Tutorial Grok

Versão 1.0

Diego Manhães Pinheiro

26/11/2009

Conteúdo

Contents

CAPÍTULO 1

Bem vindo ao tutorial Grok!

Iniciando com Zope Page Templates

Você pode encontrar introduções e mais informações sobre Zope Page Templates (ZPT, que é chamado também como TAL) em vários lugares:

http://plone.org/documentation/tutorial/zpt
http://wiki.zope.org/ZPT/FrontPage

Note que muita da informação incluída nessas introduções pode referir a conceitos não disponíveis no Grok ou no Zope Toolkit, em particular variáveis como `here` ou `template`. Os princípios básicos irão funcionar com Grok entretanto; tente ler como `context` ou `view` ao invés.

Grok é um poderoso e flexível framework de aplicativos web para desenvolvedores que usam a linguagem Python. Neste tutorial nós iremos mostrar a você as várias coisas que você pode fazer com o Grok, e como Grok pode ajudar você a construir seus aplicativos web. Nós iremos iniciar com exemplos, e iremos mostrar lentamente mais padrões complexos de uso.

Tudo que é esperado para você conhecer é a linguagem de programação Python e um entendimento de programação básica de programação web (HTML, formulários, URLs). Irá ajudar também se você é familiar com Zope Page Templates, entretanto muitos dos exemplos devem ser óbvios se você é familiar com outra linguagem de template.

Nós recomendamos que iniciantes sigam o tutorial do início ao fim. O tutorial é designado a explicar conceitos importantes de forma ordenada e lentamente definindo-os até até.

Se você é mais experiente ou somente um curioso, você pode pular e ler as partes que lhe interessam. Se alguma coisa não está clara, você sempre voltar nas seções anteriores.

Grok é baseado no Zope Toolkit . Você não precisa conhecer sobre o Zope Toolkit para seguir este tutorial. Grok é construído com a tecnologia Zope Toolkit mas expõe uma maneira especial de desenvolver. Nós acreditamos que Grok torna fácil o desenvolvimento com a tecnologia Zope Toolkit e divertido para iniciantes e desenvolvedores experientes.

CAPÍTULO 2

Iniciando com o Grok

Neste capítulo irá ajudá-lo a instalar e executar o Grok, usando a ferramenta `grokproject`. Nós criamos um novo projeto com `grokproject` e informaremos a você como fazer o projeto funcionar permitindo ter acesso a ele através de um navegador web.

2.1 Definindo o grokproject

Instalando o `easy_install`

Se você não tem `easy_install` disponível, você pode encontrar o script na página do PEAK EasyInstall.
Você precisa fazer o download de ez_setup.py . Então, você executa-o desta maneira para instalar o `easy_install` no seu sistema Python:

```
$ sudo python2.5 ez_setup.py
```

Isto irá disponibilizar o `easy_install` para você.
Nota: Algumas vezes você tem o `easy_install` instalado mais você precisa de uma nova versão do setuptools para fazer o Grok funcionar. Você pode atualizar o setuptools com:

```
$ sudo easy_install -U setuptools
```

Nota: É recomendado você instalar `easy_install` no virtualenv para isolar a instalação de bibliotecas python do seu sistema python. Isto é especialmente relevante na plataforma Mac OS X. Veja a próxima seção para mais informação.

Instalando o virtualenv

Virtualenv é uma ferramenta que permite você a isolar seu ambiente de desenvolvimento em python completamente do sistema de biblioteca Python instaladas .
Em plataformas como Mac OS X o uso de virtualenv é especialmente recomendado devido a outras versões antigas de bibliotecas (notável mente `zope.interface`) serem instaladas por operações no Mac OS X. Grok precisa de uma versão atual do `zope.interface`', gerando assim um conflito entre versões de uma mesma biblioteca. Isolá-lo do sistema Python é recomendado em ambientes Linux, pois Python é comumente instalado com sistema de gerenciamento de pacotes.
Se você não deseja usar virtualenv é sempre possível compilar e instalar em uma diferente versão do Python localmente para usar com o Grok.
Durante a instalação do Grok no Linux e Mac OS X várias bibliotecas com componentes que possuem código C são automaticamente compilados para você. No Linux você precisa ter certeza que você tem os cabeçalhos de código C instalado. (use `python2.5-dev` para a versão para o Python 2.5).
Estas instruções são escritas para um sistema no estilo Unix e irão ser difíceis de serem seguidas no Windows. Em um ambiente windows você pode pular este passo se você sabe como instalar um ambiente Python por sua conta. Entretanto, se você pretende instalar uma grande quantidade de software que usa esta mesma versão do Python, virtualenv é altamente recomendado.
Você pode instalar o virtualenv com o `easy_install`:

```
$ easy_install-2.5 virtualenv
```

O comando `virtualenv` deve agora estar disponível para você. Você pode criar uma ambiente 'caixa de areia' para usar com o Grok:

```
$ virtualenv --no-site-packages virtualgrok
```

Isto irá criar um diretório `virtualgrok` na sua localização corrente que contém seu ambiente virtual .
A opção `--no-site-packages` é importante: ela isola seu ambiente virtual de quaisquer pacotes instalados no sistema de bibliotecas.
Você deve agora ativar o ambiente virtual:

```
$ source virtualgrok/bin/activate
```

Uma vez que você tenha ativado o ambiente virtual, você pode usár o `easy_install` grokproject de maneira regular:

```
$ easy_install grokproject
```

Note que uma vez criado um projeto Grok a partir de um ambiente virtual, não é necessário ativar o ambiente virtual novamente – o projeto Grok irá saber usar o ambiente especial virtualgrok automaticamente. Você somente precisa usar a ferramenta `grokproject` diretamente.
Para mais informação, veja Usando Virtualenv para uma Instalação Grok limpa

Instalando Grok em um ambiente no estilo Unix(Linux, Mac OS X) é fácil . Muitos dessas instruções podem também funcionar em um ambiente Windows .

Vamos agora passar os pré-requisitos primeiro. Você precisa de um computador conectado a internet, pois Grok é instalado através de rede. Você irá precisar da versão 2.5 (ou 2.4) do Python instalada.

Devido ao Grok usar bibliotecas do Zope Toolkit no formato empacotadas com código fonte, você pode precisar instalar pacotes 'dev' do Python do seu sistema operacional (`python-dev` no Debian e Ubuntu, `python-2.5-dev` para o Python 2.5). Você pode precisar de um compilador C (normalmente `gcc`) instalado, pois é compilado algumas partes do Zope Toolkit durante a instalação (`build-essential` no Debian e Ubuntu). No Windows um ambiente para compilar essas partes não é necessária. Grok irá baixar e automaticamente instalar bibliotecas pré-compiladas para Windows. Finalmente, você precisará do `easy_install` instalado, o que torna fácil instalar os pacotes Python.

Quando os requisitos estivem satisfeitos, você pode instalar o grokproject:

```
$ easy_install grokproject
```

Se você está em um ambiente Unix e você não está usando o virtualenv como foi recomendado, você irá precisar requisitar direitos administrativos usando `sudo` para instalar novas bibliotecas em seu sistema Python.

Agora nós estamos prontos para criar nosso primeiro projeto Grok agora!

2.2 Criando um projeto grok

Usando paster

Para aquele que conhece o paster: `grokproject` é somente uma espécie de capa de um modelo do paster. Logo, ao invés de executar o comando `grokproject`, você pode então executar:

```
$ paster create -t grok Sample
```

Primeiramente, vamos criar um projeto Grok. Um projeto Grok é um ambiente para desenvolver usando o Grok. Em sua essência, ele é um diretório com muitos arquivos e subdiretórios nele. Vamos criar um projeto Grok chamado Sample:

```
$ grokproject Sample
```

Instalando o layout 'zopectl'

Grok usava um layout diferente usando o `zopectl`. Para usar esse layout antigo, use `grokproject` com a opção `--ctl`

```
$ grokproject --zopectl Sample
```

Isto informa ao grokproject para criar um novo sudiretório chamado `Sample` e definindo o projeto nele. Grokproject irá automaticamente instalar e definir o projeto lá . Grokproject irá automaticamente baixar e instalar as biblioteca do Zope Toolkit, assim como o Grok na pasta do projeto.

Grok irá perguntar por um usuário inicial e uma senha para o servidor. Nós iremos usar `grok` para dois:

```
Enter user (Name of an initial administrator user): grok
Enter passwd (Password for the initial administrator user): grok
```

Agora você tem que esperar enquanto grokproject baixar os arquivos e instala várias ferramentas e bibliotecas que são necessárias em um projeto Grok. Em um segundo momento que você criar um projeto Grok, ele irá executar mais rapidamente devido a ele usar as bibliotecas previamente instaladas. Depois de aguardar seu projeto Grok está pronto para o uso.

Problemas comuns ao instalar Grok

Mistura de bibliotecas
Um problema comum quando se instala o Grok é a mistura de bibliotecas. Você pode ter muitas bibliotecas instaladas em seu interpretador Python que conflitam com as que o Grok deseja instalar. Você capturará um ao iniciar um servidor Grok quando isse for o problema. Se você já instalou as bibliotecas do Zope Toolkit (ou Zope 3) anteriormente por exemplo, você pode deverá remover essas bibliotecas de seu ambiente Python, especificadamente do diretório `site-packages`.
Melhor ainda, veja a seção `virtualenv` anteriormente para usá-lo como uma maneira de isolar o Grok e suas bibliotecas do seu ambiente de sistema Python, evitando problemas como esse.
Sem ambiente de desenvolvimento Python
Grok inclui dependências que precisam ser compiladas com o Python. Isso acontece automaticamente no processo de instalação, a menos que você esteja no Windows, onde ele é suprido com versões binárias das bibliotecas requeridas.
No Debian e Ubuntu isso é o pacote `python-dev` (python2.5-dev para Python 2.5), Você pode precisar do pacote `build-essential`.

2.3 Iniciando o servidor web

Executando uma instância Grok criada com o layout ‘’zopect” antigo

Para iniciar uma instância Grok criada com o layout ‘zopectl’ antigo:

```
$ cd Sample
$ bin/zopectl fg
```

No windows para trabalhar com o `zopectl` você precisa ter certeza que você tem a biblioteca win32all instalada no seu Python. Não é requerido instalar wind32all para funcionar com a instalação via paster.

Voce pode ir para a pasta `Sample` do projeto agora e iniciar o servidor web para nosso projeto:

```
$ cd Sample
$ bin/paster serve parts/etc/deploy.ini
```

Isto irá disponibilizar o Grok na porta 8080. Você pode então acessar com o usuário `grok` e senha `grok`. Assumindo que você iniciou o servidor web em seu computador, voce pode ir por aqui:

http://localhost:8080

Esse endereço irá permite aparecer uma pequena janela de diálogo de login (username: `grok``e senha: ``grok`). Ele irá então mostrar uma interface de gerenciamento permitindo você instalar novas aplicações Grok.

Nosso aplicativo de exemplo (`sample.app.Sample`) irá estar disponível para ser adicionado. Vamos tentar fazer isso. Vá para a página de administração Grok:

http://localhost:8080

e adicione um aplicativo de exemplo. Dê a ele o nome de `test`.

Você pode agora ir para aplicação instalada se você clicar neste link. Isso irá informar a você a seguinte URL:

http://localhost:8080/test

Você deve ver uma simples página Web com o seguinte texto nela:

```
Congratulations!

Your Grok application is up and running. Edit
sample/app_templates/index.pt to change this page.
```

Agora você pode desligar o servidor a qualquer momento pressionando `CTRL-c`. Faça isso agora. Ele será desligado e iniciado várias vezes nesse tutorial.

Pratique reiniciar o servidor agora, pois você irá fazer isso muitas vezes nesse tutorial. É só parar e iniciá-lo novamente: `CTRL-c` e então `bin/paster serve parts/etc/deploy.ini` a partir do seu diretório base do seu projeto Sample.

Alternativamente, você pode usar a opção `--reload` para iniciar um monitor que busca por mudanças no seu código (somente arquivos python) e automaticamente reinicia o serviço cada vez que você faz uma mudança:

```
$ bin/paster serve --reload parts/etc/deploy.ini
```

2.4 Um projeto Grok vazio

O que são os outros diretórios e arquivos do nosso projeto ?

O que são os outros arquivos e subdiretórios em nosso diretório de projeto `Sample`? Grokproject define um projeto usando um sistema chamado zc.buildout . Os diretórios `eggs`, `develop-eggs`, `bin` e `parts` são todos definidos e mantidos pelo zc.buildout. Veja a documentação dele para mais informação de como usá-lo. A configuração do projeto e suas dependências estão no `buildout.cfg`. De qualquer maneira, por agora evite esse detalhes.

Vamos dar uma olhada em o que foi criado no diretório do projeto Sample.

Uma das coisas que o grokproject criou foi um arquivo `setup.py`. Este arquivo contém informações relacionados ao seu projeto. Esta informação é usada pelo buildout para baixar suas dependências de projeto e para instalá-lo. Você pode usar o arquivo `setup.py` para enviar seu projeto para o Índice de Pacotes Python (conhecido como PyPI).

Ainda temos o diretório `bin`. Ele contém o script de inicialização para o serviço web (`bin/paster`), assim como o executável para o sistema buildout (`bin/buildout`), que pode ser usado para refazer seu projeto (para atualizá-lo ou instalar uma nova dependência).

O diretório `parts` contém configuração e dados criados e gerenciados pelo `buildout`, como a base de dados usada, conhecida como Zope object database (ZODB), e os arquivos `ini` para serem usados com o `paster`. O código atual do projeto irá ficar dentro do diretório `src`. Nele há um diretório de pacote Python chamado `sample` com um arquivo chamado `app.py` que o grokproject criou. Vamos olhar esse arquivo:

```
import grok

class Sample(grok.Application, grok.Container):
    pass

class Index(grok.View):
    pass # see app_templates/index.pt
```

Não é muito ainda, mas suficiente para criar uma aplicação instalável e mostrar sua página de bem-vindo. Nós iremos entrar em detalhes de o que isso significa posteriormente.

Há um arquivo `__init_.py` vazio para fazer deste diretório um pacote Python.

Há também um diretório chamado `app_templates`. Ele contém um modelo chamado *`index.pt`*:

```
<html>
<head>
</head>
<body>
  <h1>Congratulations!</h1>

  <p>Your Grok application is up and running.
  Edit <code>sample/app_templates/index.pt</code> to change
  this page.</p>
</body>
</html>
```

Esse é o modelo para a a sua página de bem-vindo.

Há então um arquivo `configure.zcml`. Este arquivo irá normalmente conter poucas linhas que carregam dependências e registra sua aplicação para você. Isso significa que nós tipicamente podemos ignorá-lo, mas nós iremos mostrá-lo por boa prática:

```
<configure xmlns="http://namespaces.zope.org/zope"
           xmlns:grok="http://namespaces.zope.org/grok">
  <include package="grok" />
  <includeDependencies package="." />
  <grok:grok package="." />
</configure>
```

`configure.zcml` em aplicações não-Grok.

Em aplicações não-Grok que usam Zope Toolkit (como alguma coisa criada com Zope 2 ou Zope 3), o arquivo ZCML normalmente possui um grande papel. Ele contém diretivas que registram particulares objetos Python (tipicamente classes, como visões) com a arquitetura de componentes que é o centro do Zope Toolkit. Grok entretanto automatiza esse registro inserindo mais informação no código diretamente, ficando o arquivo ZCML muito pequeno.

Há também um diretório de nome `static`. Ele contém arquivos estáticos que podem ser usados em uma aplicação web, como imagens, arquivos css e arquivos javascript.

Entre esses arquivos, há um `app.txt`, `ftesting.zcml` e `tests.py`. Todos esses arquivos são usados para a execução automatizada de testes e podem ser ignorados por enquanto.

CAPÍTULO 3

Visualizando páginas

Mostrar páginas web é o porquê do *web* em aplicações web . Normalmente modelos HTML são usados para isso, mais Grok não pára nos modelos de página HTML. Muitas páginas web das aplicações reais irão conter lógica de apresentação complexa que é melhor tratado separando código Python em junção com modelos. Isso é especialmente importante em interações mais complexas com o usuário, como tratamento de formulários. Depois de ler esse capítulo, você deve ser capaz de escrever aplicações simples com o Grok.

3.1 Publicando uma página simples

Vamos visualizar uma página estática simples. Grok tem como propósito aplicações web e não publicar uma grande quantidade de páginas estáticas.

Vamos publicar uma página estática simples. Grok gira em torno das aplicações web e não tem como propósito publicar um grande quantidade de página estática (pré criadas). Para isso é melhor um sistema especializado nessa tarefa como o Apache. Entretanto, para desenvolver qualquer aplicação web simples nós precisamos conhecer como inserir algum HTML na web.

Como foi visto anteriormente, nossa aplicação `Sample` tem uma página inicial estática gerada pelo grokproject. Vamos mudá-la.

Para fazer isso, vá para o diretório `app_templates` em `src/sample/`. Este diretório contém os modelos de página usados para qualquer coisa definida no módulo `app`. Grok sabe associar o diretório para o módulo através de seu nome (`<nome_modulo>_templates`).

Nesse diretório nós iremos editar o modelo de página `index` para nosso objeto `Sample` da nossa aplicação. Para fazer isso, abra o arquivo `index.pt` em um editor de texto. A extensão `.pt` indica que esse arquivo é um Zope Page Template (ZPT). Nós iremos incluir somente HTML por agora, mas ZPT permite que tornemos a página dinâmica posteriormente.

Mude o `index.pt` para conter o seguinte (e muito simples) HTML:

```
<html>
<body>
<p>Hello world!</p>
```

```
</body>
</html>
```

Então recarregue a página:

http://localhost:8080/test

Você deve agora ver o seguinte texto:

```
Hello world!
```

Note que você pode mudar o modelo de página e ver os efeitos instantaneamente:

Não há necessidade de reiniciar o servidor web para ver os efeitos. Isto não é verdadeiro quando são mudanças a nível de código, por exemplo quando você adiciona um modelo. Será mostrado um exemplo disso posteriormente.

3.2 Uma segunda visão

Nossa visão é chamada `index`. Isto significa alguma coisa inconsideravelmente especial: ele é a visão padrão para a nossa aplicação. Nós podemos então acessá-la explicitamente nomeando-a:

http://localhost:8080/test/index

Se você vê aquela URL no navegador, você deve ver o mesmo resultado como anteriormente. Isto é a maneira de todos as outras visões ,que não são chamadas index, de serem acessadas.

Algumas vezes, sua aplicação precisa mais de uma visão. Um documento por exemplo, pode ter uma visão que a permite vê-lo, e uma outra chamada `edit` para modificar seu conteúdo. Para criar uma segunda visão, crie outro template chamado `bye.pt` em `app_templates`. Faça-o ter o seguinte conteúdo :

```
<html>
<body>
<p>Bye world!</p>
</body>
</html>
```

Agora nós precisamos dizer ao Grok para usar esse modelo de página. Para fazer isso, modifique `src/sample/app.py` para que possa parecer como isto:

```
import grok

class Sample(grok.Application, grok.Container):
    pass

class Index(grok.View):
    pass
```

```
class Bye(grok.View):
    pass
```

Como você pode ver, tudo que nós fizemos foi adicionar uma classe chamada `Bye` que tem como classe mãe a classe `grok.View`. Isto indica para o Grok que nós desejamos a visão chamada `bye` para a aplicação, como a classe `Index` que foi criada indicando que se queria uma visão chamada `index` . Uma *visão* é uma maneira de visualizar algum modelo, neste caso instalando na nossa aplicação `Sample`. Note que o nome da visão na URL é sempre em caixa baixa, enquanto que o nome da classe inicia com a primeira letra em caixa alta.

A definição de classe vazia acima é suficiente para o Grok olhar no diretório `app_templates` pelo `bye.pt`. A regra é que um modelo de página deve ter o mesmo nome da classe, mais em caixa baixa e com o sufixo `.pt`.

Outras linguagens de modelo de páginas

Você pode então instalar extensões para permitir o uso de outras linguagens de modelo no Grok, veja por exemplo o ‘‘megrok.genshi‘‘_ .

Reinicie o servidor web (`CTRL-C`, e então `bin/paster serve parts/etc/deploy.ini`) Você pode agora ir para a nova página chamada `bye`:

http://localhost:8080/test/bye

Quando você carregar esta página em um navegador, você deve ver o seguinte texto:

```
Bye world!
```

3.3 Tornando nossa página dinâmica

Páginas web estáticas não são amigáveis se você deseja tornar uma aplicação web dinâmica. Vamos criar uma página que mostra o resultado de um cálculo simples : `1 + 1`.

Nós iremos usar uma diretiva Zope Page Template (ZPT) para fazer isse cálculo dentro do modelo de página `index.pt` . Mude o `index.pt` para ficar como isto:

```
<html>
<body>
<p tal:content="python: 1 + 1">this is replaced</p>
</body>
</html>
```

Nós usamos a diretiva `tal:content` para substituir o conteúdo entre as marcações `<p>` e `</p>` com alguma coisa, no caso o resultado da expressão Python `1 + 1`.

Sabendo que reiniciar o servidor não é necessário para mudanças no modelo de página, você pode simplesmente recarregar a página:

http://localhost:8080/test

Você deverá ver o seguinte resultado:

```
2
```

Olhando para o código fonte da página web encontramos isto:

```
<html>
<body>
<p>2</p>
</body>
</html>
```

Como você pode ver, o conteúdo da marcação `<p>` foi substituído pelo resultado da expressão `1 + 1`.

3.4 Recursos estáticos web para a nossa aplicação web

Em páginas web reais, quase nunca uma página web publicada é somente um conteúdo HTML básico. Nós desejamos referenciar a outros recursos, como imagens, arquivos CSS ou arquivos Javascript. Vamos então adicionar algum estilo a nossa página web.

Para fazer isso , crie um diretório chamado `static` no pacote `sample` (então, `src/sample/static`). Inclua nele um arquivo chamado `style.css` e coloque nele o seguinte conteúdo:

```
body {
    background-color: #FF0000;
}
```

De forma a usá-lo, é preciso referenciá-lo no nosso `index.pt` . Mude o conteúdo do arquivo `index.pt` para ficar como isto:

```
<html>
<head>
<link rel="stylesheet" type="text/css"
      tal:attributes="href static/style.css" />
</head>
<body>
<p>Hello world!</p>
</body>
</html>
```

Agora reinicie o servidor e recarregue a página:

http://localhost:8080/test

A página web deve agora mostrar um fundo vermelho.

Você deverá notar que nós usamos a diretiva `tal:attributes` em nossa página `index.pt` agora. Isto usa a ZPT para dinamicamente gerar a ligação para o nosso arquivo `style.css`.

Vamos dar uma olhada no código fonte do página gerada:

```
<html>
<link rel="stylesheet" type="text/css"
      href="http://localhost:8080/test/@@/sample/style.css" />
<body>
<p>Hello world!</p>
</body>
</html>
```

Como você pode ver, a diretiva `tal:attributes` desapareceu e foi substituída com a seguinte URL para o estilo atual:

> http://localhost:8080/test/@@/sample/style.css

Nós não iremos entrar em detalhes da estrutura da URL aqui, mas nós iremos notar que devido a maneira que o endereço para o arquivo `style.css` é gerada irá continuar funcionando, não importando onde você instale sua aplicação (por exemplo, em um instalação usando virtual host).

Para incluir imagens e javascript é similar. Somente coloque sua imagens e arquivos na extensão `.js` no diretório `static`, e crie a URL para eles usando `static/<arquivo>` no seu modelo de página.

3.5 Usando métodos de um visão

modelos de página não associados

Se você seguiu o tutorial passo a passo, você irá ter um modelo extra chamado `bye.pt` em seu diretório `app_templates`.
Como no `app.py` não há mais classes usando-o, o modelo `bye.pt` não está mais associado. Quando você reiniciar o servidor, Grok irá informar um aviso como este:

> UserWarning: Found the following unassociated template(s) when grokking 'sample.app': bye. Define view classes inheriting from grok.View to enable the template(s).

Para ficar livre desse aviso, simplesmente remova `bye.pt` do diretório `app_templates`.

ZPT é deliberadamente limitado no que se trata do que fazer com Python. É uma boa prática usar ZPT para propósitos simples somente, e fazer qualquer coisa mais complicada em código Python. Usando ZPT com código Python externo é fácil: você deve somente adicionar métodos para a sua classe de visão e usá-la do seu modelo.

Vamos ver como isto é feito criando a página web que mostra a data e hora atual. Nós iremos usar nosso interpretador para descobrir como funciona:

```
$ python
Python 2.5.2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Nós iremos precisar da classe `datetime` do Python, logo vamos importá-la:

```
>>> from datetime import datetime
```

Note que essa expressão ultrapassa os limites do ZPT; não é permitido importar nenhum módulo Python em um ZPT. Somente expressões Python (com um resultado) são permitidos, não *expressões* como `from .. import ..`

Vamos capturar a data e hora atual:

```
>>> now = datetime.now()
```

Isso nos dá um objeto Data Hora; algo parecido com isto:

```
>>> now
datetime.datetime(2007, 2, 27, 17, 14, 40, 958809)
```

Não é agradável mostrar desta maneira em um página web, logo vamos transformá-lo em um formato mais bonito usando a capacidade de formatação do objeto `datetime`:

```
>>> now.strftime('%Y-%m-%d %H:%M')
'2007-02-27 17:14'
```

Que permite uma melhor visualização.

Não é nada novo; é somente Python. Nós iremos integrar este código em nosso projeto Grok. Vá para `app.py` e mude-o para ficar como isto:

```
import grok
from datetime import datetime

class Sample(grok.Application, grok.Container):
    pass

class Index(grok.View):
    def current_datetime(self):
        now = datetime.now()
        return now.strftime('%Y-%m-%d %H:%M')
```

Nós simplesmente adicionamos um método para a nossa classe de visão que retorna uma string que representa a data e hora atuais. Agora para incluir esta string em nossa página, mude `index.pt` para parecer como isto:

```
<html>
<body>
<p tal:content="python:view.current_datetime()">Hello world!</p>
</body>
</html>
```

Reinicie o servidor. Isto é necessário pois nós mudamos o conteúdo de um arquivo Python (`app.py`). Agora reinicie e acessa a nossa página inicial para ver se funcionou:

http://localhost:8080/test

Você deve ver uma página web com uma data e hora parecida com isto na sua tela agora:

```
2007-02-27 17:21
```

O que aconteceu aqui ? Quando visualizar uma página, a classe de visão (nesse caso `Index`) é instanciada pelo framework. O nome `view` no modelo é sempre disponível e associado com o seu modelo de página. Nós simplesmente chamamos o método em nosso modelo de página.

Há outra maneira de escrever o modelo de página que é consideravelmente menor e pode ser mais fácil de ler em muitos casos, usando uma expressão ZPT:

```
<html>
<body>
<p tal:content="view/current_datetime"></p>
</body>
</html>
```

Executando isso temos o mesmo resultado como anteriormente.

3.6 Gerando HTML a partir do Python

Enquanto normalmente você irá usar modelos de página para gerar HTML, algumas vezes você pode desejar gerar HTML mais complexos em Python e então incluí-los em página web existente. Por razões de segurança relacionadas a contra ataques de `cross-script`, TAL irá automaticamente modificar qualquer HTML delimitador de marcação para *>* ou *%lt;*. Com a diretiva `structure`, você pode dizer explicitamente para esses caracteres não serem modificados dessa forma, sendo passado literalmente para a página. Vamos ver como isto é feito. Modifique `app.py` para parecer como isto:

```
import grok

class Sample(grok.Application, grok.Container):
    pass

class Index(grok.View):
    def some_html(self):
        return "<b>ME GROK BOLD</b>"
```

e então mude `index.pt` para ficar como o seguinte:

```
<html>
<body>
<p tal:content="structure python:view.some_html()"></p>
</body>
</html>
```

Vamos dar uma olhada na nossa página web:

http://localhost:8080/test

Você deve ser o seguinte texto (em negrito):

ME GROK BOLD

Isso significa que o HTML gerado do método `some_html` foi integrado com sucesso em nossa página web. Sem a diretiva `structure`, você verá o seguinte, ao invés:

```
<b>ME GROK BOLD</b>
```

3.7 Visualizações usando Python completamente

Definindo o tipo de conteúdo

Quando gerado um conteúdo completo de uma página, várias vezes é interessante mudar o tipo de conteúdo da página para alguma coisa diferente de `text/plain`. Vamos mudar nosso código para retornar um XML simples e definir o tipo para `text/xml`:

```
import grok

class Sample(grok.Application, grok.Container):
    pass

class Index(grok.View):
    def render(self):
        self.response.setHeader('Content-Type',
                                'text/xml; charset=UTF-8')
        return "<doc>Some XML</doc>"
```

Todas as visualizações em Grok tem uma propriedade `response` que permite você manipular esse cabeçalhos de resposta.

Algumas vezes é inconveniente ter que usar o modelo para tudo Muitas vezes, nós não estamos retornando uma página. Nesse caso, nós podemos usar o método especial `render` na classe de visão.

Modifique `app.py` para que apareca como isto:

```
import grok

class Sample(grok.Application, grok.Container):
    pass

class Index(grok.View):
    def render(self):
        return "ME GROK NO TEMPLATE"
```

Se você iniciou o servido com um modelo `index.pt` residindo dentro de `app_templates`, poderá apresentar esse erro para você:

```
GrokError: Multiple possible ways to render view <class
'sample.app.Index'>. It has both a 'render' method as well as an
associated template.
```

Afim de não criar ambigüidade, Grok assim como o Python, não gera dúvidas em seus erros. Para resolver este erro, remova `index.pt``do diretório ``app_templates`.

Agora dê uma outra olhada em nossa aplicação de teste:

http://localhost:8080/test

Você deve ver o seguinte:

```
ME GROK NO TEMPLATE
```

Você deve ver isso até mesmo quando você visualiza o código-fonte da página. Ao olhar o tipo de conteúdo desta página, você irá ver que é `text/plain`.

3.8 Fazendo cálculos antes da página ser visualizada

Ao invés de calcular muitos valores em uma chamada de método no modelo de página, é mais usual efetuar o cálculo antes do modelo de página web ser processado. Desta maneira você terá certeza que um valor é somente calculado uma vez por visão, mesmo que você o use múltiplas vezes.

Você pode fazer isso definindo um método `update` na classe de visão. Modifique `app.py` para ficar como isto:

```
import grok

class Sample(grok.Application, grok.Container):
    pass

class Index(grok.View):
    def update(self):
        self.alpha = 2 ** 8
```

Isto define um nome `alpha` na visão antes do modelo ser mostrado, logo podendo ser usado pelo modelo. Você pode definí-lo com muitos nomes em `self` como você desejar.

Agora nós precisamos de um modelo chamado `index.pt` que usa `alpha`:

```
<html>
<body>
<p tal:content="python:view.alpha">result</p>
</body>
</html>
```

Reinicie o servidor e então vamos voltar a visualizar nossa aplicação:

http://localhost:8080/test

Você deve ver 256, que é 2 sobre a oitava potência.

3.9 Lendo parâmetros URL

Ao desenvolver uma aplicação web, você não deseja apenas mostrar dados, mas deseja também receber dados de entrada. Uma das maneiras mais simples para uma aplicação web receber dados é recuperando informação através de um parâmetro URL. Vamos planejar uma aplicação web que pode executar somas. Nesta aplicação, se você entrar com o seguinte URL na aplicação:

http://localhost:8080/test?value1=3&value2=5

Você deverá ver o somatório 8 como resultado na página. Modifique `app.py` para ficar como isto:

```
import grok

class Sample(grok.Application, grok.Container):
    pass

class Index(grok.View):
    def update(self, value1, value2):
        self.sum = int(value1) + int(value2)
```

Nós precisamos de um `index.pt` que usa `sum`:

```
<html>
<body>
<p tal:content="python:view.sum">sum</p>
</body>
</html>
```

Reinicie o servidor. Agora indo para a seguinte URL deve mostrar 8:

http://localhost:8080/test?value1=3&value2=5

Outros somatórios funcionam também:

http://localhost:8080/test?value1=50&value2=50

E se não suprirmos a URL com os parâmetros necessários (`value1` e `value2`) para a requisição ? Nós iremos capturar um erro:

http://localhost:8080/test

Você pode dar uma olhada na janela onde foi iniciado o servidor para ver o rastro do erro: O erro é relevantemente inaceitável:

```
TypeError: Missing argument to update(): value1
```

Nós podemos modificar nosso código para que funcione sem entrada por parâmetro:

```
import grok

class Sample(grok.Application, grok.Container):
    pass

class Index(grok.View):
    def update(self, value1=0, value2=0):
        self.sum = int(value1) + int(value2)
```

Reinicie o servidor, e veja se ele pode agora tratar a falta de parâmetros, mostrando como padrão o valor `0`.

3.10 Formulários Simples

Formulários automáticos

Criando formulários e convertendo e validando entrada de dados do usuário manualmente, como mostrando na seção anterior, pode ser desconfortável de usar . Com Grok, você pode use os sistemas *schema* e *formlib* para automatizar isto e muito mais. Isto irá ser discutido em uma seção posterior. TDB

Entrar com parâmetros pela URL não é muito amigável. Vamos usar um formulário para isso ao invés. Mude `index.pt` para conter um formulário, dessa maneira:

```
<html>
<body>
<form tal:attributes="action python:view.url()" method="GET">
  Value 1: <input type="text" name="value1" value="" /><br />
  Value 2: <input type="text" name="value2" value="" /><br />
  <input type="submit" value="Sum!" />
</form>
<p>The sum is: <span tal:replace="python:view.sum">sum</span></p>
```

```
</body>
</html>
```

Uma coisa a notar aqui é que nós dinamicamente geramos a propriedade `action` do formulário. Basicamente, nós fizemos o formulário enviar informações para seu próprio endereço. Visões Grok tem um método especial chamado `url` que pode ser usado para resgatar o endereço da visão (e outras visões que nós iremos entrar em mais detalhes posteriormente).

Deixe o `app.py` como na seção anterior, por agora. Você agora poderá ir para a página:

```
http://localhost:8080/test
```

Agora você pode enviar o formulário com alguns valores, e ver o resultado mostrado abaixo.

Entretanto, nós temos alguns padrões para tratar. Primeiramente, se nós não preenchermos com nenhum parâmetro e enviar o formulário, nós iremos capturar um erro como este:

```
File "../app.py", line 8, in update
  self.sum = int(value1) + int(value2)
ValueError: invalid literal for int():
```

Isso acontece devido ao fato de que os parâmetros foram string vazias, que não podem ser convertidas em números. Outra coisa é o fato de mostrar uma soma com valor 0 se não entramos com nenhum dado. Vamos mudar `app.py` para ter os dois casos de uso incluídos:

```
import grok

class Sample(grok.Application, grok.Container):
    pass

class Index(grok.View):
    def update(self, value1=None, value2=None):
        try:
            value1 = int(value1)
            value2 = int(value2)
        except (TypeError, ValueError):
            self.sum = "No sum"
            return
        self.sum = value1 + value2
```

Nós capturamos qualquer TypeError e ValueError aqui, logo qualquer dado errado ou faltando não irá resultar em falha. Ao invés, nós mostramos o texto “No sum”. Se nós não podemos capturar nenhum erro, a conversão para inteiro foi executada perfeitamente, e assim permitindo mostrar a soma.

Reinicie o servidor e vá para o formulário novamente e tente acessá-lo:

http://localhost:8080/test

CAPÍTULO 4

Modelos

Agora que nós sabemos como mostrar páginas web, nós precisamos ir para o que nós mostramos: os modelos. Esses modelos contém lógica independente da sua aplicação. Neste capítulo nós iremos discutir um número de ocorrências relacionadas a modelos: como suas visualizações se conectam aos modelos, e como ter certeza que os dados em seus modelos são armazenados seguramente. Como a complexidade de nossa aplicação sample cresceu, nós iremos então entrar em mais detalhes relacionados a tratamento de formulários.

4.1 Uma visão para um modelo

Até agora nós somente vimos visualizações que funcionam por sim mesmas. Em aplicações, normalmente este não é o caso - visualizações mostram informação que é armazenada de alguma maneira. Em aplicações Grok, visualizações trabalham para modelos: subclasses de `grok.Model` ou `grok.Container`. Para propósitos de discussão, nós podemos tratar a `grok.Container` como um outro tipo de `grok.Model` (mais sobre o que torna `grok.Container` especial será mostrado posteriormente). Nossa classe `Sample` é um `grok.Container`, então vamos usá-la para demostrar os princípios básicos. Vamos modificar `app.py` permitindo `Sample` disponibilizar algum dado:

```
import grok

class Sample(grok.Application, grok.Container):
    def information(self):
        return "This is important information!"

class Index(grok.View):
    pass
```

Neste caso, a informação (`"Isto é informação importante"`) é incluída dentro do código, mas você pode imaginar que a informação é recuperada de algum lugar, como uma base de dados relacional ou o sistema de arquivos.

Nós agora desejamos mostrar esta informação em nosso modelo `index.pt`:

```
<html>
<body>
<p tal:content="python:context.information()">replaced</p>
</body>
</html>
```

Reinicie o servidor. Quando você visualiza a página:

http://localhost:8080/test

Você deve ver o seguinte:

```
This is important information!
```

Anteriormente nós temos visto que você pode acessar métodos e atributos na visão usando o nome `view` em um modelo. `context` nos permite acessar informação no objeto de contexto da visão. Nesse caso é uma instância de `Sample`, nosso objeto que representa a aplicação.

Separando o modelo da visão é um importante conceito em aplicações estruturadas. A visão, juntamente com modelo de página, é responsável por mostrar a informação e a interface do usuário. O modelo representa o estado do informação (ou conteúdo) da aplicação, como documentos, entradas de blogs ou páginas wiki. O modelo não deve saber nada sobre a maneira em que é mostrado.

Esta maneira de estruturar suas aplicações permite você mudar a maneira em que o modelo é visualizado sem que o mesmo sofra alterações, mudando somente a maneira em que ele é visualizado.

Vamos fazer a visão fazer alguma coisa com a informação do modelo. Mude o `app.py` novamente:

```
import grok

class Sample(grok.Application, grok.Container):
    def information(self):
        return "This is important information!"

class Index(grok.View):
    def reversed_information(self):
        return ''.join(reversed(self.context.information()))
```

Você pode ver que é possível acessar o objeto de contexto (um instância de `Sample`) através da classe de visão, acessando o atributo `context`. Isto captura o mesmo objeto da mesma maneira em que foi usada no modelo de página anteriormente.

O que nós estamos fazendo aqui é inverter o conjunto de caracteres retornados do método `'information()` . Você pode tentar usá-lo pelo interpretador Python:

```
>>> ''.join(reversed('foo'))
'oof'
```

Agora vamos modificar o modelo de página `index.pt` que usa o método `reversed_information`:

```
<html>
<body>
<p>The information:
  <span tal:content="python:context.information()">info</span>
</p>
<p>The information, reversed:
  <span tal:replace="python:view.reversed_information()">info</span>
</p>
</body>
</html>
```

Reinicie o servidor. Quando você visualizar a página:

> http://localhost:8080/test

Agora você deve estar vendo o seguinte:

> The information: This is important information!
>
> The information, reversed: !noitamrofni tnatropmi si sihT

4.2 Armazenando Dados

Por enquanto nós só visualizamos dados que foram inseridos manualmente, ou cálculos baseados na entrada do usuário. O que fazer para *armazenar* informação, como dados inseridos pelo usuário ? A maneira mais simples de fazer isso com o Grok é usar o Zope Object Database (ZODB).

O ZODB é um banco de dados para objetos Python. Você pode armazenar qualquer objeto Python através dele, entretanto seguindo regras simples (as "regras de persistência" , que nós entraremos em detalhes futuramente). Nosso objeto aplicação `Sample` é armazenado no banco de objetos, logo podemos armazenar informação nele.

Vamos criar uma aplicação que armazene um trecho de texto para nós. Nós iremos usar uma visão para mostrar o index (`index`) e outra para editá-la (`edit`).

Modifique `app.py` para ficar como isto:

```
import grok

class Sample(grok.Application, grok.Container):
    text = 'default text'

class Index(grok.View):
    pass

class Edit(grok.View):
    def update(self, text=None):
```

```
        if text is None:
            return
        self.context.text = text
```

A classe `Sample` ganhou um atributo de classe com um valor padrão. No método `update` da classe de visão `Edit` você pode ver que o atributo `text` é atribuído ao contexto, se ele tiver algum valor enviado pelo formulário. Isto irá definir o atributo `text` na instância do objeto `Sample` na base de objetos, e irá sobreescrever o valor padrão definido como atributo de classe `text`.

Mude o modelo de página `index.pt` para ficar como isto:

```
<html>
<body>
<p>The text: <span tal:replace="python:context.text">text</span></p>
</body>
</html>
```

Isto é um modelo simples que somente mostra o atributo `text` do objeto `context` (nossa instância `Sample`).

Crie um modelo de página `edit.pt` com o seguinte conteúdo:

```
<html>
<body>
<form tal:attributes="action view/url" method="POST">
Text to store: <input type="text" name="text" value="" /><br />
<input type="submit" value="Store" />
</form>
</body>
</html>
```

Este modelo de página mostra um formulário requisitando como entrada um pequeno trecho de texto. Ele envia as informações do formulário para seu próprio endereço.

Reinicie o servidor. Vamos primeiramente visualizar a página index:

> http://localhost:8080/test

Você deve ser o texto `default text`.

Agora vamos modificar o texto acessando a página de edição da aplicação:

> http://localhost:8080/test/edit

Digite algum texto e pressione o botão "Armazenar". Como ele envia as informações para ele mesmo, você verá o formulário novamente. Após isso, vá para a página de edição.

> http://localhost:8080/test

Você agora deve ver o texto que você entrou na página de edição. Isso significa que seu texto foi armazenado com sucesso no banco de objetos! Você pode também reiniciar o servidor e voltar para a página index, e seu texto continuará lá.

4.3 Redirecionamento

Vamos criar nossa aplicação mais fácil de usar. Primeiramente, vamos mudar o modelo de página `index.pt` para que ele inclua um endereço para a página de edição. Para fazer isto, nós iremos usar o método `url` na visão:

```
<html>
<body>
<p>The text: <span tal:replace="python:context.text">text</span></p>
<p><a tal:attributes="href python:view.url('edit')">Edit this page</a></p>
</body>
</html>
```

Informando ao método `url` um argumento, ele irá retornar uma URL para a visão nomeada do mesmo objeto (`test`), logo neste caso `test/edit`. Agora vamos mudar o formulário de edição para que ele redirecione para a página `index` depois de pressionar o botão de enviar:

```
import grok

class Sample(grok.Application, grok.Container):
    text = 'default text'

class Index(grok.View):
    pass

class Edit(grok.View):
    def update(self, text=None):
        if text is None:
            return
        self.context.text = text
        self.redirect(self.url('index'))
```

A última linha é a mais recente. Nós usamos o método `url` na classe de visão para construir o endereço para a página `index`. Estando na visão, nós podemos simplesmente chamar `url` em `self`. Então, nós passamos isto para outro método especial disponível em todas as subclasses de `grok.View`, `redirect`. Nós assim diremos ao sistema para redirecionar a página `index`.

4.4 Mostrando o valor no formulário

Vamos mudar nossa aplicação para que ela mostre o que nós armazenamos no formulário de edição, não somente na página de edição.

Para fazer isso funcionar, mude o edit.py para que ele fique assim:

```
<html>
<body>
<form tal:attributes="action view/url" method="POST">
```

```
Text to store: <input type="text" name="text" tal:attributes="value python:conte
<input type="submit" value="Store" />
</form>
</body>
</html>
```

A única mudança é nós usamos `tal:attributes` para incluir o valor do atributo `texto` no formulário de contexto.

4.5 As regras de persistência

Essas são as "regras de persistência":

- **Você deve herdar a partir da classe `persistent.Persistent` se** você deseja que suas classes possam armazenar dados . A maneira mais simples de fazer isso com grok é herdar a partir da classe `grok.Model` ou `grok.Container`.
- Os objetos que você deseja armazenar devem estar conectados a outras classes persistentes que já foram armazenadas. A maneira mais simples de fazer isso com o Grok é anexá-lo de alguma maneira a um objeto `grok.Application`, diretamente ou indiretamente. Isto pode ser feito definindo-os como um atributo, ou incluíndo-os em um Container (se você fez sua aplicação subclassear de `grok.Container`).
- Para ter certeza que o ZODB sabe que você modificou um atributo mutável, é possível definir um atributo chamado `_p_changed` para `True`. Isto é somente necessário se o atributo é `Persistent` por si só. Isto não é necessário quando você cria ou sobreescreve um atributo diretamente usando =.

Se você construir o conteúdo da sua aplicação sem usar as classes `grok.Model` e `grok.Container`, você deve seguir as regras descritas. Lembre-se de definir `_p_changed` em seu métodos se você modificar uma lista em Python (com método `append`, por exemplo) ou dicionário (armazenando um valor).

O código da seção *Storing data_* é um exemplo simples. Nós não precisamos fazer nada em especial para obedecer as regras de persistência naquele caso.

Se nós usamos um objeto mutável como uma lista ou dicionário para armazenar dados, nós precisamos executar ações especiais. Vamos mudar nosso código de exemplo (baseado na seção anterior) para usar um objeto mutável (uma lista):

```
import grok

class Sample(grok.Application, grok.Container):
    def __init__(self):
        super(Sample, self).__init__()
        self.list = []

class Index(grok.View):
    pass
```

```
class Edit(grok.View):
    def update(self, text=None):
        if text is None:
            return
        # this code has a BUG!
        self.context.list.append(text)
        self.redirect(self.url('index'))
```

Nós mudamos agora a classe `Sample` para fazer uma coisa nova: ele tem um método `__init__`. Toda vez que você cria um objeto de aplicação `Sample`, ele será criado com um atributo chamado `list`, que irá conter uma lista vazia. Nós temos também temos que ter certeza que método `__init__` da superclasse continua sendo executado, usando o método regular do Python `super`. Se nós fizermos isso, nosso container poderá não ser totalmente inicializado.

Você poderá notar a pequena mudança no método `update` da classe `Edit`. Ao invés de somente armazenar o texto como um atributo do nosso modelo `Sample`, nós adicionamos cada texto que entramos para adicionar a nova `lista`.

Note que o código tem um súbito defeito, que é o porquê do nosso comentário. Nós iremos ver que o defeito é bem pequeno. Primeiramente, vamos mudar nossos modelos.

Nos mudamos `index.pt` que mostra a lista:

```
<html>
<body>
We store the following texts:
<ul>
  <li tal:repeat="text python:context.list" tal:content="text"></li>
</ul>
<a tal:attributes="href python:view.url('edit')">Add a text</a>
</body>
</html>
```

Nós então mudamos o texto do link para a página `edit` par refletir o novo comportamento de adição da nossa aplicação.

Nós precisamos desfazer a mudança para o modelo `edit.pt` que nós fizemos na última seção, pois cada vez que editamos um texto nós *adicionamos* um novo texto, ao invés de mudar o original. Conseqüentemente, não há texto para ser visualizado:

```
<html>
<body>
<form tal:attributes="action view/url" method="POST">
Text to store: <input type="text" name="text" value="" /><br />
<input type="submit" value="Store" />
</form>
</body>
</html>
```

Evolução

O que fazer quando você muda uma estrutura de armazenamento enquanto sua aplicação está em produção ? Nas próximas seções, nós iremos falar sobre o mecanismo de evolução do Zope Toolkit que permite a você atualizar objetos em base de dados existente. TDB

Vamos reiniciar o servidor. Se você está seguindo este tutorial a partir da última seção, você irá agora ver um erro quando você olhar na página inicial da aplicação:

```
A system error occurred.
```

Veja a saída que é capturada quando a página é carregada:

```
AttributeError: 'Sample' object has no attribute 'list'
```

Mas nós somente mudamos nosso objeto para tem um atributo `list`, certo ? Sim, mas somente para novas instâncias do objeto `Sample`. O que nós estamos procurando mostrar é que o objeto sample de antes continua armazenado na base de dados. Ele não tem o atributo. Isto não é um defeito na verdade (com relação ao defeito, veja adiante): é um problema de banco de dados.

O que fazer agora ? A ação mais simples a ser tomada durante o desenvolvimento é simplesmente remover nossa aplicação instalada, e criar uma nova que terá esse novo atributo. Vá para a tela de admin do Grok:

> http://localhost:8080

Selecione o objeto de aplicação (`test`) e apague o. Agora instale-o novamente, como `test`. Agora vá para a tela de edição e adicione um texto:

> http://localhost:8080/test/edit

Clique em `add a text` e adicione outro texto. Você irá ver o novo texto aparecendo na página `index`.

Tudo está ok agora, certo? Não está ! Agora nós iremos resolver o nosso defeito. Reinicie o servidor e olhe para a página inicial novamente:

> http://localhost:8080/test

Nenhum dos textos que nós adicionamos foram salvos! O que aconteceu ? Nós modificamos um atributo mutável e ele não notificou a base de dados a alteração que nós fizemos. Isso significa que a base de objetos não só manteve nossas alterações no objeto na memória, e ainda não as salvou no disco.

Nós podemos facilmente resolver adicionando uma linha para o código:

```
import grok

class Sample(grok.Application, grok.Container):
    def __init__(self):
        super(Sample, self).__init__()
        self.list = []
```

```
class Index(grok.View):
    pass

class Edit(grok.View):
    def update(self, text=None):
        if text is None:
            return
        self.context.list.append(text)
        self.context._p_changed = True
        self.redirect(self.url('index'))
```

Nós agora informamos ao servidor que o contexto do objeto mudou (pois nós modificamos um subobjeto mutável), adicionando a linha:

```
self.context._p_changed = True
```

Se você agora adicionar muitos textos e então reiniciar o servidor, você irá notar que o dado continuará: ele foi armazenado com sucesso na base de objetos.

O código mostrado é um pouco desagradável no sentido de que tipicamente queremos gerenciar o estado no código do modelo (o objeto `Sample` neste caso), e não na visão. Vamos fazer uma última mudança para mostrar como poderia ficar:

```
import grok

class Sample(grok.Application, grok.Container):
    def __init__(self):
        super(Sample, self).__init__()
        self.list = []

    def addText(self, text):
        self.list.append(text)
        self._p_changed = True

class Index(grok.View):
    pass

class Edit(grok.View):
    def update(self, text=None):
        if text is None:
            return
        self.context.addText(text)
        self.redirect(self.url('index'))
```

Como você pode ver, nós criamos um método `addText` no modelo que cuidará de incluir na lista e informar ao ZODB sobre isso. Desta maneira, qualquer código pode seguramente usar a API da classe `Sample` sem se preocupar com regras de persistência, que é responsabilidade do modelo.

4.6 Explicitamente associando uma visão a um modelo

Como Grok sabe que uma visão se relaciona a um modelo? Nos exemplos anteriores, Grok tem feito essa associação automaticamente. Grok pode fazer isso devido a ter somente um modelo definido (`Sample`). Neste caso, Grok é esperto suficiente para automaticamente associar todas as visualizações definidas no mesmo módulo. Por debaixo dos panos, Grok faz o modelo ser o *contexto* das classes de visualizações.

Tudo o que Grok faz implicitamente pode também informado ao Grok para ser feito explicitamente. Isso será bastante ajuda depois, pois você pode algumas vezes dizer o Grok o que fazer, sobrescrevendo seu comportamento padrão. Para associar uma visão com um modelo automaticamente, você deve usar a nota de classe `'grok.context`.

O que é uma nota de classe ? Uma nota de classe é uma maneira declarativa de dizer ao Grok alguma coisa sobre uma classe Python. Vamos olhar um exemplo. Nós iremos mudar `app.py` do exemplo de *Uma segunda visão* para demostrar o uso de `grok.context`:

```
import grok

class Sample(grok.Application, grok.Container):
    pass

class Index(grok.View):
    grok.context(Sample)

class Bye(grok.View):
    grok.context(Sample)
```

Este código se comporta da mesma maneira como no exemplo anterior no `Uma segunda visão`, mas tem relacionamento entre o modelo e o visão feito explicitamente, usando a nota de classe `grok.context`.

`grok.context` é somente uma nota de classe dentre muitas. Nós iremos ver outra na próxima seção.

4.7 Um segundo modelo

Como combinar modelos em uma única aplicação ?

Curioso agora sobre como adicionar modelos em uma única aplicação?
Não consegue esperar ? Olhe a seção *Containers* a seguir, ou *Traversal* posteriormente.
TDB

Nós iremos agora estender nossa aplicação com um segundo modelo. Como nós não explicamos como combinar modelos em uma única aplicação, nós iremos criar uma segunda aplicação juntamente com o nossa primeira aplicação. Normalmente nós desejamos definir duas aplicações no mesmo módulo, mas nós estamos tentando ilustrar alguns pontos, logo seja paciente.

Mude `app.py` de forma que ele pareça como isto:

```
import grok

class Sample(grok.Application, grok.Container):
    pass

class Another(grok.Application, grok.Model):
    pass

class SampleIndex(grok.View):
    grok.context(Sample)
    grok.name('index')

class AnotherIndex(grok.View):
    grok.context(Another)
    grok.name('index')
```

Agora nós definimos uma segunda classe de aplicação, `Another`. Ele herda de `grok.Application` para torná-la uma aplicação instalável.

Ele então herde de `grok.Model`. Há uma diferença entre `grok.Model` e `grok.Container`, mas para propósitos de discussão nós podemos ignorá-lo por agora. Nós mostramos que nós podemos usar `grok.Model` para uma variedade de coisas, entretanto nós poderíamos ter herdado de `grok.Container` ao invés.

Nós então definimos dois modelos de página, um chamado `sampleindex.pt`:

```
<html>
<body>
<p>Sample index</p>
</body>
</html>
```

E um chamado `anotherindex.pt`:

```
<html>
<body>
<p>Another index</p>
</body>
</html>
```

Nós nomeamos os modelos de página com os nomes das classes em minúsculo, assim os modelos de página serão associadas a elas.

Você irá notar que nós usamos `grok.context` para associar as visualizações com modelos. Nós *temos* que fazer isso aqui, evitando ambiguidades serem detectadas pelo Grok. Sem o uso de `grok.context`, nós poderíamos ver uma erro parecido com esse ao iniciar:

```
GrokError: Multiple possible contexts for <class
'sample.app.AnotherIndex'>, please use grok.context.
```

Logo, nós usamos `grok.context` para explicitamente associar `SampleIndex` com a aplicação `Sample`, e novamente associar `AnotherIndex` com a aplicação `Another`.

Nós temos outro problema: a intenção dessas visões é ser a página `index` de cada aplicação. Isto não pode ser deduzido automaticamente do nome da visão, entretanto Grok deve ter chamado as visualizações de `sampleindex` e `anotherindex` .

Nós temos outra nota de classe que pode nos ajudar aqui: `grok.name` Nós podemos usá-la nas duas classes (`grok.name('index')`) para explicitamente explicar ao Grok o que nós queremos.

Você pode agora tentar reiniciar o servidor e criar as aplicações na interface de Admin do Grok. Eles deve mostrar as páginas iniciais corretamente ao tentar visualizá-las.

Nós podemos ver que a introdução do segundo modelo tem um pouco de código complicado, contudo você irá concordar conosco que ele continua expressivo. Nós poderíamos ter esquivado do problema simplesmente incluíndo `Another` e suas visões em outro módulo como `another.py`. As visões associadas poderíam então ser incluídas em um diretório chamado `another_templates`. Frequentemente você irá ser possível estruturar sua aplicação de uma maneira que seja possa usar as convenções padrão do Grok.

4.8 Containers

Um container é um tipo especial de objeto que pode conter outros objetos. Nossa aplicação `Sample` é também um container, pois herdar de `grok.Container`. O que nós iremos fazer nesta seção é construir uma aplicação que inclui alguma coisa naquele container.

Aplicações Grok são normalmente compostas de containers e modelos. Container são objetos que podem conter modelos. Isto inclui outros containers, pois um container é um tipo especial de modelo.

Da perspectiva do Python, você pode pensar que containers são como dicionários. Eles permitem acesso um item (`container['key']`) para pegar seu conteúdo. Ele possui métodos como `keys()` e `values()`. Containers fazer muito mais do que dicionários Python: eles são persistentes, e quando voce os modifica, você não tem que usar *_p_changed* para notificar que você os modificou. Eles também podem enviar eventos especiais que você pode escutar quando items são incluídos ou removidos. Para mais sobre isso veja a seção sobre eventos (TDB).

Nosso objeto de aplicação irá ter um página index que irá mostrar a lista de itens no container. Você pode clicar em um item na lista para visualizar aquele item. Abaixo da lista, ele irá mostrar um formulário que permite você criar novos items.

Aqui está o `app.py` da nossa nova aplicação:

```
import grok

class Sample(grok.Application, grok.Container):
    pass

class Entry(grok.Model):
```

```
    def __init__(self, text):
        self.text = text

class SampleIndex(grok.View):
    grok.context(Sample)
    grok.name('index')

    def update(self, name=None, text=None):
        if name is None or text is None:
            return
        self.context[name] = Entry(text)

class EntryIndex(grok.View):
    grok.context(Entry)
    grok.name('index')
```

Como você pode ver, `Sample` está inalterada. Nós criamos também nosso primeiro objeto que não é aplicação, `Entry`. Ele é somente um `grok.Model`. Ele precisa ser criado com um argumento `text` e esse texto é armazenado nele.Logo, nós pretendemos incluir instâncias de `Entry` em nosso container `Sample`.

Depois são as visões. Nós temos uma página `index` para o container `Sample`. Quando o o método `update()` é executado com dois valores, `name` e `text`, ele irá criar uma nova instância de `Entry` com o texto informado e irá incluí-lo no container sob o nome de `name`. Nós usamos a interface similar a do dicionário de nosso `Container` para incluir o novo objeto `Entry` no container.

Aqui está o modelo associado para `SampleIndex`, `sampleindex.pt`:

```
<html>
<head>
</head>
<body>
  <h2>Existing entries</h2>
  <ul>
    <li tal:repeat="key python:context.keys()">
      <a tal:attributes="href python:view.url(key)"
         tal:content="python:key"></a>
    </li>
  </ul>

  <h2>Add a new entry</h2>
  <form tal:attributes="action python:view.url()" method="POST">
    Name: <input type="text" name="name" value="" /><br />
    Text: <input type="text" name="text" value="" /><br />
    <input type="submit" value="Add entry" />
  </form>

</body>
</html>
```

A primeira seção no modelo (<h2>Existing entries</h2>) mostra uma lista de items no container. Nós novamente usamos a API similar a uma lista usando o `keys()` para listar todos os nomes dos items em um container. Nós criamos um link para esses items usando `view.url()`.

A próxima seção (<h2>Add a new entry</h2>) mostra um formulário simples que envia dados para seu próprio endereço. Ela tem dois campos, `name` e `text`, que nós já tratamos em `update()`.

Finalmente, nós temos uma página `index` para `Entry`. Ele tem somente um modelo para mostrar o atributo `texto`:

```
<html>
<head>
</head>
<body>
  <h2>Entry <span tal:replace="python:context.__name__"></span></h2>
  <p tal:content="python:context.text"></p>
</body>
</html>
```

Reinicie o servidor e tente usar esta aplicação. Acesse sua aplicação `test`. Tenha atenção especial nas URLs.

Primeiro, nós temos a página inicial da nossa aplicação:

> http://localhost:8080/test

Quando nós criamos uma entrada chamada `hello` ` no formulário, e então clicamos na lista, você verá que a URL parece com isto:

> http://localhost:8080/test/hello

Nós estamos agora olhando para a página index da instância `Entry` chamada `hello`.

Que tipo de extensões para esta aplicação nós podemos pensar? Nós podemos criar um formulário `edit` que nos permita editar o texto das entradas. Nós podemos modificar nossa aplicação de forma que você não somente adicione instâncias de `Entry`, mas outros containers. Se você fez estas modificações, você construindo seu próprio sistema de gerenciamento de conteúdo com o Grok.