

Capítulo 12: Desenvolvendo um Produto em Python

Ao desenvolver um produto para o Plone, te permite você fazer quase qualquer coisa que você gostaria de fazer com o Plone. Utilizar o Python para desenvolver tipos de conteúdos ou ferramentas é o melhor modo para te fornecer uma maior flexibilidade. Se você tem em mente construir uma aplicação específica e que ainda não exista nenhum produto já construído e disponível, então está é sua oportunidade para adicionar está característica desenvolvendo um novo produto. Este novo produto poderia armazenar alguns tipos de conteúdos específicos para a empresa onde você trabalha ou algum tipo de manipulação única. No capítulo anterior, mostrei como personalizar um tipo de conteúdo. Esta customização permitiu a voce acrescentar ou suprimir alguns atributos diferentes ao seu tipo de conteúdo, mas provavelmente você gostaria era de desenvolver um novo tipo de conteúdo e não uma adaptação.

Neste capítulo, mostrarei dois exemplos: o tipo de conteúdo e a ferramenta. Ambos os exemplos serão relativamente simples, mesmo assim o deixará preparado para o próximo capítulo, onde mostrarei como utilizar Archetypes, um framework para Plone que permite que você gere tipos de conteúdo de uma forma rápida e simples com pequenos códigos.

Especificamente, criarei um novo tipo de conteúdo para o Plone, mostrando no código todos os passos. Este é um tipo de conteúdo muito interessante que utiliza vários blocos de construção e os incorpora em seu Plone site. Mostrei como criar um tipo de conteúdo, adicionar permissões, procurar a integração e instalação de novas ferramentas no site. Ambos os exemplos neste capítulo estão disponíveis online para você baixar, instalar, e estudar. Além disso, o Apêndice B se encontra a listagem de todos os códigos.

Desenvolvendo um Tipo de Conteúdo Padrão

Para o Web site do livro do Plone (<http://plone-book.agmweb.ca>), gostaria de ser capaz de mostrar, on-line, todos os fragmentos de código utilizados neste livro. Poderia pegar o código e simplesmente colocá-lo dentro de documentos Plone, mas o código seria mostrado sem as sintaxes em destaque. Também, todos os espaços em branco em Python seriam removidos. Um bom produto para o Plone, deveria construir algo mais apresentável. Assim eu preciso de um tipo de conteúdo que possa armazenar o fragmento de código, destacar a sintaxe, e permitir que os usuários o vejam on-line. Figura 12-1 mostra o exemplo de um produto terminado.



Figure 12-1. Um exemplo de script Python enviado para o Plone

Partindo para a concepção deste novo produto que seria itens que armazenariam fragmento de código, poderíamos definir alguns requerimentos. Especificamente, ele deve possuir os seguintes atributos:

- **ID:** Cada item (fragmento de código armazenado) terá um ID único.
- **Title:** Cada item (fragmento de código armazenado) deve ter um título.
- **Description:** Cada item deve ter uma descrição opcional.
- **Source code:** Cada item terá um atributo “código fonte” que contenha um fragmento de código.
- **Language:** A linguagem/script de programação utilizada para escrever o fragmento de código, por exemplo, Perl, Python, HTML - Hypertext Markup Language, e assim por diante.

O tipo de conteúdo pode interagir com o Plone de modo que você possa utilizar a tecnologia do Plone. Você precisará assegurar que o produto possa ser pesquisado, e possa se interagir com segurança, com o workflow, e seja persistido corretamente. Também, seria legal se os usuários pudessem fazer o upload desses fragmentos de código ao invés de ficarem recortando e colando em uma área de texto.

Ao examinar este código, eu precisei encontrar um maneira simples para transformá-lo em HTML. Isto é fácil de fazer para uma linguagem de sintaxe simples como o Python, mas se você poderá fazê-lo para outras diferentes linguagens, tais como HTML (page templates), JavaScript, Cascading Style Sheets (CSS), e assim por diante. Felizmente, o módulo SilverCity faz isto e está disponível no SourceForge (<http://silvercity.sf.net/>). Este módulo utiliza as bibliotecas C do editor de texto Scintilla transformando-o em texto de fácil legibilidade e compreensão. O fato é que não precisamos então nos preocuparmos

muito com a implementação, pois o resultado será um texto com a sintaxe em negrito para uma extensa gama de diferentes linguagens.

Observando a lista de requerimentos, você irá ver que é bastante objetiva. De fato, o ID, título, e descrição estão todas definidas na implementação Dublin Core no Plone. Então você deve apenas se preocupar com o código fonte e a linguagem. O Plone requer um ID e um título, e a lista irá te ajudar a adquirir a descrição.

Iniciar um Tipo de Conteúdo

Agora que você tem uma idéia do que é um tipo de conteúdo então criaremos um neste capítulo, desenvolvendo em Python no sistema de arquivos. Este tipo de conteúdo é também um produto, então você cria um novo diretório (por exemplo: *PloneSilverCity*) no seu diretório *product*. O nome do diretório que você criou é o nome do produto que o Zope irá importar, então nomeie com um nome que não irá te trazer futuras complicações.

Após criar o diretório, normalmente adiciono alguns arquivos e diretórios que necessitaremos. Todos os pacotes precisam de um arquivo `__init__.py` no diretório. O nome deste arquivo vem do Python e indica que este diretório é um pacote Python, portanto importável pelo Zope. Quando o pacote é importado, o Zope executa este arquivo que contém fragmentos de código que possibilita o registro do produto no Zope.

Você pode adicionar também alguns arquivos de texto tal como um *readme.txt*, *install.txt*, e assim por diante. Um outro arquivo de texto que também seria útil é o *refresh.txt*. Este arquivo deixa você juntar no módulo *Refresh* do Zope e carrega dinamicamente o produto como você desenvolveu. Isto será útil para os seus primeiros passos ao escrever uma classe, e mais tarde irei mostrar como configurá-lo no Zope.

Até agora, você criou um diretório chamado *PloneSilverCity* no diretório de produtos e que contém os seguintes arquivos, todos vazios: *readme.txt*, *refresh.txt*, *install.txt*, e `__init__.py`. Este até agora é um pacote válido que não faz absolutamente nada (mas não por muito tempo).

Desenvolvendo com ZClasses

Você está criando um tipo de conteúdo utilizando Python, você provavelmente já ouviu falar sobre ZClasses em alguma documentação ou na Internet. ZClasses é um framework existente no Zope 2 para desenvolvimento de classes pela Web. Muitas pessoas desenvolveram e distribuíram as ZClasses. Porém, não as recomendo. É difícil de desenvolvê-las utilizando as ferramentas existentes, instalá-las no código fonte, distribuí-las, e assim por diante. Quase todas as pessoas que falei sobre as ZClasses concordam que é importante o empenho para aprender como desenvolver com Python.

Se você olhar à documentação ou outra informação relacionada à ZClasses, então recomendo que resista a tentação de utilizá-las. Por esta razão, não há nenhuma história de desenvolvimento utilizando ZClasses. Se você está focado em um modo rápido para desenvolver, então dê uma olhada em Archetypes, que tem uma abordagem um pouco diferente.

Integrando o SilverCity

Antes de você se aprofundar no código Zope, pode ser útil compreender como utilizar o SilverCity. É de suma importância no desenvolvimento de qualquer software, a construção de camadas de códigos que te possibilitam testar. Por esta razão, você deve iniciar com a certeza de que você pode utilizar o SilverCity do módulo Python. Se isso funcionar, você pode seguir em frente, você construiu uma camada e poderá partir para o próximo desafio do desenvolvimento do produto.

Então, observe dentro do SilverCity. Primeiro, você tem que instalar o SilverCity, este módulo corresponde às instruções de instalação para os módulos Python como mostrado no capítulo 10. Para instalar no Windows, baixe o arquivo *SilverCity-0.9.5.win32-py2.3.exe* em <http://silvercity.sf.net> e rode o instalador gráfico. Para instalar no Linux, baixe o arquivo *SilverCity-0.9.5.tar.gz* em <http://silvercity.sf.net>. Então descompacte o arquivo e execute o programa *setup.py*. Por exemplo:

```
$ tar -zxf SilverCity-0.9.2.tgz
$ cd SilverCity-0.9.2
$ python setup.py install
...
```

Após fazer isto, você pode se certificar, com o seguinte comando no prompt do Python, em Windows ou Linux, se o SilverCity esta funcionando:

```
$ python
Python 2.3.2 (#1, Oct 6 2003, 10:07:16)
[GCC 3.2.2 20030222 (Red Hat Linux 3.2.2-5)] on linux2
Digite "help", "copyright", "credits" ou "license" para maior informação.
>>> import SilverCity
>>>
```

Isto significa que o SilverCity foi instalado com sucesso. Se você não teve um resultado parecido e não pôde importar o SilverCity, pare e solucione este caso primeiro; de outra forma nada irá rodar.

Agora você precisa compreender a API - Application Programming Interface para este módulo; Leia o exemplo do script localizado em *PySilverCity/Scripts* chamado *source2html.py*. Este script faz exatamente aquilo que queremos: Ele transforma em HTML um dado fragmento de código. Analise o fragmento de código abaixo:

```
$python source2html.py source2html.py --generator=python
```

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>source2html.py</title>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  <link
```

```

    rel="stylesheet"
    href="default.css" />
</head>
...

```

Isto significa que você apenas precisa fazer algumas leves modificações na API. Adicione um módulo chamado *source.py* no diretório *PloneSilverCity*. Nele você vai desenvolver o código que irá fornecer uma interface para a biblioteca; este novo módulo contém um código específico para o Zope ou o Plone. Este módulo possui três outros módulos principais: diz a você todas as linguagens que são possíveis utilizar, você fornece algum texto e ele retorna este texto com um tratamento específico (uma espécie de análise gramatical), e finalmente ele realmente executa a tradução.

Primeiro, adicione a função *create_generator*, que retornará “correct parser”:

```

from SilverCity import LanguageInfo
from StringIO import StringIO

def create_generator(source_file_name=None, generator_name=None):
    """ Make a generator from the given information
    about the object, such as its source and type """
    if generator_name:
        return LanguageInfo.find_generator_by_name(generator_name)()
    else:
        if source_file_name:
            h = LanguageInfo.guess_language_for_file(source_file_name)
            return h.get_default_html_generator()()
        else:
            raise ValueError, "Unknown file type, cannot create lexer"

```

Segundo, quando você esta no Plone, você precisar ser capaz de compreender exatamente quais as linguagens estão disponíveis assim você pode mostrá-la ao usuário. Desenvolva a função, *list_generators*, para retornar uma lista de linguagens:

```

def list_generators():
    """ This returns a list of generators, a generator
    is a valid language, so these are things like perl,
    python, xml etc... """
    lexers = LanguageInfo.get_generator_names()
    return lexers

```

Finalmente, a função *generate_html* recebe o arquivo fonte com uma string, um gerador e um nome de arquivo opcional. O *SilverCity* requer um arquivo tal como um *buffer* para desenvolver o conteúdo, assim você pode utilizar o módulo Python's *StringIO* para implementar isso:

```

def generate_html(source_file, generator=None, source_file_name=None):
    """ From the source make a generator
    and then make the html """

    # SilverCity requires a file like object

```

```

target_file = StringIO()
generator = create_generator(source_file_name, generator)
generator.generate_html(target_file, source_file)

# return the html back
return target_file.getvalue(), generator.name

```

Você notará que é chamada a função *create_generator* que você escreveu anteriormente para utilizar o gerador correto para esta linguagem específica. Este é todo o código que você precisa para gerar o HTML para o arquivo informado. I haven't gotten into any of the specifics of actually lexing through the source or producing the HTML; a biblioteca do SilverCity faz tudo isso para voce. Para reiterar um ponto anterior, o módulo não faz nenhuma referência ao Zope/Plone, ele é completamente independente. Você não precisa entender no detalhe o funcionamento do módulo, e sim como importar uma biblioteca desenvolvida por terceiros.

A seguir acrescentamos uma última parte de código para realizar testes. Na verdade você poderia implementar toda uma unidade de teste completa, mas isto não será tratado aqui. Ao invés acrescentaremos uma pequena porção de código para testar duas coisas: se isto funciona e se a linguagem (Python no caso) está disponível:

```

if __name__ == "__main__":
    import sys
    myself = sys.argv[0]
    file = open(myself, 'r').read()
    print generate_html(file, generator="python")
    print list_generators()

```

Se voce executar este script, ele mostrará um texto HTML com a sintaxe realçada do seu próprio código. Você apenas colocará isto num módulo Zope específico; caso contrário, terá tudo isso em um script separado sendo fácil de testar e de se promover alterações futuras.

Desenvolvendo uma Classe

Um tipo de conteúdo em Plone é apenas um objeto que possui alguns atributos específicos e algumas classes bases particulares. Não há necessidade de se preocupar com a leitura e a escrita no banco de dados, pois estas funcionalidades são tratados pela classe *Persistencebase*. Para o momento, crie, no pacote, um módulo chamado *PloneSilverCity.py*.

Primeiro, importe o módulo *source.py* que você já desenvolveu. Isto se traduz em uma única linha de código porque o módulo está contido no pacote. A linha para importar a função é a seguinte:

```

from source import generate_html, list_generators

```

Segundo, você precisará da classe *PloneSilverCity* que permite a você envolver (encapsulate) a funcionalidade que você precisa. Você precisa se preocupar com os quatro atributos desta classe:

- `id`: Armazena o ID original da instância da classe *PloneSilverCity*.
- `_raw`: Armazena o código fonte raw na classe.
- `_raw_as_html`: armazena o código fonte depois de feita a análise sintática e a transformação para HTML.
- `_raw_language`: Armazena a linguagem do código fonte.

Para cada um destes atributos, você irá desenvolver um “*accessor*”, que é uma função que retorna o valor do atributo, um artifício utilizado para melhor chamar cada atributo. Um exemplo de uma função “*accessor*” é *getLanguage*, que retorna o valor do atributo linguagem. Desenvolver a função “*accessor*” é normalmente uma ótima idéia, especialmente porque mais tarde você estará dando segurança a este método “*accessor*”. No Zope, qualquer método ou atributo que começa com underscore (“_”) não está disponível para métodos baseados em Web tal como os objetos page templates ou Script (Python). Uma boa prática é iniciar todos os seus atributos com o underscore (“_”) e então colocar a segurança ao método de acesso.

Lista 12-1 Mostra a classe básica.

Lista 12-1. Classe Python Básica

```
class PloneSilverCity:
    def __init__(self, id):
        self.id = id
        self._raw = ""
        self._raw_as_html = ""
        self._raw_language = None

    def getLanguage(self):
        """ Returns the language this code has been lexed with """
        return self._raw_language

    def getRawCode(self):
        """ Returns the raw code """
        return self._raw

    def getHTMLCode(self):
        """ Returns the html code """
        return self._raw_as_html

    def getLanguages(self):
        """ Returns the list of languages available """
        langs = []
        for name, description in list_generators():
            langs.append( {'name':lang, 'value':language} )
        langs.sort()
        return langs
```

Você tem que adicionar um outro método que permite enviar um arquivo ou uma string. Este método irá ler o arquivo e vai verificar se ele não está vazio; então será lido e o “filename” será determinado. Então o código, linguagem, e o filename serão passados para gerar uma função. Você irá armazenar todos os atributos já mencionados, como mostrado na Lista 12-2.

Lista 12-2. The Method for Handling Edits

```

def edit(self, language, raw_code, file=""):
    """ The edit function, that sets
    all our parameters, and turns the code
    into pretty HTML """
    filename = ""
    if file:
        file_code = file.read()

        # if there is a file and it's not blank...
        if file_code:
            raw_code = file_code
            if hasattr(file, "name"):
                filename = file.name
            else:
                filename = file.filename
            # set the language to None so set by SilverCity
            language = None

    self._raw = raw_code

    # our function, generate_html does the hard work here
    html, language = generate_html(raw_code, language, filename)
    self._raw_as_html = html
    self._raw_language = language

```

NOTA Well-versed Python developers may raise an issue with using *file.name* and *file.filename*. Os arquivos objetos Zope tem um atributo chamado *filename*, que representa o filename, enquanto em Python o atributo é chamado *name*. Este código irá trabalhar em Python ou Zope.

Agora você tem uma classe Python que encapsula o objeto. Neste ponto, você deve ser capaz de rodar isto do prompt Python muito facilmente e testá-lo. Por exemplo:

```

$ python
Python 2.3.2 (#1, Oct 6 2003, 10:07:16)
[GCC 3.2.2 20030222 (Red Hat Linux 3.2.2-5)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> from PloneSilverCity import PloneSilverCity
>>> p = PloneSilverCity("test.py")
>>> p.edit("python", "print 'hello world'")
>>> p.getRawCode()
"print 'hello world'"
>>> p.getHTMLCode()
'<span class="p_word">print</span>
<span class="p_default">nbsp;</span>
<span class="p_character">\'hellonbsp;world\'</span>'
>>> p.getLanguage()
'python'

```

Transformando o Pacote em um Produto

Agora você tem um simples pacote, mas não é ainda um produto Plone. Você tem que inicializá-lo com o Plone. Significa adicionar informações extras para o módulo *PloneSilverCity.py*. Especificamente, você precisa adicionar uma função “factory” que é um modelo padrão em design orientado-objeto, e contém as definições de como o objeto será criado. Assim, para o módulo *PloneSilverCity.py*, adicione o seguinte construtor para o módulo:

```
def addPloneSilverCity(self, id, REQUEST=None):
    """ This is our factory function and creates
    an empty PloneSilverCity object """
    obj = PloneSilverCity(id)
    self._setObject(id, obj)
```

A função *addPloneSilverCity* não é uma parte da classe *PloneSilverCity*. Como um construtor para a classe, ela é colocada em um módulo fora da classe. Esta função é a primeira função específica do Plone. Três parâmetros são passados para o método: o objeto *self*, a string ID para o objeto, e o *REQUEST*. O objeto *self* se refere realmente ao *contexto* que você viu anteriormente, utilizado aqui com um nome diferente. Visto que sempre os objetos serão criados dentro de uma pasta, *self* consultará a pasta na qual este objeto será criado. Esta função cria uma instância de *PloneSilverCity* chamada *obj* e o passa pelo método *_setObject* da pasta. O método *_setObject* é particular para o Zope, sendo instanciado o objeto no banco de dados e registrados os objetos na pasta “containing”.

Em seguida, adicione a informação do tipo “factory” explicado no Capítulo 11. As informações do tipo “factory” contêm todos os dados sobre o tipo de conteúdo em um dicionário; esta informação é carregada no *portal_types* quando o produto é instalado na instância Plone. Esta informação é igual a que você viu anteriormente, quando você alterou a informação do tipo “factory” pela Web.

Antes de construir a informação “factory”, normalmente crio um arquivo de configuração que contém todas as variáveis repetidas para o produto. Este arquivo é chamado *config.py*, e nele você coloca os nomes dos produtos, o nome da camada nas skins, e o nome de como irá aparecer para o usuário:

```
product_name = "PloneSilverCity"
plone_product_name = "Source Code"
layer_name = "silvercity"
layer_location = "PloneSilverCity/skins"
```

Então você ajusta a informação do tipo “factory” e utiliza estas strings. Por exemplo, o ID será *Source Code* desde que isso seja mostrado no Plone para o usuário. A seção de “actions” são um conjunto (tupla) de dicionários de todas as ações que podem ocorrer com este objeto. Quando esta “factory” é carregada no Plone, a aba Actions dentro da ferramenta *portal_types* será povoada com este conteúdo. Cada uma destas ações corresponde a um método, template, ou um script que será chamado; a maioria deles corresponde diretamente a um “page template”, o qual já foi explicado nesta seção.

Como você já sabe, uma ação é algo que os usuários podem fazer com um item no banco de dados do Plone. Usuários podem fazer duas coisas óbvias ao código fonte, como podem ver isto e ver o código destacado, e pode editar o item e enviar o “código fonte”. O Plone requer que no banco de dados uma ação seja chamada *view* e uma outra chamada *edit*. Você também já especificou uma terceira ação, que seria bom se o usuário fosse capaz de baixar o “código fonte” (source) no seu formulário original. Como em linguagens como o Python onde a formatação é essencial, isto é muito útil. Esta ação aponta diretamente ao *getRawCode*, que é o método que adquire o código raw de volta.

Cada ação tem uma permissão associada, como mostra na Lista 12-3.

Lista 12-3. A informação do Tipo Factory e Ações

```
factory_type_information = {
    'id': plone_product_name,
    'meta_type': product_name,
    'description': ('Provides syntax highlighted HTML of source code.'),
    'product': product_name,
    'factory': 'addPloneSilverCity',
    'content_icon': 'silvercity.gif',
    'immediate_view': 'view',
    'actions': (
        {'id': 'view',
         'name': 'View',
         'action': 'silvercity_view_form',
         'permissions': (view_permission,)},
        {'id': 'edit',
         'name': 'Edit',
         'action': 'silvercity_edit_form',
         'permissions': (edit_permission,)},
        {'id': 'source',
         'name': 'Source',
         'action': 'getRawCode',
         'permissions': (view_permission,)},
    ),
}
```

NOTA Até agora, o produto não pode ser importado do prompt Python porque o código esta incompleto.

Ajustando Permissões

O conceito fundamental relacionado a Web sites é que tudo e todos não são confiáveis (untrusted). Antes de qualquer propriedade ser acessada ou qualquer método ser chamado, você primeiro certifica se quem requisitou está habilitado a executar esta ação. Na maioria dos sistemas, existem três permissões: a de adicionar itens, apagar itens, e a permissão de editar itens. Uma outra permissão aplicada ao Plone: a de visualizar determinado item através da Web (ou outro protocolo). The containing folder handles deleting, which is a permission handed out in Plone to the containing folder. If you can delete anything in the folder, you can then also delete the content type you're adding here.

Assim voce tem que ocupar de 3 permissões. É normal utilizar algumas que vem com o pacote do CMF: *Add portal content*, *Modify portal content*, and *View*. Returning to the config file, voce pode adicionar permissões que voce necessitaria, tais como:

```
from Products.CMFCore import CMFCorePermissions

add_permission = CMFCorePermissions.AddPortalContent
edit_permission = CMFCorePermissions.ModifyPortalContent
view_permission = CMFCorePermissions.View
```

Isto significa que a variável *add_permission* se refere a permissão importada do *CMFCorePermissions*. Uma permissão é nada mais que apenas uma string. Utilizando a permissão “built-in” é mais conveniente e compreensível para os usuários. O Plone já é configurado para permitir a pessoa correta adicionar conteúdo utilizando a permissão *Add portal content*. Também, o workflow padrão é definido para utilizar e alterar estas permissões. Estas permissões são as que você adicionou as informações do tipo “factory”.

Se você quiser fazer sua própria permissão, você pode fazer isto facilmente. Suponha que queira a permissão *Add...* para *Add Source Code*. Então você tem que mudar o arquivo para o seguinte:

```
add_permission = "Add Source Code"
```

Após importar o produto, você tem uma nova permissão na aba *Security* combinando com a permissão “*Add Source Code*”. Por que você poderia querer isto? Bem, utilizar uma permissão que todo mundo utiliza é mais conveniente. Porém, pode ser que você queira mais qualidade ou uma segurança diferente. Por esta razão, você pode apenas criar seus ajustes de segurança.

Completando a Inicialização

Agora você precisa ajustar a inicialização do produto. Você fará isto no módulo *__init__.py* de maneira que quando o Zope analisar esse arquivo na inicialização, irá completar a inicialização do produto, como mostra a Lista 12-4.

Lista 12-4. O módulo *__init__.py*

```
import PloneSilverCity

from Products.CMFCore import utils
from Products.CMFCore.DirectoryView import registerDirectory

from config import product_name, add_permission

contentConstructors = (PloneSilverCity.addPloneSilverCity,)
contentClasses = (PloneSilverCity.PloneSilverCity,)
contentFTI = (PloneSilverCity.factory_type_information,)

registerDirectory('skins', globals())
```

```
def initialize(context):
    product = utils.ContentInit(product_name,
                                content_types = contentClasses,
                                permission = add_permission,
                                extra_constructors = contentConstructors,
                                fti = contentFTI)
    product.initialize(context)
```

O que acontece no código? Bem, realmente não é muito. Primeiro você vai fazer referência as classes e aos construtores que estão sendo usados em *contentClasses* e *contentConstructors*. Esse é um mapeamento para a função “factory” criar os objetos e a classe atual; são passadas dentro da função *ContentInit*, dentro do *initialize*, que é uma função especial que é chamada durante a inicialização do produto. O *ContentInit* faz todo o trabalho para ajustar o produto dentro do Plone. Os parâmetros para esta função são os seguintes:

- **product_name**: Nome do produto, definido no arquivo de configuração (no caso, *PloneSilverCity*).
- **content_types**: Conjunto de classes que os produtos definem.
- **permission**: Permissão necessária para criar uma instância para o objeto; no caso, é a variável *add_permission* como foi definido no *config.py*.
- **fti**: This stands for factory type information and is the dictionary of factory type information you defined in the *PloneSilverCity.py* module for the content.

Alterando os Módulos do Produto

Agora você pode retornar ao módulo *PloneSilverCity.py* e completar a tarefa de torna-lo um produto Plone. No início do módulo, você vai inserir uma instrução *import*. Esta instrução importa várias porções de código, necessárias para a inicialização do Plone, de diversos lugares/bibliotecas, como se segue:

```
from Globals import InitializeClass
from AccessControl import ClassSecurityInfo
from Products.CMFDefault.DublinCore import DefaultDublinCoreImpl
from Products.CMFCore.PortalContent import PortalContent
```

estas importações fornecem funcionalidades básicas para o produto e são comuns para a maioria dos tipos de conteúdo. As definições de importação são as seguintes:

InitializeClass: Esta função inicia a classe e aplica todas as declarações de segurança necessárias. Você especifica estas declarações de segurança utilizando a classe *ClassSecurityInfo*.

ClassSecurityInfo: Esta classe fornece uma série de métodos de segurança que vai permitir que você restrinja o acesso a métodos aos tipos de conteúdo.

DefaultDublinCoreImpl: Esta classe fornece uma implementação de metadado Dublin Core (foi mostrado no Capítulo: O Dublin Core), concede ao objeto todos os atributos e métodos Dublin Core para acessar os atributos Título, Descrição, Criador(owner) , e assim por diante.

PortalContent: Fornece a classe base para todo conteúdo de um Plone site e alguns atributos chaves necessários. Possibilita a aquisição de uma série de funcionalidade como persistência do objeto no banco de dados, catalogação para uma busca posterior e torná-lo “registrável” na ferramenta *portal_types*.

Você também necessitará importar as variáveis de configuração e de permissões. Conforme a seguir:

```
from config import plone_product_name, product_name from config import
add_permission, edit_permission, view_permission
```

Você tem ainda que adicionar as duas classes básicas para deixá-lo compatível ao Plone: *PortalContent* e *DefaultDublinCoreImpl*. Além de atribuir a classe *meta_type*. Cada produto no Zope tem um *meta_type* único:

```
class PloneSilverCity(PortalContent, DefaultDublinCoreImpl):
    meta_type = product_name
```

O Plone necessita saber qual é a classe base a ser implementada para o tipo de conteúdo:

```
__implements__ = (
    PortalContent.__implements__,
    DefaultDublinCoreImpl.__implements__
)
```

Adicionando Segurança para a Classe

Se voce decidiu atribuir ações de segurança, então temos que aplicar a classe de segurança. No ambiente de publicação com o Plone, ninguém poderá chamar qualquer método da classe através da web, a menos que o método comece com o character sublinhado. Assim você terá que proteger todo os métodos.

Para fazer isso dentro da classe, crie uma instância da classe *ClassSecurityInfo*, com a seguinte linha:

```
security = ClassSecurityInfo()
```

O ZOPE fornece para este objeto uma interface/aparato de segurança. Então você pode aplicar outros métodos para o objeto. A maneira de como implemento isso é adicionando uma linha e aplicando diretamente a segurança sob o método. Este modo é fácil para se lembrar onde a segurança é aplicada. O método *declareProtected* pega a permissão e o nome do método a ser protegido e edita o método. Assim somente poderá editar o método o usuário que puder chama-lo, isso se implementa da seguinte maneira: :

```
security.declareProtected(edit_permission, "edit")
```

Repita isto para cada método, dando a permissão apropriada e o nome do método. O único que precisa ser protegido é o *__init__* porque isto começa com um underscore (“_”), assim para aplicar toda esta segurança, você deve iniciar a classe. Sem fazer isto primeiro, toda a segurança adicional declarada *nao* será

aplicada, e seu objeto será “public” (publico ao usuário), por esse motivo não esqueça de inserir a linha abaixo:

```
InitializeClass(PloneSilverCity)
```

A API para *ClassSecurityInfo* fornece os seguintes métodos para a classe:

- **declarePublic:** Pega a lista de nomes. Todos os nomes são public (declarados publicamente) acessíveis para todos os usuários pelo código restrito e pela Web.
- **declarePrivate:** Pega a lista de nomes. Todos os nomes são private (declarados privados) e não podem ser acessíveis pelo código restrito.
- **declareProtected:** Pega a permissão e qualquer quantidade de nomes. Todos os nomes podem ser acessados apenas com a permissão dada.
- **declareObjectPublic:** Isto ajusta o objeto inteiro como public (publicamente acessível).
- **declareObjectPrivate:** Isto ajusta o objeto inteiro como private (declarado privado) e inacessível para código restrito.

Com estes métodos é possível ajustar a maioria das seguranças que você quiser.

Integrando com a Busca (Pesquisa)

No capítulo anterior mostrei como a pesquisa funciona e qual os índices que existem. Desde que os índices trabalham com os objetos Dublin Core e você usou o Dublin Core como uma classe base, seus objetos título, descrição, criador, data de modificação, e assim por diante, serão indexadas sem necessitar de trabalho extra. Também, pela herança da classe *PortalContent* a qualquer momento que o objeto for alterado, o catálogo será atualizado.

Porém, o índice *SearchableText* precisa de uma pequena alteração. Como demonstrei anteriormente, o índice *SearchableText* fornece uma busca full-text que o Plone utiliza quando a pesquisa esta sendo executada. Séria legal que o nosso “código fonte” pudesse ser encontrado através deste tipo de busca, assim se alguém fizer um upload de um fragmento de código contendo, por exemplo a instrução *import*, o serviço de busca poderia procurar por todo o fragmento de código e encontrar esta instrução. O catálogo funciona examinando o objeto e tentando encontrar coincidências entre atributos ou métodos com o nome do índice. Tudo que precisamos fazer é prover um método que retorne a valor (que seria o campo que contém o “código fonte”) que você necessita.

O melhor modo de fazer isso é criar uma string aparte dos campos que você já tem (por exemplo, o título, a descrição, e o código raw). Podemos protegê-la pela permissão *View*, assim qualquer um que possa ver o objeto também possa facilmente ver o conteúdo. O método *SearchableText* executa esta tarefa, como segue o exemplo:

```
security.declareProtected(view_permission, "SearchableText")
def SearchableText(self):
    """ Used by the catalog for basic full text indexing """
    return "%s %s %s" % ( self.Title()
                        , self.Description()
                        , self._raw
```

)

A Diferença Entre uma Classe Python e uma Classe Plone

Como você pode ver aqui existe uma grande diferença entre o produto Python e um do Plone. Entretanto, a maioria destas diferenças é sobre registros de produtos e declarações de segurança. A Lista 12-5 destaca todas as diferenças entre a implementação do Python e a do Plone.

Lista 12-5. A Versão da Classe Plone

```

from Globals import InitializeClass**
from AccessControl import ClassSecurityInfo
from Products.CMFDefault.DublinCore import DefaultDublinCoreImpl
from Products.CMFCore.PortalContent import PortalContent

from config import meta_type, product_name
from config import add_permission, edit_permission, view_permission
from source import generate_html, list_generators

factory_type_information = {
    'id': plone_product_name,
    'meta_type': product_name,
    'description': ('Provides syntax highlighted HTML of source code.'),
    'product': product_name,
    'factory': 'addPloneSilverCity',
    'content_icon': 'silvercity.gif',
    'immediate_view': 'view',
    'actions': (
        {'id': 'view',
         'name': 'View',
         'action': 'silvercity_view_form',
         'permissions': (view_permission,)},
        {'id': 'source',
         'name': 'View source',
         'action': 'getRawCode',
         'permissions': (view_permission,)},
        {'id': 'edit',
         'name': 'Edit',
         'action': 'silvercity_edit_form',
         'permissions': (edit_permission,)},
    ),
}

def addPloneSilverCity(self, id, REQUEST=None):
    """ This is our factory function and creates
    an empty PloneSilverCity object inside our Plone
    site """

    obj = PloneSilverCity(id)
    self._setObject(id, obj)

```

```

class PloneSilverCity(PortalContent, DefaultDublinCoreImpl):

    meta_type = product_name

    __implements__ = (
        PortalContent.__implements__,
        DefaultDublinCoreImpl.__implements__
    )

    security = ClassSecurityInfo()

    def __init__(self, id):
        DefaultDublinCoreImpl.__init__(self)
        self.id = id
        self._raw = ""
        self._raw_as_html = ""
        self._raw_language = None

    security.declareProtected(edit_permission, "edit")
    def edit(self, language, raw_code, file=""):
        """ The edit function, that sets
        all our parameters, and turns the code
        into pretty HTML """
        filename = ""
        if file:
            file_code = file.read()

            # if there is a file and its not blank...
            if file_code:
                raw_code = file_code
                if hasattr(file, "name"):
                    filename = file.name
                else:
                    filename = file.filename
                # set the language to None so set by SilverCity
                language = None

        self._raw = raw_code

        # our function, generate_html does the hard work here
        html, language = generate_html(raw_code, language, filename)
        self._raw_as_html = html
        self._raw_language = language

    security.declareProtected(view_permission, "getLanguage")
    def getLanguage(self):
        """ Returns the language that code has been lexed with """
        return self._raw_language

    security.declareProtected(view_permission, "getLanguages")
    def getLanguages(self):
        """ Returns the list of languages available """
        langs = []

```

```

for name, description in list_generators():
    # these names are normally in uppercase
    langs.append( {'name':lang, 'value':language } )

langs.sort()
return langs

security.declareProtected(view_permission, "getRawCode")
def getRawCode(self):
    """ Returns the raw code """
    return self._raw

security.declareProtected(view_permission, "getHTMLCode")
def getHTMLCode(self):
    """ Returns the html code """
    return self._raw_as_html

security.declareProtected(view_permission, "SearchableText")
def SearchableText(self):

    """ Used by the catalog for basic full text indexing """

    return "%s %s %s" % ( self.Title()

                        , self.Description()

                        , self._raw

                        )

InitializeClass(PloneSilverCity)

```

Adicionando Skins

Agora que você já possui o código principal, você ainda tem duas coisas a fazer: construir os skins e criar um método de instalação. Na verdade os skins é a parte mais fácil porque a maior parte do trabalho já foi feito no framework Plone. Já explicamos os skins em detalhes nos capítulos anteriores, onde foi dito como fazer um skin para um site Plone no sistema de arquivos. Cada produto necessita provêr uma interface para o Usuário - User Interface (UI), fizemos isso no próprio FSDV - File System Directory View, então vamos fazer a mesma coisa aqui.

Os skins são colocados no diretório *skins* do produto. Este nome de diretório é definido no arquivo `__init__.py` onde você registra o diretório usando a função `registerDirectory`. Se você quiser mudar o nome, certifique-se de registrá-lo. Você pode registrar em vários diretórios, mas isto é recursivo e irá registrar qualquer coisa a baixo desse diretório registrado.

A coisa mais simples de se fazer no desenvolvimento desse trabalho é adicionar um ícone para o objeto que irá aparecer no Plone. O nome do ícone já é definido na informação do tipo factory com a linha `'content_icon': silvercity.gif`, assim

tudo o que você tem que fazer é adicionar um ícone no diretório *skins* chamado *silvercity.gif*. Este ícone irá aparecer toda vez que você ver o objeto na interface do usuário no Plone. Quando o SilverCity traduz um arquivo, ele gera uma saída em código HTML utilizando tags CSS, então você deve garantir que este arquivo CSS em particular esteja disponível. Para este produto você simplesmente copia o CSS do produto SilverCity e o coloca no diretório *skins* com o nome *silvercity.css*.

Agora você vai ter que desenvolver a página de **visão e edição**. Já foi explicado isto anteriormente, e de como isto se parece com um documento, uma página de “visão” e/ou “edição”, é renderizada pelas páginas *document_view.pt* e *document_edit_form.cpt* que estão localizadas no diretório *CMFPlone/skins/plone_content* directory.

Criando a Página de Visão

Para alterar a página de visão, a partir da original, você deve fazer uma cópia para dentro do seu diretório *skins* do seu produto, renomeando-a para *silvercity_view.pt*. Assim não há a necessidade de se criar “do zero” uma nova página de visão já que elas são tão parecidas; tudo que você necessita é de realizar duas pequenas mudanças.

Como mencionado, SilverCity gera código HTML para realçar as sintaxe das instruções Python e utiliza CSS para o qual você tem um arquivo de folha de estilo. Você precisa garantir que sua página de visão irá utilizar este arquivo CSS, e o “main template” tem um slot para CSS chamado *css_slot*. Para colocar nosso arquivo CSS neste slot , você apenas fornece o caminho do arquivo CSS. Por exemplo:

```
<metal:cssslot fill-slot="css_slot">

<link
  rel="stylesheet"
  href=""
  tal:attributes="href string:$portal_url/silvercity.css" />
</metal:cssslot>
```

Aqui você está referenciando um arquivo CSS chamado *silvercity.css* localizado no diretório *skins*. O documento original mostra uma propriedade chamada *cookedBody*. Removemos esta parte do código e no lugar inserimos a função *getHTMLCode* que retorna o HTML, assim tudo o que você tem a fazer é o seguinte:

```
<div id="bodyContent">
  <div tal:replace="structure here/getHTMLCode" />
</div>
```

Você pode mudar alguma coisa específica nesta página template, por exemplo, pode ser legal mostrar a linguagem com a qual foi desenvolvido o “fragmento de código”, mostrar um ícone, e assim por diante.

Criando a página de edição

Como aconteceu com a página de “visão”, você pode pegar a página de edição e renomeando-a para *silvercity_edit_form.cpt*. Aqui o problema maior é que o formulário de edição foi concebido para utilizar um editor WYSIWYG (What You See Is What You Get) tal como o Epoz. Você vai ter que desativar isto.

This is quite a lengthy change of the page template—remember, you can get this off the Web site. Neste template, teremos que remover qualquer menção ao editor acima e substituir, fazendo referência a um simples campo do tipo “text area”. Vamos manter o nome do campo HTML porque não há uma necessidade real para alterá-lo. O documento tem no seu início uma série de seleções para o formato, quais sejam texto plano, HTML, e etc. Vamos substituir isoo por um campo do tipo drop-down para conter alguns nomes de linguagens de programação disponíveis na biblioteca do SilverCity. O método *getLanguages* escrito anteriormente retorna uma lista de todas as linguagens. Cada item é um “dicionário” que contém valores (por exemplo, CPP) e nome legível da linguagem (por exemplo, C or C++).

Listing 12-6 loops pelo método *getLanguages* escrito anteriormente..

Lista 12-6. Adicionando uma Lista Drop-Down para Selecionar a Linguagem

```
<div class="field">
  <label
    for="language"
    i18n:translate="label_silvercity_language">
Language</label>

    <div class="formHelp" i18n:translate="help_silvercity_language">
      Select the name of the language that you are adding
    </div>
    <select name="text_format"
      tal:define="l here/getLanguage">
      <option tal:repeat="item here/getLanguages"
        tal:content="item/name"
        tal:attributes="value item/value;
          selected python:test(item['value'] == 1, 1, 0)" />
    </select>
</div>
```

Quando a página de edição é submetida pelo usuário, você necessita determinar a validação e alguma ações que o formulário precisa tomar. Está validação deve checar a validade do título e do ID informados. Para o arquivo *silvercity_edit.cpt.metadata*, adicione o seguinte:

```
[validators]
validators..Save = validate_id,validate_title
validators..Cancel =
```

De onde vem todos esses validadores? O nosso formulário chama 3 validações diferentes, mas nos precisamos somente de duas. Você pode acrescentar algumas

outras e você encontra todas as validações em *plone_skins/plone_form_scripts*, sendo que o nome do objeto sempre começa com *validation*.

Agora você precisa da ação, edite o script (*document_edit.cpy*) do documento e copie-o para dentro do SilverCity. A maioria dos scripts são ótimos, assim você pode mantê-los com alguma modificação. Mude a mensagem para *Source code* no lugar de *Document*. Lista 12-7 mostra a edição do script.

Lista 12-7. Edição do Script

```
##parameters=text_format, text, file='', SafetyBelt='', ~CCC
title='', description='', id=''
##title=Edit a document

filename=getattr(file,'filename', '')
if file and filename:
    # if there is no id, use the filename as the id
    if not id:
        id = filename[max( filename.rfind('/')
                           , filename.rfind('\')
                           , filename.rfind(':') )+1:]
    file.seek(0)

# if there is no id specified, keep the current one
if not id:
    id = context.getId()

new_context = context.portal_factory.doCreate(context, id)
new_context.edit( text_format
                  , text
                  , file
                  , safety_belt=SafetyBelt )

from Products.CMFPlone import transaction_note
transaction_note('Edited source code %s at %s' % ~CCC
                (new_context.title_or_id(), new_context.absolute_url()) ~CCC
                )

new_context.plone_utils.contentEdit( new_context
                                     , id=id
                                     , title=title
                                     , description=description )

return state.set(context=new_context, ~CCC
                 portal_status_message='Source code changes saved.')
```

O script faz o seguinte. Primeiro, se existir, pega o nome do arquivo; se nenhum ID é determinado o ID é ajustado como sendo o nome do arquivo. Isto significa que se um usuário enviar o *Library.c*, o ID para o objeto será *Library.c*. Segundo, informamos ao *portal_factory* para que seja criado um objeto (veja em "Portal Factory" para mais informações sobre o que isto significa). Então ele chama o método *edit* do objeto, e chama o *contentEdit* da ferramenta *plone_utils*. Examinando mais detalhadamente a ferramenta *plone_utils*,

`contentEdit` pega os comandos dados, e se foi implementada a classe `Dublin Core`, modifica todos os atributos. Desde que você definiu o atributo `__implements__`, o método `edit` listado em 12-7 não trabalhará para você. Assim qualquer mudança no título, ID, ou descrição trará mudanças também no objeto.

Portal Factory

Existe um problema do modo com que os objetos são criados. Para você adquirir o formulário de edição, você tem que criar um objeto. Na prática, as pessoas acidentalmente criam objetos, adquirem o formulário de edição, and então o abandonam com o tipo errado. Isto é irritante e deixa os objetos sem uso e jogados no banco de dados. Ainda, ele cria, de maneira errada, um arquivo no sistema de arquivo, e então os deixam lá.

Para resolver isto, a ferramenta `portal_factory` permite a você criar objetos temporariamente. Ela cria um objeto temporário e então deixa você editá-lo. Apenas quando você tiver clicado no botão `edit` é que seu objeto será criado em definitivo. Para assinar um objeto `portal_factory`, vá para a ferramenta `portal_factory`, e no formulário selecione todos os tipos de conteúdo para os quais você deseja utilizar esta ferramenta. A única vantagem é que você necessita de se assegurar que seu editor de scripts está corretamente integrado com a ferramenta, como mostra no exemplo.

Instalando o Produto no Plone

Para instalar um produto no Plone, basta ir ao painel de controle do Plone e clicar no produto. O script vai utilizar a ferramenta `portal_quickinstaller` para instalação. Para este produto funcionar, você precisa mostrar as funcionalidades para que a ferramenta possa ler. Após, você desejará que as pessoas utilizem seu produto. Se você esta desenvolvendo apenas para uso interno e nunca irá distribuí-lo, você pode pular esse processo. Mas se você vai precisar executar estes passos de qualquer jeito, é sempre melhor construir um script para a instalação.

NOTA Quick Installer cria um método externo da função de instalação e executa-o por debaixo dos panos. Isto faz com que seja executada uma série de outras tarefas. Ou seja você pode criar um método externo também se desejar executar estas tarefas.

Para integrar com o Quick Installer, você precisa criar um módulo específico chamado `Install.py` no diretório `Extensions`. Este módulo tem que conter uma função chamada `install`. A ferramenta Quick Installer executa esta função e a saída é colocada em um arquivo no servidor. O método `install` tem que instalar o produto dentro do portal types, assim temos que adicionar um FSDV que aponta para o diretório `skins`, e adicionar este novo diretório para o skin layers.

Agora importe as funções e ajuste as variáveis como normalmente. Você tem que importar o `factory_type_information` do produto para poder utilizá-lo, como mostrado na Lista 12-8.

Lista 12-8. O Início da Função de Instalação.

```
from Products.CMFCore.TypesTool import ContentFactoryMetadata
```

```

from Products.CMFCore.DirectoryView import createDirectoryView
from Products.CMFCore.utils import getToolByName
from Products.PloneSilverCity.PloneSilverCity import factory_type_information

from Products.PloneSilverCity.config import plone_product_name, product_name
from Products.PloneSilverCity.config import layer_name, layer_location

def install(self):
    """ Install this product """

```

Após isto, tudo o mais é genérico e pode ser rodado por qualquer produto, a menos é claro que você queira fazer algo diferente na instalação. Para adicionar seu produto na ferramenta *portal_types*, você primeiro verifica se seu produto já foi registrado. Pode ser que alguém já tenha registrado algum outro produto com o mesmo nome. Se for o caso chame o método *manage_addTypeInfoInformation*, como mostrado na Lista 12-9.

Lista 12-9. Restante da Função de Instalação.

```

out = []
typesTool = getToolByName(self, 'portal_types')
skinsTool = getToolByName(self, 'portal_skins')

if id not in typesTool.objectIds():
    typesTool.manage_addTypeInfoInformation(
        add_meta_type = factory_type_information['meta_type']
        id = factory_type_information['id']
    )
    out.append('Registered with the types tool')
else:
    out.append('Object "%s" already existed in the types tool' % (id))

```

Em seguida você precisa adicionar um FSDV no diretório *skins*. Novamente, a primeira coisa é verificar se você já não possui; então você adiciona o diretório view com o seguinte:

```

if skinname not in skinsTool.objectIds():
    createDirectoryView(skinsTool, skinlocation, skinname)
    out.append('Added "%s" directory view to portal_skins' % skinname)

```

Enfim, você passa por todos os skins e adiciona seu novo FSDV para cada skin. Esta é uma função genérica; cada skin é listado como string com cada camada separada por vírgula. Tudo o que você tem que fazer é dividir a string e inserir seu novo skin após a camada *custom*, como mostrado na Lista 12-10.

Lista 12-10. Ajustando o Skin no Método de Instalação

```

skins = skinsTool.getSkinSelections()
for skin in skins:
    path = skinsTool.getSkinPath(skin)
    path = [ p.strip() for p in p.split(',') ]
    if skinname not in path:
        path.insert(path.index('custom')+1, skinname)

```

```

    path = ", ".join(path)
    skinsTool.addSkinSelection(skin, path)
    out.append('Added "%s" to "%s" skins' % (skinname, skin))
else:
    out.append('Skipping "%s" skin' % skin)
return "\n".join(out)

```

É isto. Seu produto está pronto para rodar.

Testando o Produto

Para testar, reinicie sua instância Plone de modo que leia o diretório do produto. Se você não desenvolveu seu produto na pasta apropriada de produtos, então coloque agora no lugar como parte do seu processo padrão de instalação. Se tiver algum problema com seu produto, então o Zope irá iniciar, mas o produto pode ser mostrado como corrompido no painel de controle.

Então instale-o dentro do Plone utilizando a página “Add/Remove Products” no painel de controle do Plone. Agora você deve ir a pasta e adicionar um objeto de “código fonte”. O ícone será igual no skin, e o nome é o que você definiu no sistema de arquivo. Após adicioná-lo, você poderá editá-lo. Observe que após editá-lo a URL, no final, passa a ser *silvercity_edit_form* e mostra o formulário de edição.

Você pode adicionar mais fragmentos de código, selecionar a linguagem, e clicar em “Salvar” *(Save), ou você pode enviar o arquivo (fazer o download) para o seu computador. Após clicar em “Salvar”, você irá voltar para função de visão, e o o fragmento de código será mostrado com a sintaxe destacada.

Este produto é um pequeno exemplo de como é simples desenvolver um produto no Plone. Embora tenha muitas páginas, a maioria está ajustado a infra-estrutura e aos skins. Uma das primeiras coisas que as pessoas comparam é com outras linguagens de scripting Web como PHP. Você tem que se lembrar que tendo acesso ao código do Plone, você conseguiu executar várias coisas sem ter que reescrevê-las. Especificamente, você conseguiu o seguinte:

- Procurar conteúdo de forma Full-text.
- Integração com workflow
- Integração com portal de membership e authentication
- Persistir através do banco de dados do Plone sem ter que desenvolver SQL ou fazer outro trabalho

Também, você pode completar/alterar um produto já consagrado. Por exemplo, se voce necessita de um sistema de “bug-tracking” você pode encontrá-lo no “Collector product”, ou se você necessita de gerenciador de fotografias, utilize o “CMFPhoto”. Através da utilização do framework do Plone voce tem toda uma gama de bons produtos para ser utilizado no seu site provendo-o com flexibilidade e escalabilidade

Embora este produto utilize muitos códigos já existentes, ele mostra várias aspectos importantes no desenvolvimento de um produto no Plone.

Retirando os Erros de Desenvolvimento

Se você está desenvolvendo seu próprio produto, então, em um certo momento, duas coisas vão acontecer: seu produto irá falhar, e você vai precisar resolver este erro.

Durante o desenvolvimento, você deve tentar importar, utilizando o prompt do Python, para ver como ele está funcionando, provavelmente irá ocorrer um erro. Isto acontece porque quando você faz essa importação, you'll get a cascade of Zope-related imports. Você irá se deparar com, não todos, mas alguns dos problemas/situações de erro comuns:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?

  File "PloneSilverCity/__init__.py", line 1, in ?
    import PloneSilverCity
  File "PloneSilverCity/PloneSilverCity.py", line 4, in ?
    from Globals import InitializeClass
  File "/opt/Zope-2.7/lib/python/Globals.py", line 23, in ?
    import Acquisition, ComputedAttribute, App.PersistentExtra, os
  File "/opt/Zope-2.7/lib/python/App/PersistentExtra.py", line 15, in ?
    from Persistence import Persistent
ImportError: cannot import name Persistent
```

Você pode resolver isto garantindo que antes de você importar o PloneSilverCity você importe e execute o método `startup`. Para poder importar para o Zope, você deve ter configurado o diretório do Zope em seu caminho; no meu computador, é `/opt/Zope-2.7/lib/python`. Porém, você vai então rodar através dos erros tentando importar o CMFCore se você tiver uma instância home configurada.

O modo mais fácil de importar o PloneSilverCity é executar o Zope pela linha de comando no modo debug utilizando o `zopectl`. Isto vai abrir o prompt do Python que te permitirá acessar o banco de dados do Zope diretamente do Python. O capítulo 14 explica isto em mais detalhes, mas você pode realizar esta tarefa agora. Você pode encontrar o script `zopectl` no diretório `bin` da sua instância Zope; por exemplo, no meu computador, é `/var/zope/bin`.

Lista 12-11 mostra um exemplo rodando o `zopectl` com o PloneSilverCity.

NOTA até o momento da elaboração deste livro, o `zopectl` não funciona no Windows. Porém, em Linux, é um modo conveniente testar seu código. Infelizmente, utilizar o `zopectl` requer o travamento do ZODB - Zope Object Database e, a menos que você esteja rodando o ZEO (já explicado no Capítulo 14), você não poderá fazer isto enquanto o Zope está rodando.

Lista 12-11. Retirando os Erros do Produto Utilizando o Zope

```
$ cd /var/zope/bin
$ ./zopectl debug
Starting debugger (the name "app" is bound to the top-level Zope object)
>>> from PloneSilverCity.PloneSilverCity import PloneSilverCity
>>> p = PloneSilverCity("test")
```

```
>>> p.edit("python", "import test")
>>> p.getRawCode()
'import test'
>>> p
<PloneSilverCity at test>
```

Quando seu produto falha, lhe será mostrado um “traceback” em um dos desses dois lugares: ou na página de “error log” ou na de “event logs”. Se o produto realmente estiver quebrado, o Plone não irá iniciar; e isto normalmente acontece quando existe uma falha na importação. Recomendo iniciar o Plone pela linha de comando, seja no Windows ou Linux. Olhando diretamente para o console do Plone voce obterá um feedback imediato dos erros que estão acontecendo. Por exemplo, a Lista 12-12 mostra o que acontece quando você tenta rodar o Plone SilverCity com o erro na importação.

Lista 12-12. Exemplo de Erro no Inicio

```
$ bin/runzope
-----
2003-12-19T17:44:05 INFO(0) ZServer HTTP server started at Fri Dec 19 17:44:05
2003
      Hostname: laptop
      Port: 8080
-----
2003-12-19T17:44:05 INFO(0) ZServer FTP server started at Fri Dec 19 17:44:05
2003
      Hostname: basil.agmweb.ca
      Port: 8021
-----
2003-12-19T17:44:16 ERROR(200) Zope Could not import Products.PloneSilverCity
Traceback (most recent call last):
  File "/opt/Zope-2.7/lib/python/OFS/Application.py", line 533, in
import_product
    product=__import__(pname, global_dict, global_dict, silly)
  File "/var/zope.zeo/Products/PloneSilverCity/__init__.py", line 1, in ?
    import PloneSilverCity
  File "/var/zope.zeo/Products/PloneSilverCity/PloneSilverCity.py", line 1
    import ThisModuleDoesNotExist
          ^
ImportError: No module named ThismoduleDoesNotExist
```

Neste ponto, o Zope para; você vai ter que arrumar esta importação antes de reiniciá-lo.

O próximo tipo de erro que pode ocorrer é o de programação ou erro de lógica. Suponha que seu produto adiciona dois números juntos, mas um deles é uma string (isto é um erro em Python). Um erro será detectado, o qual o Plone irá reportar na interface do usuário com o valor e o tipo do erro. Você deve clicar em *plone setup* e clicar no Error Log (Registros de Erro) para ver o traceback, encontrar o erro, e consertá-lo.

Se você mudar algo no produto, isso não vai refletir no Python; você precisa utilizar o produto chamado *Refresh* para forçar esta mudança. Esta é uma

ferramenta muito útil para novos desenvolvedores, e você pode ativá-lo pelo arquivo *refresh.txt* no seu diretório *Products*. Agora, na ZMI, selecione Control Panel, Products, PloneSilverCity (ou o nome do seu produto), e clique na aba Refresh. Se seu produto tiver o arquivo *refresh.txt*, você pode clicar no botão Refresh. O Plone então vai carregar novamente seu produto com todo o código já alterado. Se você estiver rodando o Zope em modo debug, então você pode ajustar o produto para checar dinamicamente possíveis alterações antes de ser executado.

Infelizmente, o “refresh” não soluciona tudo. Ele realiza algumas operações por de traz dos panos para que o Python possa fazer algumas coisas acontecerem. De fato o Produto Refresh pode produzir resultados inesperados que não serão resolvidos ao re-iniciar o ZOPE. Produtos relativamente simples podem não ter problemas, mas o mesmo não se pode dizer de produtos que especialmente manipulam dados.

Finalmente, se algo ocorrer de errado e você não conseguir resolver, você necessitará de retirar o debugger. Você tem vários modos de debug do Zope utilizando o Python debugger. Você pode chamar o debugger do Python adicionando a seguinte linha no código:

```
import pdb; pdb.set_trace()
```

Isto vai causar um ponto de parada (breakpoints) na execução do código, e o Zope irá parar de processar e irá executar o debugger. Ele não trabalha com um serviço ou um “daemon”. Agora se você re-criar a situação de erro, você será direcionado para o Python debugger, e assim poderá debugar o seu produto. Por exemplo, para facilitar uma hipotética correção no método *getLanguages* a e verificar se a importação do PloneSilverCity, adicionei a seguinte função de “trace”:

```
def getLanguages(self):
    """ Returns the list of languages available """
    import pdb; pdb.set_trace()
    langs = []
    ...
```

Agora quando você executar o Plone e conectar ao skin, esta função será chamada, no console:

```
--Return--
> /var/tmp/python2.3-2.3.2-root/usr/lib/python2.3/pdb.py(992)set_trace()->None
-> Pdb().set_trace()
(Pdb)
```

Você pode digitar “help” para obter a lista de opções de ajuda. Os dois principais comandos são *n* para o próximo item e *s* para passar pelo item. Por exemplo:

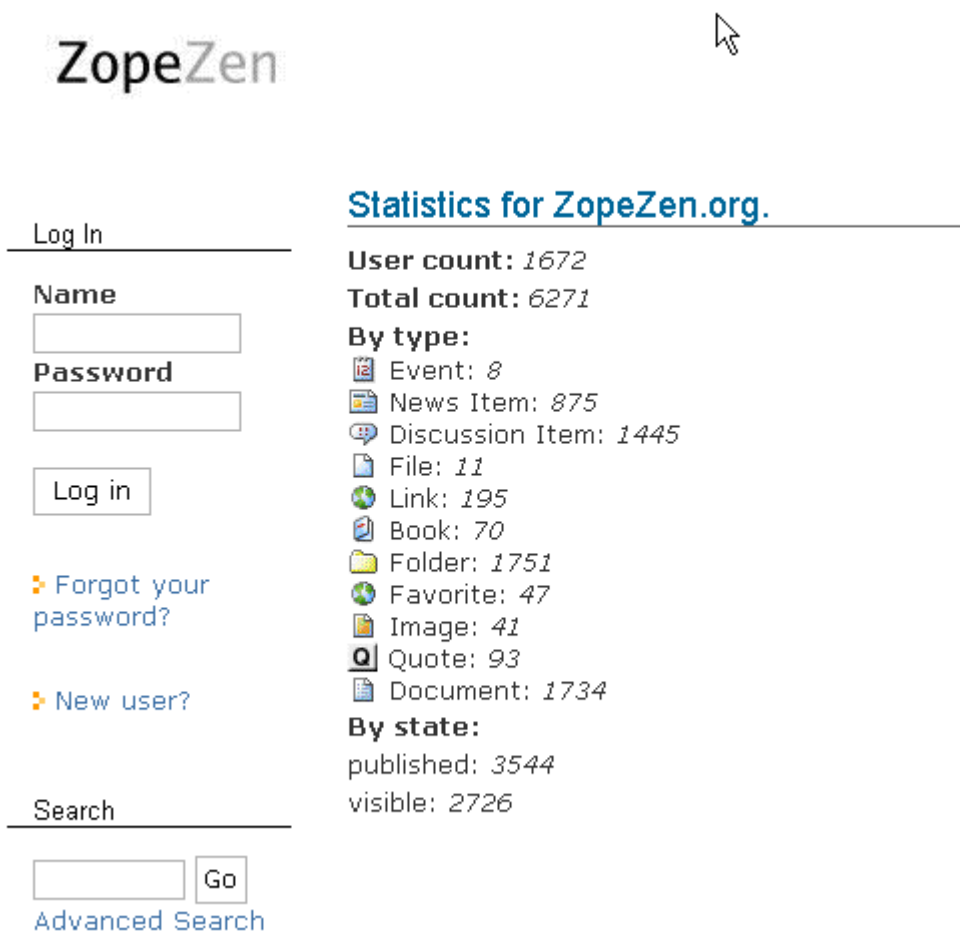
```
(Pdb) n
> /var/zope.zeo/Products/PloneSilverCity/PloneSilverCity.py(97)getLanguages()
-> langs = []
(Pdb) n
```

```
> /var/zope.zeo/Products/PloneSilverCity/PloneSilverCity.py(99)getLanguages()
-> for value, description in list_generators():
(Pdb) langs
[]
```

Para mais informações sobre o debugger, recomendo a documentação online no Python Web site (<http://python.org/doc/current/lib/module-pdb.html>). Você tem outros modos de debug no Zope, tal como utilizar o ZEO para obter um interpretador, por exemplo. O Capítulo 14 explica como utilizar o ZEO. Ambientes integrados do desenvolvedor tal como Wing (<http://wingide.com/wingide>) ou Komodo (<http://www.activestate.com/Products/Komodo>) podem, remotamente, fazer o debug nas instâncias Zope além de contar com uma ótima interface gráfica.

Escrevendo e Customizando uma Ferramenta

Escrever e customizar uma ferramenta é mais simples que desenvolver um tipo novo de conteúdo, porque pouco se tem a fazer em termos de registro de produto e a mexer na interface do usuário é relativamente simples. Por exemplo, utilizo uma simples ferramenta de estatísticas em meu ZopeZen Web site (<http://www.zopezen.org>) para dar informações da quantia de conteúdo, número de usuários, e assim por diante. Esta simples ferramenta imprime alguns números que me interessa como um administrador de site. Figura 12-2 mostra o status do meu ZopeZen.



The screenshot shows the ZopeZen website interface. On the left, there is a login form with fields for 'Name' and 'Password', a 'Log in' button, and links for 'Forgot your password?' and 'New user?'. Below the login form is a search bar with a 'Go' button and a link for 'Advanced Search'. On the right, there is a section titled 'Statistics for ZopeZen.org.' which displays the following data:

- User count: 1672
- Total count: 6271
- By type:
 - Event: 8
 - News Item: 875
 - Discussion Item: 1445
 - File: 11
 - Link: 195
 - Book: 70
 - Folder: 1751
 - Favorite: 47
 - Image: 41
 - Quote: 93
 - Document: 1734
- By state:
 - published: 3544
 - visible: 2726

Figura 12-2. PloneStats no ZopeZen

Estas são estatísticas de um sites Web, posso adquiri-las analisando os registros (logs) Web para meu servidor Plone. Ferramentas tal como Analog, Webalizer, WebTrends, e assim por diante, podendo facilmente analisar os registros do seu Plone ou Apache para você. Você pode encontrar o código para este projeto no Collective em <http://sf.net/projects/collective> no pacote PloneStats.

Iniciando a Ferramenta

Você deve colocar a ferramenta no diretório do produto do mesmo modo você faz com o tipo de conteúdo, criando um diretório dentro da instância home product. Dentro desta pasta adicione os arquivos *refresh.txt*, *install.txt*, *readme.txt*, e *__init__.py*.

Neste diretório, o módulo principal é chamado *stats.py*, que contém todo o código para criar os stats (estatísticas). Explicarei como o módulo faz a procura sem nenhuma códficação extra. Porém, desde que você esteja diretamente conectado com outras ferramentas Plone, rodar fora do Zope faz o menor sentido.

Lista 12-13 mostra o “start” da ferramenta. Esta é uma versão simples que tem dois métodos: um para retornar o número de tipos de conteúdos pelo tipo e pelo estado do workflow e o outro método para retornar o número de usuários no site.

Lista 12-13. Objeto Básico do Stats

```
class Stats:
    def getContentTypes(self):
        """ Returns the number of documents by type """
        pc = getToolByName(self, "portal_catalog")
        # call the catalog and loop through the records
        results = pc()
        numbers = {"total":len(results),"bytype":{},"bystate":{}}
        for result in results:
            # set the number for the type
            ctype = str(result.Type)
            num = numbers["bytype"].get(ctype, 0)
            num += 1
            numbers["bytype"][ctype] = num

            # set the number for the state
            state = str(result.review_state)
            num = numbers["bystate"].get(state, 0)
            num += 1
            numbers["bystate"][state] = num
        return numbers

    def getUserCount(self):
        """ The number of users """
        pm = getToolByName(self, "portal_membership")
        count = len(pm.listMemberIds())
        return count
```

Transformando o Pacote em uma Ferramenta

Para tornar um pacote numa ferramenta, você tem que fazer o mesmo processo de um tipo de conteúdo. Em outras palavras, você tem que registrar a ferramenta no módulo `__init__.py`. Apenas como no exemplo do tipo de conteúdo, você deverá criar um arquivo `config.py` que contenha todas as configurações. O arquivo `config.py` se parece com:

```
from Products.CMFCore import CMFCorePermissions

view_permission = CMFCorePermissions.ManagePortal

product_name = "PloneStats"
unique_id = "plone_stats"
```

A segurança para este produto é simples, mas isso é porque tudo é interativo com outras ferramentas e produz algumas estatísticas. Não há nada que os usuários possam adicionar, editar, apagar, ou de uma outra forma de interação. Isto significa que você só possui uma permissão a *ManagePortal*, que é a permissão que administra a configuração do Plone e é normalmente dada apenas para administradores. Significando que apenas administradores podem entrar na ZMI e ver a informação que a ferramenta fornece. Você pode facilmente adicionar um skin para seu painel de controle do Plone ou um portlet que mostra esta informação, se você quiser, no seu site.

Retornando ao `__init__.py`, adicione o código de inicialização a esta ferramenta. Há um script especial de inicialização para ferramentas, chamada *ToolInit*. Nesta Ferramenta, o arquivo `__init__.py` se parece com:

```
from Products.CMFCore import utils
from stats import Stats
from config import product_name

tools = (stats.Stats,)

def initialize(context):
    init = utils.ToolInit( product_name,
                          tools = tools,
                          product_name = product_name,
                          icon='tool.gif'
                          )
    init.initialize(context)
```

A função *ToolInit* pode pegar múltiplas ferramentas; neste caso, você só possui uma. Se você tiver múltiplas ferramentas, você pode ter apenas um nome e ícone de produto para mostrar na ZMI, que é o necessário para registrar a ferramenta. Agora você precisa completar o módulo principal para torná-lo uma ferramenta.

Alterando o Código da Ferramenta

Em seguida, adicione o código para a classe para dentro da ferramenta. Como um tipo de conteúdo, isto é o mesmo que adicionar segurança através de herança de classe base:

```

from Globals import InitializeClass
from OFS.SimpleItem import SimpleItem
from AccessControl import ClassSecurityInfo

from Products.CMFCore.utils import UniqueObject, getToolByName

```

A classe *SimpleItem* é a classe base padrão para um objeto simples (não uma pasta) em Zope. De fato, todos os tipos de conteúdos herdados de uma classe que, de algum lugar na hierarquia da classe, herda de *SimpleItem*; isto é justamente do que você não precisa, ou seja, de todos aqueles atributos extras que as outras classes fornecem. O *UniqueObject* assegura que será apenas uma instância deste objeto dentro do seu Plone site e não pode ser renomeado ou movido. Isto significa que seu objeto sempre estará disponível.

Em seguida, você importa as variáveis do arquivo de configuração. Atribuindo um ID a seu objeto, assegure que a ferramenta tenha um ID *único* no arquivo de configuração, neste caso, *plone_stats*. As duas classes base para a ferramenta são essas *UniqueObject* e *SimpleItem* classes, que é o mínimo do que você precisa. Por exemplo:

```

from config import view_permission, product_name, unique_id

class Stats(UniqueObject, SimpleItem):
    """ Prints out statistics for a Plone site """
    meta_type = product_name
    id = unique_id

```

Em seguida, você precisa ajustar a segurança, e de novo irá usar a classe *ClassSecurityInfo* para ajustar as permissões explícitas dos métodos. Por exemplo:

```

security = ClassSecurityInfo()
security.declareProtected(view_permission, 'getContentTypes')
def getContentTypes(self):
    ...

```

Adicionando alguns Usuários a Elementos de Interface

O código principal está completo, assim está na hora de mostrar alguns resultados para os usuários quando eles clicarem na ferramenta na ZMI. Para isto, você irá alterar a ZMI de modo que você possa mostrar algo.

Especificamente, você desenvolverá uma “page template” que faz o que você quer e será “fixada” na ZMI. A ZMI é uma interface de usuário pouco sofisticada que retorna páginas Web para o usuário. Tudo que você precisa fazer é desenvolver um HTML e adicionar o seguinte:

```
<span tal:replace="structure here/manage_tabs" />
```

A função *tal:replace* adquiri as abas de administração e as fazem aparecerem no topo da página. Minha ZMI executa dois métodos da ferramenta *plone_stats* e mostra os resultados para os usuários, como mostra na Lista 12-14.

Lista 12-14. Mostra a Página da Interface de Administração

```

<html>
<body>
<span tal:replace="structure here/manage_tabs" />

<p>Statistics for this Plone site.</p>

<h3>Content Types</h3>
<span tal:define="numbers here/getContentTypes">
  <p>
    Total count: <i tal:replace="numbers/total" /><br />
    Content types by type:
  </p>

  <span tal:repeat="type python:numbers['bytype'].keys()">
    <ul>
      <li>
        <span tal:replace="type" />:
        <i tal:replace="python: numbers['bytype'][type]" />
      </li>
    </ul>
  </span>

  <p>Content types by state:</p>
  <span tal:repeat="type python:numbers['bystate'].keys()">
    <ul>
      <li>
        <span tal:replace="type" />:
        <i tal:replace="python: numbers['bystate'][type]" />
      </li>
    </ul>
  </span>
</span>

<h3>Users</h3>
<p>
  User count: <i tal:replace="here/getUserCount" />
</p>

</body>
</html>

```

Isto é chamado *output.pt* e colocado dentro de um diretório *www*. Você não tem que utilizar um diretório separado, mas fica mais fácil de se lembrar.

O último passo é fixar isso na ZMI para o seu produto. Você faz isto retornando à classe *Stats* e adicionando o seguinte (primeiro importe a classe *PageTemplateFile* que pode cuidar do template do sistema de arquivos):

```
from Products.PageTemplates.PageTemplateFile import PageTemplateFile
```

Então você registra a “page template” como um método do produto que pode ser acessado. Em seguida, o método *outputPage* pode agora ser chamado pela Web, e a “page template” adequada é retornada:

```
outputPage = PageTemplateFile('www/output.pt', globals())
security.declareProtected(view_permission, 'outputPage')
```


Finalmente, as abas no topo da ZMI são determinadas por um conjunto de chamadas *manage_options* que mantém a lista de todas as abas mostradas na página. Você precisa inserir uma nova página de administração, fazendo o seguinte:

```
manage_options = (
    {'label':'output', 'action':'outputPage'},
) + SimpleItem.manage_options
```

Testando a Ferramenta

Agora a ferramenta está pronta, você pode testar se ela está funcionando. Primeiro, reinicie sua instância Plone e assim que ele ler o diretório *product* irá registrar sua nova ferramenta. Segundo, acesse a ZMI e vá para a caixa/campo “Add drop-down” no topo do canto direito. Você irá notar que o *PloneStats* está agora na lista drop-down, assim selecione ele e clique Add. O próximo formulário irá listar as ferramentas disponíveis no produto *PloneStats*; neste caso, apenas um irá aparecer, como mostra na Figura 12-3.

Add PloneStats

 Note: these are normally only useful inside a CMF site.

PloneStats

Add

Figura 12-3. Adicionando a Ferramenta.

Selecione a ferramenta, e clique no botão “Add”. Para testar se a ferramenta está funcionando, clique nela. Você deve ver uma série de estatísticas, como já visto no ZopeZen.

Esta ferramenta é simples porque, não estou certo de que apresentações às pessoas querem; se eu criar uma ferramenta padrão de relatório, então você pode utilizá-la como quiser. Algumas boas idéias de implementações futuras são as páginas no painel de controle, um pequeno portlet box, um PDF - Portable Document Format que contém ótimos gráficos e a possibilidade de se enviar um email contendo estas informações para o administrador, ou utilizar estas informações através de um programa externo, do tipo “Crystal Reports”.