Relatório de Entrega de nº 2
Produto: SPDO

Especificação dos Programas

Elaborado por
Consultor PNUD
Fabiano Weimar dos Santos

Supervisor
Claudio Morale

Contrato número
2011/000260

Versão
001

INTERLEGIS
Comunidade Virtual do Poder Legislativo

Índice

1. Introdução....3
2. Especificação dos Programas....3
2.1 SQLAlchemy....3
2.2 Zope Component Architecture - ZCA....4
2.3 Grok....5
2.4 jQuery....5
2.5 z3c.form....5
2.5.1 Exemplo: formulário de inclusão....6
2.5.2 Exemplo: formulário de alteração....7
2.6 API....8
2.7 views....10
2.7.1 Exemplo: template ZPT....11
3. Banco de Dados Versionado....13
4. Auditoria de Acesso....15
5. Integração com E-mail....16
6. Integração por Webservices....17
7. Considerações Finais....17
8. Referências....18

1. Introdução

No "Relatório de Entrega de nº 1" foram apresentados os casos de uso do sistema SPDO, assim como os modelos de dados, interfaces de usuários e uma detalhada descrição dos programas. Nesse relatório estão descritas as **especificações dos programas**, de acordo com o Termo de Referência 135641. Será dada especial atenção as tecnologias envolvidas e na forma como os programas anteriormente descritos devem ser implementados.

Esse relatório técnico assumirá que o leitor esteja familiarizado com as tecnologias Python, Zope, CMF, ZPT e os bancos de dados MySQL e PostgreSQL, requisitos obrigatórios do contrato 2011/000260 que regra esse projeto.

2. Especificação dos Programas

Alguns requisitos do sistema SPDO influenciaram significativamente nas decisões sobre como os programas deveriam ser especificados. Esse relatório explanará a respeito das especificações dos programas que compõem o sistema SPDO, explicando quais tecnologias foram adotadas para implementar os requisitos já descritos, assim como essas tecnologias foram utilizadas na prática.

2.1 SQLAlchemy

Um dos requisitos mais importantes do sistema SPDO diz respeito a utilização de bancos de dados relacionais para armazenar as informações e, ao mesmo tempo, implementar um sistema que fosse compatível com MySQL e Postgresql. Alguns sistemas mais antigos do Interlegis (como o SAPL [1]) implementam essa compatibilidade manualmente, precisando manter diferentes versões das mesmas consultas em dois "dialetos" de SQL.

Atualmente existem maneiras mais adequadas de tratar esse tipo problema em linguagens de programação orientadas a objeto. Por estarmos desenvolvendo um sistema em Python, a alternativa escolhida foi adotar o framework SQLAlchemy - The Python SQL Toolkit and Object Relational Mapper [2].

O SQLAlchemy é uma ferramenta de mapeamento objeto-relacional que implementa padrões de projeto eficientes e de alta performance, sem perder a simplicidade da linguagem Python. O SQLAlchemy suporta MySQL, PostgreSQL e muitos outros SGDB, de acordo com a tabela descrita em [3]. Essa decisão é inclusive citada em [4], onde o autor avalia diversas alternativas de integração do Zope com bancos de dados relacionais e também opta pela utilização do SQLAlchemy.

A integração do Zope com o SQLAlchemy é feita através do pacote z3c.saconfig [5],

responsável por estabelecer as conexões com os bancos de dados e integrar o sistema de controle de transação distribuída do Zope com as transações dos bancos de dados acessados através do SQLAlchemy.

Abaixo segue um exemplo de código fonte demonstrando como o SQLAlchemy deve ser utilizado para representar uma tabela, adotando a extensão "declarative" [6].

```
from sqlalchemy.ext.declarative import declarative_base
from il.spdo.history_meta import VersionedMeta
from zope.interface import implements
from il.spdo import interfaces
from il.spdo.config import TABLE_ARGS
from sqlalchemy import Column, Integer, String
from sqlalchemy.orm import relationship, backref

Base = declarative_base(metaclass=VersionedMeta)

class Area(Base):
    implements(interfaces.IArea)
    __tablename__ = 'area'
    __table_args__ = TABLE_ARGS
    id = Column(Integer, primary_key=True)
    sigla = Column(String(20), unique=True, nullable=False)
    nome = Column(String(100), unique=True, nullable=False)
    chefia_id = Column(Integer, ForeignKey('area.id'))
    chefia = relationship("Area", backref=backref('area', remote_side=id))
```

O SQLAlchemy possui uma documentação bastante completa em [7] e [8].

A implementação do modelo de dados do sistema SPDO está descrita no módulo db.py, disponível para consulta em [9].

2.2 Zope Component Architecture - ZCA

Segundo [10], o Zope Component Architecture (ZCA) é um framework destinado ao desenvolvimento de componentes, especialmente útil na implementação de grandes sistemas. Apesar do ZCA estar incluído nas versões mais modernas do Zope e do Plone [11], o ZCA não é específico do servidor de aplicações Zope e pode ser utilizado no desenvolvimento de qualquer sistema que utilize a linguagem Python.

Atualmente recomenda-se que o desenvolvimento de sistemas para a plataforma Zope utilize o ZCA. Sua correta utilização mantém o código fonte modular, extensível, reutilizável e com baixo acoplamento, características desejáveis em softwares de qualidade.

O correto entendimento do ZCA requer que o desenvolvedor compreenda e utilize alguns padrões de projeto, como: interfaces, adapters, utilities e views. Esses conceitos estão documentados em [10] [4] e foram largamente utilizados na implementação do sistema SPDO.

2.3 Grok

O ZCA é considerado um framework complexo, de difícil utilização por desenvolvedores menos experientes. Assim, optou-se por utilizar o Grok - A Smashing Web Framework [12], framework que pode ser utilizado dentro do Zope e/ou Plone.

O Grok facilita a utilização do ZCA, adotando convensões que tornam o código fonte do sistema mais simples e fácil de manter. O Grok evita que o desenvolvedor precise escrever código ZCML (linguagem de configuração, baseada em XML, utilizada pelo ZCA para configurar componentes). Isso reduz significativamente o tamanho e a complexidade do código fonte do sistema.

2.4 jQuery

Definidos os frameworks a serem utilizados nas camadas de modelo de dados (SQLAlchemy) e controle (ZCA e Grok), percebeu-se que seria necessário utilizar recursos da linguagem javascript, mais especificamente o framework jQuery [13], para melhorar a experiência dos usuários finais.

Todas as interfaces de usuários devem ser desenvolvidas utilizando Zope Page Templates – ZPT, implementadas através de views do ZCA ou, quando os templates fizerem parte do tema, implementadas como elementos da ferramenta portal_skins do CMF.

Definiu-se que o recurso jQuery Overlay [14] (que é padrão no Plone 4) seja utilizado sempre que a interface do usuário, ao implementar um caso de uso e precisar acessar um fluxo secundário, deseje evitar a mudança do contexto da navegação. Uma situação onde isso ocorre é a opção “Cadastrar Protocolo”, onde será comum acessar a opção “Cadastrar Pessoa” e não é desejável deixar o formulário de protocolo. O recurso jQuery Overlay implementa essa funcionalidade abrindo uma página como uma janela sobre a página atual. No entanto, o jQuery Overlay possui limitações: não é possível, por exemplo, abrir URLs que dependam de javascript.

Além do recurso jQuery Overlay, definiu-se que todas as informações tabulares poderão ser formatadas utilizando o recurso jQuery DataTables [15], um plug-in para a biblioteca jQuery que facilita a criação de tabelas com busca, ordenação e paginação. Um exemplo de como esse recurso pode ser utilizado encontra-se no item 2.7.1 desse relatório.

2.5 z3c.form

Uma parcela significativa do código fonte do sistema SPDO é composta por formulários e suas respectivas rotinas de validação e processamento. Optou-se pela adoção do framework

z3c.form [16] para manipulação de formulários. Recentemente esse framework foi adotado como padrão pelo projeto Plone e tem ganhado popularidade entre os desenvolvedores devido a sua flexibilidade, extensibilidade, clareza na separação de conceitos e baixo acoplamento. No plone esse framework é empacotado com o nome plone.app.z3cform [17] e é extendido através do pacote plone.directives.form [18].

Apesar de possuir extensa documentação, o z3c.form é complexo. Nesse sentido, esse relatório exemplifica nos itens 2.5.1 e 2.5.2 como os formulários do sistema devem ser implementados, de tal forma que esse código fonte possa ser adotado na construção de outros projetos.

2.5.1 Exemplo: formulário de inclusão

```
from five import grok
from zope.component import getUtility
from plone.app.layout.navigation.interfaces import INavigationRoot

from il.spdo.browser.forms import base
from il.spdo.config import MessageFactory as _
from il.spdo.interfaces import IObservacao
from il.spdo.db import Observacao
from il.spdo.log import log
from il.spdo.nav import go
from il.spdo.api import ISPDOAPI

class ObservacaoAddForm(base.AddForm):
    """Formulário de cadastro de observações.
    """

    grok.context(INavigationRoot)
    grok.name('add-observacao')
    grok.require('cmf.ManagePortal')

    schema = IObservacao
    klass = Observacao
    label = _(u'Adicionar Observação')
    description = _(u'Formulário de cadastro de observações.')

    @log
    def createAndAdd(self, data):
        api = getUtility(ISPDOAPI)
        protocolo_id = api.getProtocoloId()
        api.addObservacao(protocolo_id, data['texto'])

    def nextURL(self):
        api = getUtility(ISPDOAPI)
        protocolo_id = api.getProtocoloId()
        go('show-protocolo', id=protocolo_id)
```

2.5.2 Exemplo: formulário de alteração

```
class ObservacaoEditForm(base.EditForm):
    """Formulário de edição de uma observação.
    """

    grok.context(INavigationRoot)
    grok.name('edit-observacao')
    grok.require('cmf.ManagePortal')

    schema = IObservacao
    klass = Observacao
    label = _(u'Editar Observação')
    descrition = _(u'Formulário de edição de uma observação.')

    @log
    def applyChanges(self, data):
        content = self.getContent()
        if not content:
            return
        for k, v in data.items():
            setattr(content, k, v)
        api = getUtility(ISPDOAPI)
        content.usuario = api.getAuthId()

    def nextURL(self):
        api = getUtility(ISPDOAPI)
        protocolo_id = api.getProtocoloId()
        go('show-protocolo', id=protocolo_id)
```

Nos exemplos citados em 2.5.1 e 2.5.2 percebe-se que os formulários são codificados como classes. Isso inicialmente pode parecer estranho, pois não temos nenhum código fonte HTML. O z3c.form utiliza o conceito de "schema", gerando os campos dos formulários de acordo com uma definição de interface, como no exemplo que segue:

```
from zope import interface, schema
from plone.directives import form
from il.spdo.config import MessageFactory as _

class IBaseFormSchema(form.Schema):
    pass

class IObservacao(IBaseFormSchema):

    form.mode(id='hidden')
    id = schema.Int(
        title=_(u'ID'),
        description=_(u'Identificador da Observação.'),
```

```
        required=False)

    form.mode(protocolo_id='hidden')
    protocolo_id = schema.Int(
        title=_(u'ID do Protocolo'),
        description=_(u'Identificador do Protocolo.'),
        required=False)

    texto = schema.Text(
        title=_(u'Texto'),
        description=_(u'Texto da Observação.'))
```

Nos exemplos não é citado o código fonte das classes il.spdo.browser.forms.base.AddForm e il.spdo.browser.forms.base.EditForm, mas os mesmos estão disponíveis para consulta em [19]. Essas classes são baseadas no plone.directives.form e implementam métodos comuns a todos os formulários, como os "handlers" dos botões Cadastrar, Salvar e Cancelar. Essas classes base também definem métodos abstratos, que devem ser implementados nas classes derivadas (são os métodos que lançam a exceção "NotImplementedError").

O processamento dos formulários é implementado por dois métodos, destacados no texto em cor diferenciada. O método ObservacaoAddForm.createAndAdd é responsável pela criação das informações no banco de dados, instanciando objetos da classe db.Observacao que representa a tabela observacao (utilizando o SQLAlchemy). O método ObservacaoEditForm.applyChanges é responsável pela atualização das informações, de maneira análoga.

As validações mais simples, como a verificação de campos obrigatórios, são feitas automaticamente, de acordo com o valor do atributo "required" de cada campo. Validações mais elaboradas podem ser codificadas utilizando validadores especializados ou ainda invariantes: tipo de validador útil nas situações onde a validação de um campo depende do valor de outros campos.

A seguir são listados exemplos de validadores especializados e de invariantes, respectivamente:

```
@form.validator(field=IAddProtocolo['origem'])
def validateOrigem(value):
    if not value:
        raise interface.Invalid(_(u'Ao menos uma origem deve ser informada.'))
```

```
@interface.invariant
def vefifyAreaPai(area):
    if area.id and area.chefia_id and area.id == area.chefia_id:
        raise interface.Invalid(_(u'A área de chefia precisa ser diferente da
própria área. Se a área não possuir chefia, não selecione nenhuma área.'))
```

2.6 API

Dada a complexidade dos requisitos do projeto SPDO, optou-se por desenvolver uma API

que centraliza a implementação da lógica de negócio necessária ao funcionamento dos principais casos de uso. Essa API encontra-se em [20] é utilizada nos métodos que inserem e alteram informações nos exemplos citados em 2.5.1 e 2.5.1.

A API do sistema SPDO é implementada como uma "Global Utility", um padrão de projeto semelhante as "tools" do CMF, mas que segue recomendações de boas práticas de programação do ZCA. Esse componente é configurado utilizando diretivas do framework Grok.

Abaixo são citados trechos de código da API do sistema SPDO.

```
from zope.interface import Interface
from five import grok
from il.spdo import db
from il.spdo.config import Session

class ISPDOAPI(Interface):
    """Marker interface.
    """

class SPDOAPI(grok.GlobalUtility):
    """API SPDO.
    """
    grok.provides(ISPDOAPI)

    def _getProtocolosData(self, protocolos):
        ret = []
        for i in protocolos:
            ret.append({
               'id': i.id,
               'numero': i.numero,
               'data_protocolo': i.data_protocolo,
               'assunto': i.assunto,
               'tipodocumento': i.tipodocumento.nome,
               'situacao': i.situacao.nome,
               'url': url('show-protocolo', id=i.id),
               })
        return ret

    def getProtocolosCriadosRecebidos(self):
        """Consulta os protocolos criados ou recebidos pela área.
        """
        session = Session()
        area_id_auth = self.getAuthPessoa().area_id
        items = session.query(db.TramiteInbox).\
                filter_by(area_id=area_id_auth).all()
        return self._getProtocolosData([i.protocolo for i in items])
```

Percebe-se no código fonte do método ObservacaoAddForm.createAndAdd, citado no item 2.5.1, que a API é acessada através de uma chamada getUtility do ZCA. Abaixo, segue destacado exemplo de como a API do SPDO é acessada:

```
from zope.component import getUtility
from il.spdo.api import ISPDOAPI
api = getUtility(ISPDOAPI)
protocolo_id = api.getProtocoloId()
api.addObservacao(protocolo_id, data['texto'])
```

Nesse exemplo percebe-se que não é feito referência direta a classe que implementa a API do SPDO. Ao invés disso, utiliza-se o ZCA para ***executar métodos de uma classe que implementa a interface ISPDOAPI***. No item 2.7 desse relatório é apresentado outro exemplo de acesso a API.

Na tabela a seguir são listados os principais métodos da API do sistema SPDO:

Método	**Parâmetros**	**Descrição**
addProtocolo	tipoprotocolo, tipodocumento_id, numero_documento, data_emissao, assunto, situacao_id, origem, destino, **kwargs	Adiciona protocolo.
addObservacao	protocolo_id, texto	Adiciona observação.
addAnexos	protocolo_id, anexos	Adiciona anexos.
TramiteInicial	protocolo_id	Tramite inicial.
TramiteEnvio	protocolos, areas, acao	Tramite de envio.
TramiteRecebimento	protocolos	Tramite de recebimento.
TramiteRecuperacao	protocolos	Tramite de recuperação (recupera um protocolo enviado que não foi recebido).
getProtocolosCriadosRecebidos		Consulta os protocolos criados ou recebidos pela área.
getProtocolosNaoRecebidos		Consulta os protocolos não recebidos pela área.
getProtocolosEnviados		Consulta os protocolos enviados pela área.

2.7 views

Outra parcela significativa do código fonte do sistema SPDO consistirá de views, conceito do ZCA que permite a separação da camada de apresentação da lógica da aplicação.

Abaixo segue exemplo que ilustra como as views do projeto foram implementadas, utilizando ZCA e Grok.

```
class ProtocoloListView(grok.View, BaseListView):

    grok.name('list-protocolo')
    grok.context(INavigationRoot)
```

```
    grok.require('cmf.ManagePortal')

    dados = []
    view_sufix = 'protocolo'

    @log
    def update(self):
        self.request.set('disable_border', True)
        api = getUtility(ISPDOAPI)
        self.dados = api.getProtocolosCriadosRecebidos()
```

Percebe-se no exemplo que o método update da classe ProtocoloListView invoca o método getProtocolosCriadosRecebidos da API. Esse método já foi propositalmente citado no item 2.6 desse relatório, onde encontra-se destacado.

A segunda parte da implementação da view consiste de um template ZPT. A única diferença de uma view para um ZPT padrão, executado no contexto de aquisição do Zope, é justamente o contexto: na view temos um contexto controlado (definido pela diretiva "for" do Grok) e um namespace chamado "view", onde todas as propriedades e métodos da classe que implementa a view estão disponíveis, sem a necessidade de verificações adicionais de segurança.

Por definição, as views devem ser codificadas como classes dentro do módulo "browser" do pacote il.spdo, que por sua vez deverá ser instalado no sistema de arquivos do servidor Zope. Não faz sentido efetuar verificações de segurança em um código fonte que é executado diretamente do sistema de arquivos do servidor, pois quem tem acesso (legítimo) ao sistema operacional não deve ser considerado uma ameaça. Esse aparente "relaxamento" nas verificações de segurança das views é proposital e permite maior liberdade ao desenvolvedor, além de resultar em uma melhora significativa na performance da aplicação.

2.7.1 Exemplo: template ZPT

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
      xmlns:tal="http://xml.zope.org/namespaces/tal"
      xmlns:metal="http://xml.zope.org/namespaces/metal"
      xmlns:i18n="http://xml.zope.org/namespaces/i18n"
      lang="en"
      metal:use-macro="context/main_template/macros/master"
      i18n:domain="il.spdo">

<body>

<metal:main fill-slot="main">
    <tal:main-macro metal:define-macro="main">

    <div tal:replace="structure provider:plone.abovecontenttitle" />

    <h1 class="documentFirstHeading">Meus Protocolos</h1>
```

```
    <p class="documentDescription">Os protocolos abaixo foram criados
    pela sua área ou recebidos de outras áreas, solicitando
    providências.</p>

    <p><a tal:attributes="href string:${view/add_url}">Adicionar</a></p>

    <div tal:replace="structure provider:plone.abovecontentbody" />

    <div>
      <table class="listing" id="table-list-protocolo" width="100%">
       <thead>
         <tr>
          <th>Número</th>
          <th>Data</th>
          <th>Assunto</th>
          <th>Tipo de Documento</th>
          <th>Situação</th>
         </tr>
       </thead>
      <tbody>
        <tr tal:repeat="protocolo view/dados">
          <td><a tal:content="protocolo/numero"
                 tal:attributes="href
python:view.show_url(protocolo['id'])"></a>
          </td>
          <td tal:content="protocolo/data_protocolo"></td>
          <td tal:content="protocolo/assunto"></td>
          <td tal:content="protocolo/tipodocumento"></td>
          <td tal:content="protocolo/situacao"></td>
        </tr>
      </tbody>
     </table>
    </div>

    <script type="text/javascript">
    $(document).ready(function() {
      $('#table-list-protocolo').dataTable({
      });
    });
    </script>

    <div tal:replace="structure provider:plone.belowcontentbody" />

    </tal:main-macro>
</metal:main>

</body>
</html>
```

3. Banco de Dados Versionado

Expostas as decisões de arquitetura e padrões de projeto a serem utilizados no desenvolvimento do código fonte do sistema SPDO, convém detalhar algumas questões relacionadas com a segurança e a integridade das informações.

Num sistema de protocolo é especialmente importante saber, ou pelo menos ter como rastrear, qual usuário é o responsável pela inclusão, alteração ou exclusão de uma determinada informação. Por vezes, sistemas de protocolo podem conter informações sensíveis ou mesmo sigilosas, onde auditorias específicas podem ser necessárias.

Nesse sentido optou-se pela implementação de características temporais no bancos de dados, armazenando versões de todos os registros, juntamente com a data que esses registros foram modificados e, em algumas tabelas, quais usuários do sistema efetuaram essas modificações.

Abaixo segue exemplo que demonstra como o sistema de controle de versão do banco de dados armazena as informações durante uma alteração de um registro. A alteração ocorre na tabela "pessoa" e a versão anterior do registro alterado é inserida na tabela "pessoa_history". Em destaque, estão os campos contato (que foi atualizado) e os campos version e version_date, atualizados respectivamente com o número da versão do registro (um contador das modificações que o registro já sofreu) e a data da modificação de cada versão.

```
mysql> select * from pessoa where email = 'tiao@macale.net'\G
*************************** 1. row ***************************
          id: 2
        nome: Tião Macalé
       email: tiao@macale.net
    endereco: NULL
      bairro: NULL
         cep: NULL
      cidade: NULL
       uf_id: NULL
    telefone: NULL
    cpf_cnpj: NULL
  tipopessoa: F
     contato: Didi Mocó
     area_id: 2
       senha: 12345
     version: 2
```

```
version_date: 2011-11-24 11:35:59
1 row in set (0.00 sec)
```

```
mysql> select * from pessoa_history where email = 'tiao@macale.net'\G
*************************** 1. row ***************************
          id: 2
        nome: Tião Macalé
       email: tiao@macale.net
    endereco: NULL
      bairro: NULL
         cep: NULL
      cidade: NULL
       uf_id: NULL
    telefone: NULL
    cpf_cnpj: NULL
  tipopessoa: F
     contato: NULL
     area_id: 2
       senha: 12345
     version: 1
version_date: 2011-11-24 11:30:31
1 rows in set (0.00 sec)
```

Nas operações de exclusão de registros, o sistema insere uma cópia do registro que será excluído na respectiva tabela "_history" e depois apaga o registro da tabela original.

Essa estratégia de modelagem de banco de dados com controle de versão é especialmente interessante pois as tabelas originais do sistema sofrem apenas pequenas modificações: a adição do campo version (que tem o valor padrão 1) e do campo version_date (que tem como valor padrão a data do servidor de banco de dados). A estrutura das tabelas com sufixo "_history" é gerada a partir da estrutura das tabelas originais, copiando todos os campos e adicionando o campo version na chave primária. Isso permite que diversas versões dos registros modificados sejam mantidas nas tabelas "_history" sem impactar no funcionamento dos sistemas que utilizam e algumas vezes dependem da estrutura das tabelas originais.

Essa implementação não é nativa de nenhum banco de dados, nem de nenhuma biblioteca e foi desenvolvida como uma extensão do SQLAlchemy, que ao mesmo tempo adiciona comportamento temporal ao banco de dados e mantém a integração do SQLAlchemy consistente

com os mecanismos de controle de transação distribuída do servidor de aplicação Zope. Essa implementação não impacta na compatibilidade do sistema com os diversos SGBD suportados pelo SQLAlchemy.

A implementação do sistema de controle de versão do banco de dados foi inspirada em [21], aprimorada e integrada com o Zope utilizando classes do z3c.saconfig [5]. O código fonte dessa extensão pode ser encontrado em [22].

4. Auditoria de Acesso

Outra preocupação relevante e que deverá ser considerada na implementação do projeto SPDO diz respeito a necessidade de armazenar logs de todas as operações que modificam dados no sistema, assim como logs que representem a forma como o sistema é utilizado. Foi especificado que os programas deveriam armazenar esses logs em uma tabela do banco de dados relacional, permitindo que consultas sejam feitas de forma direta, utilizando a linguagem SQL.

A implementação desse requisito deve ser feita utilizando um decorador Python [23], cujo código fonte recomendado segue abaixo:

```
from il.spdo.db import Log
from il.spdo.config import Session
from zope.globalrequest import getRequest

def log(fn):
    def f(*args, **kwargs):
        ret = fn(*args, **kwargs)
        l = Log()
        request = getRequest()
        l.usuario = str(request.other.get('AUTHENTICATED_USER', 'Anonymous'))
        l.url = request.other.get('ACTUAL_URL')
        l.modulo = fn.__module__
        l.classe = args[0].__class__.__name__
        l.funcao = fn.__name__
        l.args = repr(list(args)[1:])
        l.kwargs = repr(kwargs)
        session = Session()
        session.add(l)
        return ret
    return f
```

O uso de decoradores permite que qualquer método de uma classe tenha sua utilização registrada, armazenando-se qual usuário executou o método, o contexto onde ocorreu a execução, data, hora, URL e até mesmo os parâmetros utilizados. Outras vantagens no uso de decoradores são a simplicidade e a legibilidade, sem prejudicar a manutenção do código fonte.

Um exemplo de utilização do decorador pode ser encontrado no item 2.7 desse relatório,

onde o método update é decorado com "@log".

5. Integração com E-mail

No "Relatório de Entrega de nº 1" item 2.1.5.11, foi especificado que o sistema SPDO deverá integrar-se com outros sistemas através de mensagens de e-mail. Abaixo segue a especificação da **estrutura de dados** que deve estar contida no corpo dessas mensagens.

```
{'origens': [{'email': 'email@origem.net', 'nome': 'Nome da Pessoa de Origem'}],
'destinos': [{'email': 'email@destino.net', 'nome': 'Nome da Pessoa de
Destino'}],
'assunto': 'Assunto...',
'observacao': '',
'numero_documento': '12345',
'data_emissao': '2011-11-23',
'situacao': 'Arquivado',
'tipodocumento': 'Carta',
'tipoprotocolo': 'E' }
```

A estrutura de dados acima é um dicionário python, com todas as chaves e valores do tipo string, exceto as chaves "origens" e "destino", onde teremos listas de dicionários.

As seguintes regras devem ser respeitadas:

1. Todas as chaves dos dicionários são obrigatórias;
2. As chaves "origens" e "destinos" não podem ser listas vazias;
3. Datas devem ser representadas no formato AAAA-MM-DD;
4. Os valores informados para as chaves "situacao" e "tipodocumento" devem estar cadastrados nas respectivas tabelas;
5. Os valores aceitos para a chave "tipoprotocolo" são: "R" (Recebido), "E" (Expedido) e "I" (Interno);
6. A chave "Assunto" possui valor obrigatório;

O corpo das mensagens de e-mail não deve conter outras informações além da estrutura de dados, que deve estar codificada utilizando o formato JSON [24], como segue:

```
>>> dados
{'origens': [{'email': 'email@origem.net', 'nome': 'Nome da Pessoa de Origem'}],
'situacao': 'Arquivado', 'assunto': 'Assunto...', 'destinos': [{'email':
'email@destino.net', 'nome': 'Nome da Pessoa de Destino'}], 'data_emissao':
'2011-11-23', 'tipodocumento': 'Carta', 'numero_documento': '12345',
'tipoprotocolo': 'E', 'observacao': ''}
>>> import json
>>> print json.dumps(dados)
{"origens": [{"email": "email@origem.net", "nome": "Nome da Pessoa de Origem"}],
"situacao": "Arquivado", "assunto": "Assunto...", "destinos": [{"email":
"email@destino.net", "nome": "Nome da Pessoa de Destino"}], "data_emissao":
```

```
"2011-11-23", "tipodocumento": "Carta", "numero_documento": "12345",
"tipoprotocolo": "E", "observacao": ""}
```

A operação inversa, ou seja, converter o corpo de uma mensagem em formato JSON para uma estrutura de dados Python, pode ser realizada como segue:

```
>>> dados
'{"origens": [{"email": "email@origem.net", "nome": "Nome da Pessoa de
Origem"}], "situacao": "Arquivado", "assunto": "Assunto...", "destinos":
[{"email": "email@destino.net", "nome": "Nome da Pessoa de Destino"}],
"data_emissao": "2011-11-23", "tipodocumento": "Carta", "numero_documento":
"12345", "tipoprotocolo": "E", "observacao": ""}'
>>> import json
>>> json.loads(dados)
{u'origens': [{u'email': u'email@origem.net', u'nome': u'Nome da Pessoa de
Origem'}], u'situacao': u'Arquivado', u'assunto': u'Assunto...', u'destinos':
[{u'email': u'email@destino.net', u'nome': u'Nome da Pessoa de Destino'}],
u'tipodocumento': u'Carta', u'numero_documento': u'12345', u'tipoprotocolo':
u'E', u'observacao': u'', u'data_emissao': u'2011-11-23'}
```

Os anexos das mensagens devem ser importados no sistema SPDO apenas se a estrutura de dados contida no corpo da mensagem estiver correta.

As demais regras sugeridas no “Relatório de Entrega de nº 1” item 2.1.5.11 também devem ser observadas.

6. Integração por Webservices

Também no “Relatório de Entrega de nº 1” item 2.1.5.12, foi especificado que o sistema SPDO deverá integrar-se com outros sistemas através de Webservices.

O sistema SPDO disponibilizará um Webservice REST, que responderá a requisições HTTP na seguinte URL: http://ip.do.servidor.spdo:porta/@@ws-add-protocolo?dados=

O parâmetro “dados” deverá receber uma string no formato JSON, de acordo com a especificação descrita no item 5 desse relatório.

7. Considerações Finais

Esse documento complementa o “Relatório de Entrega de nº 1”, especificando como os requisitos do sistema SPDO devem ser implementados na prática. Apesar de algumas tecnologias escolhidas serem consideradas complexas, acredita-se que essas escolhas refletem a complexidade do sistema que deve ser implementado e, certamente, irão representar o que existe de melhor em frameworks Python para desenvolvimento de aplicações para a plataforma Zope.

Durante a redação desse relatório, a qualidade almejada para o sistema foi considerada como fator determinante na tomada de decisões, mesmo que algumas das tecnologias escolhidas sejam potencialmente mais difíceis de implementar. Espera-se que a implementação final estará de acordo com as expectativas dos usuários e que cumpra com as determinações do contrato 2011/000260 e do Termo de Referência 135641.

8. Referências

[1] http://colab.interlegis.gov.br/wiki/ProjetoSapl
[2] http://www.sqlalchemy.org/
[3] http://www.sqlalchemy.org/docs/core/engines.html
[4] Aspeli, Martin. Professional Plone 4 Development. ISBN: 978-1-849514-42-2
[5] http://pypi.python.org/pypi/z3c.saconfig
[6] http://www.sqlalchemy.org/docs/orm/extensions/declarative.html
[7] http://www.sqlalchemy.org/docs/
[8] http://www.sqlalchemy.org/docs/orm/tutorial.html
[9] http://colab.interlegis.gov.br/browser/il.spdo/trunk/il/spdo/db.py
[10] http://www.muthukadan.net/docs/zca.html
[11] http://plone.org/
[12] http://grok.zope.org/
[13] http://jquery.com/
[14] http://flowplayer.org/tools/overlay/index.html
[15] http://datatables.net/
[16] http://pypi.python.org/pypi/z3c.form
[17] http://pypi.python.org/pypi/plone.app.z3cform
[18] http://pypi.python.org/pypi/plone.directives.form
[19] http://colab.interlegis.gov.br/browser/il.spdo/trunk/il/spdo/browser/forms/base.py
[20] http://colab.interlegis.gov.br/browser/il.spdo/trunk/il/spdo/api.py
[21] http://www.sqlalchemy.org/docs/orm/examples.html?#versioned-objects
[22] http://colab.interlegis.gov.br/browser/il.spdo/trunk/il/spdo/history_meta.py
[23] http://www.python.org/dev/peps/pep-0318/
[24] http://www.json.org/