"
    name="rss"
    template="rsstemplate.xml"
    permission="zope.View"
    class="FolderRSSView"
  />
</configure>

```



## Chapitre 11

# Récupérer l'URL d'un objet

Zope 3 propose une approche plus *propre* que la version précédente en ce qui concerne les URL des objets. Un objet donné ne conserve pas son URL comme attribut. Cette recette explique le mécanisme de calcul de l'URL d'un objet et la logique sous-jacente.

### Comprendre le principe

Chaque objet persistant de la ZODB est accédé au travers de vues, qui sont en charge de la publication. L'URL représente la location de l'objet du point de vue du navigateur. Par exemple, si l'utilisateur souhaite afficher l'objet *foo* du répertoire *bar*, il tapera : `http://server/foo/bar`. L'objet est récupéré par le publisher, qui traverse l'arborescence d'objets depuis la racine. Dans notre cas, il traverse *foo* et lui demande *bar*. *foo* fournira *bar* car son attribut `__name__` est *bar*.

Récupérer l'URL d'un objet consiste donc à faire le chemin inverse jusqu'à la racine, en construisant la chaîne avec les noms des objets traversés.

La différence majeure de ce nouveau mécanisme est que *bar* ne contient plus son URL.

### Récupérer l'URL d'un objet

L'URL de tous les objets publiables peut être récupéré par ce mécanisme. Une vue générique *AbsoluteURL* fournit cette fonctionnalité par le nom `absolute_url`.

Dans un ZPT par exemple, l'URL d'un objet peut être récupéré par : `my_object/@@absolute_url`. Dans du code Python, un appel à la fonction `absoluteURL()`, qui permet de récupérer directement le résultat de la vue, peut être utilisé.

Récupérer l'URL d'un objet :

```
>>> root = getRootFolder()
>>> from zope.publisher.browser import TestRequest
>>> request = TestRequest()
>>> from zope.app.traversing.browser.absoluteurl import absoluteURL
```

```
>>> absoluteURL(root, request)
'http://127.0.0.1'
```

## Chapitre 12

# Mettre en place une file asynchrone d'envoi de mails

Envoyer des mails est un besoin récurrent d'une application Web, que ce soit pour les notifications ou encore les envois de formulaires. Cette recette explique comment mettre en place un système asynchrone d'envoi de mails, par le biais d'une file.

### Comprendre le fonctionnement d'une file asynchrone

Dans Zope 2, l'envoi de mails dans les portails CMF était fait par l'utilisation d'un outil *singleton* appelé Mailhost et placé à la racine de l'arborescence. Cet outil effectue un appel vers le module *smtplib* de Python, qui lui-même procède à l'envoi du mail en effectuant une session telnet.

Cette technique a deux gros inconvénients pour de gros systèmes :

- L'envoi de mail peut devenir un goulot d'étranglement ;
- il n'y a aucune garantie sur les envois en cours, en cas de crash système.

Zope 3 propose une solution élégante en implémentant une file producteurs-consommateur : chaque mail à envoyer est ajouté dans la file, qui est elle-même consultée par un thread spécialisé. La file est stockée sur le système sous la forme d'un répertoire de type *Maildir*. Ce format se retrouve dans beaucoup de systèmes de stockage de mails.

Ce procédé évite les ralentissements lors de l'envoi de mails car le thread travaille de manière asynchrone sur la file. De plus, en cas de crash, les mails de la file ne sont pas perdus et sont envoyés au redémarrage du système.

### Mettre en place la file

Le paquet *zope.app.mail* fournit trois directives ZCML :

- *smtplibMailer* : permet de déclarer un mailer smtp ;
- *directDelivery* : permet de déclarer un utilitaire d'envoi de mail direct ;
- *queuedDelivery* : permet de déclarer un utilitaire d'envoi de mail par file.

La mise en place de la file se fait en déclarant un *smtp mailer*, et une file d'envoi.

## Mettre en place le mailer

Le mailer doit être configuré avec un certain nombre de paramètres :

- *name* : Un nom unique, le définissant ;
- *hostname* : le Hostname du serveur SMTP ;
- *port* : le port ;
- *username* : le nom d'utilisateur, en cas d'authentification ;
- *password* : le mot de passe.

Mise en place dans une directive ZCML :

```
>>> from zope.configuration import xmlconfig
>>> ignored = xmlconfig.string("""
...     <configure
...         xmlns='http://namespaces.zope.org/zope'
...         xmlns:mail="http://namespaces.zope.org/mail">
...         <include package="zope.app.mail" file="meta.zcml" />
...         <mail:smtpMailer name="my-mailer" hostname="localhost" port="25" />
...     </configure>
... """)
```

Notes:

La directive *include* n'est ici que pour faire fonctionner les tests, et n'a pas à être ajouté dans un environnement d'exécution normal.

## Mettre en place la file

La file doit être définie une seule fois, sa meilleure place étant le fichier *configure.zcml* principal.

Les paramètres pour la file sont :

- *name* : Son nom ;
- *permission* : la permission associée à l'envoi de mails ;
- *queuePath* : le chemin du dossier Maildir ;
- *mailer* : le nom du mailer à utiliser.

Le dossier *queuePath* est créé au premier démarrage et doit être positionné dans un endroit sécurisé du système.

Ajout de la file :

```
>>> ignored = xmlconfig.string("""
...     <configure
...         xmlns='http://namespaces.zope.org/zope'
...         xmlns:mail="http://namespaces.zope.org/mail">
...         <include package="zope.app.mail" file="meta.zcml" />
...         <mail:queuedDelivery
...             permission="zope.SendMail"
...             queuePath="./mail-queue"
...             mailer="my-mailer"
...         />
...     </configure>
... """)
```

```
...     name="my-mailer-is-rich"/>
...     </configure>""")
```

## Utiliser le système

La file peut être utilisée pour envoyer des mails, en appelant l'utilité *my-mailer-is-rich*.

*Exemple d'utilisation :*

```
>>> from zope.app import zapi
>>> from zope.app.mail.interfaces import IMailDelivery
>>> def mail(subject, body):
...     """ envoie un mail à 'tarek@ziade.org' """
...     msg = u'Subject: %s\n\n%s' % (subject, body)
...     mail_utility = zapi.getUtility(IMailDelivery, 'my-mailer-is-rich')
...     mail_utility.send('cookbook@reader-desk.org',
...                       ['tarek@ziade.org'], msg)
>>> subject = "Hello de LU"
>>> body = ""
... Cette exemple est interessant car:
... 1 - a chaque fois que les tests vont etre lances sur le livre,
...     Tarek recevra un mail. (si le thread se depeche)
... 2 - Si les lecteurs reprennent l'exemple tel quel, il en recevra d'autres
... 3 - Reste a savoir combien de temps il tiendra avant de changer cet exemple.
... """
>>> mail(subject, body)
```

