SENADO FEDERAL
Secretaria Especial do Interlegis - SINTER
Subsecretaria de Tecnologia da Informação - SSTIN

Relatório II

Programa Desenvolvido, Testado e Documentado

Consultor: Felipe Luis de Souza Vieira (felipe.tio@gmail.com)

Produto: Metabus

Contrato:2008/000750

Brasília - DF

Outubro / 2008

Sumário

1 Introdução

Este documento tem por objectivo principal apresentar os aspectos técnicos da implementação escolhida para as funcionalidade do Metabus. Ainda, são apresentados trechos de código para ilustrar e enriquecer a descrição das implemtentações adotadas.

1.1 Django

Django é o framework Python adotado como base para o desenvolvimento do Metabus. Este software além proporcionar recursos para um desenvolvimento ágil de aplicações Web, ainda sugere uma organização elegante do código fonte.

A separação de interesses sugerida nesse framework possui uma nomenclatura sutilmente diferente da nomenclatura comumente adotada por vários frameworks de aplicações Web. Ao invés do conhecido MVC (Model View and Controller) utiliza-se MTV (Model Template and View) assim o elemento 'View' do MVC chama-se 'Template' no MTV e o 'Controller' do MVC chama-se 'View' no MTV.

Processamento de requisições Web

As requisições Web são mapeadas para uma 'View' através do arquivo urls.py. Nesse arquivo podemos configurar expressões regulares que identificam cada uma das urls e associam a uma determinada função de callback (View). O código abaixo é um trecho do arquivo urls.py do módulo Interface Gráfica do Metabus e ilustra como é feita a associação de uma URL de requisição Web a uma *View*:

```
from django.conf.urls.defaults import *

urlpatterns = patterns('',
    url (r'^$', 'metabus.ui.views.home', name='home'),
    url (r'^search/$', 'metabus.ui.views.search', name='search'),
)
```

Com esse código é feita a associação da url http://metabus.aquarela3.com.br/search/ para a View metabus.ui.views.search e da http://metabus.aquarela3.com.br (raiz do site) para metabus.ui.views.home. Esta função será a responsável por processar o dados necessários para montar a resposta á requisição.

Camada para acesso aos dados

O Django oferece um conjunto de classes para acesso e manipulação dos dados contidos na base de dados. Esta camada é responsável também por tornar o sistema independente do Sistema de Gerenciamento de Banco de Bados - SGBD, desta forma, pode ser utilizado qualquer banco do dados que seja suportado pelo Django (Postgres, MySQL, SQLite3 e Oracle) e as tarefas de conexão, criação das tabelas, obtenção de dados, inserções e alterações são todas efetuadas através dessa camada de um maneira única independente do SGBD utilizado.

Abaixo segue um trecho do arquivo models.py onde são definidas as entidade relacionadas com as fontes de busca:

```
class Parser(models.Model):

    LANGUAGES = (
        ('xslt', 'XSLT'),
    )

    name = models.CharField(_('Name'), help_text=_('Name of parser'), max_length=250)
    description = models.TextField(_('Description'), help_text=_('Parser description and documentation'),
max_length=1024, blank=True)
    language = models.CharField(_('Language'), help_text=_('Parser file language'), max_length=250,
choices=LANGUAGES, default='xslt')
    file = models.FileField(_('File'), help_text=_('Parser file'), upload_to = 'parser/%Y/%M')

class Group(models.Model):
    name = models.CharField(_('Name'), help_text=_('Group name'), max_length=250)
    parent = models.ForeignKey('self', help_text=_('Parent group'), null=True, blank=True)


class Source(models.Model):

    name = models.CharField(_('Name'), help_text=_('Name of search engine'), max_length=250)
    url = models.CharField(_('Search URL'), help_text=_('URL of search engine'), unique=True, max_length=250)
    is_rss = models.BooleanField(_('Is RSS'), help_text=_('Mark it if the search engine return RSS data'),
default=True)
    op_and = models.CharField(_('AND'), help_text=_('AND operator string to this search'), max_length=250,
default='AND')
    op_or = models.CharField(_('OR'), help_text=_('OR operator string to this search'), max_length=250,
default='OR')
    cache_expires = models.IntegerField(_('Cache expires'), help_text=_('Seconds time to exipires cache
results'), default=60*60*24) # Default one day
    parser = models.ForeignKey(Parser, help_text=_('Parser file to transform the response in RSS'), null=True,
blank=True)
    group = models.ManyToManyField(Group, help_text=_('Group of this source'), null=True, blank=True)
```

Com código acima o Django provê a criação de tabelas, e o total controle sobre os dados. Para exemplificar a manipulação dos dados segue um trecho do arquivo search.py:

```
    sources = Source.objects.all()
    for source in sources:
        self.source_array.append(source.id)
```

Estrutura de templates

Os templates do Django foram desenvolvidos para estabelecer um equilíbrio entre poder e facilidade. Seu principal objetivo é gerar um arquivo baseado em texto formatado, por exemplo HTML.

Os arquivos de template contêm variáveis, quando o template é processado essas variáveis são substituídas por valores. Além de variáveis os arquivos contêm tags que controlam a lógica do template. A estrutura de templates pode ser organizada com herança de arquivos, que possibilita criar templates base com as definições de elementos comuns a todos os documentos do site, assim como blocos que permitem que outros arquivos sobrescrevam o seu conteúdo como pode ser observado no template base do metabus, apresentado no código abaixo:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-
strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en-us" xml:lang="en-us">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

    {# Block title should be overwrite #}
    <title>{% block title %}Metabus - Interlegis{% endblock %}</title>

    {# Base stylesheet could be overwrite, however the new css have to import the base stylesheet #}
    <link type="text/css" rel="stylesheet" href="{{MEDIA_URL}}css/{% block stylesheet %}base.css{%
endblock %}" />

    {# Jquery Lib #}
    <script type="text/javascript" src="{{MEDIA_URL}}js/jquery-1.2.6.js"></script>

    {# Base js definition#}
    <script type="text/javascript" src="{{MEDIA_URL}}js/base.js"></script>

    {# Template js definition. Should be overwrite to import a new js file#}
    <script type="text/javascript" src="{{MEDIA_URL}}js/{% block javascript %}blank.js{% endblock
%}"></script>
</head>

<body>
    <div id="container_dialog">
        <p></p>
    </div>

    <div id="container_top">
        <div id="container_bottom">
            <div id="container_body">

                <div id="top">
                    ...
                </div>

                <hr class="clear"/>

                <div id="header">
```

```
                    <img width="276" height="110" src="{{MEDIA_URL}}img/logo.gif" alt="Metabus"/>
                    <form id="search-form" method="GET" action="{% url search %}">
                        <fieldset>
                            <input type="text" id="keywords" name="keywords"/>
                            <input type="submit" value="search"/>
                        </fieldset>
                        <fieldset id="sources">
                            <ul>
                                {% for group in groups %}
                                <li>
                                    <input class="groups" id="{{ group.name }}" type="checkbox" value="{{
group.id }}" name="groups"/>
                                    <label class="groups" for="{{ group.name }}">{{ group.name }}</label>
                                    <fieldset class="groups">
                                        <ul>
                                            {% for source in group.sources %}
                                            <li>
                                                <input id="{{ source.name }}" type="checkbox"
value="{{ source.id }}" name="sources"/>
                                                <label for="{{ source.name }}">{{ source.name }}</label>
                                            </li>
                                            {% endfor %}
                                        </ul>
                                    </fieldset>
                                </li>
                                {% endfor %}
                                <li>
                                    ...
                                </li>
                            </ul>
                        </fieldset>
                    </form>
                </div>

                <div id="content">
                    {% block content %}
                    {% endblock %}
                </div>

                <hr class="clear"/>

                <div id="footer">
                    ...
                </div>
            </div>
        </div>
    </div>
</body>
</html>
```

Interface administrativa automática

A interface de administração do metabus é provida pelo Django que cria um site de administração a partir do modelo da base de dados, disponibilizando uma interface unificada para modificar o conteúdo da base de dados do sistema.

O arquivo admin.py contém todas as configurações que controlam como essas tabelas aparecem no painel administrativo. A interface de administração não tem a intenção de ser usada pelos visitantes do site, mas sim pelos administradores que irão adicionar e manter as fontes de busca do sistema.

O arquivo que define as configurações da interface administrativa automática é apresentado abaixo:

```
from django.contrib import admin
from django.utils.translation import ugettext_lazy as _
from metabus.source.models import Source, Parser, Group

class ParserAdmin(admin.ModelAdmin):
    fieldsets = (
        (None, {'fields': ('name', 'description',)}),
        (_('File'), {'fields': ('language', 'file')}),
    )

    verbose_name = _('Parser')

class SourceAdmin(admin.ModelAdmin):
    fieldsets = (
        (None, {'fields': ('name', 'url','cache_expires')}),
        (_('Operators'), {'classes': ['collapse'], 'fields': ('op_and', 'op_or',)}),
        (_('Parser'), {'fields': ('is_rss', 'parser')}),
        (_('Groups'), {'fields': ('group',)})
    )

    verbose_name = _('Source')


class GroupAdmin(admin.ModelAdmin):
    verbose_name = _('Group')


admin.site.register(Parser, ParserAdmin)
admin.site.register(Source, SourceAdmin)
admin.site.register(Group, GroupAdmin)
```

2 Modularização do Metabus

A modularização dos arquivos e diretórios é extremamente importante, principalmente para permitir a escalabilidade do sistema. Os módulos foram separados a fim de reutilizar o código até mesmo por aplicacões que implementem uma nova interface gráfica. Esta organização foi concebida de modo a facilitar o controle e o entendimento do todo, facilitando assim a expansão organizada e a manutenção.O repositório está dividido em dois projetos, com os seguintes arquivos e diretórios:

- mbclient: projeto da interface gráfica do sistema.

```
deploy/
    apache2
    mbclient.wsgi
__init__.py
manage.py
settings.py
ui/
__init__.py
media/
    css/
        base.css
        result.css
    img/
    js/
        base.js
templates/
    base.html
    result.html
views.py
urls.py
```

- mbserver: projeto dos servidores responsável por centralizar e requisitar paralelamente a busca nas fontes selecionadas.

```
cache/
    backends/
        __init__.py
        mb.py
    __init__.py
    models.py
    views.py
deploy/
    apache2
    mbserver.wsgi
```

```
__init__.py
logger.conf
manage.py
middleware/
    __init__.py
    setupmw.py
server/
    core/
        __init__.py
        parser.py
        pool.py
        worker.py
    __init__.py
    lib/
        __init__.py
        utils.py
    request/
        controller.py
        __init__.py
    settings.py
    urls.py
    views.py
server_lib/
    __init__.py
    search.py
    server.py
settings.py
source/
    admin.py
    __init__.py
    models.py
    views.py
urls.py
```

Os módulos e as principais aplicações serão apresentadas e detalhadas nas próximas seções.

2.1 Servidor

O servidor é responsável por pesquisar pelas palavras-chave da busca do usuário em todas as fontes cadastradas. Assim que um resultado é obtido, são aplicados parsers nesse resultado a fim de extrair as informações relevantes do resultado da busca.

O servidor possui um parser de RSS somente. Dessa forma, sempre que a fonte de busca não disponibiliza os resultados em RSS uma etapa adicional de parsing processa o resultado por meio de uma transformação XSLT, que está associada a uma fonte. Essa etapa visa transformar um documento XHTML em RSS, para que o servidor possa interpretar os resultados.

Para de obtenção de resultados das fontes de busca, o servidor mantém um pool de threads para efetuarem as requisições HTTP para as fontes. Abaixo segue um trecho do código desse pool de threads retirado do arquivo pool.py:

```
class MBThreadPool(Singleton):
    """
        This class must be Singleton, becouse it's suppose to exist only one
        of this pool per process. The total number of threads will be
        the number of process x number of MBThreadPool threads.
    """
    __pool = None

    def __init__ (self):
        if MBThreadPool.__pool is None:
            self.ioworkers = []
            self.parser = None
            self.requests = Queue(0)
            self.bg_requests = Queue(0)
            self.responses = Queue(0)

            for proc in range(settings.THREADS_NUMBER):
                w = WorkerThread(self.requests, self.bg_requests, self.responses)
                w.start()
                self.ioworkers.append(w)

            self.parser = ParserThread(self.responses)
            self.parser.start()
```

Todas as requisições a uma fonte são inseridas na fila *requests* do pool. As threads ficam verificando essa fila a procura de requisições, sempre que uma thread encontra uma requisição ela a retira da fila e efetua esta requisição. Quando essa thread termina de obter o resultado da busca ela insere o documento resultante da busca em outra fila chamada *responses*, da mesma forma existe uma thread que observa essa fila a procura de documentos obtidos quando um documento é encontrado ele entra na fase de *parsing*, onde os dados são extraídos desses documentos e armazenados no cache.

2.2 Server Lib

A arquitetura do metabus permite que vários servidores sejam cadastrados e utilizados para realizar parte do processamento das buscas. Sendo assim, tornou-se necessário centralizar a informação do cadastro de servidores. O módulo server_lib além de possuir tal cadastro, ainda é responsável pelo balanceamento de carga entre eles, distribuindo as pesquisas de maneira uniforme para todos os servidores cadastrados. Uma vez que, a comunicação entre módulo de interface gráfica e o servidor é feita exclusivamente através do server_lib.

Segue o código da função que realiza o acionamento do servidor para efetuar as buscas necessárias:

```
    def do_search(self):
        """
            Metabus server works as an asynchronous server. It's only possible
            to retrive any results after the timeout.
        """

        values = {
                    'sources' : simplejson.dumps(self.source_array),
```

```
                    'keyword' : self.keywords.encode('utf8'),
                }

        # Get a server in a RR queue
        server = Server()
        url = server.get() + 'server/process/'

        data = urllib.urlencode(values)
        req = urllib2.Request(url, data)
        response = urllib2.urlopen(req)

        self.started = True
        self.start_time = time.time()
```

Cache

Toda vez que um usuário realiza uma busca o servidor faz cálculos, requisições para outras páginas e consultas na base de dados para montar o resultado da pesquisa. Para a maioria dos sites o processamento do servidor não é um gargalo, entrentando quando pretende-se atender a um grande número de buscas o número de requisições pode ser muito alto pois cada requisição de busca feita pelo usuário pode consultar diversas fontes em apenas uma pesquisa.

Para melhorar o desempenho e evitar que pesquisas iguais sejam requisitadas mais de uma vez em um curto espaço de tempo um mecanismo de cache foi introduzido no sistema a fim de salvar resultados de buscas recentes.

O código a seguir mostra a interface das principais funções do cache do metabus baseado no sistema de cache do próprio Django:

```
class BaseCache(object):
    def add(self, key, value, timeout=None):
        """
        Set a value in the cache if the key does not already exist. If
        timeout is given, that timeout will be used for the key; otherwise
        the default cache timeout will be used.

        Returns True if the value was stored, False otherwise.
        """
        raise NotImplementedError

    def get(self, key, default=None):
        """
        Fetch a given key from the cache. If the key does not exist, return
        default, which itself defaults to None.
        """
        raise NotImplementedError

    def has_key(self, key):
        """
        Returns True if the key is in the cache and has not expired.
        """
        return self.get(key) is not None
```

O módulo de cache do metabus implementa as funções de cache definidas acima. A remoção das pesquisas salvas é feita no momento em que se obtem uma entrada do cache, se a pesquisa estiver expirada ela é removida da base de dados, como mostra o código abaixo:

```
    def get(self, key, default=None):
        try:
            key_row = Key.objects.filter(source=key["source"],keyword=key["keyword"]).get()
        except Key.DoesNotExist:
            return default
        else:
            now = datetime.now()
            if key_row.expires < now:
                Value.objects.filter(key=key_row).delete()
                key_row.delete()
                return default
            return Value.objects.filter(key=key_row).all()
```

2.3 Interface Gráfica

Como foi dito, a interface gráfica está separada da implementação do servidor para tornar mais fácil a implementação de uma outra interface. O layout do metabus é semelhante aos buscadores existentes e contém elementos simples e claros para a realização das buscas.

Independente da interface implementada deverá ser fornecido um formulário, para o usuário requisitar a busca ao servidor que retornará os resultados da pesquisa (título, descrição, link e fonte dos resultados encontrados). O código abaixo apresenta o formulario de busca gerado pelo template do metabus:

```
<form action="/search" method="get" id="search-form">
    <fieldset>
        <input type="text" name="keywords" id="keywords"/>
        <input type="submit" value="search"/>
    </fieldset>
    <fieldset id="sources">
        <ul>
            <li>
                <input type="checkbox" name="groups" value="1" id="Rio de Janeiro" class="groups"/>
                <label for="Rio de Janeiro" class="groups">Rio de Janeiro</label>
                <fieldset class="groups">
                    <ul>
                        <li>
                            <input type="checkbox" name="sources" value="7" id="Prefeitura Municipal do
Buzios"/>
                            <label for="Prefeitura Municipal do Buzios">Prefeitura Municipal do
Buzios</label>
                        </li>
                        <li>
                            <input type="checkbox" name="sources" value="6" id="Prefeitura Municipal do
Rio de Janeiro"/>
                            <label for="Prefeitura Municipal do Rio de Janeiro">Prefeitura Municipal do
```

```
Rio de Janeiro</label>
                        </li>
                    </ul>
                </fieldset>
            </li>
            <li>
                <input type="checkbox" name="groups" value="2" id="São Paulo" class="groups"/>
                <label for="São Paulo" class="groups">São Paulo</label>
                <fieldset class="groups">
                    <ul>
                        <li>
                            <input type="checkbox" name="sources" value="4" id="Prefeitura Municipal de
Diadema"/>
                            <label for="Prefeitura Municipal de Diadema">Prefeitura Municipal de
Diadema</label>
                        </li>
                        <li>
                            <input type="checkbox" name="sources" value="1" id="Prefeitura Municipal de
Mogi Mirim"/>
                            <label for="Prefeitura Municipal de Mogi Mirim">Prefeitura Municipal de Mogi
Mirim</label>
                        </li>
                        <li>
                            <input type="checkbox" name="sources" value="2" id="Prefeitura Municipal de
São Paulo"/>
                            <label for="Prefeitura Municipal de São Paulo">Prefeitura Municipal de São
Paulo</label>
                        </li>
                    </ul>
                </fieldset>
            </li>
        </ul>
    </fieldset>
</form>
```

Os dados dos formulários são enviados para uma view que faz uma requisição para o server_lib após validar os campos. O código abaixo mostra esta etapa da busca.

```
def search(request):
    params = get_search_params()
    error = False

    if 'keywords' in request.GET and len(request.GET['keywords']) > 0:
        params['search_keywords'] = request.GET['keywords']
    else:
        params['search_keywords_error']= _('Please type one keyword to search')
        error = True

    if 'sources' in request.GET and len(request.GET['sources']) > 0:
        params['search_sources'] = request.GET.getlist('sources')
    else:
        params['search_sources_error']= _('Please select one or more sources to search')
        error = True
```

```
    if error:
        t = loader.get_template('base.html')
        c = RequestContext(request, params)
        return HttpResponse(t.render(c))

    search = Search(params['search_keywords'], sources=params['search_sources'])
    search.do_search()
    params['results'] = search.get_results()

    t = loader.get_template('result.html')
    c = RequestContext(request, params)
    return HttpResponse(t.render(c))
```

3 Logging

Todos os logs do sistema são feito através do módulo nativo Python chamado *logging*. Este módulo é configuravel através de um arquivo de configuração chamado logger.conf. Segue o conteúdo desse arquivo:

```
[loggers]
keys=root,metabus

[handlers]
keys=handmetabus

[formatters]
keys=form01

[formatter_form01]
format=%(asctime)s [%(process)d] %(levelname)s %(message)s
datefmt=%Y-%m-%d %H:%M:%S
class=logging.Formatter

[handler_handmetabus]
class=handlers.RotatingFileHandler
level=DEBUG
formatter=form01
args=("/home/metabus/var/log/metabus/mbserver.log", 'a')

[logger_root]
level=NOTSET
handlers=handmetabus

[logger_metabus]
level=DEBUG
handlers=handmetabus
propagate=0
qualname=metabus
```

Os níveis de criticidade dos logs são determinados no momento de armazenar um registro de log. O nivel do log é indicado da seguinte meneira:

```
import logging
log = logging.getLogger('metabus')
log.info('Message')
log.debug('Message')
log.warning('Message')
log.error('Message')
log.critical('Message')
```

O código acima gerará os seguintes registros no arquivo de log:

```
2008-10-21 11:11:30 [5854] INFO Message
2008-10-21 11:11:30 [5854] DEBUG Message
2008-10-21 11:11:30 [5854] WARNING Message
2008-10-21 11:11:30 [5854] ERROR Message
2008-10-21 11:11:32 [5854] CRITICAL Message
```

4 Considerações Finais

Neste documento foram apresentados os pontos principais do sistema bem como o código python utilizado em cada situação. O download do Metabus pode ser feito através do repositório de código utilizando o seguinte comando:

```
svn checkout http://metabus.googlecode.com/svn/trunk/ metabus
```

Uma replicação deste código também pode ser encontrado no repositório do colab, no endereço: http://repositorio.interlegis.gov.br/metabus/trunk/

Uma instalação do Metabus está disponível em http://metabus.aquarela3.com.br. Esta instalação é utilizada para realização de testes e para antecipar uma apresentação deste produto. Este documento também está disponível em http://code.google.com/p/metabus/wiki/RelatorioDeCodigo.