Grok Community Docs Documentation

Release 1.2

2010-11, The Grok Community

November 16, 2011

CONTENTS

1 About the grok documentation **3**
- 1.1 Introduction . . . 3
- 1.2 Building the docs . . . 3

2 Data Access **5**
- 2.1 Acess content in the static directory from Python code . . . 5
- 2.2 Indexing and Searching Objects in the ZODB . . . 6
- 2.3 Basic ORM with megrok.rdb and SQLAlchemy . . . 11
- 2.4 Grok ORM with Storm . . . 18
- 2.5 How I Got Grok Talking To CAS . . . 22
- 2.6 Navigating To Transient Objects Tutorial . . . 25
- 2.7 Create simple 1:2 relationship with Megrok.rdb and SqlAlchemy (over MySql) . . . 36
- 2.8 Understanding default values for object database backed attributes . . . 41

3 Security and Authentication **45**
- 3.1 Authentication with Grok . . . 45
- 3.2 Authentication and authorization in Grok . . . 48

4 Views, Templating, Client Side **55**
- 4.1 Fanstatic resources . . . 55
- 4.2 Using a KSS plugin for Drag-and-Drop . . . 56
- 4.3 Using z3x.form with Grok . . . 59
- 4.4 Traversing subpaths in views . . . 61
- 4.5 Adding AJAX to Grok with KSS . . . 63
- 4.6 Automatic Form Generation . . . 65
- 4.7 Rest support in Grok . . . 70
- 4.8 What is XML-RPC ? . . . 72
- 4.9 Working with Forms in Grok . . . 74
- 4.10 Understanding viewlets . . . 96
- 4.11 Using Viewlets for Layout . . . 98
- 4.12 Plugging in new template languages . . . 102
- 4.13 How to internationalize your application . . . 105
- 4.14 Using sources in your forms . . . 109
- 4.15 Use the same view in multiple models . . . 110
- 4.16 Creating forms with megrok.z3cform . . . 112
- 4.17 Generate URLs with the url() function in views . . . 115

5 Life Cycle of Grok Applications **117**
- 5.1 The lifecycle of a Grok application . . . 117

5.2 Selecting the port and interface where Grok listens . . . 121
5.3 Install Grok on MS Windows . . . 123
5.4 Placing your Grok project under version control . . . 124
5.5 Profiling with Grok . . . 126
5.6 Eggs, Known Good Sets and developing with unreleased Grok source code . . . 130
5.7 How to pack your ZODB database . . . 134
5.8 Graphical debugging of Grok with Komodo IDE . . . 135
5.9 Grok, Virtual Hosting and Nginx . . . 136
5.10 Grok and Apache . . . 137
5.11 Releasing software . . . 138
5.12 Using Virtualenv for a clean Grok installation . . . 140
5.13 Install multiple Grok apps using zc.buildout . . . 142
5.14 Use Apache HTTP server with Grok (on Debian Sid) . . . 144
5.15 Legal stuff . . . 147
5.16 Set custom configurations on a system level that your application can use . . . 150

6 Testing **153**
6.1 How to test docstrings with `z3c.testsetup` . . . 153
6.2 Writing tests, discovering and running them with Grok.testing . . . 155

7 Egg Collections **159**
7.1 Dolmen . . . 159

8 Eggs in Production **169**

9 Entry Level Eggs **171**
9.1 gp.fileupload . . . 171
9.2 zope.sendmail . . . 172

10 Snippets **175**
10.1 Getting the current user/principal . . . 175
10.2 Finding out if you are in devmode . . . 175
10.3 Getting custom config from ZCML . . . 175
10.4 Getting the user's language . . . 176

11 Solving Common Tasks **177**
11.1 File Uploads . . . 177
11.2 JSON . . . 177
11.3 Sending e-mail . . . 178
11.4 Using a relationfield to express relationships between objects . . . 178
11.5 Workflow . . . 184
11.6 Exporting content as xml . . . 184
11.7 How to automatically install and maintain your Grok application in to the ZODB . . . 184

12 User Tutorials **187**
12.1 Contribute to the Grok documentation . . . 187
12.2 A Grok-Centric Explanation of Adaptation . . . 192
12.3 Macros with Grok Tutorial . . . 197
12.4 Navigating to transient objects . . . 200
12.5 Permissions Tutorial . . . 211
12.6 Musical Performance Organizer - Annotated . . . 215

13 License **269**

14 Copyright **271**

15 Indices and tables 273

Contents:

CHAPTER

ONE

ABOUT THE GROK DOCUMENTATION

Author Matthias (nitrogenycs)

Version n/a

1.1 Introduction

This documentation was created by the grok community.

If there is anything missing or unclear, please don't hesitate to _contact us: http://grok.zope.org/community ! The #grok irc channel and mailing lists are full of very helpful people.

Even better, you can help us improving the docs in an easy way. Just take a quick look at the next section.

1.2 Building the docs

To build the docs you need mercurial (hg).

Windows Users

Install mercurial if you don't have it already, create a folder and execute these commands:

```
hg clone http://bitbucket.org/jhsware/grok-doc
bootstrap.py
bin\buildout
bin\sphinx-build source build
```

Unix/Mac Users

Install mercurial if you don't have it already. Debian/Ubuntu users can do so by:

```
$ sudo apt-get install mercurial
```

Then, get the sources and build the docs:

```
$ hg clone http://bitbucket.org/jhsware/grok-doc
$ cd grok-doc
$ python bootstrap.py
$ bin/buildout
$ make
```

Now you should be able to open the docs at `build/index.html`.

The commands above pulled the sphinx source code for the documentation and built it.

Now you can just go wild and modify the files in the /source subfolder to your liking. If you are not familiar with the syntax, just take a look at the sphinx docs: http://sphinx.pocoo.org/markup/index.html .

Don't be afraid to make mistakes, your changes will be reviewed anyway.

To rebuild the local docs after changes run:

```
$ make
```

on Unix/Mac or:

```
bin\sphinx-build source build
```

on Windows.

To commit your changes you need to make a push request on bitbucket.org or ask jhsware (in #grok channel or on the mailing list) to grant you commit access to the grok-docs repository.

Thanks for helping us to improve the documentation!

CHAPTER

TWO

DATA ACCESS

Contents:

2.1 Acess content in the static directory from Python code

Author Sebastian Ware (jhsware)

Version Grok <= 1.2.x

An interesting addition to this tutorial is how to access these static resources through python code, for instance inside a grok View.

I'll quickly describe it here and you guys can massage it into mini-tutorial style whichever way you want. CAVEAT, almost nothing below was tested, I mostly read the "zope.app.publisher.browser.directoryresource" and "...fileresource" code.

say you have:

```
class MyView(grok.View):

  def someMethod(self):
       [...]
```

inside the method above, you can access the static resources through "self.static" which is a DirectoryResource, a dictionary-like object representing the "static" folder.

suppose you have an "image.png" file inside an "images" folder inside the "static" directory, you can access it like:

```
image_resource = self.static['images']['image.png']
```

or even:

```
image_resource = self.static['images/image.png']
```

The forward slash is important as this actually represents a url traversal, not a file-system traversal.

If you're not sure the image is really there, you can do:

```
image_resource = self.static.get('images/image.png', None)
```

if you got None back the image is not there.

Now that you have the image resource, what do you do with it? The main motivation for accessing the static resources is returning their URLs on the server to the browser, so you can do this by just calling the resource:

```
return image_resource() # returns the URL for the static resource.
```

If you really need to manipulate the resource in the filesystem, you can do it like this:

```
fspath = image_resource.context.path
file = open(fspath, "rb")
```

The same goes for snooping the static resource directory and subdiretories under it:

```
image_dir = self.static['images'].context.path
os.listdir(image_dir)
```

Cheers, Leo

2.2 Indexing and Searching Objects in the ZODB

Author Sebastian Ware (jhsware)

Version unkown

Indexing the contents of your objects allow
you to perform fast complex search operations.

2.2.1 Introduction

Relational databases provide ad hoc search capability by means of SQL queries. In order to perform search operations on objects stored in your ZODB you need to explicitly create indexes. These indexes will update automatically when an object is modified provided it fires the IObjectModified event. The upside to this approach is that you will be be inclined to create simple and well designed indexes that in turn will scale well.

Grok supports the vanilla indexing services available in Zope 3 straight out of the box.

- FieldIndex: search matching an entire field
- SetIndex: search for keywords in a field
- TextIndex: full-text searching
- ValueSearch: search for values using ranges

You won't be able to perform SQL-style joins to search related objects. Instead you could index an adapter with calculated properties.

2.2.2 Setup

The egg (package) containing the indexing functionality is called [zc.catalog- x.x.x-py2.x.egg]. The package is installed by including "zc.catalog" in the list "install_requires" in [setup.py]:

```
install_requires=['setuptools',
              'grok',
              'zc.catalog',
              'hurry.query',
              ],
```

The “hurry.query” package gives you some simple tools to perform advanced searching.

VERSION PROBLEMS: If you are using Grok <1.1 you need to pin down an earlier version of hurry.query in your buildout.cfg file. The error you will experience is: ComponentLookupError: (<InterfaceClass zope.app.intid.interfaces.IIntIds>,’‘)

```
[buildout]
...
versions = versions
[versions]
hurry.query = 0.9.2
```

Don’t forget to re-run buildout.

2.2.3 Example

```
# interfaces.py
class IProtonObject(Interface):
    """
    This is an interface to the class who’s objects I want to index.
    """
    body = schema.Text(title=u’Body’, required=False)

# protonobject.py
class ProtonObject(grok.Model):
    """
    This is the actual class.
    """
    interface.implements(interfaces.IProtonObject)

    def __init__(self, body):
        self.body = body

# app.py
import grok
from grok import index
from hurry import query
from hurry.query.query import Query, Text
# hurry.query is a simplified search query language that
# allows you to create ANDs and ORs.

class ContentIndexes(grok.Indexes):
    """
    This is where I setup my indexes. I have two indexes;
    one full-text index called "text_body",
    one field index called "body".
    """
    grok.site(ProtonCMS)

    grok.context(interfaces.IProtonObject)
    # grok.context() tells Grok that objects implementing
    # the interface IProtonObject should be indexed.

    grok.name(’proton_catalog’)
    # grok.name() tells Grok what to call the catalog.
    # if you have named the catalog anything but "catalog"
    # you need to specify the name of the catalog in your
    # queries.
```

```
    text_body = index.Text(attribute='body')
    body = index.Field(attribute='body')
    # The attribute='body' parameter is actually unnecessary if the attribute to
    # be indexed has the same name as the index.

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

    def search_content(self, search_query):
            # The following query does a search on the field index "body".
            # It will return a list of object where the entire content of the body attribute
            # matches the search term exactly. I.e. search_query == body
            result_a = Query().searchResults(
                               query.Eq(('proton_catalog', 'body'), search_query)
                               )

            # The following query does a search on the full-text index "text_body".
            # It will return objects that match the search_query. You can use wildcards and
            # boolean operators.
            #
            # Examples:
            # "grok AND zope" returns objects where "body" contains the words "grok" and "zope"
            # "grok or dev*" returns objects where "body" contains the word "grok" or any word
            # beginning with "dev"
            result_b = Query().searchResults(
                               Text( ('proton_catalog', 'text_body'), search_query)
                               )

            return result_a, result_b
```

2.2.4 Setting up a value index

You need to import zc.catalog to index values. First you need to create a Grok compatible index class.

```
from zc.catalog.catalogindex import ValueIndex
from grok.index import IndexDefinition
class Value(IndexDefinition):
    index_class = ValueIndex
```

Then you can use this to create your actual value index in your catalog.

```
class SiteCatalog(grok.Indexes):
    grok.site(Testvalueindex)
    grok.context(MyObject)
    grok.name('my_catalog')

    counter = Value()
```

This will index the property “counter” on objects of type “MyObject”. This index supports searches such as greater than, less than, in between. It also supports sorting.

2.2.5 Adding an index to an existing appplication

In the above example, the indexes are only added when a new application is installed. If you want to add an index to an existing application and you have a catalog this is what you do:

```
import grok
from zope.catalog.interfaces import ICatalog
from zope.component import getUtility
from zope.catalog.field import FieldIndex

app = grok.getSite()
catalog = getUtility(ICatalog, context=app)
name = 'new_index_name'
if not name in catalog:
    catalog[name] = FieldIndex(name, IMyObjects)
```

This finds your catalog and adds a FieldIndex that will index objects implementing the interface *IMyObjects*.

2.2.6 Querying the Index Using hurry.query

```
from zope.component import getUtility
from hurry.query.interfaces import IQuery
from hurry.query import value

class Index(grok.View):
    grok.context(MyApp)
    def render(self):
        mini = int(self.request.form.get('mini', 1))
        maxi = int(self.request.form.get('maxi', 99))

        query = getUtility(IQuery)
        q = value.Between(('content_index', 'counter'), mini, maxi)
        res = query.searchResults(q)
        outp = [e.counter for e in r]
        return "%s" % outp
```

This will display a list of values. If you are using hurry.query 1.1.0 or higher, you can pass sorting options to the query method. If not, you need to get the catalog and sort calling the index directly.

```
from zope.component import getUtility
from zope.catalog.interfaces import ICatalog

class Dates(grok.View):
    grok.context(MyApp)
    def render(self):
        mini = int(self.request.form.get('mini', 1))
        maxi = int(self.request.form.get('maxi', 12))
        limit = int(self.request.form.get('limit', 10))

        # Perform the query, returning a result set
        res = self.findMe(d_mini, d_maxi)

        # get the catalog
        content_catalog = getUtility(ICatalog, 'my_catalog')

        # sort the result and return limited result set
        tmp = content_catalog['published'].sort(res.uids, limit=limit)

        # create list of objects
        objs = [res.uidutil.getObject(o) for o in tmp]
        return "%s" % [e.counter for e in objs]
```

```
    def findMe(self, mini, maxi):
        q = value.Between(('content_index', 'published'), mini, maxi)
        query = getUtility(IQuery)
        r = query.searchResults(q)
        return r
```

This also shows how to find a catalog, which is useful if you want to check statistics on the index or need to update (reindex) the index.

If you want to index datetime properties, there is a datetime normalizer which I never got to work. Instead I did something like this.

```
from zope.interface import Interface
from zope import schema
class IPublished(Interface):
    published = schema.Int(title=u'Normalized datetime')

def _minuteNormalizer(dt):
    tmpin = dt.utctimetuple()[:5]
    multi = (535680, 44640, 1440, 60, 1) # Resolution in minutes
    value = sum(i*j for i,j in zip(tmpin, multi))
    return value

class Published(grok.Adapter):
    grok.implements(IPublished)
    grok.context(MyObj)
    def _published(self):
        return _minuteNormalizer(self.context.published)
    published = property(_published)

class SitePublishCatalog(grok.Indexes):
    grok.site(MyApp)
    grok.context(IPublished)
    grok.name('my_catalog')

    published = Value()
```

The SitePublishCatalog uses the IPublished() adapter to convert the datetime property "published" to an integer. In order to perform a query you will need to normalize your parameters too. Don't forget timezones or you might get unexpected results. I use the "pytz" egg to get preconfigured timezones.

```
from pytz import timezone

d_mini = datetime(2010, 1, 1, tzinfo = timezone('CET'))
d_maxi = datetime(2010, 12, 31, tzinfo = timezone('CET'))
q = value.Between(('content_index', 'published'), _minuteNormalizer(d_mini), _minuteNormalizer(d_max
query = getUtility(IQuery)
r = query.searchResults(q)
```

2.2.7 Learning More

The "hurry.query" package contains the DocTest "query.txt" that shows how to perform more complex search queries.

2.3 Basic ORM with megrok.rdb and SQLAlchemy

Author Jeffrey D Peterson

Version Grok-1.0 >= 1.0, <= 1.1

This guide will take you through the process of
mapping relational data using megrok.rdb and by
extension SQLAlchemy (SQLA).

2.3.1 Prerequisities

- Grok 1.0 or 1.1 (It may work with earlier versions but I haven't tested it.)
- megrok.rdb 0.11.0
- SQLA 0.5.7, 0.5.8; I have tested it against 0.6 as well but you have to make a change in z3c.saconfig or use the repository latest version to make it work.
- A properly configured relational database. I have tested this against Oracle 10g (10.2) and Postgresql 8.3
- At a minimum, a basic grasp of Grok, SQLAlchemy and obviously Python

2.3.2 Installation

How you install Grok is your preference, there are many good documents on how to do this. This how-to assumes you have installed Grok via virtualenv and grokproject.

Adding megrok.rdb, SQLA and the support libraries they need is very simple, just add megrok.rdb to setup.py in your project and let buildout handle the rest:

```
setup.py
setup(
      * snip *
      install_requires=['setuptools',
                        'grok',
                        'grokui.admin',
                        'z3c.testsetup',
                        'grokcore.startup',
                        # Add extra requirements here
                    'megrok.rdb',
                       ],
      * snip *
      )
```

Then re-run buildout:

```
(grokenv)zope@zope3:$ ./bin/buildout
```

This should put megrok.rdb, SQLA, z3c.saconfig and any other support packages needed in the directory you specified with –eggs-dir when you ran grokproject.

2.3.3 Imports

Note: You can find another (and eerily similar :)) example of this setup code in megrok/rdb/tests/initialization.py in your eggs-dir.

To get started we need to import some packages and modules:

```
import grok
from megrok import rdb
from z3c.saconfig import (EngineFactory, GloballyScopedSession)
from z3c.saconfig.interfaces import (IEngineFactory, IScopedSession, IEngineCreatedEvent)
```

2.3.4 Connection string

We will be using Oracle as our database for this how-to, however it should cover pretty much any DB SQLA supports:

DSN = 'oracle://username:password@database_name' My DSN is a bit different than others as it relies on Oracle TNS for some of its information, specifically the host.

Create your DSN by following the guidelines from SQLA:

```
"The URL is a string in the form dialect://user:password@host/dbname[?key=value..], where dialect
is a name such as mysql, oracle, postgresql, etc."
```

For postgres it should look like:

```
DSN = 'dialect+driver://username:password@host/database_name'
So if you use psycopg2 for your adapter, you'd have 'postgresql+psycopg2://...../database_name'
```

For more info on SQLA connections please consult the SQLA docs.

2.3.5 SQLAlchemy engines and scope

Now that we have a connection string we must setup an SQLA database engine, for this z3c.saconfig has provided us the tools:

```
engine_factory = EngineFactory(DSN, echo=True)
grok.global_utility(engine_factory, provides=IEngineFactory, direct=True)
```

Consult the z3c.saconfig code to examine the inner workings of EngineFactory, but it's main purpose is to provide an SQLA engine. Notice too that we have set echo=True, this turns on SQLA output so you can see what it is doing in the background. When you are satisfied all is well, just set it to False or remove it all together. We must also register our engine factory as a global utility, generally we are making it available globally, and more specifically so megrok.rdb, can get at it through the utility machinery.

Similarly we must also setup an SQLA session, again z3c.saconfig to the rescue:

```
scoped_session = GloballyScopedSession()
grok.global_utility(scoped_session, provides=IScopedSession, direct=True)
```

In our case the scope is global, making the session ubiquitous in our app. You are not required to to this, you are also able to scope it by site using SiteScopedSession(). Again consult the z3c.saconfig code for more specific details, particularly about scope. Moving on, we register the session as a utility to make the session available to the utility machinery scoped as we like.

2.3.6 Metadata and EngineCreatedEvents

Generally metadata is how SQLA keeps track of your classes and mappings. Here is where things can get trickier.

In most cases, you are going to only need one set of metadata. However there are special cases to consider, particularly for our example. So we will create two(2) sets of metadata.:

```
skip_create_metadata = rdb.MetaData()
create_metadata = rdb.MetaData()
```

Notice the names, the reasons for this are forthcoming...just note them for now.

Continue by setting up our events:

```
@grok.subscribe(IEngineCreatedEvent)
def create_engine_created(event):
    rdb.setupDatabase(create_metadata)

@grok.subscribe(IEngineCreatedEvent)
def skip_create_engine_created(event):
    rdb.setupDatabaseSkipCreate(skip_create_metadata)
```

Feel free to delve into how events work at your leisure, we simply need to know the basics of what's happening. Grok subscribes to (watches for) the EngineCreatedEvent, and when it happens, executes some code, in our case rdb.setupDatabase() and rdb.setupDatabaseSkipCreate(). We have two(2) events, one for each set of metadata.

So, why both sets of metadata? Why both events? For Postgres, existing views, existing tables and tables dynamically created from your classes are all handled by rdb.setupDatabase(). SQLA recognizes all reflected objects as already existing and doesn't try and create them. But with Oracle(and our example) if you reflect an existing view into some metadata and pass it through rdb.setupDatabase(), not only will SQLA reflect your views, it will also try very hard to create them as tables in your DB. This is bad. The views already exist, right? So, we need a solution, which is, hold them in a separate metadata, pass the proper metadata to rdb.setupDatabaseSkipCreate(), which skips the creation attempt, leaving you with your views reflected as you'd expect.

It's up to you how you handle reflected tables in this case. SQLA will recognize those as existing and skip the create for you. Go ahead and put them in whichever metadata you like, as either will work. Dynamically created tables, however, must be held in the metadata passed to rdb.setupDatabase() so that they are created for you.

2.3.7 The Complete Configuration

Combining all the previous code our configuration looks like this: (oracle_cfg.py):

```
import grok
from megrok import rdb
from z3c.saconfig import (EngineFactory, GloballyScopedSession)
from z3c.saconfig.interfaces import (IEngineFactory, IScopedSession, IEngineCreatedEvent)

DSN = 'oracle://username:password@database_name'

engine_factory = EngineFactory(DSN, echo=True)
grok.global_utility(engine_factory, provides=IEngineFactory, direct=True)

scoped_session = GloballyScopedSession()
grok.global_utility(scoped_session, provides=IScopedSession, direct=True)

skip_create_metadata = rdb.MetaData()
create_matadata = rdb.MetaData()
```

```
@grok.subscribe(IEngineCreatedEvent)
def create_engine_created(event):
    rdb.setupDatabase(create_matadata)

@grok.subscribe(IEngineCreatedEvent)
def skip_create_engine_created(event):
    rdb.setupDatabaseSkipCreate(skip_create_metadata)
```

2.3.8 Mapping our Classes

The heavy lifting is finished and megrok.rdb, et al, did most of it for you trust me. Now we can start developing our mapped classes.

First, we import any modules we need and our metadata:

```
from megrok import rdb

from sqlalchemy import Column, ForeignKey, Sequence
from sqlalchemy.types import Integer, String, Float
from sqlalchemy.orm import relation

from oracle_cfg import (create_matadata, skip_create_metadata)
```

Next, we create a container, you'll notice a striking similarity to grok.Container:

```
class TestSA(rdb.Container):
    pass
```

The container does what you might expect, it contains objects. Certainly, it does more than that and there are more container options than what will be presented here but we'll save that is for a future how-to. If you don't want to wait and you have to know it's full functionality now, use the source. You can find this in the eggs-dir under megrok/rdb/ in interfaces.py and components.py

Next we create our Models, these will be our mapped classes. Again these are similar to their Grok counterpart grok.Model and you can find information about them in the same place as rdb.Container. In our first Model we will describe a reflected view. The actual Oracle view name is "sales_part" it is made up of numerous columns and holds data that describes, you guessed it, parts that are available for sale.:

```
class SalesPart(rdb.Model):
    """
    Reflected view, notice the metadata used is the same as the one passed
    to rdb.setupDatabaseSkipCreate(), Don't run create_all() on this
    metadata SQLA will see this as a table that needs creation and try and
    create it in the DB, which we do not want.
    """
    rdb.metadata(skip_create_metadata)
    rdb.reflected()
    rdb.tablename('sales_part')
    rdb.tableargs(schema='app', useexisting=True)

    contract = Column('contract', String, nullable=False, primary_key=True)
    catalog_no = Column('catalog_no', String, nullable=False, primary_key=True)
```

Ok, lets examine some of the details. The first thing we see is a class definition, nothing unusual, just a normal python class that inherits from rdb.Model. Next, we see several directives or class annotations, if you are new to Grok, you will find that it is big on these. Lets look at them individually:

```
rdb.metadata(skip_create_metadata)
```

Simply put, we are telling megrok rdb, that we want to act upon the specified metadata. It's important to note the placement. If using a single metadata it is not required to define this at the model level, it could even be defined as part of your configuration (oracle_cfg.py) but since we will be exploring multiple metadata objects, this needs to be defined here and in each additional model so megrok.rdb operates on the proper metadata for each one.:

```
rdb.reflected()
```

This is telling megrok.rdb and SQLA to reflect an existing view or table.:

```
rdb.tablename('sales_part')
```

This is an optional directive, without it megrok.rdb and SQLA will look for a table/view matching the class name (salespart) but since we have an underscore in the name we use the directive to specify.:

```
rdb.tableargs(schema='app', useexisting=True)
```

Also optional depending on what you need. For Oracle if you don't connect as the schema owner, you must pass the schema value as an argument to the table, this tells SQLA where to look for the table/view and also what permissions the user you did connect with, has. In our case we are also reflecting a view here and as you will see in a moment overriding some columns. The act of overriding requires us to pass the useexisting flag to tell SQLA to use the existing columns and override any columns presented in the class itself.:

```
contract = Column('contract', String, nullable=False, primary_key=True)
catalog_no = Column('catalog_no', String, nullable=False, primary_key=True)
```

Views have no primary keys of their own. SQLA wants primary keys, so we override any columns we want as part of the primary key. As mentioned above this step requires you to pass useexisting=True to the tableargs directive.

Our second example will be a reflected table.:

```
class OutputTab(rdb.Model):
    """
    Reflected table, notice the metadata that uses the rdb.setupDatabase,
    SQLA handles the create properly because it can see this as a table
    and therefore will not try and create it. You could use either metadata
    for this case.
    """
    rdb.metadata(create_metadata)
    rdb.reflected()
    rdb.tablename('output_tab')
    rdb.tableargs(schema='app')
```

That's it, pretty simple, though there is a small difference to discuss. Notice that the useexisting flag is not there. This is a table, it has all the primary keys and other features SQLA wants so no overriding necessary, and as such, no useexisting flag required. On the flip side it isn't required to not be there either, works either way. Remember though, if for some reason you did need to override a column, you must pass it, at least for Oracle.

Our final models will be a dynamically created:

```
class MyTable(rdb.Model):
    rdb.metadata(create_metadata)
    rdb.tablename('my_table')
    rdb.tableargs(schema='app')

    id = Column('id', Integer, nullable=False, primary_key=True)
    name = Column('name', String)
    email = Column('email', String)
```

This will create the table "my_table" in the schema "app" with 3 columns, id, name and email.

What about postgres serial type? SQLA handles that for you, for Integer based primary_key columns, but if you want to use a specific sequence (and for Oracle you'd have to) you can do it this way:

```
id = Column('id', Integer, Sequence('some_id_seq'), primary_key=True)
```

Continuing on, we can add another model that references the first model:

We modify our first model like this:

```
class MyTable(rdb.Model):
    rdb.metadata(create_metadata)
    rdb.tablename('my_table')
    rdb.tableargs(schema='app')

    id = Column('id', Integer, nullable=False, primary_key=True)
    name = Column('name', String)
    email = Column('email', String)
    my_other_tables = relation('MyOtherTable', backref='my_table', collection_class='TestSA')
```

Then create the new model:

```
class MyOtherTable(rdb.Model):
    rdb.metadata(create_metadata)
    rdb.tablename('my_other_table')
    rdb.tableargs(schema='app')

    id = Column('id', Integer, nullable=False, primary_key=True)
    my_table_id = Column('my_table_id', Integer, ForeignKey('my_table.id'))
    name = Column('name', String)
    size = Column('size', Float)
```

2.3.9 Using your mapped classes

To use your mapped classes, the easiest way is through a session:

```
class TestSQLA(grok.View):
    grok.context(context_of_your_app)

    def render(self):
        session = rdb.Session()
        sp = session.query(SalesPart).all()
        msg = ''.join(['<p>%s</p>' % (o.catalog_no) for o in sp])
        return """<html><head></head><body>%s</body></html>""" % msg
```

2.3.10 The full mapping

Combining all our examples it will look like this:

```
from megrok import rdb

from sqlalchemy import Column, ForeignKey, Sequence
from sqlalchemy.types import Integer, String, Float
from sqlalchemy.orm import relation

from oracle_cfg import (create_matadata, skip_create_metadata)
```

```
class TestSA(rdb.Container):
   pass

class SalesPart(rdb.Model):
    """
    Reflected view, notice the metadata used is the same as the one passed
    to rdb.setupDatabaseSkipCreate(), Don't run create_all() on this
    metadata SQLA will see this as a table that needs creation and try and
    create it in the DB, which we do not want.
    """
    rdb.metadata(skip_create_metadata)
    rdb.reflected()
    rdb.tablename('sales_part')
    rdb.tableargs(schema='app', useexisting=True)

    contract = Column('contract', String, nullable=False, primary_key=True)
    catalog_no = Column('catalog_no', String, nullable=False, primary_key=True)

class OutputTab(rdb.Model):
    """
    Reflected table, notice the metadata that uses the rdb.setupDatabase,
    SQLA handles the create properly because it can see this as a table and
    therefore will not try and create it. You could use either metadata for
    this case.
    """
    rdb.metadata(create_metadata)
    rdb.reflected()
    rdb.tablename('output_tab')
    rdb.tableargs(schema='app')

class MyTable(rdb.Model):
    rdb.metadata(create_metadata)
    rdb.tablename('my_table')
    rdb.tableargs(schema='app')

    id = Column('id', Integer, nullable=False, primary_key=True)
    name = Column('name', String)
    email = Column('email', String)
    my_other_tables = relation('MyOtherTable', backref='my_table', collection_class='TestSA')

class MyOtherTable(rdb.Model):
    rdb.metadata(create_metadata)
    rdb.tablename('my_other_table')
    rdb.tableargs(schema='app')

    id = Column('id', Integer, nullable=False, primary_key=True)
    my_table_id = Column('my_table_id', Integer, ForeignKey('my_table.id'))
    name = Column('name', String)
    size = Column('size', Float)

class TestSQLA(grok.View):
    grok.context(context_of_your_app)

    def render(self):
        session = rdb.Session()
        sp = session.query(SalesPart).all()
        msg = ''.join(['<p>%s</p>' % (o.catalog_no) for o in sp])
        return """<html><head></head><body>%s</body></html>""" % msg
```

2.3.11 Further information

SQLAlchemy Documentation Page: http://www.sqlalchemy.org/docs/

Python Package Index: http://pypi.python.org/pypi/

Python Documentation: http://python.org/doc/

PostgreSQL: http://www.postgresql.org/

Oracle: http://www.oracle.com/

2.4 Grok ORM with Storm

Author Christian Klinger (goschtl)

Version unkown

RDBMS/ORM and Zope/Grok doesn't fit together?
With this tutorial I will show you how easy it is,
to make a simple CRUD Application with Grok and Storm.

2.4.1 Prerequisities

First we setup our database. As a lightweight solution I use sqlite. So please make sure that sqlite is installed in your system. I think sqlite should be included in every *nix distribution. After installing sqlite we set up our database for our little application.

Here are the commands for an initial setup of our database:

```
note: here we create the database in /tmp/contact.db

chrissi$ sqlite3 /tmp/contact.db
SQLite version 3.1.3
Enter ".help" for instructions
sqlite> CREATE TABLE Contacts (id INTEGER PRIMARY KEY AUTOINCREMENT, name VARCHAR(200), city VARCHAR
sqlite> .exit
```

After setting up the database let's build a new project with grokproject.:

```
chrissi$ bin/grokproject contacts
```

Fill out the asked questions from grokproject. I'm sure you have done this before a dozen times. So it should not be a problem.

After setting up our grokproject environment we have to install a stormcontainer, which acts like a normal grok.Container. The main difference is of course that stormcontainer gets its data from an RDBMS via the Storm ORM API. Unfortunately I don't have a release on pypi so we have to install it manually. Let's install the stromcontainer:

First change to our grokproject package contacts:

```
chrissi$ cd contacts/
```

In this directory we have to checkout the stromcontainer:

```
chrissi$ svn checkout http://stormcontainer.googlecode.com/svn/trunk/ stormcontainer
A    stormcontainer/bootstrap.py
A    stormcontainer/buildout.cfg
A    stormcontainer/setup.py
A    stormcontainer/src
A    stormcontainer/src/stormcontainer
A    stormcontainer/src/stormcontainer/tests
A    stormcontainer/src/stormcontainer/tests/stormcontainer.txt
A    stormcontainer/src/stormcontainer/tests/__init__.py
A    stormcontainer/src/stormcontainer/tests/test_unittests.py
A    stormcontainer/src/stormcontainer/tests/test_doctests.py
A    stormcontainer/src/stormcontainer/__init__.py
A    stormcontainer/src/stormcontainer/utils.py
A    stormcontainer/src/stormcontainer/interfaces.py
A    stormcontainer/src/stormcontainer/components.py
A    stormcontainer/.installed.cfg
```

After checkout we have to install the package stormcontainer to our python's site-packages directory. This works with the command:

```
python2.4 setup.py develop
```

OK congratulations. Now we can start to developing our contacts application.

2.4.2 Step by step

Let's change back to our contacts directory to define the interface for our contact application.:

```
../src/contacts/
```

I will walk through the package structure of contacts to give you an impression of what is required to let Grok work with Storm.:

```
configure.zcml

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

  <!-- Include the storm.zope in our application -->
  <include package="storm.zope"/>
  <include package="storm.zope" file="meta.zcml"/>

  <!-- Here is our Store this is a utility which holds the connection to DB -->
  <store name="contact" uri="sqlite:/tmp/contact.db"/>

</configure>
```

We have to include the package storm.zope. And to configure a store which is responsible for connecting the database. Take a look at the uri in the store, this points to our contact database which we have created.:

```
interfaces.py

from zope.interface import Interface
from zope.schema import Int, TextLine
```

```
class IContact(Interface):
    """ Interface for Contacts """

    id = Int(title=u"id",
                  description=u"The id of our contact",
                  readonly=True)

    name = TextLine(title=u"Name",
                    description=u"The Name of our contact",
                    required=True)

    city = TextLine(title=u"City",
                    description=u"The City of our contact",
                    required=True)
```

It's a normal interface. No ORM related dependencies.:

contact.py

```
import grok
from storm.locals import *
from interfaces import IContact

class Contact(grok.Model, Storm):
    grok.implements(IContact)
    __storm_table__ = "Contacts"  # This is the corresponding table in our DB

    id = Int(primary=True) # Here we give our attributes an "Storm" Datatype
    name = Unicode()
    city = Unicode()

class Edit(grok.EditForm):
   grok.context(Contact)
   form_fields = grok.AutoFields(IContact)
```

There is also no big deal in our model. We have to add a __storm_table__ attribute to our grok.Model this attribute reflects the corresponding table in our database. We have to give our class attributes storm datatypes.:

app.py

```
import grok
from interfaces import IContact
from stormcontainer import StormContainer
from contact import Contact

class contacts(StormContainer, grok.Application, grok.Container):
    """ this application inherits from StormContainer too """
    def __init__(self):
        super(contacts, self).__init__()
        self.setClassName('contacts.contact.Contact')
        self.setStoreUtilityName('contact')

class Index(grok.View):

    def getContacts(self):
        """ Return all Contacts of our StormContainer.
            We can use the normal container API (items, keys ...)"""
        rc=[]
        for obj in self.context.items():
```

```
            d={'uid': obj[0], 'id': obj[1].id, 'city': obj[1].city, 'name': obj[1].name}
            rc.append(d)
        return rc

class CreateContact(grok.AddForm):
    form_fields = grok.AutoFields(IContact)

    @grok.action('Create')
    def create(self, **kw):
        context = self.context
        c = Contact()
        self.applyData(c, **kw)
        context['id']= c
        self.redirect(self.url(self.context))
```

Here are the greatest changes to a normal grok application. Our application has to inherit from StormContainer, grok.Application and grok.Container. In the __init__ method of our application we set up two import parameters of our application:

```
self.setClassName('contacts.contact.Contact') --> This is our model. --> contact.py
self.setStoreUtilityName('contact') --> This is the store utility name.  --> configure.zcml
```

The method getContacts use the normal container API to get the results out of the database through the Storm API. The CreateContact grok.AddForm should also be standard.:

```
app_templates/index.pt

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

  <a href="createcontact"> Add new Contact </a>

  <table>
   <tr>
    <th> id </th>
    <th> name </th>
    <th> city </th>
    <th>  </th>
   </tr>
   <tr tal:repeat="person view/getContacts">
     <td tal:content="person/id"></td>
     <td tal:content="person/name"></td>
     <td tal:content="person/city"></td>
     <td> <a href="#" tal:attributes="href string: ${person/uid}/edit"> edit </a></td>
   </tr>
  </table>

</body>
</html>
```

No big deal here we display the results in a nice table.

Now we can start our application:

```
zopectl fg
```

Now place your browser to localhost:8080 add an contact app.

Sometimes i got this error after adding my app:

```
ValueError: database parameter must be string or APSW Connection object
```

Then we have to patch this file:

```
lib/python2.4/site-packages/storm-0.11-py2.4.egg/storm/databases/sqlite.py - on line 173
```

Just make self._filename a string:

```
raw_connection = sqlite.connect(str(self._filename),
```

Now play a bit with adding and editing an contact. You can always look with sqlite in your contact database to see what happens.

2.4.3 Further information

Visit the storm page to get more information about storm.

2.5 How I Got Grok Talking To CAS

Author Brandon Craig Rhodes

Version unkown

One user's experience connecting his Grok application to a CAS authentication web server, which he probably did the wrong way around, but which he's sharing because at least it worked.

Might be outdated! This document hasn't been reviewed for Grok 1.0 and may be outdated. If you would like to review the document, please read this post.

2.5.1 Purpose

Our campus uses a CAS server to provide single-sign-on across web applications for our users. How can a Grok application send its users to a CAS server to make them log in?

Well, knowing almost nothing about Grok or Zope, I had to put together something simple fairly quickly. In particular, this solution does not integrate with the standard and pluggable Zope authentication mechanism; instead, it does its own thing. Perhaps wiser and more experienced Zope folks can write another HOWTO explaining how to do this right! But until then, people were asking for something — anything — that would give them information with which they could get started.

2.5.2 The Three Steps

There were three steps necessary to get Grok working with CAS. They each wound up getting their own .py file the way I did it.

- auth.py

This file contains a primitive method for recording, in a user's cookie-based session, which CAS user they've authenticated themselves as (if any), and for doling that information back out when various parts of Zope call for an IAuthentication local utility.

- login.py

This file, and its accompanying template named login_templates/login.pt, creates a very simple (and, to be frank, fairly ugly) login screen. The screen consists of a single button that sends you to Webauth.

- app.py

Finally, you'll need to add one or two statements to your main app.py file in order to register the above-mentioned adapter as a local utility.

Wow, that really sounds clunky, doesn't it? And it occurs to me that, the way I've done it, the login screen doesn't remember where you were when you asked to log in, so once you're logged in you've got to navigate back to the page you came from all by yourself. I guess we'd better improve this scheme soon.

2.5.3 Remembering Who Has Authenticated

How can Grok remember who is authenticated to your site? The most convenient way to store this information is through a wonderful mechanism Zope 3 provides called ISession. The magic of this adapter is that Grok, without even being asked, takes it upon itself to mark with a cookie every user that visits your site, and then keep up with them as they move from page to page. At any time you can invoke the ISession utility in order to access a bundle of behind-the-scenes information that Grok will remember for you between the user's page visits.

And so, here is my auth.py file. It provides one or two simple functions for getting and setting a username in the user's session where they'll be safely stored, and then an IAuthentication utility that can spring into action when any part of the frameworks wants to know who is connected while the user is browsing inside of your Grok application:

```
import grok
from zope.session.interfaces import ISession
from zope.app.security.interfaces import IAuthentication
from zope.app.security.principalregistry import Principal

def get_auth_data(request):
    return ISession(request)['MyGrokApp.auth']

def get_user(request):
    return get_auth_data(request).get('user', None)

def set_user(request, user):
    get_auth_data(request)['user'] = user

class MyAuthentication(grok.LocalUtility):
    grok.implements(IAuthentication)
    grok.provides(IAuthentication)

    def unauthenticatedPrincipal(self):
        """We implement no unauthenticated principal of our own."""

    def authenticate(self, request):
        user = get_user(request)
        if user:
            title = 'User ' + user
            description = 'The user ' + user
            return Principal(user, title, description, 'aaaaa', 'bbbbb')
```

You can ignore that aaaaa and bbbbb stuff in that last Principal constructor, or put in something of your own devising. They're nonsense strings that I added because I never expected to use those last two values that you have to provide with a Principal — they're called the login and the pw — but if they ever pop up somewhere, hopefully I'll recognize those strings and know to come back here to set them to something more reasonable!

2.5.4 The Actual CAS Login Part

That previous section of this HOWTO might disappear at any time, because someone wiser will doubtless come along and point out some much easier way of keeping up with the username of who's logged in; Zope probably has one or more such ways already built in that I just didn't run across while browsing and in a hurry.

But this next bit is more important, because knowing how to "talk CAS" is the key to what we're doing here that's new. Fortunately CAS is a very easy protocol, and a complete implementation of a login page can look exactly like this:

```
import grok
from urllib import urlopen, urlencode
from mywebapp.app import MyWebApp
from mywebapp.auth import get_user, set_user

class Login(grok.View):
    grok.context(MyWebApp)
    def update(self, redirect=None, ticket=None, logout=None):
        if logout:
            set_user(self.request, None)
        elif ticket:
            data = urlencode({ 'service': self.url(), 'ticket': ticket })
            socket = urlopen('https://your.cas.server/validate', data)
            parts = socket.read().split()
            if parts and parts[0] == 'yes':
                set_user(self.request, parts[1])
                self.redirect(self.url(grok.getSite()))
        elif redirect:
            data = urlencode({ 'service': self.url() })
            self.redirect('https://your.cas.server/login?' + data)
        self.user = get_user(self.request)
```

This is very straightforward! If the user arrives with the logout parameter set, then we wipe out our knowledge of who they are.

If, instead, they look like they're returning from having just logged in to CAS and have a ticket purporting to prove their identity to us, then we ask CAS whether it will vouch that the ticket is valid; if so, then we set the current user to the one returned during ticket validation and then redirect the user back to the home page. (This is where one big improvement could take place: if the redirection was back to where the user was before they logged in.)

Else, if they have shown up with redirect set, then this means they've asked to go log in to CAS, so we redirect them there.

And if all else fails, we just show them the login page.

"But wait! What does the login page look like?!" I hear you cry. It just so happens that I have it right here (it lives, per the usual Grok conventions, in login_templates/login.pt):

```
<html>
<body>

<h1>Login Page</h1>

<p tal:condition="view/user">
 You are already logged in.<br>
 Your name is <b tal:content="view/user">username</b>.
 <form tal:attributes="action python:view.url()" method="GET">
  <input type="hidden" name="logout" value="yes" /><br />
  <input type="submit" value="Log out" />
 </form>
```

```
</p>
<p tal:condition="not: view/user">
 You are currently <b>not logged in</b>.<br>
 Press the button below to log in using CAS.<br>
 <form tal:attributes="action python:view.url()" method="GET">
  <input type="hidden" name="redirect" value="yes" /><br />
  <input type="submit" value="Log in" />
 </form>
</p>

</body></html>
```

You will want a prettier one, I have no doubt, but your logic should be the same as that shown here. If the user is not logged in, let them know you're about to send them somewhere else to check their username and password, and then provide a button that will do it. Otherwise, tell them who they are and offer them a convenient logout button.

2.5.5 Turning On The Authentication

If you install the above, then a login page will appear and users who do log in will successfully get the internal variable set that our auth.py uses to remember their username.

But, that's not enough for your application to really know who the user is! To accomplish that last step, we need to somehow deliver the user's username every time the innards of Grok or Zope tries to examine request.principal.id on a request object. (You can usually get to the request from any view through simply asking for view.request.)

And that's why we created that odd little IAuthentication utility in auth.py. To activate it, you need to import it into your main app.py and then tell your application object to make it active as a local utility:

```python
import grok
from mywebapp.auth import MyAuthentication

...

class MyWebApp(grok.Application, grok.Container):
    grok.local_utility(MyAuthentication)

...
```

And with that magic in place, you should find that users that you direct to the login page should indeed be able to log in and then be recognized by your app. Good luck.

Note that, because we've provided MyAuthentication only as a local utility, the user won't be authenticated if he moves outside of the URL of your particular Grok app that you've done this to!

2.6 Navigating To Transient Objects Tutorial

Author unkown

Version unkown

Thanks to the magic of this database, your web apps can create Python objects that are automatically saved to disk and are available every time your application runs. In particular, every grok.Model and grok.Container object you generate can be written safely to your application's Data.fs file. But sometimes you need to create objects that do not persist in the ZODB, wonderful though it is.

2.6.1 Introduction

Why instantiate objects from external data sources?

If you have already read the Grok tutorial, you are familiar with the fact that behind Grok lives a wonderful object database called the Zope Object Database, or the ZODB for short. Thanks to the magic of this database, your web apps can create Python objects that are automatically saved to disk and are available every time your application runs. In particular, every grok.Model and grok.Container object you generate can be written safely to your application's Data.fs file.

But sometimes you need to create objects that do not persist in the ZODB, wonderful though it is. Sometimes these will be objects you design yourself but have no need to save once you are done displaying them, like summary statistics or search results. On other occasions, you will find yourself using libraries or packages written by other people and will need to present the objects they return in your Grok app — whether those objects are LDAP entries, file system directory listings, or rows from a relational database.

For the purpose of this tutorial, we are going to call all such objects transient objects. This highlights the fact that, from the point of view of Grok, they are going to be instantiated on-the-fly as a user visits them. Of course, they might (or might not) persist in some other application, like a file system or LDAP server or relational database! But as far as Grok can tell, they are being created the moment before they are used and then, very often, pass right back out of existence — and out of your application's memory — once Grok is finished composing the response.

To try out the examples in this tutorial, start a Grok project named TransientApp and edit the app.py and other files that you are instructed to create or edit.

2.6.2 Choosing a method

In this tutorial, we introduce four methods for creating an object which you need to present on the web:

- Creating it in your View's update(), using no external data.
- Creating it in your View's update(), using URL or form data.
- Creating it in a Traverser that gets called for certain URLs.
- Creating it in a Container that gets called for certain URLs.

To choose among these methods, the big question you need to ask yourself is whether the object you are planning to display is one that will live at its own particular URL or not. There are three basic relationships we can imagine between an object on a web page and the URL of the page itself.

The simplest case, which is supported by the first method listed above, is when you need to create an object during the rendering of a page that already exists in your application. An example would be decorating the bottom of your front page with a random quotation selected by instantiating a RandomQuotation class you have written, so that each time a user reloads the page they see a different quote. None of the quotations would thereby have URLs of their own; there would, in fact, be no way for the user to demand that a particular quotation be displayed; and the user could not force the site to display again a quote from Bertrand Russell that they remember enjoying yesterday but have forgotten. Such objects can simply be instantiated in the update() method of your View, and this technique will be our first example below.

The situation is only slightly more complex when you need to use form parameters the user has submitted to tailor the object you are creating. This is very common when supporting searching of a database: the user enters some search terms, and the application needs to instantiate an object — maybe a SearchResult or a DatabaseQuery — using those user-provided search terms, so that the page template can loop across and display the results. The second method listed above is best for this; since the form parameters are available in the update() method, you are free to use them when creating the result object. This will be the technique illustrated in our second example below.

Finally, the really interesting case is when an object actually gets its own URL. You are probably already familiar with several kinds of object which have their own URLs on the Web — such as books on Amazon.com, photographs

on Flickr, and people on Facebook, all of which live at their own URL. Each web site has a particular scheme which associates a URL with the object it names or identifies. You can probably guess, for example, just by looking at them, which object is named by each of the following three Amazon URLs:

- http://www.amazon.com/Web-Component-Development-Zope-3/dp/3540223592
- http://www.amazon.com/Harry-Potter-Deathly-Hallows-Book/dp/0545010225
- http://www.amazon.com/J-R-R-Tolkien-Boxed-Hobbit-Rings/dp/0345340426

The Grok framework, of course, already supports URL traversal for persistent objects in the ZODB; if you create a Container named polygons that contains two objects named triangle and square, then your Grok site will already support URLs like:

- http://yoursite.com/app/polygons/triangle
- http://yoursite.com/app/polygons/square

But the point of this tutorial, of course, is how you can support URL traversal for objects which are not persistent, which you will create on-the-fly once someone looks up their URL. And the answer is that, to support such objects, you will choose between the last two methods listed above: you will either create a custom Traverser, or actually define your own kind of Container, that knows how to find and instantiate the object the URL is naming. These two techniques are described last in this tutorial, because they involve the most code.

But before starting our first example, we need to define an object that we want to display. We want to avoid choosing an obvious example, like an object whose data is loaded from a database, because then this tutorial would have to teach database programming too! Plus, you would have to set up a database just to try the examples. Instead, we need an object rich enough to support interesting attributes and navigation, but simple enough that we will not have to reach outside of Python to instantiate it.

2.6.3 Our Topic: The Natural Numbers

To make this tutorial simple, we will build a small web site that lets the user visit what some people call the natural numbers: the integers beginning with 1 and continuing with 2, 3, 4, and so forth. We will define a Natural class which knows a few simple things about each number - like which number comes before it, which comes after it, and what its prime factors are.

We should start by writing a test suite for our Natural class. Not only is writing tests before code an excellent programming practice that forces you to think through how your new class should behave, but it will make this tutorial easier to understand. When you are further down in the tutorial, and want to remember something about the Natural class, you may find yourself re-reading the tests instead of the code as the fastest way to remember how the class behaves!

The reason this test will be so informative is that it is a Python "doctest", which intersperses normal text with the example Python code. Create a file in your Grok instance named src/transient/natural.txt and give it the following contents:

2.6.4 A Simple Implementation of Natural Numbers

The "natural" module of this application provides a simple class for representing any postive integer, named "Natural".:

```
>>> from transient.natural import Natural
```

To instantiate it, provide a Python integer to its constructor:

```
>>> three = Natural(3)
>>> print 'This object is known to Python as a "%r".' % three
This object is known to Python as a "Natural(3)".
```

```
>>> print 'The number of the counting shall be %s.' % three
The number of the counting shall be 3.
```

You will find that there are very many natural numbers available; to help you navigate among them all, each of them offers a "previous" and "next" attribute to help you find the ones adjacent to it:

```
>>> print 'Previous: %r  Next: %r' % (three.previous, three.next)
Previous: Natural(2)  Next: Natural(4)
```

Since we define the set of "natural numbers" as beginning with 1, you will find that the number 1 lacks a "previous" attribute:

```
>>> hasattr(three, 'previous')
True
>>> one = Natural(1)
>>> hasattr(one, 'previous')
False
>>> one.previous
Traceback (most recent call last):
 ...
AttributeError: There is no natural number less than 1.
```

You can also ask a number to tell you which prime factors must be multiplied together to produce it. The number 1 will return no factors:

```
>>> one.factors
[]
```

Prime numbers will return only themselves as factors:

```
>>> print Natural(2).factors, Natural(11).factors, Natural(251).factors
[Natural(2)] [Natural(11)] [Natural(251)]
```

Compound numbers return several factors, sorted in increasing order, and each appearing the correct number of times:

```
>>> print Natural(4).factors
[Natural(2), Natural(2)]
>>> print Natural(12).factors
[Natural(2), Natural(2), Natural(3)]
>>> print Natural(2310).factors
[Natural(2), Natural(3), Natural(5), Natural(7), Natural(11)]
```

Each natural number can also simply return a boolean value indicating whether it is prime, and whether it is composite:

```
>>> print Natural(6).is_prime, Natural(6).is_composite
False True
>>> print Natural(7).is_prime, Natural(7).is_composite
True False
```

Next, we need to tell Grok about this doctest. Create a file in your instance named src/transient/tests.py that looks like:

```
import unittest
from doctest import DocFileSuite

def test_suite():
    return unittest.TestSuite([ DocFileSuite('natural.txt') ])
```

You should now find that running ./bin/test inside of your instance now results in a whole series of test failures. This is wonderful and means that everything is working! At this point Grok is able to find your doctest, successfully execute it, and correctly report (if you examine the first error message) that you have not yet provided a Natural class that could make the doctest able to succeed.

2.6.5 The Natural class implementation

Now we merely have to provide an implementation for our Natural class. Create a file src/transient/natural.py under your Grok instance and give it the contents:

```
class Natural(object):
    """A natural number, here defined as an integer greater than zero."""

    def __init__(self, n):
        self.n = abs(int(n)) or 1

    def __str__(self):
        return '%d' % self.n

    def __repr__(self):
        return 'Natural(%d)' % self.n

    @property
    def previous(self):
        if self.n < 2:
            raise AttributeError('There is no natural number less than 1.')
        return Natural(self.n - 1)

    @property
    def next(self):
        return Natural(self.n + 1)

    @property
    def factors(self):
        if not hasattr(self, '_factors'):  # compute factors only once!
            n, i = self.n, 2
            self._factors = []
            while i <= n:
                while n % i == 0:          # while n is divisible by i
                    self._factors.append(Natural(i))
                    n /= i
                i += 1
        return self._factors

    @property
    def is_prime(self):
        return len(self.factors) < 2

    @property
    def is_composite(self):
        return len(self.factors) > 1
```

If you try running ./bin/test again after creating this file, you should find that the entire natural.txt docfile now runs correctly!

I hope that if you are new to Python, you are not too confused by the code above, which uses @property which may not have been covered in the Python tutorial. But I prefer to show you "real Python" like this, that reflects how people actually use the language, rather than artifically simple code that hides from you the best ways to use Python. Note that it is not necessary to understand natural.py to enjoy the rest of this tutorial! Everything we do from this point on will involve building a framework to use this object on the web; we will be doing no further development on the class itself. So all you actually need to understand is how a Natural behaves, which was entirely explained in the doctest.

Note that the Natural class knows nothing about Grok! This is an important feature of the whole Zope 3 framework, that bears frequent repeating: objects are supposed to be simple, and not have to know that they are being presented

on the web. You should be able to grab objects created anywhere, from any old library of useful functions you happen to download, and suit them up to be displayed and manipulated with a browser. And the Natural class is exactly like that: it has no idea that we are about to build a framework around it that will soon be publishing it on the web.

2.6.6 Having Your View Directly Instantiate An Object

We now reach the first of our four techniques!

The simplest way to create a transient object for display on the web involves a technique you may remember from the main Grok tutorial: providing an update() method on your View that creates the object you need and saves it as an attribute of the View. As a simple example, create an app.py file with these contents:

```
import grok
from transient.natural import Natural

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

class Index(grok.View):
    def update(self):
        self.num = Natural(126)
```

Do you see what will happen? Right before Grok renders your View to answer a web request, Grok will call its update() method, and your View will gain an attribute named num whose value is a new instance of the Natural class. This attribute can then be referenced from the page template corresponding to your view! Let use write a small page template that accesses the new object. Try creating an /app_templates/index.pt file that looks like:

```
<html><body>
 <p>
  Behold the number <b tal:content="view/num">x</b>!
  <span tal:condition="view/num/is_prime">It is prime.</span>
  <span tal:condition="view/num/is_composite">Its prime factors are:</span>
 </p>
 <ul tal:condition="view/num/factors">
  <li tal:repeat="factor view/num/factors">
   <b tal:content="factor">f</b>
  </li>
 </ul>
</body></html>
```

If you now run your instance and view the main page of your application, your browser should show you something like:

Behold the number 126! It has several prime factors:

- 2
- 3
- 3
- 7

You should remember, when creating an object through an update() method, that a new object gets created every time your page is viewed! This is hard to see with the above example, of course, because no matter how many times you hit “reload” on your web browser you will still see the same number. So adjust your app.py file so that it now looks like this:

```
import grok, random
from transient.natural import Natural

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

class Index(grok.View):
    def update(self):
        self.num = Natural(random.randint(1,1000))
```

Re-run your application and hit “reload” several times; each time you should see a different number.

The most important thing to realize when using this method is that this Natural object is not the object that Grok is wrapping with the View for display! The object actually selected by the URL in this example is your TransientApp application object itself; it is this application object which is the context of the View. The Natural object we are creating is nothing more than an incidental attribute of the View; it neither has its own URL, nor a View of its own to display it.

2.6.7 Creating Objects Based on Form Input

What if we wanted the user to be able to designate which Natural object was instantiated for display on this web page?

This is a very common need when implementing things like a database search form, where the user’s search terms need to be provided as inputs to the object that will return the search results.

The answer is given in the main Grok tutorial: if you can write your update() method so that it takes keyword parameters, they will be filled in with any form parameters the user provides. Rewrite your app.py to look like:

```
import grok, random
from transient.natural import Natural

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

class Index(grok.View):
    def update(self, n=None):
        self.badnum = self.num = None
        if n:
            try:
                self.num = Natural(int(n))
            except:
                self.badnum = n
```

And make your app_templates/index.pt look like:

```
<html><body>
 <p tal:condition="view/badnum">This does not look like a natural number:
  &ldquo;<b tal:content="view/badnum">string</b>&rdquo;
 </p>
 <p tal:condition="view/num">
  You asked about the number <b tal:content="view/num">x</b>!<br/>
  <span tal:condition="view/num/is_prime">It is prime.</span>
  <span tal:condition="view/num/is_composite">Its prime factors are:
   <span tal:repeat="factor view/num/factors">
    <b tal:content="factor">f</b
    ><span tal:condition="not:repeat/factor/end">,</span>
   </span>
  </span>
```

```
 </p>
 <form tal:attributes="action python:view.url()" method="GET">
  Choose a number: <input type="text" name="n" value="" />
  <input type="submit" value="Go" />
 </form>
</body></html>
```

This time, when you restart your Grok instance and look at your application front page, you will see a form asking for a number:

```
Choose a number: __________  [Go]
```

Enter a positive integer and submit the form (try to choose something with less than seven digits to keep the search for prime factors short), and you will see something like:

```
You asked about the number 48382!
Its prime factors are: 2, 17, 1423
Choose a number: __________  [Go]
```

And if you examine the URL to which the form has delivered you, you will see that the number you have selected is part of the URL's query section:

```
http://localhost:8080/app/index?n=48382
```

So none of these numbers get their own URL; they all live on the page /app/index and have to be selected by submitting a query to that one page.

2.6.8 Custom Traversers

But what about situations where you want each of your transient objects to have its own URL on your site? The answer is that you can create grok.Traverser objects that, when the user enters a URL and Grok tries to find the object which the URL names, intercept those requests and return objects of your own design instead.

For our example application app, let's make each Natural object live at a URL like:

```
http://localhost:8080/app/natural/496
```

There is nothing magic about the fact that this URL has three parts, by the way — the three parts being the application name "app", the word "natural", and finally the name of the integer "496". You should easily be able to figure out how to adapt the example application below either to the situation where you want all the objects to live at your application root (which would make the URLs look like /app/496), or where you want URLs to go several levels deeper (like if you wanted /app/numbers/naturals/496).

The basic rule is that for each slash-separated URL component (like "natural" or "496") that does not actually name an object in the ZODB, you have to provide a grok.Traverser. Make the grok.context of the Traverser the object that lives at the previous URL component, and give your Traverser a traverse() method that takes as its argument the next name in the URL and returns the object itself. If the name submitted to your traverser does not name an object, simply return None; this is very easy to do, since None is the default return value of a Python function that ends without a return statement.

So place the following inside your app.py file:

```
import grok
from transient.natural import Natural

class TransientApp(grok.Application, grok.Container):
    pass
```

```
class BaseTraverser(grok.Traverser):
    grok.context(TransientApp)
    def traverse(self, name):
        if name == 'natural':
            return NaturalDir()

class NaturalDir(object):
    pass

class NaturalTraverser(grok.Traverser):
    grok.context(NaturalDir)
    def traverse(self, name):
        if name.isdigit() and int(name) > 0:
            return Natural(int(name))

class NaturalIndex(grok.View):
    grok.context(Natural)
    grok.name('index.html')
```

And you will only need one template to go with this file, which you should place in app_templates/naturalindex.pt:

```
<html><body>
 This is the number <b tal:content="context">x</b>!<br/>
  <span tal:condition="context/is_prime">It is prime.</span>
  <span tal:condition="context/is_composite">Its prime factors are:
   <span tal:repeat="factor context/factors">
    <b tal:content="factor">f</b
    ><span tal:condition="not:repeat/factor/end">,</span>
   </span>
  </span><br>
</body></html>
```

Now, if you view the URL /app/natural/496 on your test server, you should see:

```
This is the number 496!
Its prime factors are: 2, 2, 2, 2, 31
```

Note that there is no view name after the URL. That's because we chose to name our View index.html, which is the default view name in Zope 3. (With grok.Model and grok.Container objects, by contrast, the default view selected if none is named is simply index without the .html at the end.) You can always name the view explicitly, though, so you will find that you can also view the number 496 at:

```
http://kenaniah.ten22:8080/app/natural/496/index.html
```

It's important to realize this because, if you need to add more views to a transient object, you of course will have to add them with other names — and to see the information in those other views, users (or the links they use) will have to name the views explicitly.

2.6.9 Two final notes

In order to make this example brief, the application above does not support either the user navigating simply to /app, nor will it allow them to view /app/natural, because we have provided neither our TransientApp application object nor the NaturalDir stepping-stone with grok.View objects that could let them be displayed. You will almost always, of course, want to provide a welcoming page for the top level of your application; but it's up to you whether you think it makes sense for users to be able to visit the intermediate /app/natural URL or not. If not, then follow the example above and simply do not provide a view, and everything else will work just fine.

In order to provide symmetry in the example above, neither the TransientApp object nor the NaturalDir object knows how to send users to the next objects below them. Instead, they are both provided with Traversers. It turns out, I finally admin here at the bottom of the example, that this was not necessary! Grok objects like a grok.Container or a grok.Model already have enough magic built-in that you can put a traverse() method right on the object and Grok will find it when trying to resolve a URL. This would not have helped our NaturalDir object, of course, because it's not a Grok anything; but it means that we can technically delete the first Traverser and simply declare the first class as:

```
class TransientApp(grok.Application, grok.Container):
    def traverse(self, name):
        if name == 'natural':
            return NaturalDir()
```

The reason I did not do this in the actual example above is that showing two different ways to traverse in the same example seemed a bit excessive! I preferred instead to use a single method, twice, that is universal and works everywhere, rather than by starting off with a technique that does not work for most kinds of Python object.

2.6.10 Providing Links To Other Objects

What if the object you are wrapping can return other objects to which the user might want to navigate? Imagine the possibilities: a filesystem object you are presenting on the web might be able to return the files inside of it; a genealogical application might have person objects that can return their spouse, children, or grandparents. In the example we are working on here, a Natural object can return both the previous and the next number; wouldn't it be nice to give the users links to them?

If in a page template you naively ask your Grok view for the URL of a transient object, you will be disappointed. Grok does know the URL of the object to which the user has just navigated, because, well, it's just navigated there, so adding this near the bottom of your naturalindex.pt should work just fine:

This page lives at: <b tal:content="python: view.url(context)">url
 But if you rewrite your template so that it tries asking for the URL of any other object, the result will be a minor explosion. Try adding this to your naturalindex.pt file:

Next number: <b tal:content="python: view.url(context.next)">url
 and try reloading the page. On the command line, your application will return the exception:

```
TypeError: There isn't enough context to get URL information.
```

This is probably due to a bug in setting up location information.

Do you see the problem? Because this new Natural object does not live inside of the ZopeDB, Grok cannot guess the URL at which you intend it to live. In order to provide this information, it is best to call a Zope function called locate() that marks as object as belonging inside of a particular container. To get the chance to do this magic, you'll have to avoid calling Natural.previous and Natural.next directly from your page template. Instead, provide your view with two new properties that will grab the previous and next attributes off of the Natural object which is your current context, and then perform the required modification before returning them:

```
class NaturalIndex(grok.View):

    ...

    @property
    def previous(self):
        if getattr(self.context, 'previous', None):
            n = self.context.previous
            traverser = BaseTraverser(grok.getSite(), None)
            parent = traverser.publishTraverse(None, 'natural')
            return zope.location.location.located(n, parent, str(n))
```

```
    @property
    def next(self):
        n = self.context.next
        traverser = BaseTraverser(grok.getSite(), None)
        parent = traverser.publishTraverse(None, 'natural')
        return zope.location.location.located(n, parent, str(n))
```

This forces upon your objects enough information that Zope can determine their URL — it will believe that they live inside of the object named by the URL /app/natural (or whatever other name you use in the PublishTraverse call). With the above in place, you can add these links to the bottom of your naturalindex.pt and they should work just fine:

```
<tal:if tal:condition="view/previous">
 Previous number: <a tal:attributes="href python: view.url(view.previous)"
  tal:content="view/previous">123</a><br>
</tal:if>

Next number: <a tal:attributes="href python: view.url(view.next)"
                tal:content="view/next">123</a><br>
```

This should get easier in a future version of Grok and Zope!

2.6.11 Writing Your Own Container

The above approach, using Traversers, gives Grok just enough information to let users visit your objects, and for you to assign URLs to them.

But there are several features of a normal grok.Container that are missing — there is no way for Grok to list or iterate over the objects, for example, nor can it ask whether a particular object lives in the container or not.

While taking full advantage of containers is beyond the scope of this tutorial, I ought to show you how the above would be accomplished:

```
import grok
from transient.natural import Natural
from zope.app.container.interfaces import IItemContainer
from zope.app.container.contained import Contained
import zope.location.location

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

class BaseTraverser(grok.Traverser):
    grok.context(TransientApp)
    def traverse(self, name):
        if name == 'natural':
            return NaturalBox()

class NaturalBox(Contained):
    grok.implements(IItemContainer)
    def __getitem__(self, key):
        if key.isdigit() and int(key) > 0:
            n = Natural(int(key))
            return zope.location.location.located(n, self, key)
        else:
            raise KeyError

class NaturalIndex(grok.View):
    grok.context(Natural)
```

```
    grok.name('index.html')

    @property
    def previous(self):
        if getattr(self.context, 'previous'):
            n = self.context.previous
            parent = self.context.__parent__
            return zope.location.location.located(n, parent, str(n))

    @property
    def next(self):
        n = self.context.next
        parent = self.context.__parent__
        return zope.location.location.located(n, parent, str(n))
```

Note, first, that this is almost identical to the application we built in the last section; the grok.Application, its Traverser, and the NaturalIndex are all the same — and you can leave alone the naturalindex.pt you wrote as well.

But instead of placing a Traverser between our Application and the actual objects we are delivering, we have created an actual "container" that follows a more fundamental protocol. There are a few differences in even this simple example.

A container is supposed to act like a Python dictionary, so we have overriden the Python operation __getitem__ instead of providing a traverse() method. This means that other code using the container can find objects inside of it using the container[key] Python dictionary syntax.

A Python __getitem__ method is required to raise the KeyError exception when someone tries to look up a key that does not exist in the container. It is not sufficient to merely return None, like it was in our Traverser above, because, without the exception, Python will assume that the key lookup was successful and that None is the value that was found!

Finally, before returning an object from your container, you need to call the Zope located() function to make sure the object gets marked up with information about where it lives on your site. A Grok Traverser does this for you. Again, in most circumstances I can imagine, you will be happier just using a Traverser like the third example shows, and not incurring the slight bit of extra work necessary to offer a full-fledged container. But, in case you ever find yourself wanting to use a widget or utility that needs an actual container to process, I wanted you to have this example available.

2.7 Create simple 1:2 relationship with Megrok.rdb and SqlAlchemy (over MySql)

Author BorrajaX

Version unkown

This howto describes step by step how to create a simple 1:2 relationship using Megrok.rdb and SqlAlchemy and how to split the different classes definition in different Python modules.

I was migrating our current database from an Object-Oriented model (ZopeDB) to a relational model (MySql). At a certain point, I needed to have a Parent class that has two different instances of a Child() class inside. It couldn't be a list, it needed to be two separate instances. It took me a while to have it working, so I thought someone else could benefit from my experiences.

2.7.1 Step by step

I needed to have something that would look like:

```
from Child import Child
class Parent(object):
        def __init__(self):
                self._whateverField = "Whatever"
                self.child1 = Child()
                self.child2 = Child()
```

The MySql schema (which, in other contexts is sometimes referred to as "database") was called `test2()`

In order to move this to a relational model, I needed to do the following:

1) Create a simple file to set up the connection to the database and the metadata (the object that keeps track of the tables, the mapping Object Table, etc)

Tables.py >

```
import grok
import logging
from megrok import rdb
from z3c.saconfig import EngineFactory
from z3c.saconfig.interfaces import IEngineCreatedEvent
from z3c.saconfig.interfaces import IEngineFactory
from zope import component

log = logging.getLogger(__name__)
log.setLevel(logging.DEBUG)

DSN = "mysql://root:*******@localhost/test2?charset=utf8"

engine_factory = EngineFactory(DSN, echo=True)
grok.global_utility(engine_factory, provides=IEngineFactory, direct=True)

zepMetadata = rdb.MetaData()


@grok.subscribe(IEngineCreatedEvent)
def create_engine_created(event):
        rdb.setupDatabase(zepMetadata)

component.provideHandler(create_engine_created)
```

Comments to Tables.py:

1. I must say that I was never able to automatically call the `create_engine_created function()` (it seems the event was never triggered). That's why is manually invoked from another module (Test.py)
2. If you don't have the logging package installed, substitute log.debug, log.info... by regular prints (after all, this is a very simple howto)
3. In the DNS string I substituted my root password by * characters. Make sure you have the right configuration there. Also, be aware that is strongly discouraged using the MySql root user in production. In "real life", you should access the database with a less privileged user.

2) After setting up the database connection, prepare the two classes itself (Parent and Child):

Parent.py >

```
from megrok import rdb
from sqlalchemy import Column
from sqlalchemy import and_
from sqlalchemy.orm import relationship
from sqlalchemy.types import Integer
from sqlalchemy.types import String
from mylibraries.database.tests.Child import Child
from mylibraries.database.tests.Tables import zepMetadata

class Parent(rdb.Model):
        rdb.metadata(zepMetadata)
        rdb.tablename("parents_table")
        rdb.tableargs(schema='test2', useexisting=False)

        id = Column("id", Integer, primary_key=True, nullable=False, unique=True)
        _whateverField = Column("whatever_field", String(16)) #Irrelevant

        child1 = relationship(
                "Child",
                primaryjoin=lambda: and_((Child.parent_id == Parent.id), (Child.type == "VR")),
                uselist=True,
                collection_class=list
                )

        child2 = relationship(
                "Child",
                primaryjoin=lambda: and_((Child.parent_id == Parent.id), (Child.type == "CC")),
                uselist=True,
                collection_class=list
                )

        def __init__(self):
                print "Parent __init__"
                self._whateverField = "Water"
                self.child1 = list()
                self.child2 = list()

        def addChild(self, child):
                if child.type == "VR":
                        self.child1.append(child)
                elif child.type == "CC":
                        self.child2.append(child)
```

Comments to Parent.py:

1. We are specifying the name of table where we want our Parent instances stored (for persistence) with `rdb.tablename("parents_table")()`.
2. This is not required. If it's not specified, a parent (with lowercase letters) should be created automatically.
3. The two children are going to be lists. One of them will contain children with a type "VR" and the other will contain children with a type "CC" (just because...)
4. My modules are stored in the path mylibraries/database/tests. Please, make sure you change this to your own path when importing.

5. The lambda (callable) function used in the joins to get the children is useful to get around circular dependencies. It will delay the "check" of the existance of Child and Parent classes until they are actually defined.
6. Realize that we are using the rdb.Metadata() instance from Tables.py (not creating a new instance). The metadata is what keeps track of the mappings Class Tables. It will be automatically filled on each class.

Child.py >

```
from megrok import rdb
from sqlalchemy import Column
from sqlalchemy import ForeignKey
from sqlalchemy.types import Integer
from sqlalchemy.types import String
from mylibraries.database.tests.Tables import zepMetadata

class Child(rdb.Model):
        rdb.metadata(zepMetadata)
        rdb.tablename("children_table")
        rdb.tableargs(schema='test2', useexisting=False)
        id = Column("id", Integer, primary_key = True)
        parent_id = Column("parent_id", Integer, ForeignKey("test2.parents_table.id"))
        type = Column("type", String(2))

        def __init__(self):
                self.type = "VR"
```

Comments to Child.py:

1) One of the things that took me a while to realize is that, if you are using the directive `rdb.tableargs(schema='schema_name', useexisting=False)()`, the ForeignKey needs to be invoked by using an string of the shape "schema_name.table_name.column_name". If you're not using said directive, you can simply use "table_name.column_name" (see SqlAlchemy docs1).

2. What is done in the __init__ (assigning "VR" to the type) is merely for the example. It doesn't need to be done that way.

3) This is basically all what is needed for the database definition. Now let's prepare a file where the classes will be groked, the tables will be actually created and a little test will be performed.

Test.py >

```
from grokcore.component.testing import grok_component
import logging
import mylibraries
from mylibraries.database.tests.Child import Child
from mylibraries.database.tests.Parent import Parent
import mylibraries.database.tests.Tables


log = logging.getLogger(__name__)
log.setLevel(logging.DEBUG)

def setComponents():
        """Grok all the classes related to the database"""
        grok_component("Parent", Parent)
        grok_component("Child", Child)
```

```
def createDatabase():
        mylibraries.database.tests.Tables.create_engine_created(None)

def fillASampleParent():
        parent = Parent()
        for i in range(5):
                child = Child()
                if i % 2 == 0:
                        child.type = "VR"
                else:
                        child.type = "CC"
                parent.addChild(child)
        return parent
```

To test this in our server, let's go to app.py (where the grokserver is defined) and create a new "page" for the testing.

App.py >

```
#[ . . . ]
# Lots of useful data, classes and methods
class Test(grok.View):
        grok.context(Grokserver)
        import mylibraries.database.tests.Parent
        import mylibraries.database.tests.Child
        import mylibraries.database.tests.Test
        import mylibraries.database.tests.Tables
        from megrok import rdb
        def render(self):
                session = rdb.Session()
                mylibraries.database.tests.Test.setComponents()
                mylibraries.database.tests.Test.createDatabase()
                parent_sample = mylibraries.database.tests.Test.fillASampleParent()
                session.add(parent_sample)
                session.flush()
#[ . . . ]
# Another lot of useful data, classes and methods
```

You should be able to execute that by typing in your browser: http://server_address:port/app_name/test (change server_address, port and app_name by your own values)

After this, 5 new children should have been added to the database.

2.7.2 Further information

Basic documentation for MeGrok.

Another useful HowTo.

SqlAlchemy documentation.

And thanks to all the people who answered my questions in the following threads.

In the Grok-dev mailing list (1, 2)

In the SqlAlchemy Google Group (3, 4, 5)

2.8 Understanding default values for object database backed attributes

Author Kevin Teague (kteague)

Version unkown

Using class attributes as default values in Model objects is a common idiom, understand how this works.

Imagine you want to make a very simple Shoe Model class to define how you will manage data about Shoe objects. In Grok you would typically create an interface which described the data schema of a Shoe, and create a class that inherits from *grok.Model* that implements the Shoe data schema. While it's not strictly necessary to declare the data schema of an object in Python, and it's possible to dynamically add arbitrary data attributes to a object and store those in the Zope Object Database (ZODB), it's generally good practice to be more formal with your data schemas and only deviate from pattern when supporting a specific use-case.

```
from zope.interface import Interface
from zope import schema
import grok

class IShoe(Interface):
    kind = schema.TextLine(
        title=u'Kind of Shoe',
        default=u'Sneaker',
    )

class ClassAttributeBasedShoe(grok.Model):
    grok.implements(IShoe)
    kind = u'Sneaker'

class ObjectAttributeBasedShoe(grok.Model):
    grok.implements(IShoe)

    def __init__(self, kind=u'Sneaker'):
        self.kind = kind
```

Elsewhere you would also define your Application and perhaps a separate Shoes Container for managing collections of Shoes. Then you could create and store shoes with:

```
class CreateSingleHardcodedShoeView(grok.View):
    grok.context(ShoesContainer)

    def render(self):
        new_shoe = ClassAttributeBasedShoe()
        self.context['test_shoe'] = new_shoe
        return """
        <html><h1>
        I saved a shoe of kind %s the object database!
        </h1></html>
        """ % new_shoe.kind
```

In the above example, your IShoe schema declared that shoe data objects have an attribute named *kind* which is a single line of text, and the default value is *Sneaker*. Interfaces are only abstract, formalized descriptions of an object - the IShoe interface only states that Shoe objects should supply the value *Sneaker* as a default. It's up to the Model implementation class to actually handle the details of default values. We will look at two different ways of handling default values: using class attributes and using object attributes, as well as the subtle difference between these two implementations.

2.8.1 Class attributes and default values

In Python, class objects have attributes, and instance objects of a class have attributes. There is always only a single class object, but often you will have many instance objects of a given class. If you are familiar with working with data in a relational database, you can think of class objects as being analgous to a table, and instance objects as being analgous to rows within that table.

For a further primer on classes and objects, see the section on Classes in the Python Tutorial.

Normally, only instance objects are stored in an object database, class objects are only stored in your Python code. For example:

```
shoe = ClassAttributeBasedShoe()
return "<h1>The kind of shoe is %s</h1>" % shoe.kind
```

How does an object instance magically acquire a kind attribute in the above example? The answer is that object attributes are shadowed by class attributes. During normal instance object attribute look-up, if the object doesn't directly provide an attribute, then the class object is checked for an attribute of that name - if the class has that attribute then it is used instead. If you are using an instance object, you can always get to the shadowed class attribute value by using the magic name of *__class__*. For example, try this on Python's interactive interpreter:

```
>>> class FooBarShoe(object):
...     kind = u'Foo Bar'
...
>>> foo_shoe = FooBarShoe()
>>> foo_shoe.kind
u'Foo Bar'
>>> foo_shoe.kind = u'Extremely bling Foo Bar'
>>> foo_shoe.kind
u'Extremely bling Foo Bar'
>>> foo_shoe.__class__.kind
u'Foo Bar'
```

So what does this mean when we were using the *ClassAttributeBasedShoe* implementation to provide the default kind attribute for instance objects? Class attributes are not persisted in the ZODB! If you make twenty shoes, and they all rely on the default values, and save those twenty shoes in the database, then only fact that these objects are instances of the *ClassAttributeBasedShoe* is saved. The default value is not stored to disk for any of these objects. If you change the *ClassAttributeBasedShoe* to:

```
class ClassAttributeBasedShoe(grok.Model):
    grok.implements(IShoe)
    kind = u'Work boot'
```

Then when you retrieve all of your shoe objects, the default value will be changed for all of them.

2.8.2 Object attributes and default values

Using object attributes to store data, your objects will behave similar to a relational database. If you create and store a shoe object, then the default value for the kind attribute will be written to disk. If you later change the default value, then the kind attribute for existing objects will not change:

```
class ObjectAttributeBasedShoe(grok.Model):
    grok.implements(IShoe)

    def __init__(self, kind=u'Sneaker'):
        self.kind = kind
```

2.8.3 Which is better?

The answer is, "it depends."

Using Class attributes for default values is more efficient than object attributes, since less data is written to disk. Creating an implementation using Class attributes is also more concise and slightly more readable. However, care is necessary with Class attributes, since changing this value will change that value for **all** objects derived from that class. Sometimes though, this is exactly what you want, if you are using class attributes for default values and want to update all existing objects to that default, it's not necessary to write an migration code to update objects already stored in the ZODB.

Using object attributes for default values is going to increase the amount of data written to disk. It will make the class definition slighlty longer. However, if your goal is to permanently store the default value to disk for each object, then this is exactly what you want. For example, if you have an object which stores the version number of an program used to perform a data analysis run, then you definitely want to use object attributes, since if you used a default stored as a class attribute, older data analysis results performed with an older version number would be automagically updated to the latest value stored in the class attribute!

You can also mix-and-match class attributes and object attributes to support any specific requirements of your implementation. For example, if you like declaring defaults as class attributes, but want to copy that value into an object attribute during initialization you might write:

```
class ObjectAndClassShoe(grok.Model):
    grok.implements(IShoe)
    kind = u'Mountaineering boot'

    def __init__(self, kind=None):
        if not kind:
            kind = self.__class__.kind
        self.kind = kind
```

CHAPTER

THREE

SECURITY AND AUTHENTICATION

Contents:

3.1 Authentication with Grok

Author Martijn Faasen (faassen)

Version unkown

This document tries to explain how to set up custom authentication against your own database (may be it ZODB, a relational database or LDAP) with Grok. It doesn't go into the details of how to query the database, but shows the bits and pieces you need to integrate with Grok.

Note that the situation is currently rather low-level for Grok and that this document is rather coarse. Contributions to the document and to the authentication situation (in the form of helpful libraries) would be very welcome!

First we're going to set up an application with custom authentication:

```
import grok

from zope.app.authentication.authentication import PluggableAuthentication
from zope.app.security.interfaces import IAuthentication

class MyApplication(grok.Application, grok.Model):
    grok.local_utility(
        PluggableAuthentication, provides=IAuthentication,
        setup=setup_authentication,
        )
```

When the application is installed, a local utility will be automatically installed into it. When this local utility is installed, the `setup_authentication` function will be called to further configure it. Let's implement that now:

```
def setup_authentication(pau):
    """Set up pluggable authentication utility.

    Sets up an IAuthenticatorPlugin and
    ICredentialsPlugin (for the authentication mechanism)
    """
    pau.credentialsPlugins = ['credentials']
    pau.authenticatorPlugins = ['users']
```

3.1.1 Session-based login

The pluggable authentication system needs at least one credentials plugin, which is responsible for extracting credentials from the user's request, and one authenticator plugin, which authenticates the credentials against an actual user (principal).

Here we've configured the pluggable authentication utility to look up an `ICredentialsPlugin` utility with the name `credentials` and a `IAuthenticatorPlugin` with the name `users`.

We need to supply these utilities next. In our example we're both going to make them global utilities so they need no special setup in the `MyApplication` object above. If you need them to store data or configuration information in the ZODB however you should create them as a local utility (and also probably provide a user interface for them). This is left as an exercise to the reader.

Our credentials plugin will use the persistent user-specific session to retrieve credentials information - it's like cookie-based login:

```
from zope.app.authentication.session import SessionCredentialsPlugin
from zope.app.authentication.interfaces import ICredentialsPlugin

class MySessionCredentialsPlugin(grok.GlobalUtility, SessionCredentialsPlugin):
    grok.provides(ICredentialsPlugin)
    grok.name('credentials')

    loginpagename = 'login'
    loginfield = 'form.login'
    passwordfield = 'form.password'
```

Session-based login needs to know about a special login page where the user fills their username and password in a form, so that it can retrieve the information to store in a session. We need to supply it with the name of the form (`login`) and the name of the form fields.

Let's set up this login page next, using the `grok.Form` mechanism:

```
from zope.interface import Interface
from zope import schema

class ILoginForm(Interface):
    login = schema.BytesLine(title=u'Username', required=True)
    password = schema.Password(title=u'Password', required=True)

class Login(grok.Form):
    grok.context(Interface)
    grok.require('zope.Public')

    form_fields = grok.Fields(ILoginForm)

    @grok.action('login')
    def handle_login(self, **data):
        self.redirect(self.request.form.get('camefrom', ''))
```

The session credentials plugin will automatically redirect the user to this login form when the credentials cannot be found in the session. It's available on all objects as it's registered for `Interface`. When the user fills in the credentials they will appear as `form.login` and `form.password` in the request where the credentials plugin can find them and store them in the session.

After submitting the form successfully the user is immediately redirected to the URL in `camefrom`. This `camefrom` variable is automatically added to request when the login form is rendered and is the original page the user tried to go to when they were redirected to this one (because a login was required first). In the `login.pt` template we need

to make sure we retrieve it so that it is submitted along with the rest of the form. Let's therefore look at a bit of the `login.pt` template:

```
...code to render the formlib form...
  <input tal:condition="request/camefrom | nothing" type="hidden"
       name="camefrom" tal:attributes="value request/form/camefrom | nothing" />
```

It's also nice to have a logout page:

```
from zope.app.security.interfaces import (IAuthentication,
                                          IUnauthenticatedPrincipal,
                                          ILogout)

class Logout(grok.View):
    grok.context(Interface)
    grok.require('zope.Public')

    def update(self):
        if not IUnauthenticatedPrincipal.providedBy(self.request.principal):
            auth = component.getUtility(IAuthentication)
            ILogout(auth).logout(self.request)
```

This page logs out the user if it's an authenticated user (principal).

3.1.2 Authentication

That's what is needed for retrieving user credentials. Let's now look at how we can authenticate the user next. For this we need to set up an `IAuthenticatorPlugin` with the name `users`:

```
from zope.app.authentication.interfaces import IAuthenticatorPlugin

class UserAuthenticatorPlugin(grok.GlobalUtility):
    grok.provides(IAuthenticatorPlugin)
    grok.name('users')

    def authenticateCredentials(self, credentials):
        if not isinstance(credentials, dict):
            return None
        if not ('login' in credentials and 'password' in credentials):
            return None
        account = self.getAccount(credentials['login'])

        if account is None:
            return None
        if not account.checkPassword(credentials['password']):
            return None
        return PrincipalInfo(id=account.name,
                             title=account.name,
                             description=account.name)

    def principalInfo(self, id):
        account = self.getAccount(id)
        if account is None:
            return None
        return PrincipalInfo(id=account.name,
                             title=account.name,
                             description=account.name)
```

```
    def getAccount(self, login):
        ... look up the account object and return it ...
```

What you need to do is implement the `getAccount` method to return an instance of an account object. This object should provide a `name` attribute which is the login name under which the account is used, and a `checkPassword` method which can be used to check the password. If no such account can be found, `None` must be returned. In `getAccount` you should consult the proper database, such as the ZODB, or an LDAP database, or a relational database, so that the proper account object can be retrieved or constructed.

The structure of this particular authenticator plugin is just one example of course - you can rewrite it to suit your particular user database system. You may for instance want to separate out `checkPassword` from the account object.

There are a few bits and pieces still needed, such as the `PrincipalInfo` class referred to above:

```
from zope.app.authentication.interfaces import IPrincipalInfo

class PrincipalInfo(object):
    grok.implements(IPrincipalInfo)

    def __init__(self, id, title, description):
        self.id = id
        self.title = title
        self.description = description
        self.credentialsPlugin = None
        self.authenticatorPlugin = None
```

We'll also give an example of an account object with password management facilities (encrypting the password):

```
from zope import component
from zope.app.authentication.interfaces import IPasswordManager

class Account(grok.Model):
    def __init__(self, name, password):
        self.name = name
        self.setPassword(password)

    def setPassword(self, password):
        passwordmanager = component.getUtility(IPasswordManager, 'SHA1')
        self.password = passwordmanager.encodePassword(password)

    def checkPassword(self, password):
        passwordmanager = component.getUtility(IPasswordManager, 'SHA1')
        return passwordmanager.checkPassword(self.password, password)
```

Instead of a `grok.Model` which allows its storage in the ZODB (such as in a container), you could construct this object on the fly when needed, or you could use an ORM mapper.

3.2 Authentication and authorization in Grok

Author Jan-Wijbrand Kolman (j-w)

Version This document is based on Grok-1.2.x

This document is an attempt to explain, into as much detail as needed, how the authentication and authorisation process is handled in Grok. This explanation is based on Grok-1.2 and the specific versions of its dependencies.

Note: This document most probably contains errors, ambiguities or ommisions. The reader is encouraged to discuss this document on the grok-dev mailing list and the #grok IRC channel.

3.2.1 Authentication versus authorisation

It is important to make a clear distinction between the terms "authentication" and "authorisation". To a large extent, the Wikipedia definitions apply:

Authentication Authentication (..) is the act of establishing or confirming something (or someone) as authentic, that is, that claims made by or about the subject are true(..) This might involve confirming the identity of a person(..).

Authorization Authorization (also spelt Authorisation) is the function of specifying access rights to resources, which is related to information security and computer security in general and to access control in particular. More formally, "to authorize" is to define access policy(..)

3.2.2 Authentication

Grok - as it is built on top of the Zope Toolkit - has a particular name for the "actor" or "user" that is acting upon the application by issueing a request. This "actor" is called the *principal*.

The request object in Grok has a reference to an object that provides IPrincipal - the so called principal object. In these cases where authenticating the principal is not possible, the principal object will provide IUnauthenticatedPrincipal.

The principal is associated to the request object by several authentication attempts during the handling of a request. In other words, Grok tries to authenticate each and every request.

This is worth repeating: Grok will try to authenticate the principal for each and every request!

The authentication follows several steps:

Before traversal hook

At the very beginning of the request handling, just before object traversal is started, the publisher will call `beforeTraversal()` on the publication object (see zope.publisher.publish and zope.app.publication.zopepublication). This `beforeTraversal()` implementation will try to authenticate against the global IAuthentication utility.

By default, the global IAuthentication utility implementation that is registered in a Grok application, is that of the zope.principalregistry. It knows how to authenticate the current request using information from the application's `parts/etc/site.zcml`.

When this "top-level" authentication is succesful, no other authentication attempts are made!

Traversal

The next step in the request handling is to traverse step by step to the requested object. For each step in the traversal, a callTraversalHook() is called on the publication object. In the callTraversalHook() implementation of this hook, an authentication attempt is undertaken - at least, as long as the object currently being "traversed over" is an ISite and actually has a IAuthentication utility registered for **and** the current principal on the request still provides IUnauthenticatedPrincipal!.

In other words, the very first authentication attempt that succeeds, wins!

After traversal

After the traversal has completed (mind you, the retrieved object is not called or "rendered" just yet!), the afterTraversal() hook is called on the publication object. The afterTraversal() implementation will do yet another authentication attempt.

If authentication has not succeeded thusfar, the request object will still have the unauthenticated principal associated. If authentication would have been succesful, the request will have and object associated providing IPrincipal. The implementation for this IPrincipal depends on the IAuthentication utility that did the authentication.

IAuthentication utilities

As we have seen, in each request there are several attempts to authenticate the request. This is done by calling `authenticate()` on the found utility. If the utility returns `None` Grok will continue with the IUnauthenticatedPrincipal, if it returns an IPrincipal object that object is used for the request.

In a Grok application, there's at least a globally registered IAuthentication utility available, implemented in zope.principalregistry. Its `authenticate()` method will validate a loginname and password "pair" - also called the *credentials* - against a simple registry.

Actually retrieving the credentials however is delegated by this principal registry to the ILoginPassword adapter for the request. The ILoginPassword implementation for this adapter that is registered by default, will try to get *Basic Authentication* credentials from the request.

So, as long as the correct Basic Authentication credentials are available in the request, the principal will be authenticated.

Note: It is important to realize that other IAuthentication implementations might have a completely different strategy for authenticating requests. Later we will see how the often used PluggableAuthentication implements the `authenticate()` contract.

3.2.3 Authorization

User agents, like web browsers, do not by default provide Basic Authentication credentials in the requests they send to the application. They have to be triggered in doing so. Basic Authentication is triggered by a 401 status code in the response. But what mechanism will make sure a 401 status code is set on the response object in a Grok application?

Let's assume we have view that is publicly accessible. Let's also assume that we have a request for this view, that cannot be authenticated. The security mechanism will check whether the current request is allowed to access this particular view (exactly how this is checked is out of scope for this document. It would deserve an article on its own).

Since it is a public view, the security checking will allow it and the publisher will make sure the view is rendered.

Now suppose we have an unauthenticated request for another view, a view that is protected. Since we were not able to authenticate the request, we cannot determine whether the request should be allowed and the security mechanism will raise an Unauthorized exception.

Whenever an exception is raised in the publication process, the publisher will try to handle the situation in a clean manner by giving the publication the possibilty to do something about the error situation by calling the handleException() hook.

Part of the important house-keeping that is performed in the handleException() hook, is the lookup of a view for the current exception and having this "error" view rendered.

Grok has "error" views registered for several error situations, including a view that is specifically built to handle Unauthorized exceptions. This particular error view implementation will call unauthorized() on the request object

which will set the 401 status code on the response, in turn triggering a login/password dialog of some sort in the user agent and that is the end of this request's life-cycle.

This implies that the error view is responsible for somehow triggering the authentication challenge. If there is no such view, the result will just be a basic error message displayed, with not much more that the name of the current exception.

A new request could come in, now with credentials in the form of Basic Authentication, and the the new request cycle will follow the described process for authenticating the request. In the example of a protected view, we will now have an authenticated request and we can check whether the now-know principal actually was granted sufficient permissions to access this resource. (XXX refer to the documents about defining permissions and roles and how to restrict access on views).

3.2.4 Other implementations: the Pluggable Authentication Utility

Even though it is the default setup, actual real-world applications hardly ever use Basic Authentication for authenticating the request or the global principal registry for user management. Basic Authentication has fundamental security flaws (especially when using unencrypted connections) and so does the user experience.

One of the most often used alternative IAuthentication implementations is the flexible, but sometimes confusing, Pluggable Authentication Utility (PAU). This utility lives in the zope.pluggableauth package.

Note: This section will not have detailed explanation about installing the PAU and its plugins. This is left for a how to-type document.

Authenticating the request

As we have seen, Grok will attempt to authenticate each request by looking up an IAuthentication utility during the traversal. Suppose we have an Application object installed and an instance of the PAU registered for it.

The PAU then, will be used to authenticate the request, when traversed to this application. Its authenticate() implementation splits the authentication attempt in two steps (and plug-in points):

1. Extract credentials for the request
2. Validate the credentials

Extracting credentials

Credentials extraction is handled by a plugin point. Utilities that provide ICredentialsPlugin can be registered for this plugin point. The PAU is then configured to use these plugins, in a given order, in order to obtain credentials. The PAU will call `extractCredentials()` on each of the registered plugins and the first plugin that return a login-password pair "wins".

For this section we will look at one specific, often used, credentials extraction implementation, called SessionCredentials.

Session Credentials plugin

The SessionCredentials plugin will try to extract credentials for the request by first looking for two specific keys in the form variables of the request, namely `login` and `password`. When values are present in the request for these keys, the values are stored in the session object.

Then, an attempt is made to get the credentials from the session object as they might have been stored just now, or have been stored already there in a previous request. If the user's session object indeed contains credentials, these are handed back to the PAU. The PAU then continues with the second step, validating the login-password pair. If no credentials have been found, `None` is returned.

If `None` is returned, the PAU will try again in the next registered ICredentialsPlugin. If there are no more `ICredentialsPlugin` components registered for this PAU, the process ends, and the request will continue with an `IUnauthenticatedPrincipal` object for its associated principal.

Validating the extracted credentials

Now that the Pluggable Authentication Utility asked its credentials plugin for a login and password, the PAU needs to verify the validity of the login and password. Like for the `ICredentialsPlugin` components, the PAU will now ask each of the `IAuthenticatorPlugin` components that have been registered to verify the credentials. These components are supposed to either return an `IPrincipal` object, or `None`. In case no principal is returned, this cycle is repeated for the next `ICredentialsPlugin` component that has been registered until the list of registered components has been exhausted.

Valid credentials thus will result in an authenticated principal associated to the request. In words, the PAU now is "done", and no other authentication attempts are being made for this request during the remaining part of the traversal, a view is looked up and - if the security mechanism allow for - rendered.

Not authorized, issueing an authentication challenge

Suppose we have an unauthenticated request for a protected resource. As the security machinery cannot determine whether this request is allowed to access the resource, an `Unauthorized` exception is raised and the the publication object will attempt to render an error view for this exception. We have seen that the default view for the `Unauthorized` exception will call `unauthorized()` on the request object. This would trigger the 401 status code on the response and the user agent would display some sort of dialog asking the user for Basic Authentication credentials.

That is however not necessarily the correct response, as components registered in the PAU probably require a different method of retrieving credentials! Instead of calling `unauthorized()` on the request, error views should call the `unauthorized()` method on nearest `IAuthentication` utility.

Note: Even though the default error view to handle Unauthorized exception will call the `request.unauthorized()` method, it is probably preferrable to call the `getUtility(IAuthentication).unauthorized()` from error views.

This means, a project that wishes to use the Pluggable Authentication components effecitively needs to create and register a custom view for the `Unauthorized` exception.

Attention: *The Grok project might want to consider implementing a set of baseclasses for common error view, including one for ‘‘Unauthorized‘‘ and have it call the latter. This would unify the API for triggering the authentication challenge.*

In fact the globally registered `IAuthentication` utility (that of the zope.principalregistry) implements the `unauthorized()` method as well, and will, by delegating to the `ILoginPassword` adaptation of the request, trigger the 401 response status code.

The PAU implements the `unauthorized()` method again in a pluggable manner. It will, like for the `authenticate()` part of the `IAuthentication` contract, delegate the actual challenge() to its `ICredentialsPlugin` components.

Challenge in the SessionCredentials plugin

The SessionCredentials plugin implements the challenge essentially as a redirect to a login page. The redirect URL will - if it can be computed - include a "camefrom" URL as a parameter. The login page should be a view containing an HTML form. It should have at least have two form fields, identified by the names `login` and `password`.

The form submit action should `POST` the form to the login page itself. Since Grok will try to authenticate each and every request, the `POST` request to the login form itself will also be authenticated. And as we have seen, the credentials extraction of the SessionCredentials plugin, will try to find values for `login` and `password` keys in the request. And this very request will have these values, as we justed "POST-ed" them!

The login page thus now has an authenticated principal available on the request and can decide to, in turn, redirect to the `camefrom` URL, closing the challenge-authenticate-view cycle.

How principals are created

...

CHAPTER

FOUR

VIEWS, TEMPLATING, CLIENT SIDE

Contents:

4.1 Fanstatic resources

Publishing static resources (e.g. javascript, css, images, files) with fanstatic.

4.1.1 Fanstatic

From the fanstatic homepage at http://fanstatic.org :

> "Fanstatic is a small but powerful framework for the automatic publication of resources on a web page. Think Javascript and CSS. It just serves static content, but it does it really well."

In conjunction with the zope.fanstatic package it is the default mechanism in grok to publish your static resources. It serves the static resources via WSGI which will off-load this task from Grok, generally improving the application's performance.

4.1.2 Getting started

For a basic example run the grokproject tool. The generated project will have the infrastructure set up for serving static resources. Look at app.py and resources.py.

4.1.3 Setting up a resource library

In order to serve static content, you need to register a Fanstatic Library for each of the static directories your project may have.

Registering the 'static' directory involves three steps:

1. Add zope.fanstatic and fanstatic to the install_requires in your project's setup.py. Include zope.fanstatic in your configure.zcml.
2. Include the following code in resource.py

```
from fanstatic import Library, Resource
library = Library(PACKAGENAME, 'static')
myStyleSheet = Resource(library, 'style.css')
```

Where the PACKAGENAME should be replaced with the dotted package name in which the 'static' directory resides.

3. In the setup.py of your project, add the following entry point (again, for each 'static' directory your project may have)

```
'fanstatic.libraries': [
   'PACKAGENAME = PACKAGENAME.resource:library'
]
```

4.1.4 Using the resource library

Using the resources

```
from mypackage import resource

class Index(grok.View):
    def update(self):
        resource.myStyleSheet.need()
```

This will include a <link> to your stylesheet in the view's generated html code.

4.2 Using a KSS plugin for Drag-and-Drop

Author http://jladage.myopenid.com/

This howto extends the example described in Adding AJAX to Grok with KSS

Before you can use the KSS commands provided by the plugin a number of things have to be setup:

- Define a dependency on the plugin in *setup.py*,
- Rerun *buildout*.

4.2.1 Edit the Configuration file

Add your plugin to *setup.py*. We add *kss.plugin.yuidnd*:

```
...
install_requires=['setuptools',
                  'grok',
                  'megrok.kss',
                  'kss.plugin.yuidnd', # <---
                  ],
...
```

Now run *buildout* to make it include the extra package:

```
bin/buildout
```

4.2.2 Drag-and-Drop example

In the *<head>* section of *index.pt*, add the following list of *<script>* tags just above the *@@kss_javascript*:

```
<script src="++resource++yahoo.js" type="text/javascript"></script>
<script src="++resource++dom.js" type="text/javascript"></script>
<script src="++resource++event.js" type="text/javascript"></script>
<script src="++resource++animation.js" type="text/javascript"></script>
<script src="++resource++dragdrop.js" type="text/javascript"></script>

<tal:kss_javascript replace="structure context/@@kss_javascript" />
<link tal:attributes="href static/app.kss"
      rel="kinetic-stylesheet" type="text/kss" />
```

These scripts are the parts of the Yahoo library used by the plugin; without them, KSS will crash.

Next, add two lists of items to the bottom of *index.pt*:

```
<p>You can drag and drop items on the client.</p>

<ul id="first">
  <li id="f1">Item A</li>
  <li id="f2">Item B</li>
  <li id="f3">Item C</li>
  <li id="f4">Item D</li>
</ul>

<ul id="second">
  <li id="s1">Item 1</li>
  <li id="s2">Item 2</li>
  <li id="s3">Item 3</li>
  <li id="s4">Item 4</li>
</ul>
```

In the first how-to the *app.kss* file was added to the package (in src/ksssample). In there, add the following KSS rules.

```
li:yuidnd-dragstart {
   evt-yuidnd-dragstart-action: delete;
}

ul:yuidnd-drop  {
    evt-yuidnd-drop-action: order;
}
```

The first rule binds a *dragstart* event on all <*li*>'s. The second rule sets up a *drop* event on a container: All children of this container will be dynamically reordered when dropping an element.

After refreshing the page, start dragging elements from the first to the second list or even reorder elements within one list. This is all happening on the client side only at this moment, so it's not that useful.

To notify the server when something is successfully dragged, add the following KSS rule.

```
li:yuidnd-dragsuccess {
  action-server: drop url('@@index/@@drop');
  drop-container: pass(dropContainerId);
  drop-index: pass(dropIndex);
}
```

The *dragsuccess* event triggers an *action-server* which passes two arguments.

- container - The HTML id of the container in which the element is dropped.
- index - The place where the element is dropped in the container.

In the *app.py*, add a *drop* method to the `AppKSS` class defined there in order to process the *dragsuccess* event. The complete class should afterwards look like this (the `welcome` method is not necessary for this example):

```
from megrok.kss import KSS
...
class AppKSS(KSS):
  grok.view(Index)

  def welcome(self):
    core = self.getCommandSet('core')
    core.replaceHTML('#click-me', '<p>ME GROK KISSED !</p>')

  def drop(self, container, index):
    """This method gets called from kss and will display a message"""
    core = self.getCommandSet('core')
    core.replaceInnerHTML(
      '#message',
      '<p>You dropped something in %s at %s !</p>' % (container, index))
```

The final step is to add a placeholder *<div>* to the bottom of the *index.pt* so the *drop* method can replace it's contents:

```
<div id="message"></div>
```

Now restart Zope, because a new method has been added to the app.py. After refreshing the page, drag-and-drop an element and notice the message that is displayed.

4.2.3 Getting rid of `++resource++` links using hurry.yui

Instead of adding all the `++resource++` links in your page template manually, you can use `hurry.yui` to let that files be included more programmatically. To do this you must perform the following additional steps:

1. Edit `setup.py` to require also `hurry.yui` and `hurry.zoperesource`:

```
...
install_requires=['setuptools',
                  'grok',
                  ...
                  'megrok.kss',
                  'kss.plugin.yuidnd',
                  'hurry.zoperesource', # <----
                  'hurry.yui',          # <----
                  ],
...
```

   Rerun `buildout` afterwards:

```
$ ./bin/buildout
```

   which will add these new eggs to your runtime environment. `hurry.zoperesource` is needed to register the resources (JavaScript files) provided by `hurry.yui` automatically with Zope on startup.

   So, while `hurry.yui` brings in all the JavaScript files and their dependencies, `hurry.zoperesource` makes them known to the Zope publishing machinery.

2. Edit `app.py` and change the `Index` view to look like this:

```
# app.py
from hurry import yui

...
```

```
class Index(grok.View):
  def update(self):
      yui.animation.need()
      yui.dragdrop.need()
```

This will take care, that upon request for the `Index` view the appropriate JavaScript files for YUI drag-and-drop functionality are included in the page header.

3. Remove the `<script>` tags from `index.pt` header where `++resource++` files are included. Just leave in:

```
...
<head>
  <tal:kss_javascript replace="structure context/@@kss_javascript" />
  <link tal:attributes="href static/app.kss"
        rel="kinetic-stylesheet" type="text/kss" />
</head>
...
```

After performing these steps and starting your instance everything should work as before but you don't have to edit page templates for inclusion of cryptic JavaScript resources anymore. At least you don't have to know anymore under which resource path one can find a certain JavaScript path and `hurry.yui` will know about dependencies of JavaScript files automatically.

4.3 Using z3x.form with Grok

Author http://vimes656.myopenid.com/

4.3.1 Create a sample z3c.form project

Create a sample application with grokproject:

```
$ grokproject z3cformsample
```

4.3.2 Edit setup.py

Add the following packages:

```
...
install_requires=['setuptools',
                  'grok',
                  'zope.viewlet', # <----
                  'z3c.form',     # <----
                  'z3c.formui',   # <----
                  'z3c.macro',    # <----
                  'z3c.template', # <----
                  'z3c.layer',    # <----
                  ],
...
```

Then run the buildout:

```
$ bin/buildout -N
```

Now open the app.py file and add:

```
import grok

from zope.interface import Interface, implements
from zope import schema

from z3c.form import button, field, form, widget
from z3c.formui import layout
from z3c.formui import interfaces
from z3c.form.interfaces import IFormLayer
from z3c.layer.pagelet import IPageletBrowserLayer

from persistent import Persistent


class Z3cFormSampleLayer(IFormLayer, IPageletBrowserLayer):
    pass


class AggregatedLayer(IDivFormLayer, Z3cFormSampleLayer):
    pass


class Z3cFormSampleSkin(grok.Skin):
    grok.name('IDoubtIt')
    grok.layer(AggregatedLayer)
...
```

Here we just created the skin 'IDoubtIt'. The skin is composed of 2 layers: Z3cFormSampleLayer and the AggregatedLayer which adds simple autogenerated Div widgets. The name 'IDoubtIt' will be the one will have to use for the ++skin++ namespace traversal. In the rest of the code will use the Grok layer.

```
...
class Z3cformSample(grok.Application, grok.Container):
    grok.layer(AggregatedLayer)
    pass
```

Now we register the layer for our which has nothing, just for simplicity sake.

```
...
class Index(grok.View):
    grok.layer(AggregatedLayer)

    def items(self):
        return self.context.items()
...
```

Again, for the Index view we have to register the Aggregated Layer. The items method returns the items in

```
...
class IPet(Interface):
    name = schema.TextLine(
            title=u'name of the Pet',
            description=u'To call your pet use this name',
            required=True,)


class Pet(Persistent):
    implements(IPet)
    name = schema.fieldproperty.FieldProperty(IPet['name'])
...
```

By inheriting from Persistent we are ensuring that the objects instantiated from Something class are persisted in ZODB. This is for demo porpouses, for real applications better to use a contanier.

... class SomethingView(grok.View):

grok.layer(AggregatedLayer) grok.context(Something) grok.name('index.html')

def render(self): return '<h1>I am lazy... create a better view for this</h1>'

class AddForm(form.AddForm, grok.View): grok.layer(AggregatedLayer) fields = field.Fields(ISomething) label = u'This is a add form'

def create(self, data): return Something(**data)

def add(self, object): count = 0 while 'something-%i' %count in self.context:

count += 1;

self._name = 'something-%i' %count self.context[self._name] = object return object

def nextURL(self): return self.context[self._name].redirect('index.html')

class EditForm(form.EditForm, grok.View): grok.context(Something) grok.layer(AggregatedLayer) grok.name('edit.html') form.extends(form.EditForm) label = u'This is a edit form' fields = field.Fields(ISomething)

@button.buttonAndHandler(u'Apply and View', name='applyView') def handleApplyView(self, action):

self.handleApply(self, action) if not self.widgets.errors:

self.redirect('index.html')

Run /bin/zopectl fg and point your browser to localhost:8080/

4.4 Traversing subpaths in views

Author Peter Bengtsson

4.4.1 Purpose

A URL points to a view of an object, but if the URL contains more than that, we'll show you how to use that in the view. What we're essentially doing is using the URL and its sub-path be **positional arguments** to the view. In this how-to, we'll use an example where you want a URL that looks like this: http://localhost/app/showcalendar/2008/08/01 rather than this: http://localhost/app/showcalendar?year=2008&month=08&day=01

4.4.2 Prerequisities

A Grok app with a view.

4.4.3 Step by step

The trick is to add a method called `publishTraverse(self, request, name)` to the view class. In this view, not only do you pick up what the extra subpath is but you also modify it right there so that Grok doesn't try to find the remaining things in the URL.

So, suppose you have a view called `ShowCalendar` that displays a nice calendar based on a date (or todays date if nothing else specified):

```
class ShowCalendar(grok.View):

    def update(self):
        form = self.request.form
        if 'year' in form and 'month' in form and 'day' in form:
            self.date = datetime(int(form.get('year')),
                                 int(form.get('month')),
                                 int(form.get('day')))
        else:
            self.date = datetime.now()

    def render(self):
        return self.date.strftime('%d %B %y')
```

A quick doctest explains how it works in action:

```
>>> from zope.testbrowser.testing import Browser
>>> browser = Browser()
>>> browser.open('http://localhost/app/showcalendar')
>>> from time import strftime
>>> strftime('%d %B %y') == browser.contents
True
>>> browser.open('http://localhost/app/showcalendar?year=2007&month=7&day=2')
>>> browser.contents
'02 July 2007'
```

So far so good. Now we decide we want to extend this to work based on a URL and not CGI parameters like above. The trick is to add the `publishTraverse()` method. Do note that the `name` argument is the first part of the subpath from the left and then `request.getTraversalStack()` is the rest in **reversed order**. Here's the extended view:

```
class ShowCalendar(grok.View):

    def publishTraverse(self, request, name):
        self.traverse_subpath = request.getTraversalStack() + [name]
        request.setTraversalStack([])
        return self

    def update(self):
        form = self.request.form

        subpath = getattr(self, 'traverse_subpath', [])
        if subpath and len(subpath) == 3:
            self.date = datetime(int(subpath[2]), int(subpath[1]), int(subpath[0]))
        elif 'year' in form and 'month' in form and 'day' in form:
            self.date = datetime(int(form.get('year')),
                                 int(form.get('month')),
                                 int(form.get('day')))
        else:
            self.date = datetime.now()

    def render(self):
        return self.date.strftime('%d %B %y')
```

An extension of the doctest above explains how it works:

```
>>> browser.open('http://localhost/app/showcalendar/2009/08/01')
>>> browser.contents
'01 August 2009'
```

This demonstrates three important ideas:

- The `request.getTraversalStack()` method
- The `request.setTraversalStack()` method and notice how easy it is to use
- In the `update()` method we use `getattr(self, 'traverse_subpath', [])` rather than `self.subpath` because it might not exist if the view is published without an explicit subpath on the URL.

4.4.4 Further information

The example above is rather simple and there might be more checks you want to add and for example raise `NotFound` exception which is done like this for example:

```
from zope.publisher.interfaces import NotFound

class ShowCalendar(grok.View):
   def publishTraverse(self, request, name):
       if name != u'2008':
           # poor example but gets the job done
           raise NotFound(self.context, name, request)
       ...
```

Another important thing to consider is that the above will not work with a default view which are views that are called `Index` unless the word `index` is in the URL itself. The reason for this is that the `Index` isn't necessarily published unless it appears in the URL.

4.5 Adding AJAX to Grok with KSS

Author kteague

If you want to use Ajax functionality inside a Grok application with an approach that insulates you as much as possible from coding in Javascript, then KSS (kinetic style sheets) is for you.

KSS uses an interpreter, implemented with Javascript, that runs the KSS rules. KSS rules themselves look very much like CSS (Cascading style sheets) rules. So a web developer familiar with CSS feels quite comfortable with the KSS syntax.

KSS is an independent project which can be used with different web frameworks. You can find information, documentation and tutorial about KSS on:

http://kssproject.org

4.5.1 Create a sample KSS project

Create a buildout for your sample kss application with grokproject:

```
grokproject ksssample
```

4.5.2 Edit the Configuration file

Add megrok.kss to setup.py:

```
...
install_requires=['setuptools',
                  'grok',
                  'megrok.kss', # <---
                  ],
...
```

Now run *buildout* with:

```
$ bin/buildout
```

Introductory Example

Lets first design a very simple example. On a simple web page, add a *div* with text inside. We want that when the user clicks inside this div, its text is changed into something which comes from the server. This is the classic Ajax server request example.

To begin with this example, we first have to prepare the template to make the page KSS aware:

You add the reference to KSS engine Javascript files in your application template *index.pt*. (grokproject should have created it in the *app_templates* subdirectory of src/ksssample.)

In the <*head*> section of the template, include the following lines:

```
<tal:kss_javascript replace="structure context/@@kss_javascript" />
```

Include a kinetic stylesheet with this line of code:

```
<link tal:attributes="href static/app.kss" rel="kinetic-stylesheet" type="text/kss" />
```

The kss file is being cached by your browser. So, to be able to work with kss, disable your browser cache to avoid tearing you hair out asking yourself why your kss changes are not picked up!

Include the *div* somewhere in the body of the template, give it a unique id that you can use to associate with KSS behaviour:

```
<div id="click-me">hallo</div>
```

Create a directory called *static* in the root of your Grok application, if it doesn't already exist. Then create a file *app.kss* in this static folder:

```
#click-me:click {
     action-server: welcome url(index/@@welcome);
}
```

Create a *KSSActions* view in *app.py* with code like this:

```
from megrok.kss import KSS

class AppKSS(KSS):
    grok.view(Index)

    def welcome(self):
        core = self.getCommandSet('core')
        core.replaceHTML('#click-me', '<p>ME GROK KISSED !</p>')
```

Now, you can run your application and test whether KSS is working by clicking the *div* and seeing whether the text is replaced by *ME GROK KISSED*, which should occur without the page having to be refreshed!

Client-side action

In some cases, modifications of the UI do not need to notify or get information from the server. KSS offers client-side actions: *action-client* in the KSS rules.

Let's build an example where clicking on a tag toggles on or off the presence of a class on the given tag (and of its CSS styling).

Add the needed markup in the application template:

```
<div id="toggle-me">Click me to toggle my border.</div>
```

Then, in the *<head>* section, add the style linked to the class that will be toggled:

```
<style type="text/css">
   .border {
       border: medium solid rgb(255,0,0);
    }
</style>
```

Finally, add the client-side rule to the KSS (*app.kss*):

```
/* Toggle a class on and off */
#toggle-me:click {
    action-client: toggleClass;
    toggleClass-value: border;
 }
```

Now refresh the page in your browser (you do not need to restart your application since you did not touch Python code) and test how clicking the new text displays or hides a red border around it.

That's all for now.

Further examples can be found on http://codespeak.net/svn/kukit/kss.core/trunk/kss/core/plugins/core/demo/.

Using the Firebug Javascript debugger for development

KSS comes in two modes: production mode and devel mode.

In development mode, KSS logs a lot of messages to the firebug console. They help a lot for debugging during KSS development.

You can use @@kss_devel_mode/ui url to access the UI that sets up the development mode.

4.6 Automatic Form Generation

NOTE: This tutorial depends on z3c.widget which isn't part of the standard Grok 1.0 distribution. It would benefit from a rewrite.

Author Dirceu Pereira Tiegs

4.6.1 Introduction

Grok supports automatic form generation by working with zope.interface, zope.schema and zope.formlib. This how-to will show you how to create an application that uses this feature and also how to use some more advanced widgets than the formlib defaults.

4.6.2 Schema and Fields

Fields are components that define a model's attributes, and schemas are collections of fields. For example:

Person
name: String
birth: Date
description: Text

The model above can be translated into Grok code like this:

```
from zope import interface, schema
class IPerson(interface.Interface):
    name = schema.TextLine(title="Name")
    birth = schema.Date(title="Birth")
    description = schema.Text(title="Description")
```

Defining an interface with schema fields allows automatic form generation and validation. To do this, grok.AddForm, grok.EditForm and grok.DisplayForm are used. These components are called forms; forms are web components that use widgets to display and input data. Typically a template renders the widgets by calling attributes or methods of the displayed object.

Widgets are components that display field values and, in the case of writable fields, allow the user to edit those values. Widgets:

- Display current field values, either in a read-only format, or in a format that lets the user change the field value.
- Update their corresponding field values based on values provided by users.
- Manage the relationships between their representation of a field value and the object's field value. For example, a widget responsible for editing a number will likely represent that number internally as a string. For this reason, widgets must be able to convert between the two value formats. In the case of the number-editing widget, string values typed by the user need to be converted to numbers such as int or float.
- Support the ability to assign a missing value to a field. For example, a widget may present a `None` option for selection that, when selected, indicates that the object should be updated with the field's `missing` value.

The forms have default templates that are used if no other template is provided.

grok.AddForm and grok.EditForm use the default template [grok_egg]/templates/default_edit_form.pt.

grok.DisplayForm uses [grok_egg]/templates/default_display_form.pt.

4.6.3 Input Validation - Constraints and Invariants

A constraint is constraint (:-)) that is bound to a specific field:

```
import grok
from zope import interface, schema
import re

expr = re.compile(r"^(\w&.%#$&'\*+-/=?^_`{}|~]+!)*[\w&.%#$&'\*+-/=?^_`{}|~]+"
                  r"@(([0-9a-z]([0-9a-z-]*[0-9a-z])?\.)+[a-z]{2,6}|([0-9]{1,3}"
```

```
                  r"\.){3}[0-9]{1,3})$", re.IGNORECASE)
check_email = expr.match

class IMyUser(interface.Interface):
    email = schema.TextLine(title="Email", constraint=check_email)

class MyUser(grok.Model)
    interface.implements(IMyUser)

    def __init__(self, email):
        super(MyUser, self).__init__()
        self.email = email
```

An invariant is a constraint that involves more than one field:

```
import grok
from zope import interface, schema
from datetime import date

class IMyEvent(interface.Interface):
    title = schema.TextLine(title="Title")
    begin = schema.Date(title="Begin date")
    end = schema.Date(title="End date")

    @interface.invariant
    def beginBeforeEnd(event):
        if event.begin > event.end:
            raise interface.Invalid("Begin date must be before end date")


class MyEvent(grok.Model)
    interface.implements(IMyEvent)

    def __init__(self, title, begin, end):
        super(MyEvent, self).__init__()
        self.title = title
        self.begin = begin
        self.end = end
```

4.6.4 Example 1 - Birthday Reminder

Grok want to remember his friends's birthday, so he created a simple application to do that.

ME GROK SMASH CALENDAR!

We want to use a custom widget to select dates, so you need to add 'z3c.widget' to setup.py of your package:

```
install_requires=['setuptools',
                  'grok',
                  'z3c.widget',
                  # Add extra requirements here
                  ],
```

And run ./bin/buildout. This will install z3c.widget and make it available to your project.

app.py is pretty simple:

```
import grok
```

```
class Friends(grok.Application, grok.Container):
    pass

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

friend.py contains our content component and it's forms:

```
import grok
from zope import interface, schema
from app import Friends

from z3c.widget.dropdowndatewidget.widget import DropDownDateWidget

class IFriend(interface.Interface):
    name = schema.TextLine(title=u"Name")
    birth_date = schema.Date(title=u"Birth Date")
    description = schema.Text(title=u"Description")

class Friend(grok.Model):
    interface.implements(IFriend)

    def __init__(self, name, birth_date, description):
        super(Friend, self).__init__()
        self.name = name
        self.birth_date = birth_date
        self.description = description

class AddFriend(grok.AddForm):
    grok.context(Friends)
    form_fields = grok.AutoFields(Friend)

    # Here is the trick. You set the 'custom_widget' attribute with the custom Widget's class
    form_fields['birth_date'].custom_widget = DropDownDateWidget

    @grok.action('Add event')
    def add(self, **data):
        obj = Friend(**data)
        name = data['name'].lower().replace(' ', '_')
        self.context[name] = obj

class Edit(grok.EditForm):
    form_fields = grok.AutoFields(Friend)
    form_fields['birth_date'].custom_widget = DropDownDateWidget

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

4.6.5 Example 2 - Wiki

Grok wants to impress beautiful cavewomen with a cool Web 2.0 application, so he built a Wiki with a JavaScript enabled text editor.

ME GROK WANTS COLLABORATE AND RICH TEXT EDITOR!

You need to add 'zc.resourcelibrary' and 'z3c.widget' to setup.py of your package and run ./bin/buildout to install the new components:

setup.py

```
install_requires=['setuptools',
                  'grok',
                  'zc.resourcelibrary',
                  'z3c.widget',
                  # Add extra requirements here
                  ],
```

app.py won't contain any application logic, only the application and the default view called "index".

```
import grok

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

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

wikipage.py is almost identical to friend.py in our first example:

```
import grok
from zope import interface, schema
from app import Wiki

from z3c.widget.tiny.widget import TinyWidget

class IWikiPage(interface.Interface):
    title = schema.TextLine(title=u"Title")
    contents = schema.Text(title=u"Contents")

class WikiPage(grok.Model):
    interface.implements(IWikiPage)

    def __init__(self, title, contents):
        super(WikiPage, self).__init__()
        self.title = title
        self.contents = contents

class AddWikiPage(grok.AddForm):
    grok.context(Wiki)
    form_fields = grok.AutoFields(WikiPage)
    form_fields['contents'].custom_widget = TinyWidget

    @grok.action('Add event')
    def add(self, **data):
        obj = WikiPage(**data)
        name = data['title'].lower().replace(' ', '_')
        self.context[name] = obj

class Edit(grok.EditForm):
    form_fields = grok.AutoFields(WikiPage)
    form_fields['contents'].custom_widget = TinyWidget

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

Here is the trick: to use TinyWidget you must load it's configuration. TinyWidget uses zc.resourcelibrary to load the JavaScript editor, and zc.resourcelibrary have some dependencies (on zope.app.component and zope.app.pagetemplate). Your package's configure.zcml must be like this:

```
<configure xmlns="http://namespaces.zope.org/zope"
           xmlns:grok="http://namespaces.zope.org/grok">
  <include package="zope.app.component" file="meta.zcml" />
  <include package="zope.app.pagetemplate" file="meta.zcml" />
  <include package="zc.resourcelibrary" file="meta.zcml" />
  <include package="zc.resourcelibrary" />
  <include package="z3c.widget.tiny" />
  <include package="grok" />
  <grok:grok package="." />
</configure>
```

And we must add a directive to the AddForm template to load the TinyMCE editor. First, copy the default template:

```
$ mkdir wikipage_templates
$ cp [grok_egg]/grok/templates/default_edit_form.pt wikipage_templates/addwikipage.pt
```

Then add this directive to the <head> tag of wikipage_templates/addwikipage.pt

```
<head>
  <tal:block replace="resource_library:tiny_mce" />
</head>
```

And that's it! Now AddWikiPage uses TinyMCE to edit the "contents" field.

4.6.6 Learning More

Many topics not were covered here. You can learn more reading the source code of Zope 3 components such as zope.schema and zope.formlib. Zope is a great platform and have a pretty good automated testing culture, so you can evend read / run doctests like these:

- http://svn.zope.org/zope.schema/trunk/src/zope/schema/README.txt?rev=80304&view=auto
- http://svn.zope.org/zope.schema/trunk/src/zope/schema/fields.txt?rev=75170&view=auto
- http://svn.zope.org/zope.schema/trunk/src/zope/schema/validation.txt?rev=79215&view=auto
- http://svn.zope.org/zope.formlib/trunk/src/zope/formlib/form.txt?rev=81649&view=markup
- http://svn.zope.org/zope.formlib/trunk/src/zope/formlib/errors.txt?rev=75131&view=markup

Web Component Development with Zope 3 is a great book written by Philipp von Weitershausen (wich is a Grok core developer). While the book doesn't cover Grok directly, it covers all the underlying technology that Grok uses:

- http://worldcookery.com/

4.7 Rest support in Grok

Author faassen

REST is a way to build web services, i.e. a web application where the user is another computer, not a human being. REST takes the approach to make the web service look very similar to a normal web application, using well-known semantics of HTTP.

Grok has support that helps you implement REST-based protocols. That is, Grok doesn't actually implement any RESTful protocols itself, but it allows you to easily add them in your own application.

To implement a REST protocol, you do something very similar to implementing a skin. This way, REST requests are separated from other requests on objects. This means you can have a normal web UI with views on a set of objects in parallel to the implementation of one or more REST protocols.

Let's see how you define a REST protocol. Similar to the way skins work, first you need to define a layer. In the case of REST, your layer must derive from grok.IRESTLayer.

```
class AtomPubLayer(grok.IRESTLayer):
  grok.restskin('atompub')
```

The *restskin* directive is used to name our skin.

REST handlers are very much like views like JSON or XMLRPC views. In the case of REST, you implement the HTTP methods on the view. It needs to be in the right REST layer.

```
class MyREST(grok.REST):
  grok.context(MyContainer)
  grok.layer(AtomPubLayer)

  def GET(self):
      return "GET request, retrieve container listing"

  def POST(self):
      return "POST request, add something to container"

  def PUT(self):
      return "PUT request, replace complete contents"

  def DELETE(self):
      return "DELETE request, delete this object entirely"
```

When handling a REST request, you often want to get to the raw body of the request. You can access a special `body` attribute that contains the body as a string.

```
class MyREST2(grok.REST):
    grok.context(SomeObject)
    grok.layer(AtomPubLayer)
    def POST(self):
        return "This is the body: " + self.body
```

This body should be parsed accordingly by your REST protocol implementation - it could for instance be some form of XML or JSON.

Now you can access the object with the REST skin, through requests like this (issuing GET, POST, PUT or DELETE):

```
http://localhost:8080/++rest++atompub/mycontainer
```

As you can see, you need to use the ++rest++<restskin> pattern somewhere in the URL in order to access the REST view for your objects. If you don't like the ++rest++ bit you can also apply the skin during traversal to the object:

```
@grok.subscribe(MyApp, grok.IBeforeTraverseEvent)
def restSkin(obj, event):
    if not AtomPubLayer.providedBy(event.request):
      grok.util.applySkin(event.request,
                          AtomPubLayer,
                          grok.IRESTSkinType)
```

Note that we apply the skin during traversal of the root (application) object such that the skin is applied early. It is important to realise that all requests made to views upon MyApp **and its subobjects** will have the skin applied to it, since the traverser will always traverse `MyApp` en route. If this is a problem, you could subscribe to traversal events from more specific objects such as containers and models. However, you cannot subscribe to traversal events on your views. This is because the traverser sees views as the end of traversal, not part of it, and thus the `grok.IBeforeTraverseEvent` event will not be issued.

Alternatively, if you're using Apache, you can use a few rewrite rules (just like with skins).

Using skins like this means you could have a single object implement several different REST skins. Since layers are used, you could also compose a single REST protocol out of multiple protocols should you so desire.

If you don't explicitly set a layer using `grok.layer` for a REST subclass, it'll use the grok.IRESTLayer by default. This layer is the base of all REST layers.

Similar again to XMLRPC or JSON views, security works with all this: you can use @grok.require() on the REST methods to shield them from public use.

4.8 What is XML-RPC ?

Author admin

From the site (http://xmlrpc.com): it's a spec and a set of implementations that allow software running on disparate operating systems, running in different environments to make procedure calls over the Internet.

4.8.1 So, What is Grok?

From the site: Grok is a web application framework for Python developers. It is aimed at both beginners and very experienced web developers. Grok has an emphasis on agile development. Grok is easy and powerful.

Grok accomplishes this by being based on Zope 3, an advanced object-oriented web framework. While Grok is based on Zope 3, and benefits a lot from it, you do not need to know Zope at all in order to get productive with Grok.

So, it is cool, isn't it? :)

4.8.2 Installation

To install the latest grok, give the following command:

```
$ sudo easy_install grokproject
```

This will download and install grok for you. After this we are ready to rock...

4.8.3 Creating our first project

Let's create the project named "Foo". For that give the command:

```
$ grokproject Foo
```

This will create a subdirectory in the current directory named "Foo", then it will download Zope3 and install Grok with that which you can start working with. It will ask you a few questions like:

```
Enter module (Name of a demo Python module placed into the package) ['app.py']:
```

Press Enter for the default value. Then:

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

We typed "grok" for both the user and password.

4.8.4 Starting up Zope

Switch to the Foo directory, and give the command:

```
$ bin/zopectl fg
```

This will startup Zope for you, you can access it through a web browser pointing to http://localhost:8080/ . Then add an application named *foo*.

You can access it by http://localhost:8080/foo, it will show:

```
Congratulations!

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

Now we are going to write our xmlrpc stuffs inside it.

4.8.5 XML-RPC class

Now you can open the file src/foo/app.py in a text editor. The default is shown below.

```
import grok

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

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

We will another class which will be available through this application class, the new class should inherit `grok.XMLRPC for this`, and we will write a `say()` method. It will return "Hello World!".

```
import grok

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

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

class FooXMLRPC(grok.XMLRPC):
    """The methods in this class will be available as XMLRPC methods
       on 'Foo' applications."""

    def say(self):
        return 'Hello world!'
```

The name of the class doesn't matter, so you can give it any name. Restart the Zope in the console, and you can connect to it through any xmlrpc client.

```
#!/usr/bin/env python

import xmlrpclib

s = xmlrpclib.Server('http://localhost:8080/foo')
print s.say()
```

Run this and see!

4.8.6 Class in a different file

What if you want to write the class in a different file in the src/foo directory and still want to have the methods to be available under Foo application. For that you need to tell grok explicitly that the new class to associate it to the Foo model by using the grok.context class annotation.

4.8.7 What is a class annotation?

A class annotation is a declarative way to tell grok something about a Python class. Let's see the example, we write a Boom.py with a Boom class:

```
import grok

from app import Foo

class Boom(grok.XMLRPC):
    grok.context(Foo)

    def dance(self):
        return "Boom is dancing!!"
```

Look at the line where it says `grok.context(Foo)` this is doing all the magic. In the fooclient.py you just need to call `s.dance()` instead of `s.say()`.

So, now write your dream system.

4.9 Working with Forms in Grok

Author Kevin Teague

A walkthrough of the basics of automatically generating HTML forms using Grok, as well as a discussion of a few more advanced Form manipulations.

4.9.1 Introducing Forms

An overview of the Form infrastructure in Grok

When working with HTML Forms in Grok, they are simply treated as a special type of View. The Form base class inherits from grok.View.

Just like you might make a simple View for an Application:

```
class MammothApplication(grok.Application, grok.Container):
    """World's greatest Mammoth manager web application."""

class Index(grok.View):
    def render(self):
        return "Augh! The application, it does nothing!"
```

You can replace the default View with a Form View.:

```
class Index(grok.Form):
    "An empty Form"
```

This will render a complete - albiet completely empty - form. The default HTML rendered for a blank form will look like:

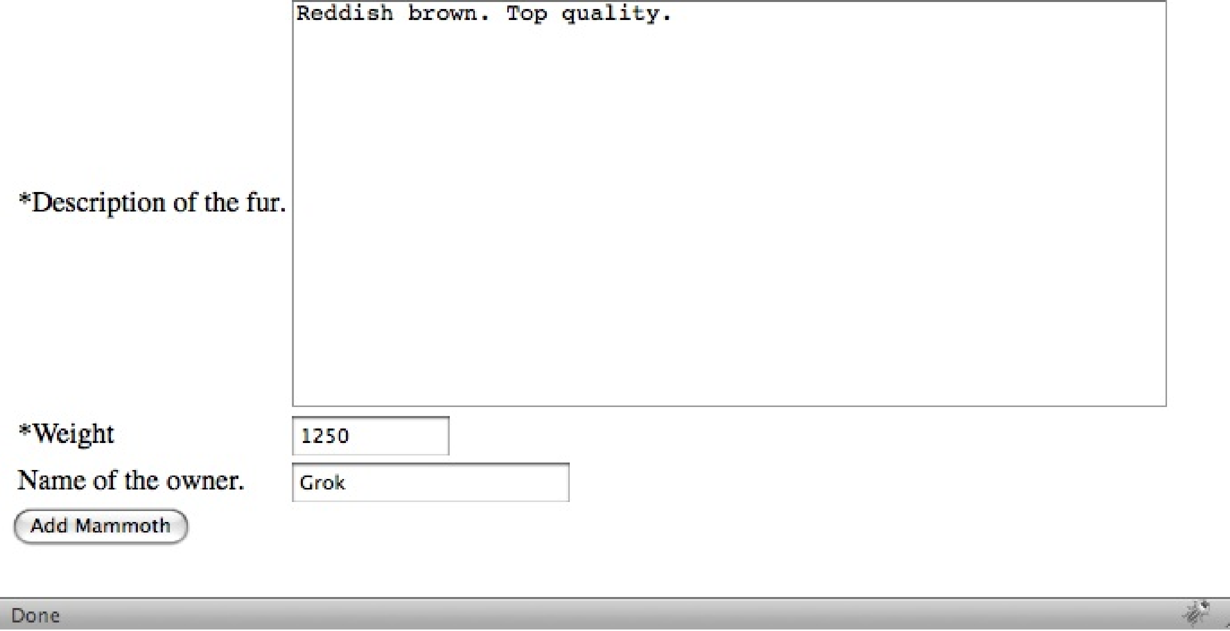
```
<html>
<head>
<base href="http://localhost:8080/mammoth/@@index" />
</head>

<body>

<form action="http://localhost:8080/mammoth/@@index"
      method="post" class="edit-form"
      enctype="multipart/form-data">

  <table class="form-fields">
    <tbody>
        <tr>
          <td class="label">
          </td>
          <td class="field">
          </td>
        </tr>
    </tbody>
  </table>

  <div id="actionsView">
  </div>

</form>

</body>
</html>
```

Forms provide the ability to customizing the template used to generate the HTML, as well as for handling the label of the form, the fields of a form, buttons of a form, the action that gets called when a form is submitted, and finally a way to automatically generate forms for creating and editing your Model objects.

Forms in Grok let you create complete working HTML Forms with very little code. Let's look at a more complete example:

```
class Edit(grok.EditForm):
    form_fields = grok.AutoFields(Mammoth)
    label = 'Edit Mammoth'

    @grok.action('Edit Mammoth')
    def edit(self, **data):
        self.applyData(self.context, **data)
        self.redirect(self.url(self.context))
```

In a short 7 lines of code this produces a complete working form that allow you to edit a Mammoth model object. When this form is viewed it looks like:



However, there are a lot of use-cases for working with forms, and this tutorial is intended to guide you through the many ways you can make the Grok Forms work towards what you want your application to do.

4.9.2 Defining Schema Fields

Schema Fields are used to describe the schema of an ordinary Python object. Before we dig into the details of Forms, it's helpful to familiarize yourself with how Grok and Zope 3 allow you to describe the schema of any ordinary Python object.

Typically in a web application, one will have the concept of a model layer - this is where the data model is described and implemented. The model layer is separate from the View and the Template layers. It knows nothing about HTML or responding to HTTP Requests, and is only concerned with maintaining the integrity of your applications data and providing core behaviour.

In Grok your model layer is typically defined as inheriting from classes which inherit from grok.Application, grok.Container and grok.Model. All of these base classes allow objects to be seamlessly stored in a database. Depending upon if you are making the distinction between the part of your Model layer which is only concerned with storing data (data model), or you are viewing the Model layer as everything which defines core application functionality (application model), then you may also consider your model layer as also including Adapters, Utilities and Subscribers.

Objects in your data model layer are usually going to want to provide a formal description of the type of data that they contain. In Grok this formal description of the data that a model object contains is called a schema. In relational database terms, you can think of a schema as being similar to a CREATE TABLE statement. It describes a collection of data fields.

When creating a schema in a relation database you might write:

```
CREATE TABLE mammoth (
    furryness   text NOT NULL DEFAULT 'Brown. Average quality.',
    weight      integer NOT NULL,
    owner       varchar(200),
);
```

The equivalent declaration as Zope schema would be:

```
from zope import schema
from zope import interface

class IMammoth(interface.Interface):
    furryness = schema.Text(
        title = u'Furryness',
        default = u'Brown. Average quality.',
    )
    weight = schema.Int(
        title = u'Weight',
    )
    owner = schema.TextLine(
        title = u'Owner',
        required = False,
    )
```

However, Grok goes one step farther in it's seperation of concerns than most other web frameworks and doesn't tightly tie the use of data schemas to just objects that map to a database and are part of a specific Object Relation Mapper (ORM). Schemas can be used with any ordinary Python object.

Consider three sources of Python objects which contain and deal with data: a Model object which is stored in a database, a Form object which is submitted from an HTTP Request, and a call to an external web service which pulls data in to the application via HTTP. All three types of objects contain data, so it's helpful to be able to use the same system for formally describing the data in any Python object.

This schema feature is provided by the zope.schema package. This package extends the notion of Interfaces as defined in the zope.interface package. Remember, Interfaces are a way of formally describing a set of method signatures and attributes. An orindinary Attribute defined by zope.interface only allows you to descirbe the name and doc string of an Attribute. Schemas extend the basic Attribute of interfaces to allow for more detailed descriptions. An Attribute which has this extended description is called a Schema Field (or just Field). It allows for additional descriptions of an Attribute such as title, required and default.

If we were only working with simple Interfaces, and we wanted to describe a the data that a Mammoth object provided we would write:

```
from zope import interface

class IMammoth(interface.Interface):
    "Describes a Mammoth"
    furryness = interface.Attribute("Furryness.")
    weight = interface.Attribute("Weight")
    owner = interface.Attribute("Owner")
```

This is very generic though. Let's go back to the zope.schema version of our Mammoth and enrich the description of the data even further:

```
class IMammoth(interface.Interface):
    "Describes a Mammoth"
    furryness = schema.Text(
        title = u'Description of the fur.',
```

```
        description = u"""
This field is primarily used by cavemen to aid in sorting and processing
the mammoths in the spring during fur harvesting season.""",
        required = True,
        default = u'Brown. Average quality.',
    )
    weight = schema.Int(
        title = u'Weight',
        description = u'Measured in Kilograms',
        required = True,
    )
    owner = schema.TextLine(
        title = u'Name of the owner.',
        description = u'Kept as a pet unless the owner is very hungry.',
        required = False,
    )
```

By extending the description of an ordinary Python attribute, we can use that information to automatically map data into an ORM, a web service call, or generate an HTML Form.

4.9.3 Further Reading and Reference

For more reading see the documentation that is part of the zope.schema package.

For now though it is helpful to remember that a Schema Field is simply an extensible description of a Python attribute. Every Schema Field description in an Interface must inherit from zope.schema.interfaces.IField. This interface looks like:

```
class IField(Interface):
    """Basic Schema Field Interface.

    Fields are used for Interface specifications.  They at least provide
    a title, description and a default value.  You can also
    specify if they are required and/or readonly.

    The Field Interface is also used for validation and specifying
    constraints.

    We want to make it possible for a IField to not only work
    on its value but also on the object this value is bound to.
    This enables a Field implementation to perform validation
    against an object which also marks a certain place.

    Note that many fields need information about the object
    containing a field. For example, when validating a value to be
    set as an object attribute, it may be necessary for the field to
    introspect the object's state. This means that the field needs to
    have access to the object when performing validation::

         bound = field.bind(object)
         bound.validate(value)

    """

    def bind(object):
        """Return a copy of this field which is bound to context.
```

```
    The copy of the Field will have the 'context' attribute set
    to 'object'.  This way a Field can implement more complex
    checks involving the object's location/environment.

    Many fields don't need to be bound. Only fields that condition
    validation or properties on an object containing the field
    need to be bound.
    """

title = TextLine(
    title=_(u"Title"),
    description=_(u"A short summary or label"),
    default=u"",
    required=False,
    )

description = Text(
    title=_(u"Description"),
    description=_(u"A description of the field"),
    default=u"",
    required=False,
    )

required = Bool(
    title=_(u"Required"),
    description=(
    _(u"Tells whether a field requires its value to exist.")),
    default=True)

readonly = Bool(
    title=_(u"Read Only"),
    description=_(u"If true, the field's value cannot be changed."),
    required=False,
    default=False)

default = Field(
    title=_(u"Default Value"),
    description=_(u"""The field default value may be None or a legal
                    field value""")
    )

missing_value = Field(
    title=_(u"Missing Value"),
    description=_(u"""If input for this Field is missing, and that's ok,
                      then this is the value to use""")
    )

order = Int(
    title=_(u"Field Order"),
    description=_(u"""
    The order attribute can be used to determine the order in
    which fields in a schema were defined. If one field is created
    after another (in the same thread), its order will be
    greater.

    (Fields in separate threads could have the same order.)
    """),
    required=True,
```

```
        readonly=True,
        )

    def constraint(value):
        u"""Check a customized constraint on the value.

        You can implement this method with your Field to
        require a certain constraint.  This relaxes the need
        to inherit/subclass a Field you to add a simple constraint.
        Returns true if the given value is within the Field's constraint.
        """

    def validate(value):
        u"""Validate that the given value is a valid field value.

        Returns nothing but raises an error if the value is invalid.
        It checks everything specific to a Field and also checks
        with the additional constraint.
        """

    def get(object):
        """Get the value of the field for the given object."""

    def query(object, default=None):
        """Query the value of the field for the given object.

        Return the default if the value hasn't been set.
        """

    def set(object, value):
        """Set the value of the field for the object

        Raises a type error if the field is a read-only field.
        """
```

4.9.4 Form Fields extend the information about a Schema Field

Understanding the difference between a form field and a schema field.

Schemas are a collection of schema fields. Schema fields are an extended, abstract description of a Python attribute. From this, we can use schema fields as the basis for automatically generating HTML Forms.

While schema fields provide a rich set of information about attributes, there is even more information that you might want to have when using those schema fields in a form. Thus form objects themselves consist of an ordered collection of form fields. Form fields are distinct from schema fields, and are used to enhance information about a schema field in the context of using that field in a specific Form.

The difference between schema fields and form fields sounds a little confusing, but you can simply auto-generate the form fields from the schema fields.

Typically information in a schema fields provide information about the data Model, independant of how that information might be presented in a user interface. The descriptions for schema fields might only be helpful in the context of a developer or internal business person, but be a poor fit for the user's of the application. However, in many applications you may decide that you don't need this additional seperation of information and wish to directly reuse the information in schema fields to automatically render the user interface and autogenerate the form fields from them.

Form fields contain an attribute named field which is the Schema field object which defines the data for the form field. In addition to this field the other attributes provided by a form field is described in the

zope.formlib.interfaces.IFormField Interface.:

```
class IFormField(interface.Interface):
    """Definition of a field to be included in a form

    This should not be confused with a schema field.
    """

    __name__ = schema.ASCII(
        constraint=reConstraint('[a-zA-Z][a-zA-Z0-9_]*',
                                "Must be an identifier"),
        title = u"Field name",
        description=u"""\
        This is the name, without any proefix, used for the field.
        It is usually the same as the name of the for field's schem field.
        """
        )

    field = interface.Attribute(
        """Schema field that defines the data of the form field
        """
        )

    prefix = schema.ASCII(
        constraint=reConstraint('[a-zA-Z][a-zA-Z0-9_]*',
                                "Must be an identifier"),
        title=u"Prefix",
        description=u"""\
        Form-field prefix.  The form-field prefix is used to
        disambiguate fields with the same name (e.g. from different
        schema) within a collection of form fields.
        """,
        default="",
        )

    for_display = schema.Bool(
        title=u"Is the form field for display only?",
        description=u"""\
        If this attribute has a true value, then a display widget will be
        used for the field even if it is writable.
        """
        )

    for_input = schema.Bool(
        title=u"Is the form field for input?",
        description=u"""\
        If this attribute has a true value, then an input widget will be
        used for the field even if it is readonly.
        """
        )

    custom_widget = interface.Attribute(
        """Factory to use for widget construction.

        If not set, normal view lookup will be used.
        """
        )

    render_context = schema.Choice(
```

```
        title=u"Should the rendered value come from the form context?",
        description=u"""\

        If this attribute has a true value, and there is no other
        source of rendered data, then use data from the form context
        to set the rendered value for the widget.  This attribute is
        ignored if:

        - There is user input and user input is not being ignored, or

        - Data for the value is passed to setUpWidgets.

        If the value is true, then it is evaluated as a collection of bit
        flags with the flags:

        DISPLAY_UNWRITEABLE
            If the field isn't writable, then use a display widget

            TODO untested


        SKIP_UNAUTHORIZED
            If the user is not priviledges to perfoem the requested
            operation, then omit a widget.

            TODO unimplemented

        """,
        vocabulary=schema.vocabulary.SimpleVocabulary.fromValues((
            False, True,
            DISPLAY_UNWRITEABLE,
            SKIP_UNAUTHORIZED,
            DISPLAY_UNWRITEABLE | SKIP_UNAUTHORIZED,
            )),
        default=False,
        missing_value=False,
        )

    get_rendered = interface.Attribute(
        """Object to call to get a rendered value

        This attribute may be set to a callable object or to
        a form method name to call to get a value to be rendered in a
        widget.

        This attribute is ignored if:

        - There is user input and user input is not being ignored, or

        - Data for the value is passed to setUpWidgets.

        """
        )
```

4.9.5 Creating a Simple Form

Grok provides four types of Form components, grok.Form, grok.AddForm, grok.EditForm and grok.DisplayForm. The latter three are all specializations of the basic grok.Form class and the grok.Form class itself is a specialization of the grok.View class. This means that all of the methods and attributes available in grok.View are available in any Grok Form. You can rely on self.context to represent the model object that the form is acting upon, and self.request to contain the current HTTP request object.

This also means that in simple cases, it's not necessary to have both a View class and a Form class, these can both be neatly tied into a single Form class. For more complex use cases it is possible to instantiate and work with several Forms from within one View, this is described later in the tutorial.

We'll now create the "Mammoth manager", a simple Grok application that lets us maintain data about a collection of Mammoths. Our starting application looks like:

```
import grok
from zope import schema
from zope import interface

class IMammoth(interface.Interface):
    "Describes a Mammoth"
    furryness = schema.Text(
        title = u'Description of the fur.',
        description = u"""
This field is primarily used by cavemen to aid in sorting and processing
the mammoths in the spring during fur harvesting season.""",
        required = True,
        default = u'Brown. Average quality.',
    )
    weight = schema.Int(
        title = u'Weight',
        description = u'Measured in Kilograms',
        required = True,
    )
    owner = schema.TextLine(
        title = u'Name of the owner.',
        description = u'Kept as a pet unless the owner is very hungry.',
        required = False,
    )

class MammothApplication(grok.Application, grok.Container):
    """World's greatest Mammoth manager web application."""

class Mammoth(grok.Model):
    grok.context(MammothApplication)
    grok.implements(IMammoth)

    furryness = u''
    weight = 0
    owner = u''

class MammothForm(grok.Form):
    grok.context(MammothApplication)
    grok.name('index')
    form_fields = grok.AutoFields(Mammoth)
```

You should already be familiar with the basics of a simple Grok application, the interesting part is the MammothForm class. For now we are declaring that this Form is named index so that it the default view for our application is this simple Form.

We've added one new line to the Form:

```
form_fields = grok.AutoFields(Mammoth)
```

The form_fields attribute in a form must be an object that implements IFormFields. The easiest way to generate a list of form fields that conforms to the IFormFields interface is to use the convenience function grok.AutoFields() upon a model object. This will generate form fields from all schema fields that the object provides.

Often you don't want to use every schema field in a model object. You can use the select() and omit() methods to choose just the fields you want:

```
# ask for fields by name
form_fields = grok.AutoFields(Mammoth).select('furryness','owner')

# or choose all fields and remove the unwanted ones
form_fields = grok.AutoFields(Mammoth).omit('weight')
```

When we view our Grok application, the MammothForm view will be rendered and it looks like this:

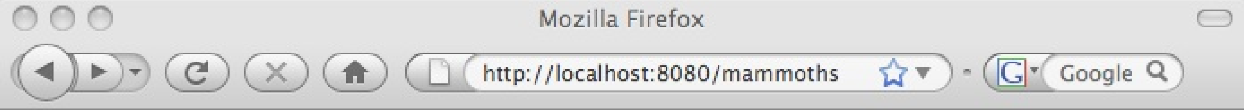
This is a start, but there is a problem with this form. It doesn't have a submit button! Let's extend MammothForm so that it can handle the creation of a new Mammoth.:

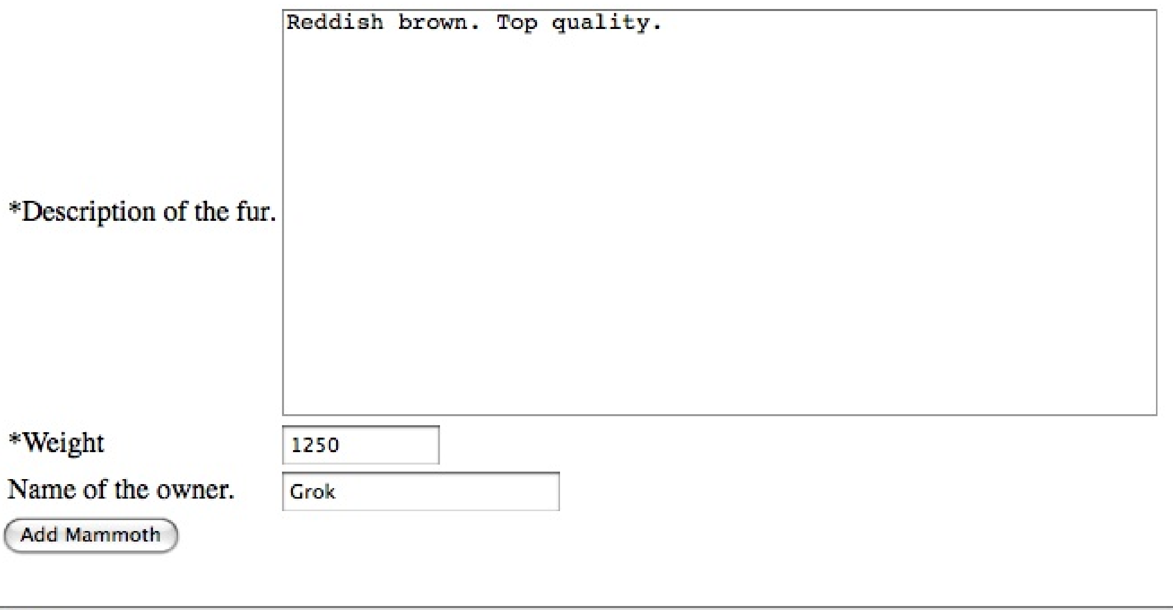
```
class MammothForm(grok.AddForm):
    grok.context(MammothApplication)
    grok.name('index')
    form_fields = grok.AutoFields(Mammoth)
    label = "Let's Make a Mammoth"

    @grok.action('Add Mammoth')
    def add(self, **data):
        mammoth = Mammoth()
        self.applyData(mammoth, **data)
        import datetime
        name = str(datetime.datetime.now()).replace(' ','-')
        self.context[name] = mammoth
```

```
        return self.redirect(self.url(self.context[name]))

class MammothView(grok.View):
    "Display a Mammoth"
    grok.context(Mammoth)
    grok.name('index')

    def render(self):
        return """
<html><body>
    <p><b>Furryness:</b> %s</p>
    <p><b>Weight:</b> %s kilograms</p>
    <p><b>Owner:</b> %s</p>
</html></body>""" % (
                self.context.furryness,
                self.context.weight,
                self.context.owner,
        )
```

What's changed? The base class of form to is now an grok.AddForm to indicate that this form is intended for the creation of new Mammoth objects. We have also given our form a label attribute, this value will be displayed at the top of our form to improve the user-interface. We have created an add method that will be called when the form is submitted. The button in our form is automatically rendered and wire it up to the add method by using the grok.action decorator. This decorator takes a single argument which will be used as the name of the button.

Our new form, once filled out and ready for submission, looks like this:





The add method we created uses another feature of forms, the applyData() method. This is a convience method for

automatically takign data submitted by the form in the request and setting it as corresponding attributes in the the data object. Calling this method also sends out a grok.IObjectMovedEvent if any of the data has changed.

4.9.6 Customising the default Form templates

Learn how to tweaking the template used to render a form. Useful for integrating a form into your applications custom layout.

Grok provides default templates that are used to render a form. While they are functional, they are also very plain. Furthermore, they have no way of hooking into headers, footers, sidebars and other layout elements of your application.

It's intended that you override the default templates by providing your own templates. You can tell a form to use a custom template by providing a template attribute in the form:

```
class Form(grok.Form):
    template = grok.PageTemplateFile('custom_edit_form.pt')
```

It's helpful to start by making a copy of the default templates in your application, and then add your customizations as needed.

The default edit form is (part of the grokcore.formlib package):

```
<html>
<head>
</head>

<body>
<form action="." tal:attributes="action request/URL" method="post"
      class="edit-form" enctype="multipart/form-data">

  <h1 i18n:translate=""
    tal:condition="view/label"
    tal:content="view/label">Label</h1>

  <div class="form-status"
    tal:define="status view/status"
    tal:condition="status">

    <div i18n:translate="" tal:content="view/status">
      Form status summary
    </div>

    <ul class="errors" tal:condition="view/errors">
      <li tal:repeat="error view/error_views">
         <span tal:replace="structure error">Error Type</span>
      </li>
    </ul>
  </div>

  <table class="form-fields">
    <tbody>
      <tal:block repeat="widget view/widgets">
        <tr>
          <td class="label" tal:define="hint widget/hint">
            <label tal:condition="python:hint"
                   tal:attributes="for widget/name">
              <span class="required" tal:condition="widget/required"
              >*</span><span i18n:translate=""
```

```
                            tal:content="widget/label">label</span>
            </label>
            <label tal:condition="python:not hint"
                   tal:attributes="for widget/name">
              <span class="required" tal:condition="widget/required"
              >*</span><span i18n:translate=""
                            tal:content="widget/label">label</span>
            </label>
          </td>
          <td class="field">
            <div class="widget" tal:content="structure widget">
              <input type="text" />
            </div>
            <div class="error" tal:condition="widget/error">
              <span tal:replace="structure widget/error">error</span>
            </div>
          </td>
        </tr>
      </tal:block>
    </tbody>
  </table>

  <div id="actionsView">
    <span class="actionButtons" tal:condition="view/availableActions">
      <input tal:repeat="action view/actions"
             tal:replace="structure action/render"
             />
    </span>
  </div>
</form>

</body>
</html>
```

The default display form is (part of the grokcore.component package):

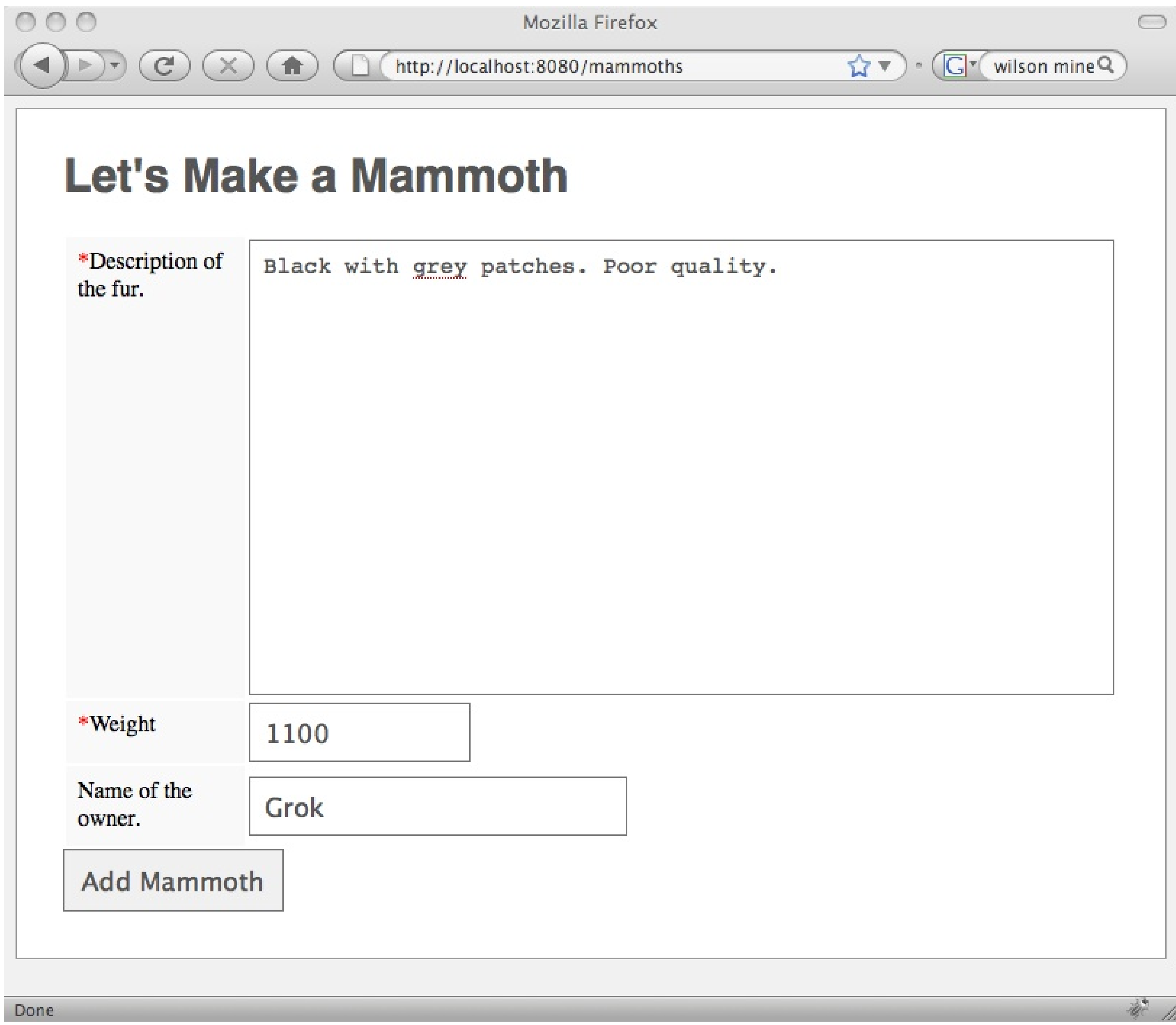
```
<html>
<head>
</head>

<body>
  <table class="listing">
    <thead>
      <tr>
        <th class="label-column"> </th>
        <th> </th>
      </tr>
    </thead>
    <tbody>
      <tal:block repeat="widget view/widgets">
        <tr tal:define="odd repeat/widget/odd"
          tal:attributes="class python: odd and 'odd' or 'even'">
          <td class="fieldname">
            <tal:block content="widget/label"/>
          </td>
          <td>
            <input tal:replace="structure widget" />
          </td>
        </tr>
```

```
      </tal:block>
    </tbody>
    <tfoot>
      <tr class="controls">
        <td colspan="2" class="align-right">
          <input tal:repeat="action view/actions"
            tal:replace="structure action/render" />
        </td>
      </tr>
    </tfoot>
  </table>
</body>
</html>
```

For this example we'll just put a snippet of CSS into our customised edit form right below that body tag that looks like:

```
<style type="text/css">
    body {
        background: #f1f1f1;
    }
    form {
        background: #ffffff;
        border: 1px solid #999999;
        padding: 2em;
    }
    form h1 {
        font-family: sans-serif;
        font-weight: bold;
        color: #555555;
        margin-top: 0;
    }
    form .label {
        vertical-align: top;
        padding: 0.5em;
        background: #f9f9f9;
    }
    form .required {
        color: red;
    }
    form textarea {
        border: 1px solid #777777;
        color: #555555;
        font-size: 1.2em;
        line-height: 1.2em;
        padding: 0.5em;
    }
    form input {
        border: 1px solid #777777;
        color: #555555;
        font-size: 1.1em;
        line-height: 1.2em;
        padding: 0.5em;
    }
</style>
```

Now our MammothForm has a shiny new look and feel:



4.9.7 Customising individual Form Fields

Learn how to customise individual form fields.

Form fields are rendered into HTML by widgets. A widget is similar to a view, except that they are designed to work specifically upon schema fields.

For example, a Text schema field has a TextAreaWidget that is used for input. A DateTime schema field has a DateTimeWidget and a DateTimeDisplayWidget, depending upon whether the field is meant for input or is being displayed as read-only.

The setupWidgets method of the Form class is automatically called for you before the form is rendered. This method is intended to be overridden, so that you can customise the input or display of individual form fields by using modified or different widgets.:

```
def setUpWidgets(self, ignore_request = False):
    super(MammothForm, self).setUpWidgets(ignore_request)
    self.widgets['furryness'].width = 20
```

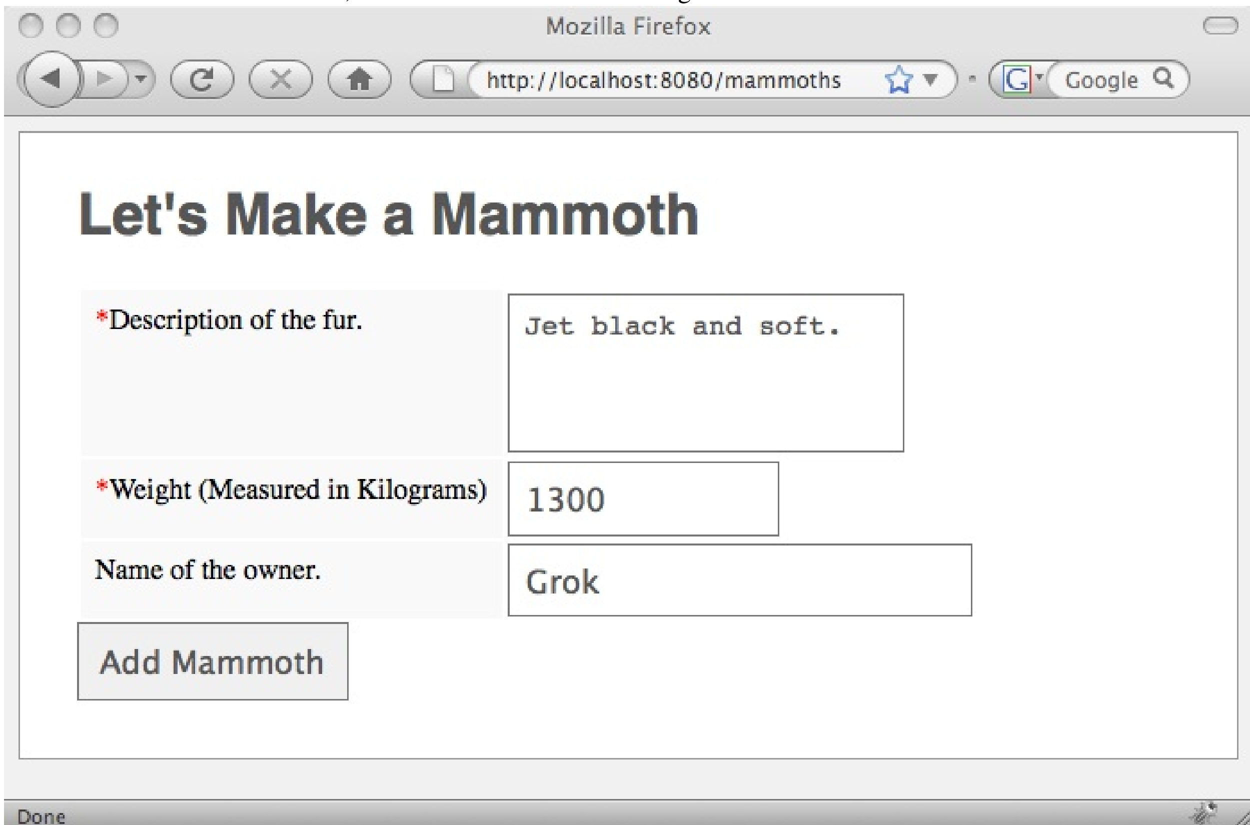
When overridding setupWidgets, first use super() to call the default setupWidgets method. This will set a widgets

attribute on your form, and each widget can be referred to by the name of the form field that it is rendering.

In the Mammoth manager application, we want to make the TextArea widget used for the furryness field smaller. We also want to display both the title and the description from the schema field for the weight attribute.:

```
class MammothForm(grok.AddForm):
    grok.context(MammothApplication)
    grok.name('index')
    form_fields = grok.AutoFields(Mammoth)
    label = "Let's Make a Mammoth"
    template = grok.PageTemplateFile('custom_edit_form.pt')

    def setUpWidgets(self, ignore_request = False):
        super(MammothForm, self).setUpWidgets(ignore_request)
        self.widgets['furryness'].width = 20
        self.widgets['furryness'].height = 3
        self.widgets['weight'].label = '%s (%s)' % (
            self.form_fields['weight'].field.title,
            self.form_fields['weight'].field.description,
        )

    @grok.action('Add Mammoth')
    def add(self, **data):
        mammoth = Mammoth()
        self.applyData(mammoth, **data)
        import datetime
        name = str(datetime.datetime.now()).replace(' ','-')
        self.context[name] = mammoth
        return self.redirect(self.url(self.context[name]))
```

Now we have a smaller text box, and a custom label for the weight:



Remember that every form field is going to have a field attribute that refers to the schema field that the form field is being applied to. Form fields and the widgets used to render each of those fields are unique to every form in your application. The schema fields however are global and are typically used to describe your data model, and this information may be used by many different components. You want to make sure that you only modify form fields and widgets within your form and never modify a schema field. Otherwise these changes will likely produce unwanted side-effects.

4.9.8 Creating custom widgets and overriding widgets globally

How to create new widgets, as well as replace existing widgets with your own custom versions. The widgets used to render form fields in Grok are supplied by the zope.app.form package. You can create your own custom widget and override the default widget on a per field basis, or override that widget globally for all fields of a particular type.

4.9.9 Creating a new widget

Making a widget is fairly straightforward. Every widget implements the zope.app.form.interfaces.IWidget interface. There are number of useful existing widget implementations and base classes to make it easier to implement this interface.

Let's say that we wanted to make a custom text input widget where the default display width is longer. We could write this widget as:

```
from zope.app.form.browser.textwidgets import TextWidget

class LongTextWidget(TextWidget):
    displayWidth = 35
```

Then use it in form on a per field basis with by setting the custom_widget attribute.:

```
class MammothForm(grok.AddForm):
    form_fields = grok.AutoFields(Mammoth)
    form_fields['owner'].custom_widget = LongTextWidget
```

4.9.10 Overriding widgets globally

If you want to override every TextLine form field globally, then you need to register an adapter as an override using ZCML. Create a file in your project named overrides.zcml and put the following adapter in it:

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

    <adapter
        for="zope.schema.interfaces.ITextLine
             zope.publisher.interfaces.browser.IBrowserRequest"
        provides="zope.app.form.browser.interfaces.ITextBrowserWidget"
        factory="gum.widgets.LongTextWidget"
        permission="zope.Public"
        />

</configure>
```

The you need to tell Zope 3 to include that configuration and have it override the existing configuration with the includeOverrides directive. Change the configure.zcml file in your package to read:

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

    <include package="grok" />
    <includeDependencies package="." />
    <grok:grok package="." />
    <includeOverrides file="overrides.zcml" />

</configure>
```

Now every TextLine form field will use your custom widget. Note that this is very global - so if you have multiple apps within the same server instance, they will all be updated.

4.9.11 Widget interfaces

For reference, the IWidget, IInputWidget and IDisplayWidget interfaces are shown below. Also note that IView in IWidget is from the zope.component.interfaces.IView interface. This interface is quite simple and only declares a generic context and request attribute:

```
class IWidget(IView):
    """Generically describes the behavior of a widget.

    Note that this level must be still presentation independent.
    """

    name = Attribute(
        """The unique widget name

        This must be unique within a set of widgets.""")

    label = Attribute(
        """The widget label.

        Label may be translated for the request.

        The attribute may be implemented as either a read-write or read-only
        property, depending on the requirements for a specific implementation.

        """)

    hint = Attribute(
        """A hint regarding the use of the widget.

        Hints are traditionally rendered using tooltips in GUIs, but may be
        rendered differently depending on the UI implementation.

        Hint may be translated for the request.

        The attribute may be implemented as either a read-write or read-only
        property, depending on the requirements for a specific implementation.

        """)

    visible = Attribute(
        """A flag indicating whether or not the widget is visible.""")

    def setRenderedValue(value):
```

```
        """Set the value to be rendered by the widget.

        Calling this method will override any values provided by the user.

        For input widgets (`IInputWidget` implementations), calling
        this sets the value that will be rendered even if there is
        already user input.

        """

    def setPrefix(prefix):
        """Set the name prefix used for the widget

        The widget name is used to identify the widget's data within
        input data.  For example, for HTTP forms, the widget name is
        used for the form key.

        It is acceptable to *reset* the prefix: set it once to read
        values from the request, and again to redraw with a different
        prefix but maintained state.

        """

class IInputWidget(IWidget):
    """A widget for editing a field value."""

    required = Bool(
        title=u"Required",
        description=u"""If True, widget should be displayed as requiring input.

        By default, this value is the field's 'required' attribute. This
        field can be set to False for widgets that always provide input (e.g.
        a checkbox) to avoid unnecessary 'required' UI notations.
        """)

    def getInputValue():
        """Return value suitable for the widget's field.

        The widget must return a value that can be legally assigned to
        its bound field or otherwise raise ``WidgetInputError``.

        The return value is not affected by `setRenderedValue()`.
        """

    def applyChanges(content):
        """Validate the user input data and apply it to the content.

        Return a boolean indicating whether a change was actually applied.

        This raises an error if there is no user input.
        """

    def hasInput():
        """Returns ``True`` if the widget has input.

        Input is used by the widget to calculate an 'input value', which is
        a value that can be legally assigned to a field.
```

```
        Note that the widget may return ``True``, indicating it has input, but
        still be unable to return a value from `getInputValue`. Use
        `hasValidInput` to determine whether or not `getInputValue` will return
        a valid value.

        A widget that does not have input should generally not be used
        to update its bound field.  Values set using
        `setRenderedValue()` do not count as user input.

        A widget that has been rendered into a form which has been
        submitted must report that it has input.  If the form
        containing the widget has not been submitted, the widget
        shall report that it has no input.

        """

    def hasValidInput():
        """Returns ``True`` is the widget has valid input.

        This method is similar to `hasInput` but it also confirms that the
        input provided by the user can be converted to a valid field value
        based on the field constraints.
        """

class IDisplayWidget(IWidget):
    """A widget for displaying a field value."""

    required = Bool(
        title=u"Required",
        description=u"""If True, widget should be displayed as requiring input.

        Display widgets should never be required.
        """)
```

4.9.12 Using multiple schemas with a Form

Demonstrating using multiple schemas within the same form.

Forms can use fields from multiple schemas. We have seen how grok.AutoFields() can be used to automatically provide all form fields from a class. If that class implements more than one schema, then all fields from all schemas will be used to generate form fields (without any overlap). Grok also provides the grok.Fields() convenience method which let's us build form fields from schema fields directly.

Let's say that in the Mammoth manager application we want to automatically send an email notification when a new Mammoth is added, and we'd like to provide a field in the form where a message can be appended to the email. Both grok.AutoFields and grok.Fields return objects that implement the IFormFields interface. This interface supports addition (+), so we can add two collections of form fields together.

We could change the form_fields attribute of the MammothForm class to read:

```
form_fields = grok.AutoFields(Mammoth) + grok.Fields(
    message = schema.Text(
        title=u'Email Message', required=False
    ),
)
```

Now when the form is submitted, the data dictionary will contain a key named message with the value entered into the "Email Message" field.

4.9.13 Editing two Model objects in one Form

With a simple grok.Form or grok.AddForm we can generate whatever set of form fields we need. But with a grok.EditForm it's necessary for the form to bind the fields to the object it's editing, so that the edit widgets are pre-filled with existing data.

What if we wanted to create a form capable of editing two different Model objects at the same time? How can we tell the form to read and write data to both Model objects?

The answer is to make an EditForm for one primary object, and then create an Adapter that allows us to adapt that primary object to our secondary object.

In the Mammoth manager application we'll add a location attribute to the MammothApplication. This way the cave men using the application can easily update the application with the location of the herd whenever they are making an edit to an existing mammoth. We'll need to make an adapter which can take a Mammoth object and extend it with the IHerdLocation interface. We can think of this as the primary object being edited (Mammoth) as providing the attribtues of the secondary object (MammothApplication) via our adapter.

First we update the application object:

```
class IHerdLocation(interface.Interface):
    "Where's the mammoth herd at?"
    location = schema.Text(
        title = u'Herd Location',
        required = False,
    )

class MammothApplication(grok.Application, grok.Container):
    """World's greatest Mammoth manager web application."""
    grok.implements(IHerdLocation)
    location = u''
```

Then add an EditMammothForm. The only thing special about this form is that it has form fields for both the Mammoth and IHerdLocation. Once we have the right Adapter in place, the Form class is smart enough that if a field is not provided by the primary object then it will try and adapt the primary object to the interface that contains the field of the secondary object.:

```
class EditMammothForm(grok.EditForm):
    grok.context(Mammoth)
    grok.name('edit')
    form_fields = grok.AutoFields(Mammoth) + grok.Fields(IHerdLocation)
    label = 'Edit Mammoth'
    template = grok.PageTemplateFile('custom_edit_form.pt')

    @grok.action('Edit Mammoth')
    def edit(self, **data):
        self.applyData(self.context, **data)
        self.redirect(self.url(self.context))
```

Finally we need to provide an adapter that can take a Mammoth object, and provide the IHerdLocation interface. We'll implement this by creating a property in the adapter which delegates gettting/setting of the location attribute to the MammothApplication object.:

```
class MammothHerdLocationAdapter(grok.Adapter):
    "Allows us to edit a MammothApplication via a Mammoth"
    grok.context(Mammoth)
    grok.implements(IHerdLocation)

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

```
        self.app = context.__parent__

    def _get_location(self): return self.app.location
    def _set_location(self, value): self.app.location = value
    location = property(_get_location, _set_location)
```

Now when we edit a Mammoth, the location field will be bound to an instance of the MammothHerdLocationAdapter.

4.9.14 Accessing a Form within another View

Forms are a specialization of the grok.View base class. You can use a form programmatically from within another View if required.

All Forms subclass from grok.View, so they can be instantiated and rendered programmatically, just like any other View.

Let's say that you wanted to embed MammothForm within another View. You can simply instantiate the form, passing it the same context and request objects that your View was instantiated with.:

```
class FormWrappedView(grok.View):
    grok.context(MammothApplication)
    grok.name('wrap')

    def render(self):
        form = MammothForm(self.context, self.request)
        form.template = grok.PageTemplateFile('bare_edit_form.pt')
        return """<html><body>%s</body></html>""" % form()
```

Once you have a form object, you can edit it's attributes. For example, change the template attribute to use a different template. Calling a form object itself will first update the form, setting up the widgets and processing data from the request, and then call and return the render() method. Howewver, the normal grok.View convention of first calling an a custom update() method if it exists to do any pre-processing is stil followed. If a URL redirect was generated during the update, it is allowed to happen without interruption and normal form processing doesn't take place.

Forms are grokked during start-up, which means like everything else they are registered with the Zope Component Architecture. That means that you can instantiate a Form object without needing to hard-code the class name in the calling View. This can be useful, since you can refactor your code and move or rename the Form class without needing to change the calling View code.:

```
from zope import component
form = component.queryMultiAdapter(
    (self.context, self.request),
    name='index'
)
```

4.10 Understanding viewlets

Author Unknown

Viewlets is a flexible way to compound html
snippets. Learn what they are and how they work.

4.10.1 Introduction

A viewlet is a component that represents an HTML snippet. It has basically the same purpose as a macro, but is more clean, powerful and flexible. Viewlets in Grok are synonymous to viewlets in Zope 3. It's only how to create and register them that is different, but they work exactly the same way.

Viewlets are typically used for the layout of the web site. Often all the pages of the site has the same layout with header, one or two columns, the main content area and a footer.

Viewlets are grouped by adding them to viewlet managers. Viewlets can have a certain order within a group. The viewlet manager has a name and that is what you refer to in a page template when you want to use viewlets. When you ask the viewlet manager to be rendered or inserted into your page template, what will happen is that update() of each viewlet is called. After that render() of each viewlet will be called and the return values of each render() is added to the output. You can provide your own render() method if you want or you can provide a template and that will be called.

4.10.2 Example

This is an example of how you can use viewlets. Two viewlets (Fred and Wilma) are ordered in a viewlet manager. To try this out, create a new project with "grokproject viewlettest" and add these three files:

```
#
# app.py
#
import grok

class Counter(grok.Model):
    def __init__(self):
        self.count = 0

class GrokExample(grok.Application, grok.Container):
    def __init__(self):
        grok.Application.__init__(self)
        grok.Container.__init__(self)
        self.fred = Counter()
        self.wilma = Counter()

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

class CountersManager(grok.ViewletManager):
    grok.name('counters')
    grok.context(GrokExample)

class Fred(grok.Viewlet):
    grok.viewletmanager(CountersManager)
    grok.context(GrokExample)
    grok.template('fred_template')
    grok.order(2)

    def update(self):
        self.context.fred.count += 1
        self.counter = self.context.fred.count

class Wilma(grok.Viewlet):
    grok.viewletmanager(CountersManager)
    grok.context(GrokExample)
```

```
    grok.order(1)

    def update(self):
        self.context.wilma.count += 1

    def render(self):
        return """<div style='float: left; width:200px; height:200px'>
                  <h1>Wilma</h1>
                  <div style='font-size:200%%'>%d</div>
                  </div>""" % self.context.wilma.count
```

Page templates:

```
#
# app_templates/fred_template.pt
#
<div style="float: left; width:200px; height:200px">
    <h1>Fred</h1>
    <div style="font-size: 200%;" tal:content="view/counter" />
</div>

#
# app_templates/index.pt
#
<html>
    <body>
        <h1>Grok viewlets</h1>
        <div style="width: 400px" tal:content="structure provider:counters" />
    </body>
</html>
```

What happens when you visit http://localhost:8080/exampleapp/

- The 'index' view of the application is automatically used when nothing is specified and it will render the index.pt template.
- The index.pt template says it wants to insert whatever is returned by the viewlet manager called 'counters'.
- 'counters' is a group of two viewlets and the Wilma viewlet has lower order than Fred so the 'counters' viewlet manager process Wilma first.
- The viewlet manager will call update() on the Wilma viewlet first and update Wilma's counter. The manager will then call update() on the Fred viewlet which will cause an increment of Fred's counter.

4.11 Using Viewlets for Layout

Author Unknown

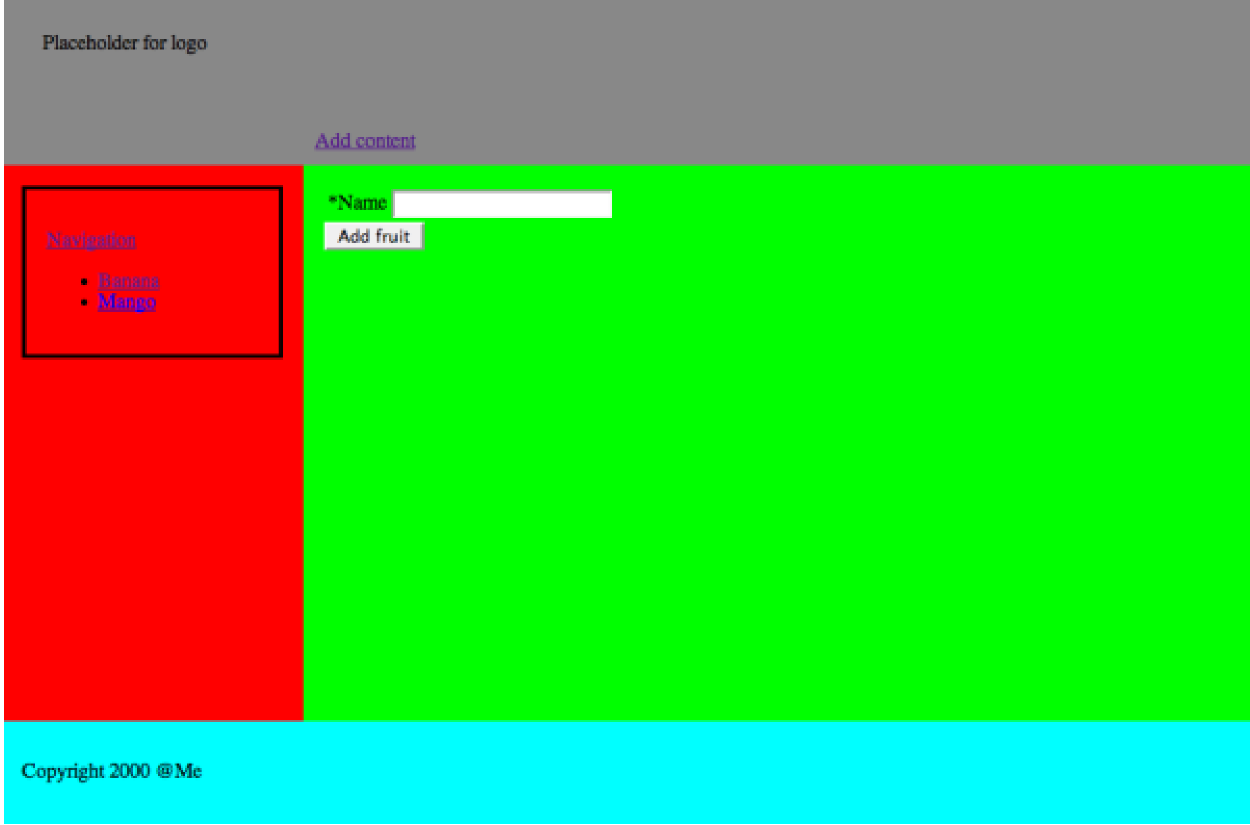
Viewlets can be used instead of macros to
make a flexible layout. This is an example
application that shows how viewlets can be used.

The code can be found here: http://svn.zope.org/grokapps/SimpleViewletDemo

To try it out, do something like this:

```
$ svn co svn://svn.zope.org/repos/main/grokapps/SimpleViewletDemo
$ cd SimpleViewletDemo
$ python bootstrap.py
$ ./bin/buildout
$ ./bin/zopectl fg
```

This how-to explains how this application is built and how viewlets are used to get this layout and behaviour.



4.11.1 Viewlets and Viewlet Managers

A viewlet is a component that represents an HTML snippet. Viewlets are multi adapters (context, request, view, viewlet manager) so they only show up when they are registered for the interfaces or classes that currently applies. Viewlets are grouped using viewlet managers. On the picture above each coloured area is a viewlet manager. There are five viewlet managers and they are referred to by name: header, left, main, footer and head. Only four of them are visible, 'head' is only including stylesheets. The 'left' viewlet manager has two viewlets, a navigation viewlet and a login viewlet, but only one is visible on the picture. The left viewlet manager might show one of the viewlets, both or none of them. It depends completely on how the viewlets are registered. When you click on 'Navigation' in the navigation viewlet the login viewlet will show up. That's because the login viewlet was only registered for the Application model and not the Fruit model.

A viewlet is often associated with a template which will render the HTML and use data from the viewlet object. Instead of using a template it's also possible to have a render method defined in the viewlet. In this example application most viewlets use templates, but there are two that uses the render method instead. You can't use both, grok will complain and refuse to start up.

4.11.2 What happens?

When you visit the frontpage of the application the 'index' view is invoked. The 'index' view uses the master.pt template for its HTML output. The master.pt is what decides the layout of the page. It adds some div tags and asks viewlet managers to put HTML in each slot.

For example, the template asks for HTML output from the viewlet manager called 'left':

```
<tal:left replace="structure provider:left" />
```

This will ask the viewlet manager called 'left' to insert its input here. The viewlet manager will see which viewlets are associated with it and are registered for the current context, request and view. The results (HTML snippets) of all viewlets are concatenated and inserted into the template output.

The master.pt template is used for all views so they all get the same layout. It's the viewlets' responsibility to decide what content to display. When you're not using viewlets (for layout purpose) it's generally the views' responsibility to provide content.

4.11.3 Stylesheets

Viewlets can be used to dynamically add stylesheets or javascripts in the <head> tag of the HTML. In this example application the frontpage will use app.css and when visiting a fruit object it will use fruit.css. That is the reason why the colours change when you navigate to a fruit:

```
class Head(grok.ViewletManager):
    grok.name('head')

class Title(grok.Viewlet):
    grok.viewletmanager(Head)

class AppCSS(grok.Viewlet):
    grok.viewletmanager(Head)
    grok.context(App)

class FruitCSS(grok.Viewlet):
    grok.viewletmanager(Head)
    grok.context(Fruit)
```

The viewlet manager is called 'head' and three viewlets are associated with it. One CSS viewlet is registered for the application model and the other CSS viewlet is registered for the fruit model so they will never be used both at the same time. The title viewlet will be used on all pages because of the grok.context(Interface) directive which is found higher up in app.py. You can add as many viewlets as you want. Each CSS viewlet has a template which will include the CSS link into the HTML.

4.11.4 Forms

Viewlets make it possible to use grok.AddForm in just a part of the page. It's not as easy as it could be, but it works fine and is pretty straightforward:

```
class Admin(grok.View):
    grok.template('master')

class AddFruit(grok.Viewlet):
    grok.viewletmanager(MainArea)
    grok.context(App)
    grok.view(Admin)
```

```
    def update(self):
        self.form = getMultiAdapter((self.context, self.request), name='addfruitform')

        self.form.update_form()

        if self.request.method == 'POST':
            app = get_application(self.context)
            self.__parent__.redirect(self.__parent__.url(obj=app))

    def render(self):
        return self.form.render()

class AddFruitForm(grok.AddForm):
    form_fields = grok.AutoFields(Fruit)

    @grok.action('Add fruit')
    def add(self, **data):
        obj = Fruit(**data)
        name = data['name'].lower().replace(' ', '_')
        self.context[name] = obj
```

First you can see that an admin view is registered. That is what you visit when you click the "Add Content" link in the example application. When you visit the admin page or view it will call the master template. As you can see the AddFruit viewlet is registered for the main content area viewlet manager, the App model class and the 'admin' view. This means that the AddFruit viewlet is only shown when visiting /admin on the root of the site and it will show up in the main content area (the green area).

The AddFruit viewlet just delegates its HTML rendering to the AddFruitFrom class which is the form. If the request method is POST (the user submitted the form) the user is redirected to the frontpage and will see the new fruit in the navigation.

4.11.5 Context dependent viewlets

In this application the main content area shows different text depending on the model you are visiting:

```
class FruitContent(grok.Viewlet):
    grok.viewletmanager(MainArea)
    grok.context(Fruit)
    grok.template('fruit')


    def update(self):
        self.name = self.context.name
```

This viewlet is only registered for the Fruit model. This means that this viewlet will be responsible for showing the relevant information about the currently selected fruit. When visiting the frontpage (the application model) the Content viewlet (see app.py) is used instead.

Notice that there is no special index view for the Fruit model. It uses the same index view as the App object. This is because of the grok.context(Interface) directive high up in app.py. The directive has the effect that the index view is used for all models (contexts).

In the update() method we add all data to self that we want available in the template.

In the login viewlet below we can see that it's associated with the App model and the 'index' view. That means that it will only appear on the frontpage. If we didn't specify grok.view(Index) the login viewlet would still be visible on

the 'admin' view. So to be sure that login should only be shown on the frontpage we need both grok.context() and grok.view():

```
class Login(grok.Viewlet):
    grok.viewletmanager(LeftSidebar)
    grok.context(App)
    grok.view(Index)
    grok.order(2)
```

4.11.6 Ordered viewlets

In the login viewlet above you can see that order 2 is specified. The navigation portlet has order 1. This means that the 'left' viewlet manager will return the output of the navigation viewlet first and then the login viewlet.

4.12 Plugging in new template languages

Replace the default template language.

4.12.1 Introduction

Grok, like the Zope 3 framework on which it is built, uses Zope Page Templates as its default templating language. While you can, of course, use whatever templating language you want in Grok by calling it manually, you can also "plug in" a template language such that both inline templates and templates stored in files are automatically linked with your Views - just like inline grok.PageTemplates and files with the .pt extension are by default.

4.12.2 Existing template extensions

There are already some template language extensions available for Grok. See

- Chameleon: megrok.chameleon http://pypi.python.org/pypi/megrok.chameleon
- Genshi: megrok.genshi http://pypi.python.org/pypi/megrok.genshi
- Jinja: megrok.jinja http://pypi.python.org/pypi/megrok.jinja

Note, that in grok projects you have to include these extensions before you grok your own package, i.e. in configure.zcml a line like:

```
<include package='megrok.genshi' />
```

must appear before everything else to make the extensions work properly.

4.12.3 A simple template engine

For demonstration purposes we create a very plain template engine that returns any text passed formatted uppercase:

```
class UpperCaseTemplate(object):
    """A template engine that renders to uppercase."""

    def __init__(self, sourcetext):
        self._text = sourcetext
```

```
    def render(self, **namespace):
        return self._text.upper()
```

As you can see this is plain Python. In real world usage you will normally use a thirdparty-package like the ones mentioned above which provide 'real' template functionality.

4.12.4 Inline templates

"Inline" templates are templates that you create right in your Python code - for example, by instantiating the default grok.PageTemplate class with a literal string value as its argument. Such templates are automatically associated with nearby View classes: if you create a View named Mammoth and next to it instantiate a template named mammoth, then Grok will use them together.

To enable such automatic association for a new templating language, you need to write a subclass of grokcore.view.components.GrokTemplate. You will need to override three methods. The setFromFilename and setFromString methods should each load the template from disk or a given string, depending on method. Your render method should run the template with the dictionary of values returned by self.getNamespace() and return the resulting string.

Here is an example of a minimal page template integration, assuming that your template engine is available as UpperCaseTemplate:

```
import os
from grokcore.view.components import GrokTemplate

class MyPageTemplate(GrokTemplate):

    def setFromString(self, string):
        self._template = UpperCaseTemplate(string)

    def setFromFilename(self, filename, _prefix=None):
        file = open(os.path.join(_prefix, filename))
        self._template = UpperCaseTemplate(file.read())

    def render(self, view):
        return self._template.render(**self.getNamespace(view))
```

With this class finished you can create an inline template, like this:

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

aview = MyPageTemplate('<html><body>Some text</body></html>')
```

The result of the rendered inline template will appear when you request the @@aview and look like this:

```
<HTML><BODY>SOME TEXT</BODY></HTML>
```

You can also create a filebased template, inline. In this case you only have to pass the filename as filename parameter:

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

aview = MyPageTemplate(filename='lasceaux.html')
```

Here the complete contents of the local file lasceaux.html will appear uppercase when you browse the view.

4.12.5 Templates in the templates/ directory

The most common use case, however, is to place templates for a file foo.py in the corresponding foo_templates directory. Grok, of course, already recognizes that files with a .pt extension each contain a Zope Page Template. To tell Grok about a new file extension, simply register a global utility that generates a MyPageTemplate when passed a filename. That utility needs to implement the ITemplateFileFactory interface which is found in grokcore.view.interfaces.

Let's register a .uct (for 'upper case template') filename extension:

```
from grokcore.view.interfaces import ITemplateFileFactory

class MyPageTemplateFactory(grok.GlobalUtility):

    grok.implements(ITemplateFileFactory)
    grok.name('uct')

    def __call__(self, filename, _prefix=None):
        return MyPageTemplate(filename=filename, _prefix=_prefix)
```

When your module gets grokked, Grok will discover the MyPageTemplateFactory class, register it as a global utility for templates with the .uct extension, and you can start creating .uct files in the template directory for your class.

That's all you basically need! Have fun!

4.12.6 Advanced usage: namespaces

Often you do not only want text to be turned uppercase but produce different output based on other attributes of the context object, the name of the view or whatever. In other words: templates are often used in a certain environment or namespace.

A namespace is actually only a dict containing environment parameters like the context object used, the view and similar often Zope-related things. Your template engine might expect those (or other) parameters to process templates correctly or provide useful stuff.

Here is another (silly) template plugin, which replaces all occurences of <TIMESTAMP> with a timestamp passed via namespace. The timestamp is expected to be the usual 9-tuple:

```
import time
class DatedTemplate(object):
    """A template engine that replaces <TIMESTAMP>s."""

    def __init__(self, sourcetext):
        self._text = sourcetext

    def render(self, **namespace):
        timestamp = time.strftime('%c', namespace['timestamp'])
        return self._text.replace('<TIMESTAMP>', timestamp)
```

Our template extension now has to provide this timestamp in the namespace. The handling happens in the render() method only:

```
import os
from grokcore.view.components import GrokTemplate

class MyTimestampedTemplate(GrokTemplate):

    def setFromString(self, string):
        self._template = DatedTemplate(string)
```

```
    def setFromFilename(self, filename, _prefix=None):
        file = open(os.path.join(_prefix, filename))
        self._template = DatedTemplate(file.read())

    def render(self, view):
        namespace = self.getNamespace(view)
        namespace.update(dict(timestamp= time.gmtime()))
        return self._template.render(**namespace)
```

Of course you can modify the values in namespace during rendering.

Any view that uses this template type like this:

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

adatedview = MyTimestampedTemplate('The datetime is: <TIMESTAMP>')
```

will now get something like this:

```
The dateime is: Wed Sep 23 23:23:23 2023
```

depending on the datetime of access. There is nearly no limit for more useful scenarios.

4.13 How to internationalize your application

In this howto, you will learn how to internationalize your code, extract translatable strings and translate your application into an other language.

In this example, a project HelloWorld was created with grokproject HelloWorld.

4.13.1 Internationalizing Strings in Python Code

You need to use a MessageFactory that marks all the strings you want to be able to translate:

```
__init__.py
from zope.i18nmessageid import MessageFactory
HelloWorldMessageFactory = MessageFactory('helloworld')

app.py
from HelloWorld import HelloWorldMessageFactory as _
```

Now to internationalize your strings in Python code, change:

```
message = u'Hello World'
```

to:

```
message = _(u'Hello World')
```

You will have in your generated POT:

```
msgid "Hello World"
msgstr ""
```

or you can define both a msgid and default:

```
message = _(u'hello_msg', default=u'Hello World')
```

You will have in your generated POT:

```
#. Default: "Hello World"
msgid "hello_msg"
msgstr ""
```

If you need to insert some data into your message you can accomplish this, too

```
who = u'World'
message = _(u'hello_msg', default=u'Hello ${who}', mapping={ 'who' : who })
```

4.13.2 Example

In your code you will have:

app.py

```
class Index(grok.View):
    def update(self):
        self.message = _(u'Hello World')
```

And your templates:

app_templates/index.pt

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

To internationalize text in a page template, you add the i18n:domain attribute to the html-tag. Then you mark each tag who's text you want to translate with the attribute i18n:translate:

```
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en"
    xmlns:tal="http://xml.zope.org/namespaces/tal"
    xmlns:i18n="http://xml.zope.org/namespaces/i18n"
    lang="en"
    i18n:domain="helloworld">
<head>
    <title>My project</title>
</head>
<body>
  <a tal:attributes="href python: view.url('creategame')"
     i18n:translate="link_create_new_game"
     >Create new game</a>
  <a tal:attributes="href python: view.url('createcontact')"
     i18n:translate=""
     >Create new contact</a>
</body>
</html>
```

You will have in your generated POT:

```
#. Default: "Create new game"
msgid "link_create_new_game"
msgstr ""

msgid "Create new contact"
msgstr ""
```

Using Zope 3 style translation domains

4.13.3 Resource

http://worldcookery.com/files/fivei18n/ (Zope 3 part)

4.13.4 Setting the locales directory

To activate translations you need to let Grok know what translation domain you are using and where the translation files are. src/helloworld/configure.zcml should look like this:

```
<configure xmlns="http://namespaces.zope.org/zope"
           xmlns:grok="http://namespaces.zope.org/grok"
           xmlns:i18n="http://namespaces.zope.org/i18n">
  <include package="grok" />
  <includeDependencies package="." />
  <grok:grok package="." />
  <i18n:registerTranslations directory="locales" />
</configure>
```

Extracting strings with z3c.recipe.i18n:i18n

Grokproject has created two files in your HelloWorld/bin-folder called: i18nextract and i18nmerge.

1 – Building the POT file

In your project directory, to extract all internationalizable strings:

```
./bin/i18nextract
```

This creates the src/helloworld/locales directory and generates src/helloworld/locales/helloworld.pot

2 – Creating new translation

Begin to translate into french:

```
mkdir -p src/helloworld/locales/fr/LC_MESSAGES/
msginit -i src/helloworld/locales/helloworld.pot -o src/helloworld/locales/fr/LC_MESSAGES/helloworld
```

translate src/helloworld/locales/fr/LC_MESSAGES/HelloWorld.po with poEdit, KBabel...

NOTE: If you use an editor such as poEdit, it will generates your compiled .mo translation files when saving, this saves you having to compile them manually.

To use default text strings that you have put in your PageTemplates and python-code you need to create a translation for your default language and generate the .mo file, BUT you don't translate any of the strings.

http://markmail.org/message/ozesj6jxqlbugtql

3 – Updating existing translation

To update all PO files, in your buildout directory:

```
./bin/i18nextract
./bin/i18nmergeall
```

4 – Generating MO files

To finish, you have to generate a MO file for each PO file:

```
cd locales/fr/LC_MESSAGES/
msgfmt -o helloworld.mo helloworld.po
```

If you want to generate all mo file, you can use this command:

```
for po in `find . -name "*.po"` ; do msgfmt -o `dirname $po`/`basename $po .po`.mo $po; done
```

As mentioned previously, this is done automatically by poEdit when you save changes.

Restart your server and see the translation in french.

There is a new feature in zope.i18n 3.5.0, the changelog says: "Feature: Added optional automatic compilation of mo files from po files. You need to depend on the zope.i18n [compile] extra and set an environment variable called zope_i18n_compile_mo_files to any True value to enable this option."

4.13.5 How Does the Server Know What Language is Used?

Grok/Zope3 uses the Accept-Language header of the HTTP-request to determine what language to present to the user. This is set by the browser and in Macosx you change this in System Preferences / International / Language. In order to test your translations you probably need to change there and restart the browser.

So the HTTP request comes with a field for the user's preferred language, set by the browser.

You can retrieve the user's preferred languages like

```
import grok
from zope.i18n.interfaces import IUserPreferredLanguages

class ShowLanguageView(grok.View):
    grok.context(grok.Application)

    def render(self):
        return str(IUserPreferredLanguages(self.request).getPreferredLanguages())
```

If you want to override these languages in your application, e.g. by setting it to a fixed value or by fetching the user's preferred language from some storage, you can do it like this:

```
import grok
from zope.publisher.interfaces.http import IHTTPRequest
from zope.i18n.interfaces import IUserPreferredLanguages

class PreferredLangugageAdapter(grok.Adapter):
    grok.context(IHTTPRequest)
    grok.implements(IUserPreferredLanguages)

    def getPreferredLanguages(self):
        # example: fix language to German
        #  The sequence is sorted in order of quality, with the most preferred
        #  languages first.
        return ['de-de', 'de']

        # If you have a custom principal object where you save the user's pre-
```

```
        #  ferred language on, you could also do something like this:
        #
        # request = self.context
        # return [ request.principal.preferred_language, 'en-us', 'en' ]
```

Now your application will use the best fitting language computed from the values returned by this adapter.

To see how you can internationalize and localize your page templates, please the corresponding "Internationalization" chapter.

For more information be sure to check out the zope.i18n package: http://pypi.python.org/pypi/zope.i18n .

4.14 Using sources in your forms

Author jmichiel

This How To explains how one can use the zc.sourcefactory package to fill out lists and choices with dynamic content.

4.14.1 Prerequisities

You need to know how to work with zope.schema and grok.Form or its derivatives.

4.14.2 Step by step

Imagine you are writing an Issue Tracker in Grok. Your definition of an 'issue' might look like this:

```
class IIssue(Interface):
        title = schema.TextLine(title=u'Title', description=u'The issue in short')
        severity = schema.Choice(values=(u'Minor', u'Major', u'Blocker'), title=u'Severity',
                                                 description=u'How severe is this issue')
        description = schema.Text(title=u'Description', description=u'The complete description of the
```

(Of course an issue should also have a state, but that is besides the focus of this howto, so I left it out to keep things simple). The *severity* attribute is the one of interest here. We used a hard-coded tuple to define the possible options. Hard-coded generally is not such a bright idea. Imagine you have a user who wants 5 levels of severity? He would be forced to edit code! This is where sources come in!

Installing zc.sourcefactory

Edit your setyp.py and add the zc.sourcefactory to *install_requires*:

```
install_requires=['setuptools',
                  'grok',
                  'grokui.admin',
                  'z3c.testsetup',
                  'zc.sourcefactory',
                  # Add extra requirements here
                  ],
```

Note: for Grok 1.0a1, at the time of writing, if you would not run buildout, it would get you zc.sourcefactory version 0.4.0, which is, alas, incompatible with some packages Grok 1.0a1 installs by default. To counter this add this line in the [versions] section of your versions.cfg: zc.sourcefactory = 0.3.5

Now run bin/buildout.

Using Sources

Now we're ready to start using Sources! Add this to your code:

```
from zc.sourcefactory.basic import BasicSourceFactory

class SeveritySource(BasicSourceFactory):
    def getValues(self):
        return (u'Minor', u'Major', u'Blocker')
```

And change your schema-definition of severity to this:

```
severity = schema.Choice(source=SeveritySource(), title=u'Severity', description=u'How severe is this
```

This tells the Choice field to get its values from the *SeveritySource*. The getValues function is needed to get at the actual values this source is all about and should return something iterable. Of course this is still hard-coded, but it already separates the definition of severity away from the definition of Issue. This is also a great way to do some agile coding: first make a hard-coded source so you can implement Issue, and implement the whole severity part later, without needing to change your Issue definition again.

Making the Source truly Dynamic

Imagine you implemented Severity like this:

```
class ISeverity(Interface):
    label = schema.TextLine(title=u'Label')
    description = schema.Text(title=u'Description')
```

and that you have a SeverityContainer called 'severities' under your application, with the above mentioned severities. Now you can change your SeveritySource to this:

```
class SeveritySource(BasicSourceFactory):
    def getValues(self):
        return grok.getSite()['severities'].values()
```

However if you try your issue add form now, you would see a dropdown list with 3 empty values. What went wrong? The getValues function now returns objects of class Severity, but the droplist doesn't know how to display these! To solve this you can add a getTitle function to your SeveritySource:

```
def getTitle(self, value):
    return value.label
```

This function accepts a value from the list of values that is returned by getValues, and allows you to return an appropriate title for it. Checking the add form now gets us back where we started. But if you now add a Severity to your SeverityContainer, it will show up in this list, without having you changing any code!

4.14.3 Further information

The docs of the zc.sourcefactory package can be found in the PyPI: http://pypi.python.org/pypi/zc.sourcefactory/0.3.5 .

4.15 Use the same view in multiple models

Author Peter Bengtsson

If you have multiple models each with their own views, perhaps you want all views to have access to one view in common.

4.15.1 Purpose

You want all views in all models of an application to have access to a view without having to redefine this view in each model. For example you have a master template that defines the viewlet managers or `METAL` macros and you want all "sub views" to be able to use this.

4.15.2 Prerequisities

Suppose you have a simple Grok application up and running and you've defined another model other than the application one called `Term` in a file called `term.py` like this:

```
## app.py

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

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

class Master(grok.View):
   # see app_templates/master.pt
   pass

-------%<-----------------------------

## term.py

class Term(grok.Model):
   pass

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

Now, the `App` model has two views: `index` and `master`. The `Term` model has only one view: `index`. In both files called `index.pt` it says something like this on the very first line:

```
<html metal:use-macro="context/@@master/macros/page">
```

That works fine for `app_templates/index.pt` but will give you a `TraversalError` if you try to do the same in `term_template/index.pt`.

4.15.3 Step by step

The solution is really simple. Basically, change the `Master` view to this:

```
from zope.interface import Interface

class Master(grok.View):
   grok.context(Interface)
```

With this you'll be able to reach `context/@@master` in any of the templates of your application which achieves the goal. Think of `grok.context(...)` as a filter mechanism that attaches itself into the "namespace tree" of your application at a certain "branch". By saying the context is `Interface` you're almost attaching the view to the trunk of the trunk.

4.15.4 Further information

An alternative solution would be to use a base class and attach the `Master` view to that. Then you let all other models also subclass this base class and they get the shared functionality too. That approach has the advantage that it's more "standard pythonic" but you're then not using the powerful patterns that Grok offers plus you have to remember to add the base class on all other models which adds noise to the code.

4.16 Creating forms with megrok.z3cform

Author timo

4.16.1 Purpose

There are different ways to create forms in Grok. The z3c.form form building framework provides a flexible and powerful way to create forms. You can find the code for this tutorial here: http://svn.zope.org/repos/main/grokapps/tutorialmegrokz3cform/trunk/

4.16.2 Prerequisities

Before we can start to create forms, we need to create a grok project with the necessary dependencies.

Creating a Grok Project:

```
grokproject tutorialmegrokz3cform
```

Adding megrok.z3cform package Dependencies to our Buildout

Since the megrok.z3cform.* packages are not officially released yet, we use mr.developer to automatically check out the packages from the Zope Subversion repository. We do this by adding the following lines to our buildout.cfg, right after the "versions=versions" line:

buildout.cfg:

```
[buildout]
...
versions = versions
extensions += mr.developer
auto-checkout =
    megrok.z3cform.base
    megrok.z3cform.layout

[versions]
z3c.pagelet = 1.0.3

[sources]
megrok.z3cform.base      = svn http://svn.zope.org/repos/main/megrok.z3cform.base/trunk
megrok.z3cform.layout    = svn http://svn.zope.org/repos/main/megrok.z3cform.layout/trunk
```

```
# do "bin/develop up -v" to update all the checkouts
...
```

We also want to use megrok.layout in order for our layout, so we add it to the package dependecies of our setup.py:

setup.py:

```
install_requires=[...
                  # Add extra requirements here
                  ...
                  'megrok.layout',
                  'megrok.z3cform.base',
                  'megrok.z3cform.layout',
                  ],
```

Run buildout to fetch the dependencies:

```
$ bin/buildout
```

4.16.3 Creating the Form

After creating the initial package, we can start to build our application by defining an interface. We will build a very simple database application to store contacts.

Creating an Interface

We define an interface for a contact with a subject and a text field.

interfaces.py:

```
import os
import zope.interface
import zope.schema

class IContact(zope.interface.Interface):
    """A z3c.form contact form."""

    name = zope.schema.TextLine(
        title=u'Name',
        description=u'Name of the person.',
        required=True)

    description = zope.schema.TextLine(
        title=u'Description',
        description=u'Description of the person',
        required=True)
```

Creating a Contact Class

Now we create a contact class that implements the interface we just defined.

contact.py:

```
import grok
import persistent
import zope.interface
```

```
from zope.schema import fieldproperty
from examplemegrokz3cform import interfaces

class Contact(grok.Model):
    grok.implements(interfaces.IContact)

    name = fieldproperty.FieldProperty(interfaces.IContact['name'])
    description = fieldproperty.FieldProperty(interfaces.IContact['description'])

    def __init__(self, name, description):
        self.name = name
        self.description = description
```

Creating Forms

Now we can create add, edit, and display forms for our contact class.

browser.py:

```
import grok
import megrok.z3cform.base

import contact
import interfaces
import datetime

from z3c.form import field, form, button, widget
from examplemegrokz3cform.app import Examplemegrokz3cform
from grokcore.component import global_adapter
from z3c.form.interfaces import IAddForm

class ContactAddForm(megrok.z3cform.base.PageAddForm):
    """ A sample add form."""
    grok.context(Examplemegrokz3cform)

    label = u'Contact Add Form'
    fields = field.Fields(interfaces.IContact)

    def create(self, data):
        return contact.Contact(**data)

    def add(self, object):
        count = 0
        while 'contact-%i' %count in self.context:
            count += 1;
        self._name = 'contact-%i' %count
        self.context[self._name] = object
        return object

    def nextURL(self):
        return self.redirect(self.url(self.context[self._name]))


class ContactEditForm(megrok.z3cform.base.PageEditForm):
    grok.context(contact.Contact)
    grok.name('edit.html')
    form.extends(form.EditForm)
```

```
    label = u'Contact Edit Form'
    fields = field.Fields(interfaces.IContact)

    @button.buttonAndHandler(u'Apply and View', name='applyView')
    def handleApplyView(self, action):
        self.handleApply(self, action)
        if not self.widgets.errors:
            self.redirect(self.url(self.context, name='index'))


class ContactDisplayForm(megrok.z3cform.base.PageDisplayForm):
    """ A simple Display Form"""
    grok.context(contact.Contact)
    grok.name('index')
    template = grok.PageTemplateFile('display.pt')
    fields = field.Fields(interfaces.IContact)
```

We have to add a default form layer, otherwise you will get a ComponentLookupError. (Why is this? Anyone?)

configure.zcml:

```
<configure xmlns="http://namespaces.zope.org/zope"
           xmlns:grok="http://namespaces.zope.org/grok">
  <include package="grok" />
  <include package="megrok.z3cform.base" file="default_form_layer.zcml" />
  <includeDependencies package="." />
  <grok:grok package="." />
</configure>
```

4.17 Generate URLs with the url() function in views

Author Peter Bengtsson, Jan-Jaap Driessen

The 'url()' function makes it really easy to generate a URL for an object or by name or a combination of an object and a name. We'll also show how to append a query string to the URL by passing a dictionary to the 'url()' function.

4.17.1 Purpose

You want to easily construct URLs in your view classes and view templates.

4.17.2 Prerequisities

A running Grok site with one or more views.

4.17.3 Step by step

The url() function is really easy to use and you should be able to throw it an object, a name or neither and it will construct the URL for you. If you omit the object parameter it will default to "self" where "self" is either the object you're doing this in or the view you're calling it form. As an added bonus you can use url() to construct a URL with a CGI query string that is URL encoded.

Suppose you have a view class called "FooView" and an accompanying template file called fooview.pt. If you want to add a URL to the template itself you can do that with this TAL code.

The following combinations of object, name and data are allowed:

```
<a tal:attributes="href view/url">Link to this page</a>
```

This is the same as using the python: syntax:

```
<a tal:attributes="href python: view.url()">Link to this page</a>
```

When not specifying the object, url() defaults to the *view* itself:

```
<a tal:attributes="href python: view.url(view)">Link to this page</a>
```

To make a url to the container of the view, use *context*:

```
<a tal:attributes="href python: view.url(context)">Link to the container</a>
```

To make a url to a different view on the current container, use *context* and name:

```
<a tal:attributes="href python: view.url(context, 'add')">
  Link to adding to the container
</a>
```

Using the *data* argument, a query string is appended to the generated URL:

```
<a tal:attributes="href python: view.url(context, 'add', data={'a':1, 'b':2})">
  Link to adding to the container, with a=1&amp;b=2
</a>
```

You can supply unicode and lists in the data argument. These are properly escaped.

```
<a tal:attributes="href python: view.url(data={'name':u'Pjötr'})">
  ?name=Pj%C3%B6tr
</a>

<a tal:attributes="href python: view.url(data={'id':[1,2,3]})">
  ?id=1&amp;id=2&amp;id=3
</a>

<a tal:attributes="href python: view.url(data={'name':[u'Pjötr', 'Sonja']})">
  ?name=Pj%C3%B6tr&amp;name=Sonja
</a>
```

4.17.4 Further information

You can still use Zope's automatic type casting of parameters:

```
<a tal:attributes="href python: view.url(data={'amount:int': 25})">
  ?amount%3Aint=25
</a>
```

CHAPTER

FIVE

LIFE CYCLE OF GROK APPLICATIONS

Contents:

TODO: Break out parts not covered by other docs.

5.1 The lifecycle of a Grok application

Author reinoud

This how-to tries to give an overview of what has to be done in which phase of the lifetime of a typical Grok application.

5.1.1 Step by step

An application usually has different phases in its lifetime.

5.1.2 Starting development of a new application

Usually, an application starts with a realy bright idea. Let's suppose you have that idea and have made the decision to implement it in Grok. So you want to start the development. Now there is a decision to make: do you want to use buildout or virtualenv (or both?).

Virtualenv is a tool to build an isolated local Python environment that is not inflenced by the global site packages. It allows you to install (versions of) packages that are local to the aplication you are developing. It is really easy to use and great if you want to start coding right now and not bother with buildout configurations (yet). It provides a sandbox where you can safely plan without causing problems outside it. Buildout is a advanced tool to create a virtual environment, install all dependencies of your application in a reliable and reproducable way. It is usable both for development and deployment. It is more difficult to use than virtualenv, but provides lots more flexibility. It is most likely the tool to use for serious applications. It allows for formalized upgrade procedures, version pinning and is extendable with external recipes. It can maintain different configurations for development and production environments. The combination of the two tools is preferable. If you use builduit within a virtualenv environment, you are sure to explicitely include all the needed packages that might be installed in your operating system, but are not on the production server where your application with be rolled out. You will start with an empty list of site packages so you will have to include the needed ones.

And it gets even better: grokproject is a tool around buildout that sets up a Grok environment for you. It saves you a lot of typing and you use the normal builduit after creation of your project.

So in short (assuming you already have easy_installed virtualenv and grokproject):

```
$ mkdir myproject
$ virtualenv --no-site-packages myproject
New python executable in myproject/bin/python2.4
Installing setuptools............done.
$ . myproject/bin/activate
(myproject)$ grokproject --no-buildout myproject
Enter module (Name of a demo Python module placed into the package) ['app.py']:
Enter user (Name of an initial administrator user): admin
Enter passwd (Password for the initial administrator user): mysecretpassword
Enter eggs_dir (Location where zc.buildout will look for and place packages) ['/home/username/buildou
(myproject)$ cp myproject/buildout.cfg myproject/buildout.org
(myproject)@iss$ sed 's/^eggs-directory = .*//' myproject/buildout.org  > myproject/buildout.cfg

(the last two lines remove a line from buildout.cfg that you do not want)
```

5.1.3 Developing with a team and version control

Even when you're working alone, it is a very good idea to use version control. We use the svn tool here, for users of other systems the commands might be slightly different. What is important is what you do and don't want to put in version control.

Since you've not run buildout at this moment, no auto generated content or downloaded dependencies are present. This might be a good time to run your first checkin. Checkin these files:

svn add buildout.cfg setup.py src/myproject Other developers start basically with the same steps by creating a virtualenv and a grokproject on their own environment. Then they do a checkout of your code and start coding.

5.1.4 Running buildout

Before you can actually run buildout you first have to install it:

```
easy_install zc.buildout
```

And after that you can run it:

```
bin/buildout
```

(this can take a while; a lot of software is downloaded)

5.1.5 Make a release

After some coding, you and your team probably want to release something. Of course you have the latest version of all the softeware of your team and all test pass. Do not forget to fill in all the required information in setup.py, and write release notes. Usually at this moment you create a tag in the revision control system to be able to easily identity the current state of all the files (something like !.0-RELEASE)

In this example we will make both a source and a binary release:

```
(myproject)$ python setup.py sdist
running sdist
running egg_info
[...]
Writing myproject-0.0/setup.cfg
creating dist
tar -cf dist/myproject-0.0.tar myproject-0.0
```

```
gzip -f9 dist/myproject-0.0.tar
removing 'myproject-0.0' (and everything under it)
```

Now you have a tgz file in your dist directory. To change the version number edit your setup.py (before creating the dist). Since this is a python script you can off course automate this to get the revision number or tag name from a version control substitution like $Revision:$. This left as an excersize to the reader.

To make a binary (egg) distribution:

```
(myproject)reinoud@iss$ python setup.py bdist_egg
running bdist_egg
running egg_info
[...]
copying src/myproject.egg-info/requires.txt -> build/bdist.freebsd-6.2-RELEASE-i386/egg/EGG-INFO
copying src/myproject.egg-info/top_level.txt -> build/bdist.freebsd-6.2-RELEASE-i386/egg/EGG-INFO
creating 'dist/myproject-0.0-py2.4.egg' and adding 'build/bdist.freebsd-6.2-RELEASE-i386/egg' to it
removing 'build/bdist.freebsd-6.2-RELEASE-i386/egg' (and everything under it)
```

Now you have an egg in the dist directory.

5.1.6 Creating a local package index

(todo: make a HTML file with links)

5.1.7 Specifying search paths for development eggs

In the [buildout] section: devel = ../foo

5.1.8 Namespace packages

Add this to your setup.py:

namespace_packages = ['foobar'], And add this to your src/foobar/__init__.py:

__import__ ('pkg_resources').declare_namespace(__name__)

5.1.9 Extending buildout with new recipies

(todo)

5.1.10 Adding scripts to the bin directory using entry points

Add this to your setup.py:

```
entry_points = """
[console_scripts]
foo=myproject.scripts.foo:main"""
```

When you run buildout again you will find a new script bin/foo that will call the main function in src/myproject/scripts/foo.py

5.1.11 Upgrading your buildout.cfg

5.1.12 Setting up a production environment and installing the first release

Note that there is a limit to what you can rollout within a Python application. You might need shared libraries to connect to a relation database or shared libraries that are needed for XML parsing. Since every operating system uses its own way for the installation of packages, you cannot easily automate this in a cross-platform way. However: somtimes a different version of a shared library is needed than the one that might be installed on the system (and might be needed by other applications). You can include a recipe to build shared libraries in you buildout configuration. But not all operating systems include a C compiler by default...

In this example we'll try to avoid needing root rights as much as possbile. This makes it easier to install multiple applications under different user accounts, or enables you to upgrade your application without the need to wake-up the system administrator... Having root rights only makes it easier.

Note that the people that maintain the production server might have other ideas than developers about what should be installed where. While a deveper might be used to run everything from (a subdirectory of) his home directory on a unix system, a system administrator might prefer a path somewhere near /usr/local/application. Another point is whether to use centrally installed python packages for all applications on a server, or let every application use its own. The virtualenv makes is possible to use your own, and run the application from a central /usr/local or from a home directory of a dedicated account on the server. The developer does not have to bother with such decisions. If the system administrator wants to use centrally administered packages, he should not use virtualenv at installation time. That might save diskspace, but can also cause problems with some package managers on some Unix systems.

First we start with a clean virtual environment to make sure the application does not bother existing applications and vice versa:

```
easy_install virtualenv
virtualenv --no-site-packages myproject
. myproject/bin/activate
```

We can now either use an SVN checkout, which contains both the buildout configuration and and the application packages, or we can use the source distribution. In case you want to use SVN, the procedure is the same as in a development environment.

We will assume that you want to deploy an official distribution. In this case you want to have received the buildout.cfg from the developers.

To start a Grok application, you use paster in the bin directory. To start it as a daemon run

> bin/paster serve parts/etc/deploy.ini –daemon

5.1.13 Starting the application from a @reboot crontab entry

If the application developer has root rights on the server, he will typically start the daemon from something like /usr/local/etc/rc.d, according to what is usual on the Unix flavor.

But when you have installed the application in the home directory of a user, and do not have root rights you can also use another trick: create a crontab entry (for that user) that starts the application at boot time. Most modern crontab implementations allow something like this:

```
@reboot /home/myapplicaionuser/myapplication/bin/paster serve parts/etc/deploy.ini --daemon
```

5.1.14 Apache config

It is very common to run a Grok or Zope application behind a Apache server. Usually the Grok process listens to a not-reserved high portnumber (like 8080) that any user can open. It is a good idea to only let the localhost connect to

that port. In that way the Apache server (that is started with root privileges to open ports like 80 or 443) can redirect the requests with mod_rewrite.

This usualy means that at installation time the apache configfile has to be edited once.

5.1.15 Install an upgrade

The nice thing about buildout is that most of the things are installed automatically. But that also includes your configuration files. So if you have edited them, be sure to back them up before upgrading.

5.1.16 Rollback an upgrade

5.1.17 Administration of a Grok application

Once the application is running, the administrator still has some work to do:

5.1.18 What to backup

Grok applications usually store at least something in the Zope database. That is a file called Data.fs in the var/filestorage subdirectory of your application.

Log files go (by default) to the var/log directory.

Configuration files are in the same place: parts/etc

5.2 Selecting the port and interface where Grok listens

Author Brandon Craig Rhodes

Hosts which use the Transmission Control Protocol (TCP) atop the Internet Protocol (IP) for communication receive connections through devices called *interfaces* which each specify a *port numbers* that they want to connect to. For example, on the machine on which I am typing, there are several interfaces (this is Linux, so I can list them with the old-fashioned command `ifconfig -a` or, instead, with the modern command `ip a`):

```
lo    ("loopback" interface)
eth0  (my Ethernet card)
wlan0 (wireless LAN card)
```

Port numbers are simple integers, and generally each new application that comes out on the Internet claims a port number on which it listens for incoming connections. Some of the most popular, just to give some examples, are:

```
22   SSH (Secure Shell)
25   Email server
53   Domain name service (DNS)
80   Web server (HTTP)
443  Secure web server (HTTPS)
```

On typical Unix systems, you can see a full list of port numbers in the file `/etc/services`.

Whenever you run a Grok instance, it will open a port on your machine on which it can receive, and respond to, incoming HTTP requests. Its default behavior is to listen for connections coming in on *any* interface as long as the connection is aimed at port 8080, which is a common port for use by unofficial web services since it looks kind of like the number 80, but is greater than 1024 and is, therefore, a port that normal users can open without needing administrator privileges on the machine.

But very often you'll want to change this default.

First, it's not terribly safe to have a web application you're still developing listening to connections from the whole entire world! You will probably want to have you development Grok instances just listen on the "loopback" interface of your computer, since that way Grok will only see connections coming from other programs on your computer (like your web browser), and not connections from hackers scanning your machine to find web applications to explore and exploit.

Secondly, you might want to change the port number because you find that port 8080 is already in use by another process. Its official designation is that it's the HTTP proxy port, and if your machine is already running a web proxy there, your Grok application might not even start.

Thirdly, if you want several Grok instances running on you machine, then only the first one can grab and use port 8080, so the others will have to be configured to use some other port.

The solution is to edit your project's `buildout.cfg` and look for the `[zopectl]` section, which defines basic information about the little web server that Grok comes with by default. You need to add a line that looks like:

```
address = localhost:8012
```

This will make the whole `[zopectl]` section, at least on the version of Grok I'm using, look something like:

```
[zopectl]
recipe = zc.zope3recipes:instance
application = app
zope.conf = ${data:zconfig}
address = localhost:8012
```

The `address` can either be a plain port number, like `8012`, or can specify both an interface's IP address followed by a port number, like `localhost:8012` or `my.host.com:80`.

After changing the `address`, **you must re-run buildout for your change to have any effect!** There have been many occasions on which I have adjusted this setting, and then been puzzled about why it is having no effect, because I forgot to re-run the `./bin/buildout` command.

Note that, if you include the IP address and colon instead of just a bare port number, you are indirectly specifying which interface Grok will listen on by naming the IP address associated with the interface. This might confuse you at first. It means that you are *not* giving an actual interface name — Grok will not understand if you tell it to listen on `lo` or `eth0` or `wlan0`. Instead, you have to look at the output of a command like `ifconfig -a` or `ip a`, find the IP address associated with the interface, and then name either the raw IP address or else a host name that points at that IP address.

For example, since most machines define `localhost` as being the IP address `127.0.0.1` (check out the file `/etc/hosts` on a typical Unix machine to verify that your machine defines `localhost` this way), the following two `address` lines are equivalent:

```
address = 127.0.0.1:8080
address = localhost:8080
```

Note that all of this about the `buildout.cfg` only applies if you are using Grok's built-in web server. If you are instead running it under a paste-based WSGI pipeline, or inside of Apache, or under a CherryPy web server, or through some other exotic mechanism, then what you put in the `[zopectl]` section of your `buildout.cfg` will probably not matter at all. (Or your `buildout.cfg` might not even have that section!) Instead, you would consult the documentation for the particular web server you're using to find out how to set its listening address.

But the central lesson of this HOWTO is the same regardless of the web server you use: letting a web application listen on all of your computer's interfaces is dangerous if the application is not finished, polished, tested, and ready for attacks from the very clever criminals on the Internet who will test and probe every page, every form, and every script for some way to exploit a flaw. Until your application is ready to go live, you should have it listen only on `localhost` for connections coming directly from your machine.

If your web browser runs on a different machine than the one on which you're developing your Grok application, but you still want the safety of having Grok listen only on the loopback interface, there are ways that you can VPN or tunnel a connection to the development machine so that your web browser's connections look like they're coming from "localhost". Though port-forwarding is a big issue that I cannot cover in detail here (try a Google search), I should mention that I myself usually develop my Grok apps on a server that's a different machine than my desktop where Firefox runs, and that the following command lets me cause port 8080 on my desktop to actually make a request to port 8080 on the server from its own loopback interface:

```
ssh -N -L 8080:localhost:8080 my.server.com
```

Note that a properly configured firewall is another way to prevent hosts out on the Internet from connecting to your development Grok instances.

Finally, even when someone is finished and ready to deploy their Grok app, an experienced web developer will often leave it listening on `localhost` and protect it by putting it behind another HTTP implementation as a proxy — such as Apache, Squid, Pound, or Varnish. But that is a large topic that deserves a HOWTO of its own!

5.3 Install Grok on MS Windows

Author Roger Erens

This HowTo gives instructions for installing Grok on the various flavours of MS Windows.

Please note that this HowTo is a _WORK IN **PROGRESS_** during the alpha releases of Grok version 1.0 and grokproject version 1.0.

TBD denotes To Be Discussed/Determined/Developed/Described/Done

5.3.1 Procedure

0. You can install the software in a folder for which you have write access rights. If you want all users on your machine to be able to run Grok, make sure you have administrator rights during the installation.
1. Install Python. Currently, version 2.6.x and version 2.5.x are supported. Note that these links point to installers for the ordinary x86 processor architecture.
2. Install version 2.6 or version 2.5 of the win32all package that goes with the python version you chose above.
3. Install setuptools2.6 or setuptools2.5.
4. Open up a command window and change to the folder where you want to place the sandbox that will contain your Grok project(s). In this HowTo we will assume this folder is C:\

   ```
   cd c:\
   ```

5. Install virtualenv by using the command:

NOTE: As of Grok 1.2, you do not need to use virtualenv. Grok is automatically isolated from the system python environment.

   easy_install virtualenv

6. Create an isolated environment (sandbox) using the command:

   ```
   virtualenv <grok_sandbox>
   ```

   where you substitute <grok_sandbox> with your name of choice for the sandbox.

7. Activate the environment:

```
cd <grok_sandbox>
Scripts\activate
```

8. Obtain grokproject, the tool that will download and install Grok for you. It will also create a Grok project for you to get you started:

```
easy_install grokproject
```

Work-around for issue with step 9.

When the process stalls because of some error saying that a temporary folder cannot be removed, we have currently the following work-around:

```
cd <project_name>
<project_name>\bin\buildout.exe
```

9. Set up your project.

   Use the grokproject tool to set things up for you:

```
grokproject <project_name>
```

   Where you substitute <project_name> with your name of choice for the project.

   Answer the questions.

10. Starting up Grok

    Start the server:

```
cd <project_name>
bin\paster serve parts\etc\deploy.ini
```

    Visit Grok's applications page to see what you have achieved!

5.3.2 Reporting issues

Report any issue in the bug tracker, please put the word 'Windows' in its title. Also, feel free to notify us on the mailing list.

Finally, there is also the possibility to share your insights with us on the Wiki-page.

5.4 Placing your Grok project under version control

Author Brandon Craig Rhodes

Very often, programmers find themselves wanting to save their projects to a version control system like Subversion, Bazaar, Git, or CVS. But it can be confusing to know what to save out of all the files that grokproject creates under your project directory! And, if you version-control the wrong ones, you could find that your project can be difficult to get working again when you check it back out.

The key is to recognize that much of your Grok project is created automatically from its buildout.cfg, and therefore should not be version-controlled. There are three reasons for not saving these files. First, they are each already defined or described by your buildout.cfg, and it is therefore redundant to place them in persistent storage. Second, they

are often files which need to be different on every computer on which you place them because they have file names or paths to executables (like the Python interpreter) that are likely to be different on the various machines to which people check the project out. And, third, having the files in version control means that your version-control system and buildout will be constantly fighting as they overwrite the files' contents with new and old data, which can make deployment quite difficult.

Therefore, the files that you want to version-control are generally these:

```
# Save these files

buildout.cfg
setup.py
src/yourproject/everything except *.pyc files
```

That is, you want to save the two files buildout.cfg and setup.py that define your project and describe how it should get built and run, and you want to save all of the source files and application content that you have generated under the src/ directory. But you should not save anything inside of the following directories, which are all auto-generated files that might have to look different on each system on which you check out and build your Grok project:

```
# Do NOT save these files

bin/
develop-eggs/
parts/
src/yourproject.egg-info/
```

By omitting these directories, you leave buildout free to create these files afresh as they need to be on each system where you place your project.

- Option #1: Rebuilding your project with the buildout command

Because you will need these files re-created when you check out your project, you will need to get a working copy of buildout available. Some people, I have heard, including people of quite high reputation in the Grok community, actually use easy_install to make the buildout program permanently available on their system:

```
# How to install the buildout command permanently
$ easy_install zc.buildout
```

Having this command available on your system means that you will typically find yourself doing something like this when you are ready to start work on a project that has been sitting in version control:

```
# Getting to work again

$ svn co http://.../myproject
$ cd myproject
$ buildout
$ bin/zopectl fg
```

- Option #2: Using a bootstrap file to avoid installing buildout

Another method for checking your project back out, and that avoids having to install the buildout command on your system at all, is to place a copy of the file bootstrap.py in your project:

```
# Grabbing bootstrap.py for your project

$ cd my-project
$ wget http://svn.zope.org/*checkout*/zc.buildout/trunk/bootstrap/bootstrap.py
```

You can then check this file in with the rest of your project, along with the other files you version-control, and it will be available whenever you check the project out. Then, re-activating your project just goes like this:

```
# Building out a freshly checked-out project

$ python ./bootstrap.py
$ ./bin/buildout
$ ./bin/zopectl fg
```

Using bootstrap.py means that anyone can work on your project without having to worry about installing Python commands on their system that they might never have heard of.

5.4.1 Backing up your live application data

Note that, as you run your project in production, you will want to be very careful to back up the following file:

```
# Back this file up, it contains your application data!

parts/data/Data.fs
```

But as one does not typically use version control to back up raw application data, you should not actually place this file under version control! Instead you will copy it safely to another machine, or make sure it gets copied and placed on a back-up tape, to assure that you can restore your site with its content should the machine on which it is running ever be destroyed.

5.5 Profiling with Grok

Author Uli Fouquet

When it comes to a web framework, profiling is not as easy as with commandline or desktop applications. You normally want to trigger certain requests and see, in which parts of your code how much time or memory was spent.

Here is how you can do this.

5.5.1 Prerequisites

Before we start, we apparently need something to profile: your application. So if you haven't done it yet, create a typical Grok project:

```
$ grokproject Sample
```

This will create our project in the `Sample/` folder. We assume, that the created project is based on `paster`, which is the default as of `grokproject v1.0a2`.

If you have an older version of *grokproject* installed, update your install by:

```
$ easy_install -U grokproject
```

You should be able to start/stop the created project.

5.5.2 Installing a profiler

There are some profiling tools available with every Python installation. With web-frameworks, however, we often want to check only certain requests. This is difficult with the regular profiling tools, but with *paster* we luckily have a pipleine mechanism, where we can put in a profiler, which is even configurable over web frontend: *repoze.profile*

Install *repoze.profile*

In the `buildout.cfg` of your project add the `repoze.profile` egg to list of eggs of your application. Look out for a section named `[app]`, which could read like this:

```
...
[app]
recipe = zc.recipe.egg
eggs = cptapp
       z3c.evalexception>=2.0
       Paste
       PasteScript
       PasteDeploy
       repoze.profile
interpreter = python-console
...
```

Here the added `repoze.profile` line is important.

Now run:

```
$ bin/buildout
```

to fetch the egg from the net if it is not already available and to make it known to the generated scripts.

Create a profiler.ini

To make use of the new egg we must tell *paster*. This is done by an appropriate initialization file we create now:

```
# profiler.ini
[pipeline:main]
pipeline =
         egg:repoze.profile#profile
         egg:sample

[server:main]
use = egg:Paste#http
host = 127.0.0.1
port = 8080

[DEFAULT]
# set the name of the zope.conf file
zope_conf = %(here)s/zope.conf
```

It is crucial, that you use the name of your project egg here in the pipeline. As we created a project named `Sample`, our egg is named `sample`.

Put this new file in the same directory as where your `zope.conf` lives (not: `zope.conf.in`). For projects created with *grokproject* >= *v1.0a2* this is `etc/`, newer projects might use `parts/etc/`.

5.5.3 Start Profiling

With the given setup we can start profiling by:

```
$ bin/paster serve etc/profiler.ini
```

If your `profiler.ini` file resides elsewhere, you of course must use a different location.

The server will start as usual and you can do everything you like with it.

Browsing the Profiler

To get to the profiler, enter the following URL:

> http://localhost:8080/__profile__

This brings us to the profiler web frontend. If you have browsed your instance before, you will get some values about the timings of last requests. If not, then browse a bit to collect some data. The data is collected 'in background' during each requests and added to the values already collected.

Profiling a certain view

Say we want to profile the performance of the index view created by the default application. To do this, we first have to install an instance of our `Sample` application.

So go to the admin interface (http://localhost:8080/applications) and add an instance of your application under the name `app` (you can actually use any name you like, of course).

Now we can access

> http://localhost:8080/app

and the usual index page will appear.

If we go back to the profiler, however, we will see the summed up values of all requests we did up to now. Including all the actions in the admin interface etc. we are not interested in.

We therefore clear the current data by clicking on `clear`.

Now we access the page we want to examine directly and go to the above URL directly.

When we now go back to the profiler, we only see the values of the last request. That's the data we are interested in.

5.5.4 Profiling mass requests

Very often a single request to a view does not give us reliable data: too many factors can influence the request to make its values not very representative. What we often want are **many** requests and the average values appearing here.

This means for our view: we want to do several hundreds requests to the same view. But as we are lazy, we don't want to press the reload button several hundred or even thousand times. Luckily there are tools available, which can do that for us.

One of this tools is the apache benchmarking tool `ab` from the apache project. On Ubuntu systems it is automatically installed, if you have the apache webserver installed.

We can trigger 1,000 requests to our index page now with one command:

```
$ ab -n1000 -c4 http://127.0.0.1/app/@@index
```

This will give us 1,000 requests, of which at most four are triggered concurrently, to the URL http://127.0.0.1/app/@@index. Please don't do this on foreign machines.

The result might look like this:

```
Benchmarking 127.0.0.1 (be patient)
Completed 100 requests
Completed 200 requests
Completed 300 requests
Completed 400 requests
Completed 500 requests
Completed 600 requests
```

```
Completed 700 requests
Completed 800 requests
Completed 900 requests
Finished 1000 requests


Server Software:        PasteWSGIServer/0.5
Server Hostname:        127.0.0.1
Server Port:            8080

Document Path:          /app/@@index
Document Length:        198 bytes

Concurrency Level:      4
Time taken for tests:   38.297797 seconds
Complete requests:      1000
Failed requests:        0
Write errors:           0
Total transferred:      448000 bytes
HTML transferred:       198000 bytes
Requests per second:    26.11 [#/sec] (mean)
Time per request:       153.191 [ms] (mean)
Time per request:       38.298 [ms] (mean, across all concurrent requests)
Transfer rate:          11.41 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0    0   0.0      0       0
Processing:    94  152  17.3    151     232
Waiting:       86  151  17.3    150     231
Total:         94  152  17.3    151     232

Percentage of the requests served within a certain time (ms)
  50%    151
  66%    153
  75%    156
  80%    158
  90%    176
  95%    189
  98%    203
  99%    215
 100%    232 (longest request)
```

Also this benchmarking results can be interesting. But we want to know more about the functions called during this mass request and how much time they spent each. This can be seen, if we now go back to the browser and open

> http://localhost:8080/__profile__

again.

5.5.5 Turning the Data into a Graph

We now want to turn the data into a graph. To do this, we first have to 'export' the data from the web framework.

Getting the Data out of the Web

The web frontend provided by `repoze.profile` is very comfortable and nice for analyzind ad-hoc. But sometimes we want to have the data 'exported' to process it further with other tools or simply archiving the results.

Luckily we can do so by grabbing the file `wsgi.prof` which contains all the data presented in the web interface. This file is created in the project directory (here: `Sample/`).

Be careful: when you click `clear` in the webinterface, then the file will vanish. So copy it to some secure location where we can process the data further.

Because `repoze.profile` makes use of the standard Python profiler in the `profile` or `cProfile` module, the data in the `wsgi.prof` file conforms to output generated by this profilers.

Converting the Data into `dot`-format

One of the more advanced tools to create graphs from profiling information is `dot`. To make use of it, we first have to convert the data in `wsgi.prof` into something `dot`-compatible.

There is a tool available, which can do the job for us, a Python script named `GProf2Dot` which is available here:

> http://code.google.com/p/jrfonseca/wiki/Gprof2Dot

Download the script from:

> http://jrfonseca.googlecode.com/svn/trunk/gprof2dot/gprof2dot.py

We can now turn our profiling data into a `dot` graph by doing:

```
$ python grprof2dot.py -f pstats -o wsgi.prof.dot wsgi.prof
```

This will turn our input file `wsgi.prof` of format `pstats` (=Python stats) into a dot-file named `wsgi.prof.dot`.

Converting the `dot` file into Graphics

Now can do the last step and turn our dot file into a nice graphics file. For this we need of course the `dot` programme, which on Ubuntu systems can be easily installed doing:

```
$ sudo apt-get install dot
```

Afterwards we do the final transformation by:

```
$ dot -Tpng -omygraph.png wsgi.prof.dot
```

This will generate a PNG file.

All the used tools (`ab`, `dot`, `gprof2dot`) provide a huge bunch of options you might want to explore further. This way we can generate more or less complete graphs (leaving out functions of little impact), coulours etc.

In the end you hopefully know more about your application and where it spends its time.

5.6 Eggs, Known Good Sets and developing with unreleased Grok source code

Author Kevin Teague

Grok releases are distributed as Python eggs. This gives you the flexibility to easily control what versions of each of the individual python packages that are used to make up a Grok application. Learn why this is a desirable goal, and how you can use this to develop your Grok application based on unreleased versions of Grok checked out from subversion.

5.6.1 Eggs and Versions

Eggs are packaged python projects. They are a collection of Python files and other project files with a set of associated package data. They are the way that the parts of Grok and other Python projects that Grok depends upon are distributed. Before Phillip J. Eby started the setuptools project that allows for the creation and distribution of Python eggs, Python developers distributed their code as a single, monolithic archive (e.g. zip files or tarballs). This system had the drawback that to upgrade to a new version of the framework, you had to download the entire framework - even if you were only updating to a maintenance release that contained just a few small bug fixes. Let's look at how Zope 3 was originally distributed:

Release	Size	Package Versions
Zope-3.1.0.tgz	3.86 MB	• zope.tal = 3.1.0 • zope.schema = 3.1.0 • zope.event = 3.1.0
Zope-3.2.1.tgz	6.23 MB	• zope.tal = 3.2.1 • zope.schema = 3.2.1 • zope.event = 3.2.1
Zope-3.2.2.tgz	6.24 MB	• zope.tal = 3.2.2 • zope.schema = 3.2.2 • zope.event = 3.2.2

Wow, to get the upgrade to the maintenance release between 3.2.1 and 3.2.2 you had to download and install over 6 MB of data. Worse, can you tell how much the zope.tal or zope.schema packages changed between release 3.1 and 3.2? The medium bump in version number indicates that there has been some interesting feature improvements and bug fixes happening. But this version number is global - perhaps people were perfectly happy with the 3.1 version of zope.event and it had no changes in it between the 3.1 and 3.2 releases. Yet it still got a medium increase in it's version number. Wouldn't it be good if only the packages that change had their version numbers updated?

5.6.2 Known Good Sets

Starting in 2006 work was done to break Zope 3 up so that it could be managed and distributed as individual Python eggs. This work was completed in 2007 and Zope 3.4a was the first release to use a completely egg-based system. Now each part of the framework can be assigned it's own version number, and managed independently. With this system it is very easy to upgrade to a newer version of just one package in your application, without being forced to upgrade *everything* else in the framework.

When you take a collection of Python eggs that make up a framework or an application and produce a list of eggs and version numbers that are tested to work together this is called a *Known Good Set* (KGS).

Grok 0.10.1 was the first Grok release to be composed of a Known Good Set. Let's look at a portion of the set of eggs in this release, and a set of eggs in the Grok 0.11 release:

Release	Size	Package Versions
Grok-0.10.1	• 1.9 MB • 0.4 MB • 1.3 MB	• grok = 0.10.1 • martian = 0.9 • zope.tal = 3.4.0b1
Grok-0.11	• 1.9 MB • 0.4 MB • 1.3 MB	• grok = 0.11 • martian = 0.9.1 • zope.tal = 3.4.0b1

This is much better than the monolithic tarball of the earlier Zope 3 releases as you now have additional information easily available. You can see that the martian package has had a single maintenance release made between Grok 0.10 and Grok 0.11, and that the zope.tal package has not changed at all between releases.

The full set of eggs and version numbers that made up the official Grok releases can be found at http://grok.zope.org/releaseinfo/

5.6.3 Managing eggs with zc.buildout

The zc.buildout tool was created primarily as a way of making it easy to collect together a set of eggs usable by a Python application. While a full discussion of zc.buildout is beyond the scope of this tutorial, we can look at how you can use this tool to work with eggs that are newer than the ones supplied in a known good set.

If you have created a Grok application using grokproject, then you will have a *buildout.cfg* file with these lines in it:

```
[buildout]
extends= http://grok.zope.org/releaseinfo/grok-0.11.cfg
versions = versions
```

This tells your buildout configuration that it is extending the Grok 0.11 configuration. The Grok release configuration files only ever contain a single section labelled [versions]. The *versions = versions* line tells buildout that we want to use all the eggs with the versions specified by a Grok release.

What if you wanted to update just a single egg to use a newer version than the one specified in an official Grok release? You would add the following below your [buildout] section:

```
[versions]
martian = 0.9.2
```

Run *./bin/buildout* to update your application and you will be using version 0.9.2 of martian instead of 0.9.1. The version syntax of buildout also allows you to specify minimum and maximum versions. You could write:

```
[versions]
martian >= 0.9.2, < 1.0
```

This would ensure that you used at least the 0.9.2 version, and when you ran *./bin/buildout* to update your application, it would upgrade to the newest maintenance release made in the 0.9 series. It would not upgrade to a 1.0 release though.

5.6.4 Trail blazing (working with unreleased eggs)

Cave men beat well-worn paths to the watering hole, and developing with tested releases of Grok is equivalent to staying on this beaten path. But what if you want to go on a hunting expedition for mammoths and these animals don’t frequent the watering hole near the cave?

Let's look at an example of the Viewlets work done at the Snow Sprint 2008. How can we try out this experimental release in your project? First you will need to checkout that feature branch somewhere inside your project, the *src* directory is the standard location for the parts that you are actively working on:

```
$ cd ~/buildouts/my-grok-project/src
$ svn co svn://svn.zope.org/repos/main/grok/branches/snowsprint-viewlets2/ grok
```

You will notice that checking out Grok also checks out Martian using the *svn:externals* feature. Tell buildout that you want to use these subversion checkouts for development by adding the paths to them in the *develop* setting. The versions required by this feature branch also requires some newer eggs which are specified inside the *grok/versions.cfg* file. Change your *extends* setting from using official Grok release versions to point to these development versions:

```
[buildout]
develop = . src/grok src/grok/martian
extends = src/grok/versions.cfg
```

Run *./bin/buildout* to update your application. There is just one last tricky bit that we are missing. If you look at *parts/app/runzope*, the Python program which launches your Grok application you will see:

```
sys.path[0:0] = [
  '/home/username/buildouts/my-grok-project/src',
  '/home/username/buildouts/my-grok-project/grok/src',
  '/home/username/buildouts/shared/eggs/martian-0.9.3-py2.4.egg',
  ...
]
```

This is because the *src/grok/versions.cfg* file is specifying martian 0.9.3, while the version in development may declare that it is a newer version. You can tell what the development version of martian is by looking at it's *setup.py* file:

```
setup(
   name='martian',
   version='0.9.4dev',
   ...
)
```

You can declare that you want to use this version by overriding the value in your buildout configuration file. However, if you do newer checkouts of martian this version number may be updated. If you specify nothing as a version, then the newest version will be used, which is what you want:

```
[versions]
martian =
```

Now run *./bin/buildout* one more time and you are all set to develop with a feature branch of Grok.

5.6.5 Developing with a development.cfg file

What if you'd like to keep your *buildout.cfg* file clean, as you are using that file for doing production installations of your project and you don't want to forget to undo the changes to your *buildout.cfg* file when you are done experimenting?

You can have a separate file, typically named *development.cfg* (or *dev.cfg* if you are a lazy typist) that overrides just the development changes that you have made. Putting all of the changes above together it would look like this:

```
[buildout]
develop = . src/grok src/grok/martian
extends = buildout.cfg src/grok/versions.cfg

[versions]
```

```
grok =
martian =
```

Now a production buildout will be the default *buildout.cfg* file only, and you explicity specify a development buildout with *./bin/buildout -c development.cfg*. Remember, you can also have a *~/.buildout/default.cfg* file that contains defaults specific to your own personal set-up. This way you can separate the configuration neatly between:

- Your own personal setup in *~/.buildout/default.cfg*
- Active development configuration in *project/development.cfg*
- Default configuration in *project/buildout.cfg*

Happy mammoth hunting!

5.7 How to pack your ZODB database

Author Sebastian Ware

The ZODB grows with each write operation. In order to reduce the size of the data.fs file, you need to perform a "pack" operation.

Packing from a grok.View class can be done with the following method:

```
self.request.publication.db.pack()
```

Each update is stored by appending the new state of the updated objects to the end of the data.fs file. This exposes an object history, allowing ZODB to offer an object undo feature and also multi version concurrency control.

A disadvantage with this approach is that the database file grows with each write. This requires you to "pack" the database to reduce the size by removing old copies of objects.

You can access the current database from a view by:

```
self.request.publication.db
```

The database exposes many features that you might find interesting, such as cache parameters and undo services. However, the only method that we need to pack the database is conveniently called just that:

```
class Pack(grok.View):

    def render(self):
        self.request.publication.db.pack()
```

This is what the documentation says:

```
pack(self, t=None, days=0) method of ZODB.DB.DB instance
    Pack the storage, deleting unused object revisions.

    A pack is always performed relative to a particular time, by
    default the current time.  All object revisions that are not
    reachable as of the pack time are deleted from the storage.

    The cost of this operation varies by storage, but it is
    usually an expensive operation.

    There are two optional arguments that can be used to set the
    pack time: t, pack time in seconds since the epcoh, and days,
    the number of days to subtract from t or from the current
    time if t is not specified.
```

5.8 Graphical debugging of Grok with Komodo IDE

Author Sebastian Ware

Set up the Komodo IDE graphical debugger for your Grok project.

1. Set the PYTHONPATH

1. either in the terminal before you execute the "/bin/paster serve" command

   http://community.activestate.com/forum-topic/debugging-zope-apps

2. or (if you start Grok from Komodo) by adding PYTHONPATH with a path to the

   Komodo debugger library. Go to "Preferences / Environment / New..." and

   add Name= PYTHONPATH, Value= /path/to/debugger_library

   The Komodo Python debugger library is called "dbgp". In Macosx you can find it

   the Komodo IDE.app package under Contents/SharedSupport/. I actually copied it

   to another location but I don't think that is necessary.

2. Make sure that the Komodo debugger listens to the right port

Go to "Preferences / Debugger / Connection" and choose "a specific port". I selected port 9000 (but I think anything above 1024 should be ok in Macosx).

3. Add the following two lines anywhere in a .py file in your project (outside any class or method definition):

   import dbgp.client dbgp.client.brk(host="localhost", port=9000)

Note that it supports remote debugging, but that requires you opening the firewall on the computer running the debugger.

4. Start Grok from within Komodo IDE (I have a python virtualenv installation)

I use the following macro (search for "macro" in Komodo help to learn how to create a macro) and have a keybinding to "shift-command r":

```
// Macro recorded on Tue Jun 19 2007 21:08:09 GMT+0200 (CEST)
var appname = 'MyApp';
var sandbox = '/path/to/my/projects';
var pythonCmd = sandbox + '/bin/python';
var startCmd = sandbox + '/' + appname + '/bin/paster serve ' + sandbox + '/' + appname + '/parts/etc
komodo.assertMacroVersion(3);
ko.run.output.kill(-1);
if (komodo.view) { komodo.view.setFocus() };
setTimeout(function(ko, win) {
    ko.run.runEncodedCommand(win,  pythonCmd + ' ' + startCmd);
}, 200, ko, window);
// Note "200" is the delay before starting the server, allowing the output.kill command to
// execute properly. You might want to extend this on a slower computer.
```

Grok will pause execution at the "client.brk" command added in step 3. You can add and remove your break points and press the "Go / continue debugging" button in the toolbar. The Komodo debugger supports regular, conditional, variable based, function call, function return and exception base breakpoints. (At the time of writing I have only set regular line based breakpoints.)

Once the Komodo debugger breaks at a break point you can:

step into/over/return from statements inspect local/global/watch variables check the call stack enter interactive mode to perform arbitrary python code (this gives you a Python prompt, not the pdb prompt)

5.9 Grok, Virtual Hosting and Nginx

Author Sebastian Ware

Configuring the super fast and lightweight Nginx HTTP server to support virtual hosting.

Zope 3 and thus Grok supports virtual hosting straight out of the box. The super fast and lightweight Nginx HTTP server does this too. In fact, it is so easy to congfigure even a caveman could do it... so I did.

This is a stripped down version of [nginx.conf] with only the relevant statements included. Add these in the appropriate places to the default nginx.conf and you will be cooking in no time.

```
http {
  upstream mygrokinstance {
    server 127.0.0.1:8080;
  }
  server {
    listen       80;
    server_name  www.example.com;
    location / {
      proxy_pass http://mygrokinstance/grok_application_name/++vh++http:www.example.com:80/++;
    }
  }
}
```

Explanation:

2-4: Defining a backend server *mygrokinstance*. This also allows me to add simple load-balancing (round-robin and client IP) by adding an extra line for each server.

5-11: Server directives...

6: The port that Nginx listens to.

7: The domain name used to access the application.

8-10: The rewrite rule.

8: `location /` corresponds to anything. `location /application` would equal anything starting with `www.example.com/application`

9: The rewrite rule, where:

`mygrokinstance` is the backend server that we defined on row 3-5

`grok_application_name/` is the application name since I don't want to have to include it in the URLs

`http:www.example.com:80/` tells zope to generate the URLs without the application name

More on Nginx:

http://wiki.codemongers.com/Main

Virtual hosting in Zope 3:

http://wiki.zope.org/zope3/virtualhosting.html

The Nginx upstream command:

http://wiki.codemongers.com/NginxHttpUpstreamModule

5.10 Grok and Apache

Author Matthias

Configuring the Apache HTTP server with grok.

There are two approaches to make your Grok application work with Apache: either through mod_wsgi or by using mod_rewrite and mod_proxy.

If you choose the mod_wsgi way then your application will run directly under Apache.

If you choose the mod_rewrite way, you will run your application as a paster server (probably the same server you are currently using for development). Then you make Apache a proxy to the paster server. This means Apache takes any incoming requests and forwards them to the paster server running on localhost. The results from paster are then returned to Apache which returns them to the browser. This way paster is not exposed directly to the internet.

This document will cover different aspects of integration between Grok and Apache. If there are missing spots, please post on the Grok mailing list so we can fill them in.

5.10.1 Virtual hosting (mod_wsgi)

If you want to just get started with grok and mod_wsgi you don't need any virtual hosting setup. E.g. you want to be able to access example.com/grokui . However if you want something like example.com to serve a certain grok application of yours, you'll have to setup virtual hosting.

Setting up virtual hosting for using Grok, Apache and mod_wsgi, Apache configuration:

```
<VirtualHost *:80>
    Servername example.com
    RewriteEngine on

    # Possible values include: debug, info, notice, warn, error, crit,
    # alert, emerg.
    #LogLevel warn
    #ErrorLog /home/myuser/example.site/log/error.log
    #CustomLog /home/sites/example.site/log/access.log combined


    # An example setup with two grok applications
    # Both are served from a dedicated apache daemon process. Both sites use separate python
    #  interpreters. They are completely isolated from each other

    ## Application 1 ##
    # Here: served under example.com/apps/fritz
    WSGIDaemonProcess example-fritz user=www-data threads=2
    RewriteRule ^/apps/fritz(.*) /apps/fritz/++vh++http:example.com:80/apps/fritz/++$1 [L,PT]
    Alias /apps/fritz /home/myuser/example.site/apps/fritz/parts/wsgi_app/wsgi/
    <Directory /home/myuser/example.site/apps/fritz/parts/wsgi_app>
           WSGIApplicationGroup example-fritz
           WSGIProcessGroup example-fritz
           WSGIPassAuthorization On
           SetHandler wsgi-script
           Options ExecCGI FollowSymLinks
           Order allow,deny
           Allow from all
    </Directory>
```

```
    ## Application 2 ##
    # Here: served under example.com/apps/heinz
    WSGIDaemonProcess example-heinz user=www-data threads=2
    RewriteRule ^/apps/heinz(.*) /apps/heinz/++vh++http:example.com:80/apps/heinz/++$1 [L,PT]
    Alias /apps/heinz /home/myuser/example.site/apps/heinz/parts/wsgi_app/wsgi/
    <Directory /home/myuser/example.site/apps/heinz/parts/wsgi_app>
           WSGIApplicationGroup example-heinz
           WSGIProcessGroup example-heinz
           WSGIPassAuthorization On
           SetHandler wsgi-script
           Options ExecCGI FollowSymLinks
           Order allow,deny
           Allow from all
    </Directory>


</VirtualHost>
```

5.10.2 Virtual hosting (mod_rewrite/mod_proxy)

Setting up virtual hosting for using Grok, Apache and mod_rewrite/mod_proxy and paster running on localhost:8080, Apache configuration:

```
<VirtualHost *:80>
    SSLDisable
    Servername example.com
    RewriteEngine on

    # Application: served under example.com/apps/fritz
    RewriteRule ^/apps/fritz(/?.*)  http://localhost:8080/apps/fritz/++vh++http:www.example.com:80/a
    CustomLog /var/log/apache/example.com/access.log combined
    ErrorLog /var/log/apache/example.com/error.log
</VirtualHost>
```

5.10.3 Virtual hosting (general)

Virtual hosting in Zope 3:

http://wiki.zope.org/zope3/virtualhosting.html

5.11 Releasing software

Author unknow

Steps to take when releasing software.

When releasing software, the following steps should be taken:

1. Make sure all automated tests of the package pass.
2. Make sure the package has a `LICENSE.txt` or similar file. Some distributors like Ubuntu require such a file in sources. See legal stuff for details.
3. Fill in the release date in `CHANGES.txt`. Make sure the changelog is complete.

4. Make sure the package metadata in `setup.py` is up-to-date. You can verify the information by re-generating the egg info:

   ```
   python setup.py egg_info
   ```

   and inspecting `src/EGGNAME.egg-info/PKG-INFO`. You should also make sure the that the long description renders as valid reStructuredText. You can do this by using the `rst2html.py` utility from docutils:

   ```
   python setup.py --long-description | rst2html.py > test.html
   ```

   If this will produce warnings or errors, PyPI will be unable to render the long description nicely. It will treat it as plain text instead.

5. Create a release tag.

6. Get a separate checkout of the release tag for creating the distribution tarball and eggs. It is important that you don't do this on the trunk or release branch to avoid

   (a) forgetting to tag the release at all.

   (b) forgetting to clean up the build directory that distutils and setuptools create. Failure to do so may result in old artefacts in eggs.

   (c) forgetting to check in files that are needed by `setup.py` or as package data. Setuptools will only include them in the distribution if they are checked into subversion.

   In the checkout of the tag perform the following steps:

   (a) Remove the "dev" marker from the version in setup.py

   (b) Commit these changes. It's acceptable that these changes modify the tag since they're purely related to release management.

   (c) Create a distribution and upload it to PyPI using the following command:

   ```
   python setup.py register sdist upload
   ```

   If the package contains C extensions, you need to upload a binary Windows egg as well:

   ```
   python setup.py bdist_egg upload
   ```

   This may require the help from someone with a Windows installation and proper tools (Visual C).

   Binary eggs for Linux or MacOSX should **never** be uploaded because those platforms vary too much to be binary-compatible with each other, due to varying UCS support, different libc versions and linking models (framework / non-framework).

   (a) Back on the trunk or the release branch, increase the version number in `setup.py` to the *next* release while preserving the dev marker. The convention is that the trunk or release branch always points to the upcoming release, *not* the one that has been released already. So if you've just released version 3.4.1, you should change `setup.py` to read:

   ```
   setup(
       name='...',
       version='3.4.2dev',
       ...
       )
   ```

In `CHANGES.txt` add a *new* section for the upcoming release. The release date for that should say "unreleased" so that committers recording their changes won't accidentally put their entry in the section for an already released version. For example:

```
3.4.2 (unreleased)
------------------

* ...

3.4.1 (2007-01-24)
------------------

* Fixed bug in the foo adapter.

* Added a bar utility for optimized kaboodling.

3.4.0 (2006-09-13)
------------------

Initial release as separate egg.
```

Important: Once released to PyPI or any other public download location, a released egg may never be removed, even if it has proven to be a faulty release ("brown bag release"). In such a case it should simply be superseded immediately by a new, improved release.

5.12 Using Virtualenv for a clean Grok installation

Author unknow

NOTE: As of Grok 1.2, you do not need to use virtualenv. Grok is automatically isolated from the system python environment.

5.12.1 Benefits of Using Virtualenv

Virtualenv is a python tool that allows you to create isolated python environments. It is great for simplifying the environment for your Grok installation so that you don't run into version or permissions problems. It can also be used as a solution for installing python packages on machines where you do not have write access to the site-packages directory.

- It's **really easy to use** virtualenv
- Virtualenv can allow you to have different Zope versions on your system (or any conflicting python software). This can be very useful if you run Plone and Grok on the same machine, or if you use a stand alone Zope version that is different from the Zope version that Grok uses.
- Virtualenv will allow you to install Grok on a system to which you do not have write access to the system wide python directory.

Example of conflicting packages

You have already installed Zope 3.1 for a previous development task. This was installed system wide and has placed a number of packages in your site-packges directory. Grok currently requires the Zope Toolkit (ZTK). Running **grokproject** produces an error and aborts it's process.

Virtualenv provides a fix by allowing you to use your Python installation without the packages installed in the default site-packages directory.

5.12.2 What you need

A working Python installation with virtualenv and easy_install is required.

You can install virtualenv with easy_install:

```
$ easy_install virtualenv
```

If you have newer versions of Python on your system, then you can target the required version of easy_install with:

```
easy_install-2.6 virtualenv
```

Debian specific instructions

If you are using a Debian-based system, you can use the apt package management tool to install virtualenv. For Debian this is available on the unstable branch (Sid) as **python-virtualenv**:

```
# apt-get install python-virtualenv
```

The current stable release of Debian 4.0 (Etch) does not have the python-virtualenv package available, but does have easy_install available, so you can install it by running:

```
# apt-get install python-setuptools
# easy_install virtualenv
```

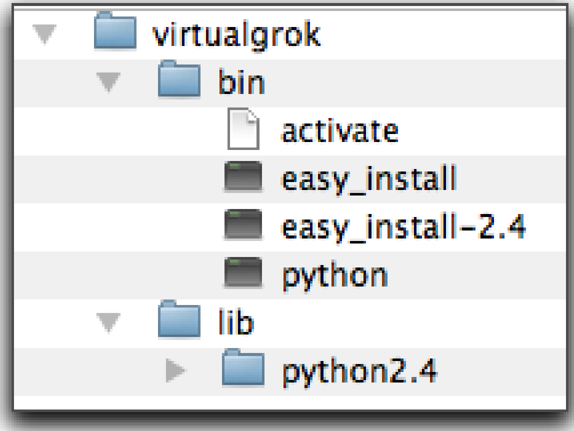
5.12.3 Creating a new Python sandbox

Create the virtual python environment

Run the virtualenv command to create a new, clean Python environment by the name of 'virtualgrok':

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

This creates a directory called 'virtualgrok' with two folders within it, lib and bin. This will look like this:



NOTE: Grok 1.1 requires python 2.6 or 2.5 (2.4 deprecated) so you would have easy_install-2.5/-2.6 and python2.5/2.6.

The optional **–no-site-packages** switch means that none of your existing Python site-packages will be available in your new Python environment. This is desirable if you already have conflicting Python packages installed that you want to avoid (or might install such packages in the future).

Some Zope developers do not use this switch as they have installed a second Python (or sometimes many more!) separate from the system Python for developing Python web applications with. They only install the more difficult to install packages into their development Python such as PIL, XML, database and LDAP packages available and then use a separate virtual environment for each major Zope project they work on. Zope developer Lennart Regebro has termed this approach as "having both a belt and suspenders".

Activate your virtual python environment

VirtualEnv creates a shell file called **activate** which you can optionally use to alter your PATH environment variable so that you do not have to type the full path to your new Python (e.g. /home/username/virtualgrok/bin/python). Let's look at how you can use the **source** shell command with this activate file to make the new python the default:

```
$ echo $PATH
usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/X11/bin
$ which python
/usr/bin/python
$ source virtualgrok/bin/activate
(virtualgrok)$ echo $PATH
/home/username/virtualgrok/bin:usr/bin:/bin:/usr/sbin:/sbin:/usr/local/bin:/usr/X11/bin
(virtualgrok)$ which python
/home/username/virtualgrok/bin/python
(virtualgrok)$ deactivate
$
```

Install grokproject

Install grokproject making sure that you use the easy_install provided by your new virtual Python environment:

```
(virtualgrok)$ easy_install grokproject
```

At this point you may want to continue following the Grok Tutorial from where it leaves of with the install instructions.

5.12.4 Further information

Thanks to j-w from the #grok IRC channel on Freenode.org for much help. And to Ian Bicking for creating VirtualEnv.

http://pypi.python.org/pypi/virtualenv

http://grok.zope.org/doc/current/tutorial.html#setting-up-grokproject

http://peak.telecommunity.com/DevCenter/EasyInstall#installing-easy-install

This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States License.

5.13 Install multiple Grok apps using zc.buildout

Author unknow

Grok is packaged as Python eggs. zc.buildout is a tool for managing these eggs, and let's you quickly try out or develop a Grok-based project.

Might be outdated! This document hasn't been reviewed for Grok 1.0 and may be outdated. If you would like to review the document, please read this post.

5.13.1 Before you start

You will need a clean installation of Python 2.6 (or 2.5) to use. You can either build your own Python from source, or use an existing system Python install and isolate yourself from any system-wide packages using VirtualEnv.

5.13.2 Install a Grok application using zc.buildout

Almost all Grok applications are developed using zc.buildout. You will know if a Grok application has been developed using zc.buildout if you see a file named `buildout.cfg` at the top level of a project's development directory.

If you see a `buildout.cfg` file you will also often see a scripot named `bootstrap.py` or a directory named `bootstrap` with this file in it. If you don't find this program you can download a copy.

Let's walk through the process of installing the Adder application.

```
$ svn co svn://svn.zope.org/repos/main/grokapps/Adder Adder
...
$ cd Adder
$ python bootstrap.py
...
$ ./bin/buildout
$ ./bin/zopectl fg
```

That's it! You should have a complete, running Grok application with just five commands.

5.13.3 Sharing Eggs: Installing multiple Grok applications

Grok requires quite a few Python eggs to be installed. Every Grok application that you try out will require a complete set of eggs to be downloaded over the internet. This can be quite time consuming and takes up disk space. Fortunately, zc.buildout can be configured to put all of the eggs in a common shared directory. When you install your second Grok application using a shared eggs directory it will install very quickly. Running `./bin/buildout` to install a new Grok application on a modern computer should only take about five seconds.

To tell zc.buildout that you would like it to use a shared eggs directory, you will need to create a file called `default.cfg` in the top level of your home directory inside a directory named `.buildout`. Then you will need to add this text to the file:

```
[buildout]
eggs-directory = <path-to-shared-eggs>
```

This path can be to anywhere on your system, the default for grokproject is to use a directory named `<your-home-directory>/buildout-eggs`, but you can place this anywhere you like.

5.13.4 Faster, buildout, faster!

You can increase the speed of running buildout a little bit more by adding two more options to your `~/.buildout/default.cfg` file. The `download-cache` option will tell buildout to cache the files it downloads from the internet before unpacking them into your eggs directory. The `newest` option can tell buildout that

upon subsequent reruns of the `./bin/buildout` command not to use the internet to automatically check for and fetch the newest versions of eggs.

Your final buildout file may look something like this:

```
[buildout]
newest = false
eggs-directory = /Users/kteague/buildouts/shared/eggs
download-cache = /Users/kteague/buildouts/shared/cache
```

5.13.5 Keeping bootstrap.py close at hand

If you work frequently with zc.buildout, you may find it handy to keep a single copy of `bootstrap.py` at hand. You can place this file somewhere on your system, and then on UNIX-like systems create a shell alias to allow you to quickly run bootstrap with your current development version of Python. Place a line such as the following in your shell profile (this is found at `~/.bashrc` on Linux, or `~/.profile` on Mac OS X):

```
alias bootit="/Users/kteague/buildouts/shared/python-2.6.0/bin/python \
/Users/kteague/buildouts/shared/bootstrap.py"
```

see also:

Using Virtualenv for a clean Grok installation

NOTE: As of Grok 1.2, you do not need to use virtualenv. Grok is automatically isolated from the system python environment.

5.14 Use Apache HTTP server with Grok (on Debian Sid)

Author unknow

This Grok How-To gives a step-by-step explanation of how to install and configure Apache HTTP server version 2.2 on Debian Sid to serve Grok Web Applications using the mod_rewrite method.

5.14.1 Apache on Debian Sid - Using Apache HTTP server with our Grok web application

Question: Why does one want to use Apache to serve our Grok web application?

Answer: There are multiple possible reasons.

- A Grok specific reason is that you cannot currently customise the base URL of Grok apps and serve them from the root server directory "/ ".
- Your website requires more than one framework or language from the same server. [1]
- Zope's Zserver is not as robust as Apache. [2]
- Zope's SSL is not as robust as Apache's. [2]
- Apache HTTP server uses caching to speed up requests. [2]

There are three methods available for combining Apache and Grok:

1. Mod_rewrite

2. ProxyPass
3. cgi

This how-to deals only with the mod_rewrite method as the other two are impractical. [1,2]

5.14.2 Steps involved:

1. Install Apache
2. Configure Apache

No configuration of Zope or Grok should be required! Zope uses something called virtual hosting to greatly simplify this procedure for us. [3] It works by having Apache request a URL which passes to Zope any configuration it needs in order to serve us our Grok application the way we want it.

5.14.3 Step One: Apache Installation

The first step is to install the Apache HTTP server, thankfully Debian makes this easy:

```
apt-get install apache2
```

Debian then does all the work and the important directories at the end of the installation are:

```
/etc/apache2 - for configuration
```

```
/var/log/apache2 - For logs
```

5.14.4 Step Two: Apache Configuration

The Debian package comes pre-configured to a certain extent so be sure to familiarize yourself with this in the /etc/apache2 directory at some point in the future. The Apache2 package also creates for you an empty file **/etc/apache2/httpd.conf** which is where you can add all the custom configuration you want for your apache server.

Useful syntax [4]:

- # - comment
- \ - line continuation

Here is a copy of my httpd.conf file:

```
LoadModule rewrite_module /usr/lib/apache2/modules/mod_rewrite.so
LoadModule proxy_module /usr/lib/apache2/modules/mod_proxy.so
LoadModule proxy_http_module /usr/lib/apache2/modules/mod_proxy_http.so

ProxyRequests off
ServerName example.com

<VirtualHost *>
ServerName example.com
RewriteEngine On
RewriteLog "/var/log/apache2/rewrite.log"
RewriteLogLevel 1
RewriteRule ^(/?.*) \
http://localhost:8080/grokapp/++vh++http:example.com:80/++$1 \
[P,L]
</VirtualHost>
```

It is important to point out that Apache will not reload it's configuration files until it is restarted, this can be accomplished with the command 'Apache2ctl graceful'. [5]

These few lines of configuration are all that's required to set up apache serve your Grok application.

Understanding httpd.conf

Lines 1-3 Load the necessary Apache modules.

ProxyRequests off – this command is in the interest of safety to prevent your HTTP server from turning into an open proxy if you misconfigure it.

ServerName is a variable that's supposed to be declared both inside and outside the VirtualHost declaration.

The **VirtualHost** statement is used to compartmentalize Apache configuration, see the main docs. [6]

RewriteEngine On – turns on mod_rewrite.

RewriteLog "/var/log/apache2/rewrite.log" - makes mod_rewrite log it's activities consistent with Debian practice.

RewriteLogLevel – Choose 0-9 for how much gets logged. Start with 9 until everything is working then switch to your desired production level probably zero or one.

Configuring mod_rewrite

RewriteRule – This configuration takes two arguments in the form of Perl regular expression statements. [7,8]

The first regex statement a pattern that determines if mod_rewrite will process a URL, the second regex statement is a pattern that specifies how the URL will be processed.

In this particular case there is a third optional argument '[P,L]' which passes two options to mod_rewrite which force it to proxy and make this the last processing rule if applied.

You can have several RewriteRule statements and they are processed in the order they are found in the config file. Please refer to mod_rewrite docs for more info. [9]

The RewriteRule required will be different but similar for every grok user. If you need help with your RewriteRule there is a useful utility that will write it for you called the witch. [10]

The only problem with the RewriteRule witch is that she is written for zope2. Grok is based on zope3 and zope3 virtual hosting syntax has changed slightly so you will have to modify the second argument to look like it does in my sample configuration. [3]

Once Apache is configured, you're done. Everything in Grok should work automatically. In my case I can now access my Grok application which is being served by Zserver at http://example.com:8080/grokapp through Apache by going to http://example.com

If you need to troubleshoot, remember the apache logs are located in /var/log/apache2/

Good Luck!

References

1 http://www.zope.org/Members/asv/Personal%20Docs/HowTo.2004-04-07.3618

2 http://wiki.zope.org/zope2/ZopeAndApache

3 http://wiki.zope.org/zope3/virtualhosting.html

4 http://httpd.apache.org/docs/2.2/configuring.html

5 http://httpd.apache.org/docs/2.2/

6 http://httpd.apache.org/docs/2.2/vhosts/

7 http://www.perl.com/doc/manual/html/pod/perlre.html

8 http://www.troubleshooters.com/codecorn/littperl/perlreg.htm

9 http://httpd.apache.org/docs/2.2/mod/mod_rewrite.html

10 http://betabug.ch/zope/witch

This work is licensed under a Creative Commons Attribution-Noncommercial-Share Alike 3.0 United States License.

see also:

Grok, Virtual Hosting and Nginx

Configuring the super fast and lightweight Nginx HTTP server to support virtual hosting.

5.15 Legal stuff

Author Uli Fouquet

Version n/a

About license files, source headers, fair use and friends

5.15.1 Licenses, Fair Use and Friends

If you're a developer, if you plan to create a new software package and chances are, that this package will be made publicly available, then you have to fiddle with licenses and legal stuff.

Here you can find some hints, how to design your software package layout in a way, that satisfies most people that are seriously concerned with legal issues. The hints given here are often so-called "good practices" known to work. This means: they are rules, not laws. In the end it is up to you.

In short, it comes down to this:

When you create a package:

1. add a `LICENSE.txt` or similar file to the root of your project (i.e. beside `setup.py` and `buildout.cfg`).

   If you reuse code from others with different licenses, add these licenses as well. You might then have a `ZPL.txt`, `LGPL.txt` in your project root.

   Warning: Projects hosted on `svn.zope.org` should usually be covered by ZPL only. See your Zope Committer Agreement for exceptions from this rule.

2. set the apropriate attributes in your project's `setup.py` like this:

```
#...
license='ZPL 2.1',
classifiers=[
    ...
    'License :: OSI Approved :: Zope Public License',
    ...
    ]
#...
```

or similar, depending on the license you picked. Make sure the license you set here is the same as mentioned in `LICENSE.txt`.

If your code is covered by multiple licenses, you should add an appropriate classifier for each and set the `license` attribute to something like `'ZPL + BSD'`.

3. [maybe optional] mark each header file in your sources with a license header like this (if you use ZPL as license):

```
##############################################################################
#
# Copyright (c) 2010 Zope Foundation and Contributors.
# All Rights Reserved.
#
# This software is subject to the provisions of the Zope Public License,
# Version 2.1 (ZPL).  A copy of the ZPL should accompany this distribution.
# THIS SOFTWARE IS PROVIDED "AS IS" AND ANY AND ALL EXPRESS OR IMPLIED
# WARRANTIES ARE DISCLAIMED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
# WARRANTIES OF TITLE, MERCHANTABILITY, AGAINST INFRINGEMENT, AND FITNESS
# FOR A PARTICULAR PURPOSE.
#
##############################################################################
```

This step is *not optional* if your project contains code covered by different licenses. Adding this header you tell which of your source files is covered by which license.

If all your code is covered by one single license, you don't have to do this (but you can).

Again, the text you use in header depends on the license you've chosen in step one.

5.15.2 Licenses

Each package that goes into public should be licensed. This way you can tell people the conditions under which they can use, reuse, modify and distribute your code.

Who is allowed to choose a license?

The only one that has the right to choose a license for a package is the author of the package. If there are several authors, all of them have to agree with a license (or a license switch).

What license should I pick?

In the grok eco-sphere we usually use open-source licenses like BSD, GPL, LGPL, or ZPL. You can pick whatever license you want as long as you have the right to do so and the license is compatible with your hosting location and other special circumstances.

***Warning:* People that host their code on `svn.zope.org` should** pick the ZPL! See your committer agreement for exceptions from this rule.

Why is setting the license in ''setup.py'' not enough?

Because some distributors like Debian, Ubuntu, etc. require explicit license files for every source package they package in their distributions. If you add a license file and release your project on PyPI, the license file will become part of the uploaded sources and hence satisfy these requirements.

Can I use multiple licenses with a project hosted on ''svn.zope.org''

Yes, you can. But it's a bit more complicated. If possible, you schould try to avoid such situations.

If, however, you have to make use of some code already covered by a non-ZPL license, you can avoid submitting the code to `svn.zope.org` by providing a script in your project, that downloads and packages the additional code on your local workstation. This code can then be tested, built and submitted to PyPI (which has no license restriction).

See hurry.yui (section about preparing releases near bottom) for an example.

5.15.3 Code from others on `svn.zope.org`

You can and you should use code from others. Hosting your code on `svn.zope.org` you might try to follow some rules as follows.

Am I allowed to reuse code already hosted on ``svn.zope.org`` code in my project?

Of course you can reuse code that's already hosted on `svn.zope.org`. The best way to make clear that you use other peoples code here is to svn-copy it over from the other location on `svn.zope.org` to your project. This way others can follow the history of your code better:

```
$ svn copy -m "Copy foobar from baz." \
              svn+ssh://myusername@svn.zope.org/repos/main/foo/trunk/src/foo/bar/baz.py \
              svn+ssh://myusername@svn.zope.org/repos/main/myproj/trunk/src/myproj/baz.py
```

If that would be too much (as you might only need a little function from another module, for instance), you might want to add a comment telling where the function comes from. This might also be helpful when problems occur due to changed dependencies/changes in other packages.

```
# Copied verbatim from zope.app.spammachine.
# This way we don't depend on z.a.spammachine anymore, as we only need this piece.
def spamtheworld(target_num='20 zillions'):
  """Spam the world.
  """
  ...
```

In other words: if you use code from others, tell it somehow.

Am I allowed to commit code from other locations (outside ``svn.zope.org``) to ``svn.zope.org``?

As long as the other author(s) agrees and the foreign code is not covered by a non-ZPL license, it *might* be okay. You might want to ask on `zope-dev` first, before you submit such code.

Submitting code that can also be found at other (non `svn.zope.org`) locations is not a problem, if the code is a small snippet that represents an 'obvious' solution for some problem. I.e. if your code would become artificially blown-up only to avoid repetition, then it's likely that this is a case of fair-use.

If you want to submit code that is not yours, not covered by fair-use, and not covered by ZPL, then this *is* almost certainly a problem, even if the original author agrees with your 'borrowing'.

If in doubt, ask on `zope-dev` first before submitting.

5.15.4 Code on `svn.zope.org`

Code hosted on `svn.zope.org` should comply to the general policy of the Zope Foundation (see below).

For developers there is a package available to ease this task.

1. Download and build `zope.repositorypolicy`:

   ```
   $ svn co svn+ssh://<USERNAME>@svn.zope.org/repos/main/zope.repositorypolicy
   $ cd zope.repositorypolicy/trunk
   $ python bootstrap.py
   $ ./bin/buildout
   ```

   This generates helper scripts to check your project for repository compliance.

2. Run:

```
$ ./bin/zope-org-check-project <PATH-TO-CHECKOUT-OF-MY-PROJECT>
```

This will tell you what is missing or wrong in your package from `zope.repositorypoliiy`-point of view.

If you get no output, nothing is wrong with your package and you're done.

If you get some output, you can fix the problems manually or run:

```
$ ./bin/zope-org-fix-project <PATH-TO-CHECKOUT-OF-MY-PROJECT>
```

to automatically add missing files, fix headers or existing LICENSE files, etc.

After running `zope-org-fix-project` you should make sure, that any new generated files are registered with SVN and update the history (`CHANGES.txt`).

3. Commit the changes.

5.15.5 Appendix: files, headers, forms

On http://foundation.zope.org/agreements you can find (amongst other things):

- The Zope Foundation Committer Agreement
- Zope Public License (current version, 2.1 as time of writing)

There are also general hints for new contributors and contributors-to-be:

http://docs.zope.org/developer/becoming-a-contributor.html

5.16 Set custom configurations on a system level that your application can use

Author Peter Bengtsson

Certain properties are best stored persistently inside your application. Other properties are more appropriate to store on a "system" level.

5.16.1 Purpose

You want to set a configuration variable that affects how your Grok application is running. Perhaps you have two instances running the same application on different servers. One which should log all emails sent through your application since you're doing some statistics or debugging and one where you want to leave it as is.

One alternative is to set it as a persistent property inside the application itself as part of the ZODB but that would mean changes to the functionality or perhaps you feel that the configurations you want to set are a sys admins problem and not the application manager.

Another alternative is to set environment variables in your system and then use `os.environ` to pick up the values but this is generally a bad idea since environment variables are difficult to "connect" to the application and it's really hard to tell if them have been set before your application starts.

5.16.2 Step by step

First of all, the key is to use the file `zope.conf` in `<zopeinstance>/parts/zopectl/` **but** this file is auto-generated by buildout which means that if you make any changes to it (in fact, any changes inside `parts/*`) you're likely to loose them the next time your run buildout.

The ticket is to change the blueprint of your project: `buildout.cfg`. By default it probably looks something like this:

```
[zopectl]
recipe = zc.zope3recipes:instance
application = app
zope.conf = ${data:zconfig}
```

This is your chance to add your own lines to go *into* `zope.conf` which goes on the next line like this:

```
[zopectl]
recipe = zc.zope3recipes:instance
application = app
zope.conf = ${data:zconfig}
            <product-config name_of_product_or_app>
              debug-email-sending 1
              dump-directory ${buildout:directory}/parts/dumps
            </product-config>
```

It might seem a little strange at first but is incredibly powerful and works well. Run buildout again and check for yourself that those extra lines were included by buildout by taking a look at `parts/zopectl/zope.conf`.

Now you can reach these inside your application and use as you please:

```
from zope.app.appsetup.product import getProductConfiguration
# see parts/zopectl/zope.conf
config = getProductConfiguration('name_of_product_or_app')

...
DEBUG_EMAIL_SENDING = config.get('debug-email-sending', False)
DUMP_DIRECTORY = config.get('dump-directory', '/tmp/dumps')
```

The function `getProductConfiguration()` will return `None` if there is no configuration under that name which means that you an `AttributeError` if you call `.get()` on it. So, if you envision that you might not set a configuration at all you should probably add these two lines so it looks like this:

```
config = getProductConfiguration('name_of_product_or_app')
if config is None:
    config = {} # config.get() below will return the default values
```

5.16.3 Further information

At the time of writing this I can't find any documentation other than the Zope 3 API doc which is rather sparse. However, the syntax couldn't be easier to use. It's just a matter of remembering that this great tool is there.

EDIT

This how-to was rewritten after first publish by help of Philipp von Weitershausen. Thanks Philipp!

CHAPTER

SIX

TESTING

HowTos related to testing.

Contents:

6.1 How to test docstrings with `z3c.testsetup`

Author Peter Bengtsson, Uli Fouquet

Version This document is based on Grok <= 1.0

Intended Audience Developers

6.1.1 Purpose

Show how to include the doc strings of your classes into the test suite with *z3c.testsetup*.

Note: The usage of *z3c.testsetup* for testing is generally *deprecated* as Grok in general will switch to wider spread Python testing frameworks like *py.test* or *nose* in future. Heading for new life forms and new civilizations, to boldly go where no man has gone before, we start by using plain unittests and will then leave the sector of *zope.testrunner* deploying fancier forms of testing. If you want to follow this trek, please do not use *z3c.testsetup* (any more). Use plain unittests or *py.test* or *nose* instead and simply ignore this HowTo.

6.1.2 Prerequisities

easy_install and an Internet connection.

6.1.3 Step by step

We're assuming that you don't already have a Grok project up and running but if you already do it should be easy you understand where this all fits in. For the sake of simplicity we'll go through from the start. The first thing is to install grokproject and then modify it's setup.py:

```
$ cd /tmp
$ grokproject Sample
$ cd Sample
$ emacs setup.py
```

The necessary change is to include *z3c.testsetup* in the 'install_requires' (if it is not there already):

```
install_requires=['setuptools',
                  'grok',
                  'z3c.testsetup',
                  ],
```

Once you've added that run *buildout* again:

```
$ ./bin/buildout
```

Next add a file called `tests.py` which according to `buildout.cfg` automatically picks up. Let it contain the following code:

```
import z3c.testsetup

test_suite = z3c.testsetup.register_all_tests('sample',
                                              extensions=['.py','.txt','.rst'],
                                              )
```

The key here is to including doc string tests inside the Python files is in adding `.py` to the parameter `extensions`.

The next step is to modify the file `app.py` a little bit so that it includes a naive but working test. So change `app.py` to look like this:

```
"""
:Test-Layer: unit

Documentation for the module.
"""

class Sample(grok.Application, grok.Container):
   """
   Test something

     >>> from sample.app import Sample
     >>> sample= Sample()
     >>> sample.foo()
     'test'

   """
   def foo(self):
       return 'test'
```

The next step is of course to write the interesting code and a more interesting test. But here's what you expect to happen when you run the test runner:

```
$ ./bin/test
Running tests at level 1
Running unit tests:
  Running:
.
  Ran 1 tests with 0 failures and 0 errors in 0.004 seconds.
```

6.1.4 Further information

The usage of *z3c.testsetup* for testing is generally deprecated as Grok in general will switch to wider spread Python testing frameworks like *py.test* or *nose* in future.

6.2 Writing tests, discovering and running them with Grok.testing

Author ulif

6.2.1 Prerequisites

You need Grok 0.13 or newer installed. If you are developing on an older version of Grok, have a look at z3c.testsetup How-To instead.

To follow along with this How-To, you can install a blank new project called Sample.:

```
$ cd /tmp
$ grokproject Sample
$ cd Sample
```

6.2.2 Step by step

You don't have to download anything else, because Grok already includes all the packages you need.

Test that the project's testrunner works when you run ./bin/test. You should see something like this:

```
$ ./bin/test
Running tests at level 1
Total: 0 tests, 0 failures, 0 errors in 0.000 seconds.
```

Now you're ready to start adding your tests. There are several different kinds of tests and we'll scratch the surface of each one. The tests are either doctest or python and the difference is that you either write them in pure python or as text that you embedd in the doc strings of your classes or in separate plain-text (.txt) files. grok.testing is about discovering tests, not writing them or running them.

Let's write a very simple unit test for the app.py that has been created.:

```
$ cd src/sample
$ mkdir app_tests; cd app_tests
$ touch __init__.py
$ emacs test_app.py
```

Notice the importance of creating an __init__.py file in that new directory to make it a valid Python package. Here's some example code that is really silly but at least proves that the test is found during runs of testrunner:

```
"""
Do a Python test on the app.

:Test-Layer: python
"""
import unittest
from sample.app import Sample

class SimpleSampleTest(unittest.TestCase):
    "Test the Sample application"

    def test1(self):
        "Test that something works"
        grokapp = Sample()
        self.assertEqual(list(grokapp.keys()), [])
```

The next thing is to tell grok.testing where to find this test. That's achieved by creating a file called tests.py in the 'src/sample' directory with the following content:

```
import grok
test_suite = grok.testing.register_all_tests('sample')
```

The 'sample' string here denotes our `sample` package, which we want to be scanned for test files. We could also pass any other package name, which is available at runtime, in 'dotted name' notation.

That's it! How cool is that? I love the Just Works'ism of writing "register_all_tests('sample')" and it does it.

Let's now crack on with a "doc test". One way to get started on a doc test is to run grok in debug mode (typing './bin/zopectl debug') and when you're done, copy and paste what you've written in the interactive prompt. Here's a really simple doctest copy and paste after having run ./bin/zopectl debug. I call this file doctest.txt and I place it in the app_tests directory too:

```
Do a simple doctest test on the app.
************************************
:Test-Layer: unit

When you create an instance there are no objects in it::

   >>> from sample.app import Sample
   >>> grokapp = Sample()
   >>> list(grokapp.keys())
   []
```

Make sure it is found by the testrunner:

```
$ ./bin/test
Running tests at level 1
Running unit tests:
  Running:
..
  Ran 2 tests with 0 failures and 0 errors in 0.007 seconds.
```

These tests are just making sure that you're code is working but we haven't yet tried to run any tests in a full blown Grok environment. That's called functional testing (or integration testing). The first test we'll make is a functional test in python. The setup of the complete Grok environment is called the "functional layer" Fortunately grokproject sets one up for you automatically which you can use in your functional test cases. The magic you need is the instance object *FunctionalLayer* which you'll find in the file testing.py. Again pay attention to the marker in the doc string is set to *python* and also bare in mind that the test is extremely simple. Save this as functional.py in the app_tests/ directory.:

```
"""
Do a functional test on the app.

:Test-Layer: python
"""
from sample.app import Sample
from sample.testing import FunctionalLayer
from zope.app.testing.functional import FunctionalTestCase
class SampleFunctionalTest(FunctionalTestCase):
    layer = FunctionalLayer
class SimpleSampleFunctionalTest(SampleFunctionalTest):
    """ This the app in ZODB. """
    def test_simple(self):
        """ test creating a Sample instance into Zope """
        root = self.getRootFolder()
```

```
        root['instance'] = Sample()
        self.assertEqual(root.get('instance').__class__, Sample)
```

Apologies for the stupidity of the test but at least it's picked up the next time we run bin/test:

```
$ ./bin/test
Running tests at level 1
Running unit tests:
  Running:
..
  Ran 2 tests with 0 failures and 0 errors in 0.021 seconds.
Running sample.testing.FunctionalLayer tests:
  Set up sample.testing.FunctionalLayer in 1.756 seconds.
  Running:
.
  Ran 1 tests with 0 failures and 0 errors in 0.022 seconds.
Tearing down left over layers:
  Tear down sample.testing.FunctionalLayer ... not supported
Total: 3 tests, 0 failures, 0 errors in 1.956 seconds.
```

As you can see, running the functional test is a lot slower. The first two tests took 0.007 seconds and now all three tests took 1.96 seconds. This is why developers sometimes refer to unit tests as "fast tests".

Choosing to write unit tests versus functional tests depends upon what you want to achieve with your testing. Unit tests verify the correct behaviour of code in isolation, useful for ensuring that your code is loosely coupled. They are a quicker form of checking for simple errors than clicking around in a web browser, and they can help find bugs by exposing unseen edge cases. Functional testing ensures that the parts of your application work together as a whole, and are useful for ensuring that your application behaves as desired. If you are new to testing, try both forms of testing, as you will often find yourself mixing and matching from both test approaches depending upon what you want your tests to do.

The final type of test is a functional doc test which is really sexy in its simplicity. Create a file in app_tests/ called functional.txt and let it have the following content:

```
Do a functional doctest test on the app.
****************************************

:Test-Layer: functional

Test creating a Sample instance into Grok::

   >>> from sample.app import Sample
   >>> root = getRootFolder()
   >>> root['instance'] = Sample()
   >>> root.get('instance').__class__.__name__
   'Sample'
```

We now have one python unit test, a doc test, a python functional test and a function doc test. Let's check that they all run:

```
$ ./bin/test
Running tests at level 1
Running unit tests:
  Running:
..
  Ran 2 tests with 0 failures and 0 errors in 0.005 seconds.
Running sample.FunctionalLayer tests:
  Set up sample.FunctionalLayer in 1.755 seconds.
  Running:
```

```
.
  Ran 1 tests with 0 failures and 0 errors in 0.005 seconds.
Running sample.testing.FunctionalLayer tests:
  Tear down sample.FunctionalLayer ... not supported
  Running in a subprocess.
  Set up sample.testing.FunctionalLayer in 1.771 seconds.
  Running:
.
  Ran 1 tests with 0 failures and 0 errors in 0.004 seconds.
  Tear down sample.testing.FunctionalLayer ... not supported
Total: 4 tests, 0 failures, 0 errors in 4.896 seconds.
```

ZopeXMLConfigurationError in ftesting.zcml

If you run into an error stating:

```
ZopeXMLConfigurationError: File ".../ftesting.zcml", line 11.2-13.8
    ConfigurationError: ('Invalid value for', 'component', "ImportError: Couldn't import
zope.app.securitypolicy.zopepolicy, No module named securitypolicy.zopepolicy")
```

chances are, that you are using Grok 0.12 and grokproject. Unfortunately the ftesting.zcml generated by grokproject includes a wrong package. To fix this edit the *ftesting.zcml* in your application root and replace:

```
<securityPolicy
    component="zope.app.securitypolicy.zopepolicy.ZopeSecurityPolicy"
    />
```

with:

```
<securityPolicy
    component="zope.securitypolicy.zopepolicy.ZopeSecurityPolicy"
    />
```

i.e.: remove the 'app' package and all should work again.

6.2.3 Further information

This how-to is meant to be an introduction to get you started on writing tests, how those tests are discovered, and how you can run them. To help making this How-to as short as possible, I've skipped...

- The discussing the various options for z3c.testsetup which is the base of grok.testing.
- The options on the test runner (./bin/test –help).
- Doctests embedded as docstrings inside the Grok classes.
- Browser testing with requests (example here)

Hopefully this how-to can improve over time to make things even simpler and dummy-proof but looking back you'll have to admit that we managed to get a lot done with very little configuration work.

CHAPTER

SEVEN

EGG COLLECTIONS

Contents:

7.1 Dolmen

7.1.1 Tutorial

Getting started with Dolmen

Author Vincent Fretin

Contributors Aroldo Souza-Leite

Version This tutorial has been tested with dolmenproject 1.0a3 and Dolmen 0.5.4.

(last modified: 2011-11-16 03:33:48)

Dolmen is set of thin enhancement layers, that is, very modular, task specific libraries, based on Grok. Among other things, you can use the **Dolmen** libraries to gradually create a **Grok** based CMS.

Dolmen is strongly based on Grok, but you don't need to have experience with **Grok** to start working with **Dolmen**. This tutorial can also be seen as manual for creating your first **Grok** site and start playing with **Grok** using a **Dolmen** site (or simply a **Dolmen**) as a **Grok** sample site.

Here you will learn how to create your first **Dolmen** render it in the browser. You do it by first creating a **Dolmen** project on the file system using the *dolmenproject* script. For this, you have to install a Python virtual environment, activate it and then install *dolmenproject* in this environment.

The examples in this tutorial are done on the operating system Linux Ubuntu, a Linux distribution based on Debian. But as Python and **Grok** run independently from the operating system, it should be easy to take virtually the same steps on a MacOS or MSWindows machine.

Preconditions

In order to install a **Dolmen** for the first time, you need:

- Python-2.6 (see Python)
- the Python tool *virtualenv* installed in your system Python (see virtualenv)
- a separate Python virtual environment
- the Python imaging tool *PIL* installed for your installeed Python.

- the tool *dolmenproject* installed in the Python virtual environment

Python, lxml and PIL

We need to install some packages via the package manager:

```
$ sudo apt-get install build-essential python2.6 python2.6-dev \
    python-imaging libxslt1-dev libxml2-dev
```

We use python-imaging to install PIL and libxslt1-dev libxml2-dev to be able to build lxml.

The Python virtual environment

The main Python system must include a tool called *virtualenv* that creates a separate Python environment. A separate Python virtual environment can be used among other purposes for testing a Python library without changing the main Python configuration on your computer. For the installation of the *virtualenv* utility in your main Python system see the virtualenv documentation on the Python site.

Here and in then next examples, *$* stands for your usual operating system's usual command prompt, whereas *(dolmenenv)$* stands for the prompt indicating that the virtual environment is active.

```
$ virtualenv -p /usr/bin/python2.6 dolmenenv
$ source dolmenenv/bin/activate
(dolmenenv)$
```

A label *(dolmenenv)* will appear on the left of your usual command prompt. This is a sign that you are working in the separate Python environment you created. In order to deactivate it, type *deactivate* and you will see your usual operating system prompt again. Now change the current directory to the location where you want to create **Dolmen** projects:

```
(dolmenenv)$ cd /path/to/my/projects
```

Installing the *dolmenproject* script

You can easy_install dolmenproject:

```
(dolmenenv)$ easy_install dolmenproject
```

Development version of dolmenproject

If you want to work on *dolmenproject*, you can install it from the *git* repository.

```
(dolmenenv)$ git clone git://devel.dolmen-project.org/dolmenproject.git
(dolmenenv)$ cd dolmenproject
(dolmenenv)$ python setup.py install
```

To update *dolmenproject* afterwards, issue `git pull` in the directory and run `python setup.py install` again.

```
(dolmenenv)$ git pull
(dolmenenv)$ python setup.py install       # with the dolmenenv enabled.
```

Using *dolmenproject* to create the Dolmen project

Make sure you changed to the directory where you want to create your **Dolmen** projects and that the Python virtual environment *(dolmenenv)* is active.

Create a **Dolmen** project by calling `dolmenproject` with the project name as a parameter. Here we call it *MyDolmen* but of course you can give it a name of your choice.

```
(dolmenenv)$ dolmenproject MyDolmen
```

You can use an older KGS like this:

```
(dolmenenv)$ dolmenproject \
--version-url="http://www.dolmen-project.org/kgs/dolmen-kgs-0.5.4.cfg" \
MyDolmen
```

Or use development version like this:

```
(dolmenenv)$ dolmenproject \
--version-url="http://gitweb.dolmen-project.org/misc.git?a=blob_plain;f=dolmen-kgs-1.0dev.cfg;hb=HEAD
MyDolmen
```

In the process of creating the **Dolmen** project, you are asked for a user name and a password as a site adminstrator. You have to remember these credentials later on.

Disable the virtual environment:

```
(dolmenenv)$ deactivate
```

Then change to the MyDolmen project root to bootstrap your project with distribute (-d option) and run buildout:

```
$ cd MyDolmen
$ python2.6 bootstrap.py -d
$ bin/buildout
```

Run the **Dolmen** site like this:

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

For development purpose, run it with the following command:

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

This command causes the **Dolmen** process to be more verbose, so that you are told about what is happening through messages in the console.

Point your browser to http://localhost:8080 to see the page being served. You are going to be asked for your credentials. Here you can test your **Dolmen** server by creating a test **Dolmen** application (give it some name like *mydolmen*).

Then click on 'mydolmen' to see your first **Dolmen** application.

There is almost no content in 'mydolmen', but it's already a fully functional **Dolmen** application.

Use a skin

To use a skin for your Dolmen project, you have to do some work on `setup.py` of the MyDolmen root and then change the working directory to `src/mydolmen` to do some work there.

```
$ cd src/mydolmen
```

Add to `setup.py`:

```
menhir.skin.lightblue
```

Add to `configure.zcml`:

```
<include package="menhir.skin.lightblue" />
```

and replace:

```
<!-- Skin  -->
<includeOverrides
    package="dolmen.app.layout"
    file="skin.zcml"
    />
```

by:

```
<!-- Skin  -->
<includeOverrides
    package="menhir.skin.lightblue"
    file="skin.zcml"
    />
```

You can try the *menhir.skin.snappy* skin too, but it may not be up to date with the latest Dolmen changes.

You can remove 'dummydolmen' by pointing your browser back to http://localhost:8080 , checking the 'delete' box and submitting it. You will implement your **Dolmen** application in the following parts of this tutorial.

Customizing your MyDolmen site

Author Vincent Fretin

Contributors Aroldo Souza-Leite

Version unknown

You will customize your **Dolmen** application in the following parts of this tutorial.

Creating your own Dolmen site

In *src/mydolmen/app.py* module (remove all existing code), implement a **Dolmen** site by inheriting from the `Dolmen` class:

```
from dolmen.app.site import Dolmen

class MySite(Dolmen):
    title = u"My project site"
```

But MySite will have a default factory and so will appear in the Add menu. To disable the auto creation of the factory, change it like this:

```
from dolmen.app.site import Dolmen
from dolmen import content

class MySite(Dolmen):
    content.nofactory()
    title = u"My project site"
```

For now, you only have the admin:admin account.

You can add a PAU to your application:

```
from dolmen.app.site import Dolmen
from dolmen import content
from dolmen.app.authentication import initialize_pau
from zope.authentication.interfaces import IAuthentication
from zope.pluggableauth import PluggableAuthentication as PAU
import grok

class MySite(Dolmen):
    content.nofactory()
    title = u"My project site"
    grok.local_utility(PAU, IAuthentication, setup=initialize_pau, public=True, name_in_container="u
```

And you need *menhir.contenttype.user* to create a user directory and create users.

Add to configure.zcml:

```
<include package="dolmen.app.authentication" />
<include package="menhir.contenttype.user" />
```

Add to setup.py:

```
'zope.authentication',
'zope.pluggableauth',
'dolmen.app.authentication',
'menhir.contenttype.user',
```

You have to recreate your application. You can now create a user directory (users) and create some users.

For the user, you can download the original image, with the namespace traverser *download* and the field name *portrait*:

http://localhost:8080/demo/users/vincentfretin/++download++portrait

Anywhere in the portal, you can access to an avatar, a square image:

http://localhost:8080/demo/++avatar++vincentfretin

It's useful to develop a commenting system, like *menhir.simple.comments*.

Adding content types

We'll add an *Image* content type.

Add to *setup.py*:

```
menhir.contenttype.image
```

Add to *configure.zcml*:

```
<include package="menhir.contenttype.image" />
```

Run the buildout. Restart the site. You can now add images.

Creating content types

Creating a folder and a content:

```
from dolmen.file import ImageField
from dolmen.blob import BlobProperty
from zope import schema
from zope.container.constraints import contains
from zope.interface import Interface

class IArtBook(Interface):
    contains(".IPicture")

class ArtBook(content.Container):
    grok.implements(IArtBook)
    content.name(u"My documents")
    content.require("dolmen.content.Add")

class IPicture(content.IBaseContent):
    description = schema.Text(title=u"Description")
    image = ImageField(title=u"Image")

class Picture(content.Content):
    content.schema(IPicture)
    content.name(u'My document')
    content.require("dolmen.content.Add")
    image = BlobProperty(IPicture['image'])
```

We inherit from `content.IBaseContent` which contains the title. We define explicitly a BlobProperty for the image here. The `content.schema` directive will do the grok.implements too.

ArtBook container can contain only Picture objects. Picture can be added everywhere.

Add to *configure.zcml*:

```
<include package="dolmen.blob" />
```

and in your *setup.py*:

```
dolmen.blob
dolmen.file
zope.container
zope.interface
zope.schema
```

For the image, you have access to thumbnails like this:

- http://localhost:8080/demo/image/++thumbnail++image.square
- http://localhost:8080/demo/image/++thumbnail++image.large
- etc.

Changing the view of a content type

To change the view for a content type:

```
from dolmen.app.layout import models

class PictureView(models.Index):
    grok.context(IPicture)

    def render(self):
        return "hello"
```

Include the type Image only in a IArtBook folder

Example:

```
from menhir.contenttype.image import IImage, Image
from zope.container.constraints import contains, containers
from zope.interface import Interface, classImplements


class IArtBook(content.IBaseContent):
    contains(".IPicture", IImage)

class IImageConstraints(Interface):
    containers(IArtBook)

classImplements(Image, IImageConstraints)
```

Include slimbox preview for an image

Code:

```
from megrok.resource import component_includes
from menhir.contenttype.user import UserView
from menhir.contenttype.image import ImagePopup
component_includes(UserView, ImagePopup)
```

It is already done in the *menhir.skin.snappy* theme, but not in the *menhir.skin.lightblue* theme.

Other functionalities

If buildout says that a package was not found, it's probably the package doesn't have a release yet. You normally add it to auto-checkout in `sources.cfg` to fix that.

There are some packages you can add to your buildout to add functionalities to your application. Simply add the package to `setup.py` and `configure.zcml`:

Functionalities:

- dolmen.app.breadcrumbs: breadcrumbs
- dolmen.app.search: search box
- dolmen.app.viewselector: add a view menu
- menhir.simple.livesearch: add a livesearch to the searchbox
- menhir.simple.navtree

Functionalities in development, may be buggy:

- dolmen.app.clipboard: add cut, copy, paste functionalities
- dolmen.app.metadatas: metadata tab on simple content type to edit dublin core metadata.

Content types:

- menhir.contenttype.document
- menhir.contenttype.file
- menhir.contenttype.folder

- menhir.contenttype.rstdocument
- menhir.contenttype.image
- menhir.contenttype.photoalbum

And other packages used in snappy demo:

- menhir.simple.tag
- snappy.*

You have the full list at http://gitweb.dolmen-project.org/?o=project

Differences between Zope2 and Grok

Author Vincent Fretin

Version unkown

Get the user id

Zope 2:

```
self.request.AUTHENTICATED_USER.getId()
```

Dolmen:

```
self.request.principal.id
```

Check if user is anonymous

Zope 2:

```
portal_membership.isAnonymousUser()
```

What this code do is actually:

```
u = getSecurityManager().getUser()
return u is None or u.getUserName() == 'Anonymous User'
```

Dolmen:

```
from zope.authentication.interfaces import IUnauthenticatedPrincipal
IUnauthenticatedPrincipal.providedBy(self.request.principal)
```

Example how to retrieve the email from the authenticated user

Zope 2 / Plone:

```
self.request.AUTHENTICATED_USER.getProperty('email')
```

Dolmen:

```
from menhir.contenttype.user.user import IUser
email = IUser(self.request.principal).email
```

Get the absolute ur of an object

Zope 2:

```
self.context.absolute_url()
```

Grok:

```
from zope.traversing.browser.absoluteurl import absoluteURL
absoluteURL(self.context, self.request)
```

or simply in a grok.View:

```
self.url(self.context)
```

Relations between data

Author unknown

Version unknown

dolmen.relations is a thin layer above *zc.relation*, allowing a simple and straightforward implementation of standalone relationships between objects.

Relationships are very important when you have objects in your application which reference other objects. Imagine you have an Employee object. Each Employee has an attribute "supervisor" which points to another Employee (or None).

You might want to perform miscellaneous queries on that. For example you might want to know all the employees supervised by a certain person. While you could just loop over all employees and check if the "supervisor" attribute matches the supervisor you want to query, this is going to be slow with many objects.

You also might want to perform more complex searches, like find the chain of control upwards from a certain employee.

All of this (and much more) can be done with dolmen.relations. To see how it is used check the README.txt in the dolmen.relation package.

Since dolmen.relation builds on zc.relation it inherits all of its power. I strongly recommend you to read the extensive zc.relation docs at http://pypi.python.org/pypi/zc.relation . They will show you all the queries that can be performed and expand on the Employee example from above. Take your time to reads these docs, it's not the easiest topic, but it really pays off understanding relationships.

7.1.2 Indices and tables

- *genindex*
- *modindex*
- *search*

7.1.3 Support

channel #dolmen at irc.freenode.net

CHAPTER

EIGHT

EGGS IN PRODUCTION

Contents:

CHAPTER

NINE

ENTRY LEVEL EGGS

Contents:

9.1 gp.fileupload

Author unknown

Version unknown

NOTE: Incomplete. Maybe needs zip with full example code.

Using a middleware for larger file uploads is advisable. Very long uploads will block server threads and create long running transactions. This is something you usually want to avoid.

You also might want to show a progress bar to your users. For this you need to be able to query the current upload progress.

gp.fileupload does exactly this. It integrates into the WSGI stack. Basically, when a user uploads a file the gp.fileupload stage of the pipeline will process the incoming data and provide a way to query progress. Once the upload is complete your application will receive a POST request with the filename of the file stored in a temporary folder. You can then use it for your own purposes.

Just add gp.fileupload to your setup.py. Then add a snippet like below to your debug.ini/deploy.ini .

Snippet from debug.ini:

```
[filter:fileupload]
use = egg:gp.fileupload
tempdir= %(here)s/tmp/
upload_to = %(here)s/myapp.upload.files/
include_files = jquery.fileupload.*
exclude_paths = /@@ /.+\.direct
max_size = 500

[pipeline:main]
pipeline = fileupload egg:myapp#debug
```

This puts the fileupload module in the WSGI stack.

From your grok.View class serve html code like this:

```
<form enctype="multipart/form-data" method="POST" action=".?gp.fileupload.id=1">
    <input type="file" name="file" />
    <input type="submit" />
</form>
```

Where 1 is the session id. The session id **must** be a digit.

When the form is submitted, you can use some ajax stuff to get the stats of the upload with the url:

```
http://yourhost/gp.fileupload.stat/1
```

This will return some JSON data like:

```
{'state': 1, 'percent': 69}
```

state can have the following values:

- *0*: nothing done yet.
- *1*: upload is active
- *-1*: file is larger than max_size.

You can use this to display the upload progress.

For detailed information please take a look at these two pages:

_gp.fileupload documentation: http://www.gawel.org/docs/gp.fileupload/ and _PyPI page of gp.fileupload: http://pypi.python.org/pypi/gp.fileupload .

9.2 zope.sendmail

Author unknown

Version unknown

9.2.1 Example config

```
<configure xmlns="http://namespaces.zope.org/zope"
           xmlns:grok="http://namespaces.zope.org/grok"
           xmlns:mail="http://namespaces.zope.org/mail">

  <include package="zope.sendmail" file="meta.zcml" />
  <include package="zope.sendmail" />

  <grok:grok package="." />

  <mail:smtpMailer
      name="myapp.smtp"
      hostname="smtp.myhost.com"
      port="25"
      username="user@myhost.com"
      password="topsecret"
      />

  <mail:queuedDelivery
      name="myMailer"
      permission="zope.Public"
      mailer="myapp.smtp"
      queuePath="./mailqueue"
      />

</configure>
```

9.2.2 Example integration with grok

```
import grok
from zope.component import getUtility, getMultiAdapter
from interfaces import IMyMailer
from zope.sendmail.interfaces import IMailDelivery

import email.MIMEText
import email.Header
import email.utils

defaultSender = ( 'My app', 'user@myhost.org' )

class MyMail(grok.GlobalUtility):
    grok.implements( IMyMailer )

    def __init__(self, defaultSender = defaultSender):
        self.sender = defaultSender

    def send_mail(self, recipient, subject, body, sender = None, format = 'html'):
        ''' format can be 'html' or 'plain' for example '''
        if sender is None:
            sender = self.sender
        mailer = getUtility( IMailDelivery, 'myMailer' )
        msg = email.MIMEText.MIMEText( body.encode('UTF-8'), format, 'UTF-8' )
        msg["From"] = email.utils.formataddr( sender )
        msg["To"] = recipient
        msg["Subject"] = email.Header.Header(subject, 'UTF-8')
        mailer.send( sender[1], [recipient], msg.as_string() )

    def send_mail_with_view(self, context, request, viewname, recipient, defaultSubject, sender = Nor
        view = getMultiAdapter( (context, request), name = viewname )
        if getattr( view, 'mailSubject', None ) is None:
            view.mailSubject = grok.title.bind().get(view, default = defaultSubject)
        mailBody = view()
        return self.send_mail( recipient, view.mailSubject, mailBody, sender, 'html' )

# somewhere in your app...
user = getUserHere()
mailer = getUtility( IMyMailer )
mailer.send_mail_with_view( PasswordReset(user, newPassword), self.request, 'passwordresetemail', use
```

CHAPTER

TEN

SNIPPETS

Snippets are short pieces of code which are generally useful. Examples are getting the current user or finding out if you are running in devmode.

Contents:

10.1 Getting the current user/principal

You often might want to get the user currently accessing your page. The best way is to fetch the user (or principal) is via the page's request. It's done like this:

```
self.request.principal
```

If you don't have a request at hand (you really should have one) you can use this somewhat hackish workaround:

```
import zope.security
principal=zope.security.management.getInteraction().participations[0].principal
```

If you are using Dolmen and menhir.contenttype.user you can use code like this to get the menhir.contenttype.user.User object:

```
def getCurrentUser(request):
    # todo: could use principal.id instead of this magic looking stuff
    return request.principal.__parent__[request.principal.__name__]
```

10.2 Finding out if you are in devmode

If you start grok via debug.ini it will run in devmode by default. Devmode is useful for giving more debug information for example.

To test whether your application is running devmode, try this piece of code:

```
from zope.applicationcontrol.runtimeinfo import RuntimeInfo
def in_dev_mode():
    return RuntimeInfo(None).getDeveloperMode() == 'On'
```

10.3 Getting custom config from ZCML

If you want to retrieve some configuration options from zcml use this code:

```
from zope.app.appsetup.product import getProductConfiguration
config = getProductConfiguration('database')      # plug your own name in here
dbname = config.get('dbname', 'default')          # query your own attribute here
```

10.4 Getting the user's language

See *How to internationalize your application*.

CHAPTER

ELEVEN

SOLVING COMMON TASKS

This document provides a list of common tasks you might encounter during development of your application. Examples include a file-upload feature, exporting your data, sending mails or sending JSON responses.

Contents:

11.1 File Uploads

While you can upload files with grok out-of-the-box it needs a bit more work if you want to support uploads of larger files or want to display progress information to your user.

This document holds a list and discussion of packages which are used to solving this task.

11.1.1 gp.fileupload

This is the only package I am aware of currently which solves this problem. It integrates into the wsgi stack. For a complete tutorial, please see the *gp.fileupload* tutorial.

Personal opinion: Is easy to use and works well. Suitable for production use.

11.2 JSON

When developing ajax applications you often have to deal with JSON data.

This document holds a list and discussion of packages which are used to solving this task.

11.2.1 grok.JSON

Grok's built-in support for views which want to return JSON data.

11.2.2 simplejson

Is a python package which can convert python objects to and from json data.

11.2.3 jsonor

http://bitbucket.org/faassen/jsonor

Implementation of a json schema implementation that can also do content conversion for JSON input, along with some other features.

11.2.4 megrok.attributetraverser

http://svn.zope.org/Sandbox/cklinger/megrok.attributetraverser/

11.3 Sending e-mail

You often want to send e-mails from your application, e.g. for user registration or for sending reminders.

This document holds a list and discussion of packages which are used to solving this task.

11.3.1 zope.sendmail

For a full tutorial see *zope.sendmail* .

http://pypi.python.org/pypi/zope.sendmail

Personal opinion: Easy to configure and use.

11.4 Using a relationfield to express relationships between objects

Author Unknown

Version unknown

11.4.1 Introduction

It is sometimes desirable to somehow have objects store references to other objects. There are different ways to do this, some better than others, some easier than others. I'll show you one way of doing it, based on the z3c.relationfield and z3c.relationfieldui packages. I won't claim it's the best way out there, but I certainly found it easy!

11.4.2 Prerequisites

You should know how to work with schema and AddForms based on them. You should also know about catalogs, indexes and integer IDs.

11.4.3 How Relations work

The z3c.relationfield (currently) provides 3 fields you can use as schema fields in your interface definitions: Relation, RelationChoice and RelationList. The value(s) they store are RelationValue objects.

You could store relations to objects by using standard python references, by doing

```
a = A()
b = B()
...
b.rel = a
```

This will work perfectly, and even if 'a' and 'b' are already stored separately in the ZODB and afterwards related like that, the ZODB is smart enough not to store a copy of 'a', but a real reference to 'a'. (which is, by the way, a common misconception about the ZODB [1]). But this has a problem: if you later intend to remove 'a' from the ZODB (by removing it from its container), it won't actually be deleted. You won't find it under its previous container, but 'b.rel' will still yield the 'a' object, as 'b' holds a reference to it.

This is the same problem normal indexes in catalogs have, and they solved it by using integer IDs for objects: each object that is entered into the ZODB gets its own unique ID. If you have this ID, you can later look up the object it represents, whenever you need it.

The same problem can usually use the same solution, so a RelationValue merely stores the integer ID of the 'to' object. As the RelationValue is meant to be stored as an attribute of the 'from' object, storing the id of the 'from' object doesn't make much sense, so a real reference is stored here. With the 'to_object' and 'from_object' attributes you can get to the actual objects of the RelationValue, 'to_id' and 'from_id' gives access to the integer IDs.

This, of course requires you to have an Unique Integer ID utility activated in your app. You can do this by configuring your application like this:

```
from zope.app.intid.interfaces import IIntIds
from zope.app.intid import IntIds

class Relations(grok.Application, grok.Container):
  grok.local_utility(IntIds, provides=IIntIds)
```

11.4.4 Searching relations

A very handy and important feature of zc.relation is the relation catalog (zc.relation.catalog.Catalog): it enables you to search for specific relations, like 'Give me all employees that have to report to this manager'. However, zc.relation requires you still to define indexes, it just provided the catalog functionality. Luckily z3c.relationfield provides a RelationCatalog that derives from zc.relation.catalog.Catalog, but automatically creates the needed indexes. So creating the catalog is quite easy:

```
from zc.relation.interfaces import ICatalog
from z3c.relationfield import RelationCatalog

class Test(grok.Application, grok.Container):
  grok.local_utility(IntIds, provides=IIntIds)
  grok.local_utility(RelationCatalog, provides=ICatalog)
```

z3c.relationfield also defines the appropriate event handlers so that all objects that have relations are automatically indexed. This is done by using the 'marker interfaces' IHasOutgoingRelations, IHasIncomingRelations, IHasRelations. The first one indicates that the object refers to another object, the second one that the object can be referred to by another object. The 3rd one is a combination of both

11.4.5 Relation and RelationChoice

Enough theory, let's define an application to use our relations in. Imagine you want an app in which everyone can have a buddy. So every 'buddy' object can have a 'link' to another buddy object. The relationfieldui[4] package has 2 possible schema fields for this: Relation and RelationChoice. The difference is really about what widget will be used to render it in a form. Relation will render a textbox and a button that will popup a window in which you are to select

an object and then via AJAX fill in the path to the object. The other one however, works just like an ordinary Choice field, so we'll use that one. Let's define the buddy interface like this:

```
from zope.interface import Interface
from zope import schema
from z3c.relationfield import RelationChoice

class IBuddy(Interface, IHasRelations):
    name = schema.TextLine(title=u'Name')
    buddy = RelationChoice(title=u'Buddy', source = BuddySource(), required = False)
```

Note the use of the IHasRelations marker interface.

As the RelationChoice field works like a normal Choice field, you need to specify a set of values, a vocabulary or a source where it will draw it's values from. I use a source, and the relationfieldui defines a handy baseclass you can use: RelationSourceFactory.

```
from z3c.relationfieldui import RelationSourceFactory

class BuddySource(RelationSourceFactory):
    def getTargets(self):
        return [b for b in grok.getSite().values() if IBuddy.providedBy(b)]
    def getTitle(self, value):
        return value.to_object.name
```

All you need to do is define a getTargets method that returns an iterable to all the objects that can be a target for your relation, and a getTitle method that can turn a RelationValue into a human readable text to show in the drop- down list.

Another thing you'll be needing is an IObjectPath implementation, like this:

```
from z3c.objpath.interfaces import IObjectPath
from z3c.objpath import path, resolve

class ObjectPath(grok.GlobalUtility):
  grok.provides(IObjectPath)
  def path(self, obj):
      return path(grok.getSite(), obj)
  def resolve(self, path):
      return resolve(grok.getSite(), path)
```

This class can create a path to an object or an object from a path. A path is really how to get to the object in the ZODB. The RelationSourceFactory uses this to create tokens from objects.

Now let's implement IBuddy and create an AddForm and a View:

```
class Buddy(grok.Model):
  grok.implements(IBuddy)

class AddBuddy(grok.AddForm):
  grok.context(grok.Container)
  grok.name('add')
  form_fields = grok.Fields(IBuddy)

  @grok.action('Add')
  def Add(self, **data):
      buddy = Buddy()
      self.applyData(buddy, **data)
      self.context[buddy.name] = buddy
      self.redirect(self.url(buddy))
```

```
class BuddyView(grok.View):
  grok.context(IBuddy)
  grok.name('index')
```

The buddyview template may look like this:

```
<html>
<head></head>
<body>
<h1 tal:content="context/name"/>
<span tal:condition="context/buddy/to_object | nothing">
 <dl>
       <dt><em>My Buddy:</em></dt>
       <dd  tal:content="context/buddy/to_object/name"/>
 </dl>
</span>
</body>
</html>
```

The main Index view can be changed to

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

  def buddies(self):
      return [b for b in grok.getSite().values() if IBuddy.providedBy(b)]
```

Your main index view template can look like this

```
<html>
<head></head>
<body>
<h1>Buddies</h1>
<ul>
<li tal:repeat="buddy view/buddies"><a tal:attributes="href python:view.url(buddy)" tal:content="budd
</ul>
<a tal:attributes="href python:view.url('add')">Add buddy</a>
</body>
</html>
```

Now start the server, add your app (I named it 'test'), and go to http://localhost:8080/test. Click 'Add Buddy' and you should see a form. Enter the name and click add. Of course you can't select a buddy yet, as there are none. If you now go back to the add form (and refresh the page), you'll notice that the buddy you added before can now be selected as the buddy for the buddy you are to create.

11.4.6 Am I my buddy's buddy?

How can we now find out who I am a buddy of? By searching the relation catalog! Add this method to the BuddyView:

```
from zope import component

  def referers(self):
      catalog = component.getUtility(ICatalog)
      intids = component.getUtility(IIntIds)
      return list(catalog.findRelations({'to_id': intids.getId(self.context)}))
```

Change the buddyview template to this:

```
<html>
<head></head>
<body>
<h1 tal:content="context/name"/>
<span tal:condition="context/buddy/to_object | nothing">
 <dl>
       <dt><em>My Buddy:</em></dt>
       <dd  tal:content="context/buddy/to_object/name"/>
 </dl>
</span>
<span tal:define="referers view/referers" tal:condition="referers | nothing">
       <dt><em>I am a buddy of</em></dt>
       <dd tal:repeat="ref referers" tal:content="ref/from_object/name"/>
</span>
</body>
</html>
```

11.4.7 Hiding the relation inside the implementation

You most probably noticed the gratuitous use of 'to_object' to actually get to the referred object. This can be quite annoying: imagine you want to get the name of your buddy's buddy:

> me.buddy.to_object.buddy.to_object.name

Not so nice...

I'll present you a way of getting rid of this by hiding the entire relation inside the application so that my grandfathers again become me.father.father and me.mother.father, so to speak. I also feel that using the RelationChoice directly in the interface is like forcing the users of the interface to know how to use z3c.relationfield, while all they want is to get to the actual objects you refer to. You might also want to refactor an existing app that implemented references in a different way (for instance by simply storing the id) to zc.relationfield without wanting to change any existing interfaces.

Remove the IHasRelations from the IBuddy interface:

```
class IBuddy(Interface):
  name = schema.TextLine(title=u'The Name')
  buddy = schema.Choice(title=u'My Buddy', required=False, source = BuddySource())
```

And change the BuddySource to an 'ordinary' source:

```
from zc.sourcefactory.basic import BasicSourceFactory

class BuddySource(BasicSourceFactory):
  def getValues(self):
      return grok.getSite()['buddies'].values()
  def getTitle(self, value):
      return value.to_object.name
```

Now we need to adapt the IBuddy implementation, which is the Buddy class. We still will be using a RelationValue somewhere in the background, and use that each time the buddy attribute is accessed. You might be tempted to try something like this:

```
class Buddy(grok.Model):
  grok.implements(IBuddy, IHasRelations)
  buddy_rel = Relation()
```

However, this won't work, because the relationfield packages automatically creates the relation indexes by looking up all fields of the *interfaces* the newly created object implements and checking if they provide IRelation (or IRelationList). In this case, there is no *interface* that has any, so the relation catalog will not be correctly updated.

The trick here is to first subclass the IBuddy interface, and then implement that new interface:

```
class IRelationBuddy(IBuddy, IHasRelations):
  buddy_rel = Relation()

  def referers():
      "iterate over all referers"

class Buddy(grok.Model):
  grok.implements(IRelationBuddy)

  def setBuddy(self, obj):
      intids = component.getUtility(IIntIds)
      self.buddy_rel = RelationValue(intids.queryId(obj))
  def getBuddy(self):
      return self.buddy_rel.to_object
  buddy=property(getBuddy, setBuddy)

  def referers(self):
      catalog = component.getUtility(ICatalog)
      intids = component.getUtility(IIntIds)
      return [r.from_object for r in catalog.findRelations({'to_id': intids.getId(self)})]
```

As you may have noticed, I also moved the 'referers' method from the view to the IRelationBuddy class.

Now you can change the buddyview.pt into:

```
<html>
 <head></head>
 <body>
 <h1 tal:content="context/name"/>
 <span tal:condition="context/buddy | nothing">
  <dl>
      <dt><em>My Buddy:</em></dt>
      <dd  tal:content="context/buddy/name"/>
  </dl>
 </span>
 <span tal:define="referers context/referers" tal:condition="referers | nothing">
      <dt><em>I am a buddy of</em></dt>
      <dd tal:repeat="ref referers" tal:content="ref/name"/>
 </span>
 </body>
 </html>
```

Notice the lack of 'to_object' and 'from_object'!

I'll leave the use of the RelationList as an exercise for the reader...

11.4.8 References

[1] http://faassen.n–tree.net/blog/view/weblog/2008/06/20/0 [2] http://pypi.python.org/pypi/zc.relation [3] http://pypi.python.org/pypi/z3c.relationfield [4] http://pypi.python.org/pypi/z3c.relationfieldui

11.5 Workflow

Often you might want to control the way your content is created. For this you can create workflows. They define how a content object can be modified and by which persons for example.

This document holds a list and discussion of packages which are used to solving this task.

11.5.1 zope.wfmc

http://pypi.python.org/pypi/zope.wfmc

Big package. GUI tools for editing workflows available.

11.5.2 hurry.workflow

http://pypi.python.org/pypi/hurry.workflow

Personal opinion: Seems to be a bit complicated sometimes.

11.5.3 repoze.workflow

http://pypi.python.org/pypi/repoze.workflow

Workflow definition in ZCML possible.

11.5.4 trollfot's workflow

Ask him :)

11.6 Exporting content as xml

Sometimes you need to interface with external applications. Often this can be done via xml, so you need a way to convert your content data to xml.

This document holds a list and discussion of packages which are used to solving this task.

11.6.1 z3c.schema2xml

http://pypi.python.org/pypi/z3c.schema2xml

Allows you to convert your schema-defined objects to xml. Can be extended with custom serializers.

11.7 How to automatically install and maintain your Grok application in to the ZODB

Use the zope.generations infrastructure to automatically install and upgrade your application data when Grok starts.

11.7.1 Purpose

When I develop my Grok applications I need for my app to be already availabe after first Grok startup and I also need a decent way of safely upgrade application's data after the deployment of any new release of application's sources. This is a tiny how to that explains howto use the zope.generations infrastructure in order to do so.

11.7.2 Prerequisities

To demonstrate the usage of zope.generations we will create a grokproject called example. We need the zope.generations package available in our example grokproject. We can archive this by adding the package zope.generations to our setup.py install_requires:

```
install_requires=['setuptools',
                  ...
                  # Add extra requirements here
                  'zope.generations',
                  ],
```

11.7.3 Step by step

For installing an app without the grok-admin-interface we have to define a package e.g. example.generations. This package will contain a utility which manages the update and installation process, and little scripts which does the work.

Let's take a look in the generations package:

```
-- generations
|   -- __init__.py
|   -- evolve1.py
|   -- install.py
|   -- util.py
```

Ok let's create the Manager which holds all information in the util.py

```
import grok
from zope.generations.generations import SchemaManager
from zope.generations.interfaces import ISchemaManager


schemaManager = SchemaManager(
    minimum_generation=0,
    generation=0,
    package_name='example.generations'
)


grok.global_utility(
    schemaManager,
    provides=ISchemaManager,
    name="example",
    direct=True
)
```

This means we register an instance of SchemaManager as global utility. This utility is picked up on startup. As it is the first run, the utility will look for a module called install.py:

```
from example.app import Example
from zope.generations.utility import getRootFolder

def evolve(context):
    root = getRootFolder(context)
    root['app'] = Example()
    print "CREATED App"
```

I think it's not difficult to understand what goes on here. This script is found and executed by the SchemaManager on startup. It get's the root folder and creates an instance of our Example Application it it. Internally the SchemaManager saves that the script was executes, so it get's executed one times.

Step 2 how can we update some attributes on startup: We have to tell the SchemaManager that he should look for new scripts on startup. So let's modify the our SchemaManager in the util.py package

```
schemaManager = SchemaManager(
    minimum_generation=1,
    generation=1,
    package_name='example.generations'
)
```

As you can see we increase the 'generation', 'minimum_generation' attribute to 1. This means that on startup the SchemaManager will look for a module called 'evolve1.py'.

This is our sample evolve1.py:

```
from zope.generations.utility import getRootFolder

def evolve(context):
    root = getRootFolder(context)
    app = root['app']
    app.title = u"Hello World"
```

We simply get our Example Application from the root folder, and add an attribute title with the value "Hello World" to it.

11.7.4 Further information

You can have a look in the tests of zope.generations.

CHAPTER

TWELVE

USER TUTORIALS

Contents:

12.1 Contribute to the Grok documentation

Author Uli Fouquet

Guidelines, policies and help for how you can contribute to the Grok documentation effort.

Note: The community driven documentation is being migrated to a sphinx-based setup. This tutorial needs to be updated accordingly.

Contents:

12.1.1 Basic Guidelines

Note: The community driven documentation is being migrated to a sphinx-based setup. This tutorial needs to be updated accordingly.

About our editoring policies, preferred documentation format and content licensing.

Overview

The Documentation section of the Grok web site welcomes your contributions. We welcome any documentation that another Grokker may find helpful.

You will need to register with this web site before you can contribute. A person in the Documentation group will review your content it publish it. The Documentation group is also a very open group, so if you've contributed some content please ask a Grok web site person and they'll add your account to the Grok Documentation group. This will give you full permissions over the Documentation area of the Grok web site.

There is an effort to maintain community documentation that was announced on the mailing list:

Announcement Grok-doc

Content Types

We have two main types of content you can add to the Documentation section right now:

- Create a How-To. A single page of documentation. Bite-sized help that should focus on a single task.
- Create a Tutorial. A multi-page work of documentation. If you are tackling a sizeable asset of documentation, then we recommend that you use the Tutorial content type even if you only write a single page to begin with. The Tutorial is useful if you later want to add pages that are only example code, or want to expand your documentation into a more complete work.

In a few places on the documentation area we currently dislpay "How-To" or "Tutorial" but we may remove these at a later date, since people browsing through documentation will only be interested in navigation by topic or audience and not by content-type.

ReStructured Text

The preferred format for documentation is using ReStructured Text (ReST). You can enable this for your account by going to "Preferences" -> "Personal Preferences" and choose "Basic HTML textarea editor".

If you do not know restructured text, it's a standard Python documentation format that allows you to apply structure to plain text. The ReStructured Text Primer serves as a quick introduction to the format. Also the ReST Extensions page will tell you how you can format your ReST documentation so that source code will have line numbers and syntax highlighting applied to it (yummy!).

While ReStructured Text is our preferred format, if you are only comfortable working in a WYSIWIG editor such as Kupu or already have documentation in HTML format we still welcome your contribution to the site. Our site editors may later on convert HTML into ReStructured Text format where appropriate.

We may take selected content in ReStructured Text format on the web site, and include this as part of the Grok package itself inside the /doc/ directory for the convenience of developers who prefer to access documentation in plain-text.

Documentation Web Site Bugs

There are currently a couple of small bugs in our web site that you may run into. We intend to fix these bugs, but for now ...

When making subsequent edits to ReStructured Text formatted documents, please take care this format is selected before you hit "Save". There is bug with the web site at the moment where it defaults to Plain Text.

You are also prompted for a Format for the Summary field. This field *must* be plain text. The Summary field is displayed in lots of areas other than the web page, such as RSS feeds, search engine results and other contexts where the only format is plain text.

Editing and Comments

Any documentation which has typos or incorrect information is free to be corrected by anyone in the Documentation group. In addition, members of the Documentation group may incorporate or respond to comments made to documentation to better clarify or improve existing documentation. We will remove comments from the site after we incorporate them into the main body of the text.

12.1.2 ReStructured Text (ReST) Extensions

Note: The community driven documentation is being migrated to a sphinx-based setup. This tutorial needs to be updated accordingly.

Write documentation that makes use of the extended set of text-roles and directives available on this web site.

How to write docs with extended ReST

Before you proceed, you should be familiar with the standard roles and directives provided by the standard `docutils` package.

Primers are available all over the net. The page:

```
http://docutils.sourceforge.net/rst.html
```

might serve as a starting point.

The extensions of this package provide additional `docutils` roles and directives, that are related to API entities like functions, classes etc.

Using Roles

Roles are written by enclosing a role-keyword in colons (':') followed by a term that is enclosed in backticks:

```
:<rolename>:`<term>`
```

In the rendered document, the rolename should disappear and the term itself be rendered in a special fashion.

Example: a role:

```
Here we talk about the function :func:`my_function`
```

This renders as:

Here we talk about the function `my_function()`

In this example, the *func* role is used to express that *my_function* is a function name. You can also use whitespaces in the term:

Example: role-term with whitespace:

```
A more elaborated description: :func:`my_function(param1, param2)`
```

which renders as:

A more elaborated description: `my_function(param1, param2)()`

There are many more roles than only the *func* role defined in this package. Please see *README.txt* for a complete list.

Using Directives

Directives are used to mark a whole block of text as special. Their general syntax is as follows:

```
.. <directive-name>:: <directive-header>

   <text-block>
```

It is very important, that the text block is indented. This way, the parser knows that it belongs to the directive.

Example: simple directive:

```
Normal text.

.. function:: my_func(param1,[ param2=None])

   A function to do something.

   *param1* -- a parameter.

   *param2* -- an optional parameter.

   Use this function with care.

Normal text again.
```

This will render as:

> Normal text.
>
> **my_func** (*param1*[, *param2=None*])
>
> > A function to do something.
> >
> > *param1* – a parameter.
> >
> > *param2* – an optional parameter.
> >
> > Use this function with care.
>
> Normal text again.

In this example the *function* directive is used to indicate, that a complete function description is given. In the text block you can write, whatever you like, as long as it is indented correctly.

Some directives require a text block (or 'body' or 'content') to exist. This includes the version-related directives *versionadded*, *versionchanged* and *deprecated* as well as the *seealso* directive.

Some directives require a heading. This includes the version-related directives, which take the heading as a version number.

Example: 'versionadded' directive:

```
Normal text.

.. versionadded:: 0.11

   This function was added because of trouble with the core modules.

Normal text again.
```

This renders to:

> Normal text. New in version 0.11. Normal text again.

You can (and should) nest directives:

Example: nested directives:

```
Normal text.

.. function:: my_func(param1,[ param2=None])

  A function to do something.

  *param1* -- a parameter.
```

```
  *param2* -- an optional parameter.

  Use this function with care.

  .. versionadded:: 0.11

     This function was added because of trouble with the core modules.

Normal text again.
```

This is rendered as:

> Normal text.
>
> **my_func** (*param1* [, *param2=None*])
>
> A function to do something.
>
> *param1* – a parameter.
>
> *param2* – an optional parameter.
>
> Use this function with care. New in version 0.11.
>
> Normal text again.

As you can see, nesting is done simply by more indenting parts. Here, the *versionadded* directive is written as part of the function-description.

Syntax highlighting

Syntax highlighting is enabled by using the *sourcecode* or *code-block* directive (both names are aliases), with a language name as 'heading':

Example: code block with syntax highlighting:

```
Normal text.

.. code-block:: python

  class Cave(object):
      pass

Normal text (continued).
```

This will be rendered as:

> Normal text.
>
> ```python
> class Cave(object):
>     pass
> ```
>
> Normal text (continued).

In this example *python* is the option, that tells *pygments*, the syntax- highlighting engine, that works in background, which type of code you want to be highlighted.

Other supported code types are:

- *pycon* (Python console sessions, the stuff with >>>)
- *pytb* (Python tracebacks)

- *sh* (Bash or other shell scripts)
- *bat* (DOS/Windoes batch files)
- *html* (HTML)
- *xml* (XML)
- *css* (Cascaded stylesheets)
- *js* (Javascript, aka ECMA-script)
- *c* (C-code)
- *cpp* (C++-code)
- ... dozens of code types more.

A complete list can be found here:

```
http://pygments.org/docs/lexers/
```

If you have some kind of code, that is not supported natively, for example a more esoteric configuration file, then you can use 'text'.

To enable linenumbers, you can use the *:linenos* option.

Example: Code with linenumbers:

```
.. code-block:: python
   :linenos:

   class Cave(object):
       pass
```

This will be rendered as:

```
1  class Cave(object):
2      pass
```

Note, that the linenumber are 1-based, i.e. first linenumber will be the number one (not zero).

There are some more directives than only the ones described above defined in this package. Please see *README.txt* for a complete list.

12.2 A Grok-Centric Explanation of Adaptation

Author Kevin Teague

Modified Jul 19, 2008 07:08 AM

Contributors Danny Navarro

Adaptation is a central concept in the Zope 3 Component Architecture. This explanation approaches it from a Grok perspective.

Contents:

12.2.1 How do Adapters compare to the MVC architecture?

In Grok you have View and Model components. Applications written in them are also likely to generate additional code that needs to sit between your Views and your Models. Adapters are one solution for keeping this code out of your Models or Views and putting them into a place where they can easily be reused. Put as simply as possible, an Adapter takes the Interface of an existing component and adapts it to provide another Interface.

If you are used to other web frameworks that use a Model-View-Controller (MVC) architecture, such as Ruby on Rails, you may be thinking, “Isn’t the glue code between the Model and the View called a Controller?” Adapters differ from common MVC web architecture in that your Views can either access the Model directly, or extend the Model using one or more Adapters as intermediate actors. Adapters differ from Controllers in that they can adapt any type of component, and not just a Model component.

The Zope Component architecture approach page says, “The main idea in the Zope Component Architecture is the use of components, rather than multiple-inheritance for managing complexity”. One of the benefits of Adapters over multiple-inheritance is that you can extend the behavior of a Class without needing to modify the code for that Class. If you are acting upon a component provided by someone else, and that component has a well defined interface, then adaption is a much more robust way of extending the behaviour of that component than using meta programming techniques such as monkey patching.

12.2.2 A real world example of Adaption

Imagine that you are about to give a presentation at a meeting. You’ve brought your laptop, and the room has a projector and a cable to plug your laptop into the projector. However, the cable you brought won’t plug into the projector because your cable only interfaces with a DVI port, and the projector only interfaces with a VGA port.

So you ask of the room, “does anyone have a DVI to VGA **adapter**?”

An adapter? Let’s state the problem of the projector and the laptop in Grok. Of course it’s impossible to implement a hardware problem in software (duh!) so for this example we won’t have any implementation details:

```
import grok
from zope.interface import Interface

class IVGAPort(Interface):
    "Video connection using VGA"

    def connectToVGA(cable):
        "Connect to a VGA video cable"

class IDVIPort(Interface):
    "Video connection using DVI"

    def connectToDVI(cable):
        "Connect to a DVI video cable"

class DVIToVGAAdapter(grok.Adapter):
    "Use your DVI laptop with a VGA projector"
    grok.adapts(IDVIPort)
    grok.provides(IVGAPort)

    def connectToVGA(cable):
        # self.context will be a reference to the laptop object

def plugLaptopIntoProjector(laptop, projector):
    "Display your presentation on the wall"
```

```
    vga_capable = IVGAPort(laptop, None)
    if vga_capable is None:
        print "Uh oh, looks like you forget to bring an adapter."
    else:
        vga_capable.connectToVGA(projector)
```

A common idiom for asking the Zope 3 component architecture to perform adaptation is to instantiate a new Interface with the object that you wish to adapt as an argument. We did this in the 'plugLaptopIntoProjecter' function with the line:

```
vga_capable = IVGAPort(laptop, None)
```

The vga_capable variable is now an object that provides the IVGAPort. If the laptop already implements IVGAPort, then this will be the laptop object itself. If the laptop provides IDVIPort, then the DVIToVGAAdapter will be used to extend IDVIPort to provide IVGAPort. The second optional argument (`None`) will be returned if the laptop doesn't provide an IVGAPort and no suitable adapter can be found. If adaptation was successful, you shouldn't care if you have a Laptop object or a DVIToVGAAdapter object returned - your only concern is that you have an object that provides the requested IVGAPort interface that provides the necessary `connectToVGA(projector)` method.

The DVI to VGA Adapter

Let's review the DVIToVGAAdapter class that we created.

```
class DVIToVGAAdapter(grok.Adapter):
    "Use your DVI laptop with a VGA projector"
    grok.adapts(IDVIPort)
    grok.provides(IVGAPort)

    def connectToVGA(cable):
        # self.context will be a reference to the laptop object
```

When you inhert from grok.Adapter, Grok will do two things. One is to register your Adapter with the Zope 3 Component Architecture so that it can look-up the right adapter for the requested interface. The second is to reference the object that is being adapted to an attribute named `context`. In Grok, the Adapter base class that you inherit from is very simple and looks like this:

```
class Adapter(object):

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

This is just a convention used to save some typing. If you wish, you may supply your own constructor to call the object being adapted something specific to your adapter. For example, in the above example, we could add the following constructor to our DVIToVGAAdapter Class:

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

Adaptation is about requiring an object which provides one interface, and extending it so that you get a new interface. We declare that we *require* an object that provides an IDVIPort interface - this is declared with the directive `grok.adapts(IDVIPort)`. We are *extending* the IDVIPort interface to provide a new interface, IVGAPort - this is declared with the directive `grok.provides(IVGAPort)`. Finally, the adapter is responsible for fulfilling the interface that it declares it implements. This implementation will typically use `self.context` attribute - which is the object that is being adapted.

It is possible to use either `grok.provides(IVGAPort)` or `grok.implements(IVGAPort)` directives for the DVIToVGAAdapter. In a more complex example it's possible that an adapter provides multiple interfaces, but

only one interface is meant to be used for the purposes of adaptation. The `grok.provides` directive explicitly declares the interface to be used for adaptation, however if an Adapter only declares that it implements a single

12.2.3 Adapters in a Web Application

While using physical adapters to connect laptops is something that we can understand, where would using adapters in a web application make sense?

While adapters can perform any action, one use for them is to add additional, reusable HTML specific functionality to your application.

Let's say that you have a Model component called Article. You've developed two different Views for your Article: PlainArticleView and FancyArticleView. Your Article has a keywords property, and you'd like to render the HTML meta tag using the keywords from an Article. You might write:

```
class PlainArticleView(grok.View):
    # lots of other code here
    def meta_tags(self):
        if len(self.context.keywords) > 0:
            return '<meta name="keywords" content="%s" />' % ','.join(
                self.context.keywords()
            )

class FancyArticleView(grok.View):
    # lots of other code here
    def meta_tags(self):
        if len(self.context.keywords) > 0:
            return '<meta name="keywords" content="%s" />' % ','.join(
                self.context.keywords()
            )
```

Well, that doesn't look so good does it? We've stated the logic of how to convert our keywords into a meta tag in two different places. Oh the horrors, what will all those DRY loving Web 2.0 programmers think of us? Let's write an Adapter that can render our meta tags:

```
class IMetaTags(Interface):
    def render(self):
        "HTML Meta Tags"

class MetaTags(grok.Adapter):
    "HTML Meta tags"
    grok.implements(IMetaTags)
    grok.adapts(IArticle)

    def render(self):
        if len(self.context.keywords) > 0:
            return '<meta name="keywords" content="%s" />' % (
                ','.join(self.context.keywords()
            )

class PlainArticleView(grok.View):
    # lots of other code here
    def meta_tags(self):
        return IMetaTags(self.context).render()

class FancyArticleView(grok.View):
    # lots of other code here
```

```
    def meta_tags(self):
        return IMetaTags(self.context).render()
```

Now our presentation logic is stated in only one place, which is good. Although you might be thinking, “Is this worth all the trouble, my PHP-loving Web 1.0 programmers are saying I’m just making my code harder to understand.” This is a good point, and sometimes KISS is the enemy of DRY. But let’s say that our application is expected to have a long lifecycle, where you know you’ll be wanting to extend the application functionality frequently. Let’s see how this extra work we have done can help us down the road. You’ve added a description property to your Article, and you’d like to have a description meta tag. So you change the MetaTags adapter to look like this:

```
class MetaTags(grok.Adapter)
    "XHTML Meta tags"
    grok.implements(IMetaTags)
    grok.adapts(IArticle)

    def render(self):
        html = ''
        if len(self.context.keywords) > 0:
            html += '<meta name="keywords" content="%s" />' % (
                ','.join(self.context.keywords()
            )
        if len(self.context.description) > 0:
             html += '<meta name="description" content="%s" />' % (
                 ','.join(self.context.keywords()
             )
        return html
```

Now both of your Views have been updated, and you only need to change the code in one place. Later on you add new HowTo Model to your application, and you create some Views for it. You are careful to make sure that your HowTo model has both a keywords and a description property, and now you just need to allow your adapter to work with both the HowTo and the Article models. Interfaces can inherit from other interfaces, just like a normal Python Class, so you can make explicit the relationship between your metadata and your content types.

```
from zope.interface import Interface
from zope.schema import List, TextLine, SourceText

class IMetadata(Interface):
    "Metadata about a content type"
    description = TextLine(title=u'short summary of the content',)
    keywords = List(
                    title=u"Keywords",
                    unique=True,
                    value_type=TextLine(title=u"Keyword"),
                    required=False
                  )

class IArticle(IMetadata):
    body = SourceText(title=u'Article Body')

class IHowTo(IMetadata):
    body = SourceText(title=u'HowTo Body')
```

Finally you just need to change the line of your MetaTags adapter from **grok.adapts(IArticle)** to **grok.adapts(IMetadata)**. Now all of the Views that you write for your HowTo can reuse the code that you were using to support the Views for your Article.

12.2.4 Further Reading

There are some more documents that can help you grasping the concept of adaptation

This tutorial is taken from a Kevin Teague blog entry

zope.component/README.txt A discussion on Adapters that shows more API examples of adapters in action.

Adaptation For Busy People A description of the purpose and usage of Zope's adaptation machinery in fewer than 1500 words.

Using Grok to walk like a duck A presentation that takes you through different design paradigms to enhance code reusability. Good to compare adaptation with other reuse techniques common in Python.

12.3 Macros with Grok Tutorial

Author Unknown

Macros are a way to define a chunk of presentation in one template, and share it in others. Changes to the macro are immediately reflected in all of the places, that share it.

Contents:

12.3.1 Introduction

Macros are a way to define a chunk of presentation in one template, and share it in others. Changes to the macro are immediately reflected in all of the places, that share it.

Such, a developer can easily write some sort of 'page-skeleton' (the macro), which can be reused in other pages to take care, for instance, for a certain and consistent look-and-feel of a whole site with less maintenance.

Technically, macros are made available by `METAL`, a Macro Expansion for TAL, which ships with nearly every Zope installation. They can be used in Zope Page Templates. See the TAL and METAL pages for details.

12.3.2 Defining a simple macro

In Grok macros are defined in usual views, where the associated page templates contain *metal*-statements to define the desired macros. Macros generally are attributes of the page template wherein they are defined, but to get them rendered, we usually use views.

We define a view `MyMacros` the usual way in `app.py`:

```
1  import grok
2
3  class Sample(grok.Application, grok.Container):
4      pass
5
6  class Index(grok.View):
7      pass # see app_templates/index.pt
8
9  class MyMacros(grok.View):
10     """The macro holder"""
11     grok.context(Sample) # The default model this view is bound to.
```

In the associated page template `app_templates/mymacros.pt` we define the macros we like to have available. You define macros with the `METAL` attribute:

```
metal:define-macro="<macro-name>"
```

and the slots therein with:

```
metal:define-slot=<slot-name>
```

Let's define a very plain page macro in `app_templates/mymacros.pt`:

```
<html metal:define-macro="mypage">
  <head></head>
  <body>
    The content:
    <div metal:define-slot="mycontent">
      Put your content here...
    </div>
  </body>
</html>
```

Here we defined a single macro `mypage` with only one slot `mycontent`.

If we restart our Zope instance (don't forget to put some `index.pt` into `app_templates/`) and have created our application as `test`, we now can go to the following URL:

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

and see the output:

```
The content:
Put your content here...
```

Allright.

12.3.3 Referencing a simple macro

How to reference a simple macro?

In `index.pt` we now want to *use* the macro defined in `mymacros.pt`. Using a macro means to let it render a part of another page template, especially, filling the slots defined in `mymacros.pt` with content defined in `index.pt`. You call macros with the `METAL` attribute:

```
metal:use-macro="<macro-location>"
```

Our `app_templates/index.pt` can be that simple:

```
<html metal:use-macro="context/@@mymacros/macros/mypage">
</html>
```

Watching:

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

should now give the same result as above, although we didn't call `mymacros` in browser, but `index`. That's what macros are made for.

When we fill the slots, we get different content rendered within the same macro. You can fill slots using:

```
metal:fill-slot="<slot-name>"
```

where the slot-name must be defined in the macro. Now, change `index.pt` to:

```
<html metal:use-macro="context/@@mymacros/macros/mypage">
  <body>
    <!-- slot 'mycontent' was defined in the macro above -->
    <div metal:fill-slot="mycontent">
      My content from index.pt
    </div>
  </body>
</html>
```

and you will get the output:

```
The content:
My content from index.pt
```

The pattern of the macro reference (the <macro-location>) used here is:

```
context/<view-name>/macros/<macro-name>
```

whereas `context` references the object being viewed, which in our case is the `Sample` application. In plain English we want Zope to look for a view for a `Sample` application object (`test`) which is called `mymacros` and contains a macro called `mypage`.

The logic behind this is, that views are always registered for certain object types. Registering a view with some kind of object (using `grok.context()` for example), means, that we promise, that this view can render objects of that type. (It is up to you to make sure, that the view can handle rendering of that object type).

It is not a bad idea to register views for interfaces (instead of implementing classes), because it means, that a view will remain usable, while an implementation of an interface can change. [FIXME: Is this a lie?] This is done in the section Defining 'all-purpose' macros.

12.3.4 Background: how grok.View and macros interact

The template attribute and the macro shortcut convention

In the case of `grok.View` views, they subclass the Zope 3 `BrowserPage` objects which provide an attribute `template`, which is the associated page template. The associated page template in turn has got an attribute `macros`, which is a dictionary containing all the macros defined in the page template with their names as keys (or `None`).

This means, that you can also reference a macro of a `grok.View` using:

```
context/<view-name>/template/macros/<macro-name>
```

Grok shortens this path for you by mapping the `macros` dictionary keys to the associated `grok.View`. If you define a macro `mymacro` in a template associated with a `grok.View` called `myview`, this view will map `mymacro` as an attribute, so that you can ask for the attribute `mymacro` of the *view*, where as it is in fact an attribute of the associated template.

Such, you can write in short for the above pattern:

```
context/<view-name>/macros/<macro-name>
```

View names always start with the 'eyes' (`@@`) which is a shortcut for `++view++<view-name>`.

12.3.5 Defining 'all-purpose' macros

To define an 'all-purpose' macro, i.e. a macro, that can render objects of (nearly) any type and thus be accessed from any other page template, just set a very general context for your macro view:

```
1 from zope.interface import Interface
2 import grok
3
4 class Master(grok.View):
5     grok.context(Interface)
```

and reference the macros of the associated pagetemplate like this:

```
context/@@master/macros/<macro-name>
```

Because the macros in `Master` now are 'bound' (in fact their view is bound) to `Interface` and every Grok application, model or container implements some interface, the `Master` macros will be accessible from nearly every other context. `Master` promises to be a view for every object, which implements `Interface`.

12.4 Navigating to transient objects

Author Brandon Craig Rhodes

Thanks to the magic of this database, your web apps can create Python objects that are automatically saved to disk and are available every time your application runs. In particular, every grok.Model and grok.Container object you generate can be written safely to your application's Data.fs file. But sometimes you need to create objects that do not persist in the ZODB, wonderful though it is.

Contents:

12.4.1 Introduction

Why instantiate objects from external data sources?

If you have already read the Grok tutorial, you are familiar with the fact that behind Grok lives a wonderful object database called the Zope Object Database, or the ZODB for short. Thanks to the magic of this database, your web apps can create Python objects that are automatically saved to disk and are available every time your application runs. In particular, every `grok.Model` and `grok.Container` object you generate can be written safely to your application's `Data.fs` file.

But sometimes you need to create objects that do not persist in the ZODB, wonderful though it is. Sometimes these will be objects you design yourself but have no need to save once you are done displaying them, like summary statistics or search results. On other occasions, you will find yourself using libraries or packages written by other people and will need to present the objects they return in your Grok app — whether those objects are LDAP entries, file system directory listings, or rows from a relational database.

For the purpose of this tutorial, we are going to call all such objects *transient* objects. This highlights the fact that, from the point of view of Grok, they are going to be instantiated on-the-fly as a user visits them. Of course, they might (or might not) persist in some other application, like a file system or LDAP server or relational database! But as far as Grok can tell, they are being created the moment before they are used and then, very often, pass right back out of existence — and out of your application's memory — once Grok is finished composing the response.

To try out the examples in this tutorial, start a Grok project named `TransientApp` and edit the `app.py` and other files that you are instructed to create or edit.

12.4.2 Choosing a method

In this tutorial, we introduce four methods for creating an object which you need to present on the web:

- Creating it in your View's `update()`, using no external data.
- Creating it in your View's `update()`, using URL or form data.
- Creating it in a `Traverser` that gets called for certain URLs.
- Creating it in a `Container` that gets called for certain URLs.

To choose among these methods, the big question you need to ask yourself is whether the object you are planning to display is one that will live at its own particular URL or not. There are three basic relationships we can imagine between an object on a web page and the URL of the page itself.

The simplest case, which is supported by the *first* method listed above, is when you need to create an object during the rendering of a page that already exists in your application. An example would be decorating the bottom of your front page with a random quotation selected by instantiating a `RandomQuotation` class you have written, so that each time a user reloads the page they see a different quote. None of the quotations would thereby have URLs of their own; there would, in fact, be no way for the user to demand that a particular quotation be displayed; and the user could not force the site to display again a quote from Bertrand Russell that they remember enjoying yesterday but have forgotten. Such objects can simply be instantiated in the `update()` method of your View, and this technique will be our first example below.

The situation is only slightly more complex when you need to use form parameters the user has submitted to tailor the object you are creating. This is very common when supporting searching of a database: the user enters some search terms, and the application needs to instantiate an object — maybe a `SearchResult` or a `DatabaseQuery` — using those user-provided search terms, so that the page template can loop across and display the results. The *second* method listed above is best for this; since the form parameters are available in the `update()` method, you are free to use them when creating the result object. This will be the technique illustrated in our second example below.

Finally, the really interesting case is when an object actually gets its own URL. You are probably already familiar with several kinds of object which have their own URLs on the Web — such as books on Amazon.com, photographs on Flickr, and people on Facebook, all of which live at their own URL. Each web site has a particular scheme which associates a URL with the object it names or identifies. You can probably guess, for example, just by looking at them, which object is named by each of the following three Amazon URLs:

```
http://www.amazon.com/Web-Component-Development-Zope-3/dp/3540223592
http://www.amazon.com/Harry-Potter-Deathly-Hallows-Book/dp/0545010225
http://www.amazon.com/J-R-R-Tolkien-Boxed-Hobbit-Rings/dp/0345340426
```

The Grok framework, of course, already supports URL traversal for persistent objects in the ZODB; if you create a `Container` named `polygons` that contains two objects named `triangle` and `square`, then your Grok site will already support URLs like:

```
http://yoursite.com/app/polygons/triangle
http://yoursite.com/app/polygons/square
```

But the point of this tutorial, of course, is how you can support URL traversal for objects which are *not* persistent, which you will create on-the-fly once someone looks up their URL. And the answer is that, to support such objects, you will choose between the last two methods listed above: you will either create a custom `Traverser`, or actually define your own kind of `Container`, that knows how to find and instantiate the object the URL is naming. These two techniques are described last in this tutorial, because they involve the most code.

But before starting our first example, we need to define an object that we want to display. We want to avoid choosing an obvious example, like an object whose data is loaded from a database, because then this tutorial would have to teach database programming too! Plus, you would have to set up a database just to try the examples. Instead, we need an object rich enough to support interesting attributes and navigation, but simple enough that we will not have to reach outside of Python to instantiate it.

12.4.3 Our Topic: The Natural Numbers

To make this tutorial simple, we will build a small web site that lets the user visit what some people call the natural numbers: the integers beginning with 1 and continuing with 2, 3, 4, and so forth. We will define a Natural class which knows a few simple things about each number - like which number comes before it, which comes after it, and what its prime factors are.

We should start by writing a test suite for our `Natural` class. Not only is writing tests before code an excellent programming practice that forces you to think through how your new class should behave, but it will make this tutorial easier to understand. When you are further down in the tutorial, and want to remember something about the `Natural` class, you may find yourself re- reading the tests instead of the code as the fastest way to remember how the class behaves!

The reason this test will be so informative is that it is a Python "doctest", which intersperses normal text with the example Python code. Create a file in your Grok instance named `src/transient/natural.txt` and give it the following contents:

```
A Simple Implementation of Natural Numbers
```

The "natural" module of this application provides a simple class for representing any postive integer, named "Natural".

```
>>> from transient.natural import Natural
```

To instantiate it, provide a Python integer to its constructor:

```
>>> three = Natural(3)
>>> print 'This object is known to Python as a "%r".' % three
This object is known to Python as a "Natural(3)".
>>> print 'The number of the counting shall be %s.' % three
The number of the counting shall be 3.
```

You will find that there are very many natural numbers available; to help you navigate among them all, each of them offers a "previous" and "next" attribute to help you find the ones adjacent to it.

```
>>> print 'Previous: %r  Next: %r' % (three.previous, three.next)
Previous: Natural(2)  Next: Natural(4)
```

Since we define the set of "natural numbers" as beginning with 1, you will find that the number 1 lacks a "previous" attribute:

```
>>> hasattr(three, 'previous')
True
>>> one = Natural(1)
>>> hasattr(one, 'previous')
False
>>> one.previous
Traceback (most recent call last):
 ...
AttributeError: There is no natural number less than 1.
```

You can also ask a number to tell you which prime factors must be multiplied together to produce it. The number 1 will return no factors: >>> one.factors []

Prime numbers will return only themselves as factors: >>> print Natural(2).factors, Natural(11).factors, Natural(251).factors [Natural(2)] [Natural(11)] [Natural(251)]

Compound numbers return several factors, sorted in increasing order, and each appearing the correct number of times:

```
>>> print Natural(4).factors
[Natural(2), Natural(2)]
```

```
>>> print Natural(12).factors
[Natural(2), Natural(2), Natural(3)]
>>> print Natural(2310).factors
[Natural(2), Natural(3), Natural(5), Natural(7), Natural(11)]
```

Each natural number can also simply return a boolean value indicating whether it is prime, and whether it is composite.

```
>>> print Natural(6).is_prime, Natural(6).is_composite
False True
>>> print Natural(7).is_prime, Natural(7).is_composite
True False
```

Next, we need to tell Grok about this doctest. Create a file in your instance named `src/transient/tests.py` that looks like:

```
import unittest
from doctest import DocFileSuite

def test_suite():
    return unittest.TestSuite([ DocFileSuite('natural.txt') ])
```

You should now find that running `./bin/test` inside of your instance now results in a whole series of test failures. This is wonderful and means that everything is working! At this point Grok is able to find your doctest, successfully execute it, and correctly report (if you examine the first error message) that you have not yet provided a `Natural` class that could make the doctest able to succeed.

12.4.4 The Natural class implementation

Now we merely have to provide an implementation for our Natural class.

Create a file `src/transient/natural.py` under your Grok instance and give it the contents:

```
class Natural(object):
    """A natural number, here defined as an integer greater than zero."""

    def __init__(self, n):
        self.n = abs(int(n)) or 1

    def __str__(self):
        return '%d' % self.n

    def __repr__(self):
        return 'Natural(%d)' % self.n

    @property
    def previous(self):
        if self.n < 2:
            raise AttributeError('There is no natural number less than 1.')
        return Natural(self.n - 1)

    @property
    def next(self):
        return Natural(self.n + 1)

    @property
    def factors(self):
        if not hasattr(self, '_factors'):  # compute factors only once!
            n, i = self.n, 2
```

```
            self._factors = []
            while i <= n:
                while n % i == 0:          # while n is divisible by i
                    self._factors.append(Natural(i))
                    n /= i
                i += 1
        return self._factors

    @property
    def is_prime(self):
        return len(self.factors) < 2

    @property
    def is_composite(self):
        return len(self.factors) > 1
```

If you try running `./bin/test` again after creating this file, you should find that the entire `natural.txt` docfile now runs correctly!

I hope that if you are new to Python, you are not too confused by the code above, which uses `@property` which may not have been covered in the Python tutorial. But I prefer to show you "real Python" like this, that reflects how people actually use the language, rather than artifically simple code that hides from you the best ways to use Python. Note that it is *not* necessary to understand `natural.py` to enjoy the rest of this tutorial! Everything we do from this point on will involve building a framework to use this object on the web; we will be doing no further development on the class itself. So all you actually need to understand is how a `Natural` behaves, which was entirely explained in the doctest.

Note that the `Natural` class knows nothing about Grok! This is an important feature of the whole Zope 3 framework, that bears frequent repeating: objects are supposed to be simple, and not have to know that they are being presented on the web. You should be able to grab objects created anywhere, from any old library of useful functions you happen to download, and suit them up to be displayed and manipulated with a browser. And the `Natural` class is exactly like that: it has no idea that we are about to build a framework around it that will soon be publishing it on the web.

12.4.5 Having Your View Directly Instantiate An Object

We now reach the first of our four techniques!

The simplest way to create a transient object for display on the web involves a technique you may remember from the main Grok tutorial: providing an `update()` method on your View that creates the object you need and saves it as an attribute of the View. As a simple example, create an `app.py` file with these contents:

```
import grok
from transient.natural import Natural

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

class Index(grok.View):
    def update(self):
        self.num = Natural(126)
```

Do you see what will happen? Right before Grok renders your View to answer a web request, Grok will call its `update()` method, and your View will gain an attribute named `num` whose value is a new instance of the `Natural` class. This attribute can then be referenced from the page template corresponding to your view! Let use write a small page template that accesses the new object. Try creating an `/app_templates/index.pt` file that looks like:

```
<html><body>
 <p>
  Behold the number <b tal:content="view/num">x</b>!
  <span tal:condition="view/num/is_prime">It is prime.</span>
  <span tal:condition="view/num/is_composite">Its prime factors are:</span>
 </p>
 <ul tal:condition="view/num/factors">
  <li tal:repeat="factor view/num/factors">
   <b tal:content="factor">f</b>
  </li>
 </ul>
</body></html>
```

If you now run your instance and view the main page of your application, your browser should show you something like:

Behold the number 126! It has several prime factors:

- 2
- 3
- 3
- 7

You should remember, when creating an object through an `update()` method, that a new object gets created every time your page is viewed! This is hard to see with the above example, of course, because no matter how many times you hit "reload" on your web browser you will still see the same number. So adjust your `app.py` file so that it now looks like this:

```
import grok, random
from transient.natural import Natural

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

class Index(grok.View):
    def update(self):
        self.num = Natural(random.randint(1,1000))
```

Re-run your application and hit "reload" several times; each time you should see a different number.

The most important thing to realize when using this method is that this `Natural` object is *not* the object that Grok is wrapping with the View for display! The object actually selected by the URL in this example is your `TransientApp` application object itself; it is this application object which is the `context` of the View. The `Natural` object we are creating is nothing more than an incidental attribute of the View; it neither has its own URL, nor a View of its own to display it.

12.4.6 Creating Objects Based on Form Input

What if we wanted the user to be able to designate which Natural object was instantiated for display on this web page?

This is a very common need when implementing things like a database search form, where the user's search terms need to be provided as inputs to the object that will return the search results.

The answer is given in the main Grok tutorial: if you can write your `update()` method so that it takes keyword parameters, they will be filled in with any form parameters the user provides. Rewrite your `app.py` to look like:

```
import grok, random
from transient.natural import Natural

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

class Index(grok.View):
    def update(self, n=None):
        self.badnum = self.num = None
        if n:
            try:
                self.num = Natural(int(n))
            except:
                self.badnum = n
```

And make your `app_templates/index.pt` look like:

```
<html><body>
 <p tal:condition="view/badnum">This does not look like a natural number:
  &ldquo;<b tal:content="view/badnum">string</b>&rdquo;
 </p>
 <p tal:condition="view/num">
  You asked about the number <b tal:content="view/num">x</b>!<br/>
  <span tal:condition="view/num/is_prime">It is prime.</span>
  <span tal:condition="view/num/is_composite">Its prime factors are:
   <span tal:repeat="factor view/num/factors">
    <b tal:content="factor">f</b
    ><span tal:condition="not:repeat/factor/end">,</span>
   </span>
  </span>
 </p>
 <form tal:attributes="action python:view.url()" method="GET">
  Choose a number: <input type="text" name="n" value="" />
  <input type="submit" value="Go" />
 </form>
</body></html>
```

This time, when you restart your Grok instance and look at your application front page, you will see a form asking for a number:

```
Choose a number: __________  [Go]
```

Enter a positive integer and submit the form (try to choose something with less than seven digits to keep the search for prime factors short), and you will see something like:

```
You asked about the number 48382!
Its prime factors are: 2, 17, 1423
Choose a number: __________  [Go]
```

And if you examine the URL to which the form has delivered you, you will see that the number you have selected is part of the URL's query section:

> http://localhost:8080/app/index?n=48382

So none of these numbers get their own URL; they all live on the page `/app/index` and have to be selected by submitting a query to that one page.

12.4.7 Custom Traversers

But what about situations where you want each of your transient objects to have its own URL on your site?

The answer is that you can create `grok.Traverser` objects that, when the user enters a URL and Grok tries to find the object which the URL names, intercept those requests and return objects of your own design instead.

For our example application `app`, let's make each `Natural` object live at a URL like:

```
http://localhost:8080/app/natural/496
```

There is nothing magic about the fact that this URL has three parts, by the way — the three parts being the application name `"app"`, the word `"natural"`, and finally the name of the integer `"496"`. You should easily be able to figure out how to adapt the example application below either to the situation where you want all the objects to live at your application root (which would make the URLs look like `/app/496`), or where you want URLs to go several levels deeper (like if you wanted `/app/numbers/naturals/496`).

The basic rule is that for each slash-separated URL component (like `"natural"` or `"496"`) that does not actually name an object in the ZODB, you have to provide a `grok.Traverser`. Make the `grok.context` of the Traverser the object that lives at the previous URL component, and give your Traverser a `traverse()` method that takes as its argument the next name in the URL and returns the object itself. If the name submitted to your traverser does not name an object, simply return `None`; this is very easy to do, since `None` is the default return value of a Python function that ends without a `return` statement.

So place the following inside your `app.py` file:

```
import grok
from transient.natural import Natural

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

class BaseTraverser(grok.Traverser):
    grok.context(TransientApp)
    def traverse(self, name):
        if name == 'natural':
            return NaturalDir()

class NaturalDir(object):
    pass

class NaturalTraverser(grok.Traverser):
    grok.context(NaturalDir)
    def traverse(self, name):
        if name.isdigit() and int(name) > 0:
            return Natural(int(name))

class NaturalIndex(grok.View):
    grok.context(Natural)
    grok.name('index.html')
```

And you will only need one template to go with this file, which you should place in `app_templates/naturalindex.pt`:

```
<html><body>
 This is the number <b tal:content="context">x</b>!<br/>
  <span tal:condition="context/is_prime">It is prime.</span>
  <span tal:condition="context/is_composite">Its prime factors are:
   <span tal:repeat="factor context/factors">
    <b tal:content="factor">f</b
```

```
    ><span tal:condition="not:repeat/factor/end">,</span>
   </span>
  </span><br>
</body></html>
```

Now, if you view the URL /app/natural/496 on your test server, you should see:

```
This is the number 496!
Its prime factors are: 2, 2, 2, 2, 31
```

Note that there is no view name after the URL. That's because we chose to name our View index.html, which is the default view name in Zope 3. (With grok.Model and grok.Container objects, by contrast, the default view selected if none is named is simply index without the .html at the end.) You can always name the view explicitly, though, so you will find that you can also view the number 496 at:

```
http://kenaniah.ten22:8080/app/natural/496/index.html
```

It's important to realize this because, if you need to add more views to a transient object, you of course will have to add them with other names — and to see the information in those other views, users (or the links they use) will have to name the views explicitly.

Two final notes:

- In order to make this example brief, the application above does not support either the user navigating simply to /app, nor will it allow them to view /app/natural, because we have provided neither our TransientApp application object nor the NaturalDir stepping-stone with grok.View objects that could let them be displayed. You will almost always, of course, want to provide a welcoming page for the top level of your application; but it's up to you whether you think it makes sense for users to be able to visit the intermediate /app/natural URL or not. If not, then follow the example above and simply do not provide a view, and everything else will work just fine.
- In order to provide symmetry in the example above, neither the TransientApp object nor the NaturalDir object knows how to send users to the next objects below them. Instead, they are both provided with Traversers. It turns out, I finally admin here at the bottom of the example, that this was not necessary! Grok objects like a grok.Container or a grok.Model already have enough magic built-in that you can put a traverse() method right on the object and Grok will find it when trying to resolve a URL. This would not have helped our NaturalDir object, of course, because it's not a Grok anything; but it means that we can technically delete the first Traverser and simply declare the first class as:

  ```
  class TransientApp(grok.Application, grok.Container):
      def traverse(self, name):
          if name == 'natural':
              return NaturalDir()
  ```

  The reason I did not do this in the actual example above is that showing two different ways to traverse in the same example seemed a bit excessive! I preferred instead to use a single method, twice, that is universal and works everywhere, rather than by starting off with a technique that does not work for most kinds of Python object.

12.4.8 Providing Links To Other Objects

What if the object you are wrapping can return other objects to which the user might want to navigate?

Imagine the possibilities: a filesystem object you are presenting on the web might be able to return the files inside of it; a genealogical application might have person objects that can return their spouse, children, or grandparents. In the example we are working on here, a Natural object can return both the previous and the next number; wouldn't it be nice to give the users links to them?

If in a page template you naively ask your Grok view for the URL of a transient object, you will be disappointed. Grok *does* know the URL of the object to which the user has just navigated, because, well, it's just navigated there, so adding this near the bottom of your `naturalindex.pt` should work just fine:

```
This page lives at: <b tal:content="python: view.url(context)">url</b><br>
```

But if you rewrite your template so that it tries asking for the URL of any other object, the result will be a minor explosion. Try adding this to your `naturalindex.pt` file:

```
Next number: <b tal:content="python: view.url(context.next)">url</b><br>
```

and try reloading the page. On the command line, your application will return the exception:

```
TypeError: There isn't enough context to get URL information.
This is probably due to a bug in setting up location information.
```

Do you see the problem? Because this new `Natural` object does not live inside of the ZopeDB, Grok cannot guess the URL at which you intend it to live. In order to provide this information, it is best to call a Zope function called `locate()` that marks as object as belonging inside of a particular container. To get the chance to do this magic, you'll have to avoid calling `Natural.previous` and `Natural.next` directly from your page template. Instead, provide your view with two new properties that will grab the `previous` and `next` attributes off of the `Natural` object which is your current context, and then perform the required modification before returning them:

```python
class NaturalIndex(grok.View):

    ...

    @property
    def previous(self):
        if getattr(self.context, 'previous', None):
            n = self.context.previous
            traverser = BaseTraverser(grok.getSite(), None)
            parent = traverser.publishTraverse(None, 'natural')
            return zope.location.location.located(n, parent, str(n))

    @property
    def next(self):
        n = self.context.next
        traverser = BaseTraverser(grok.getSite(), None)
        parent = traverser.publishTraverse(None, 'natural')
        return zope.location.location.located(n, parent, str(n))
```

This forces upon your objects enough information that Zope can determine their URL — it will believe that they live inside of the object named by the URL `/app/natural` (or whatever other name you use in the `PublishTraverse` call). With the above in place, you can add these links to the bottom of your `naturalindex.pt` and they should work just fine:

```html
<tal:if tal:condition="view/previous">
 Previous number: <a tal:attributes="href python: view.url(view.previous)"
  tal:content="view/previous">123</a><br>
</tal:if>
Next number: <a tal:attributes="href python: view.url(view.next)"
 tal:content="view/next">123</a><br>
```

This should get easier in a future version of Grok and Zope!

12.4.9 Writing Your Own Container

The above approach, using Traversers, gives Grok just enough information to let users visit your objects, and for you to assign URLs to them.

But there are several features of a normal `grok.Container` that are missing — there is no way for Grok to list or iterate over the objects, for example, nor can it ask whether a particular object lives in the container or not.

While taking full advantage of containers is beyond the scope of this tutorial, I ought to show you how the above would be accomplished:

```
import grok
from transient.natural import Natural
from zope.app.container.interfaces import IItemContainer
from zope.app.container.contained import Contained
import zope.location.location

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

class BaseTraverser(grok.Traverser):
    grok.context(TransientApp)
    def traverse(self, name):
        if name == 'natural':
            return NaturalBox()

class NaturalBox(Contained):
    grok.implements(IItemContainer)
    def __getitem__(self, key):
        if key.isdigit() and int(key) > 0:
            n = Natural(int(key))
            return zope.location.location.located(n, self, key)
        else:
            raise KeyError

class NaturalIndex(grok.View):
    grok.context(Natural)
    grok.name('index.html')

    @property
    def previous(self):
        if getattr(self.context, 'previous'):
            n = self.context.previous
            parent = self.context.__parent__
            return zope.location.location.located(n, parent, str(n))

    @property
    def next(self):
        n = self.context.next
        parent = self.context.__parent__
        return zope.location.location.located(n, parent, str(n))
```

Note, first, that this is almost identical to the application we built in the last section; the `grok.Application`, its `Traverser`, and the `NaturalIndex` are all the same — and you can leave alone the `naturalindex.pt` you wrote as well.

But instead of placing a `Traverser` between our `Application` and the actual objects we are delivering, we have created an actual "container" that follows a more fundamental protocol. There are a few differences in even this simple example.

- A container is supposed to act like a Python dictionary, so we have overriden the Python operation `__getitem__` instead of providing a `traverse()` method. This means that other code using the container can find objects inside of it using the `container[key]` Python dictionary syntax.
- A Python `__getitem__` method is required to raise the `KeyError` exception when someone tries to look up a key that does not exist in the container. It is *not* sufficient to merely return `None`, like it was in our `Traverser` above, because, without the exception, Python will assume that the key lookup was successful and that `None` is the value that was found!
- Finally, before returning an object from your container, you need to call the Zope `located()` function to make sure the object gets marked up with information about where it lives on your site. A Grok `Traverser` does this for you.

Again, in most circumstances I can imagine, you will be happier just using a Traverser like the third example shows, and not incurring the slight bit of extra work necessary to offer a full-fledged container. But, in case you ever find yourself wanting to use a widget or utility that needs an actual container to process, I wanted you to have this example available.

12.5 Permissions Tutorial

Author Luis De la Parra, Uli Fouquet, Jan-Wijbrand Kolman, Sebastian Ware

Grok comes with authorization capabilities out of the box. The authorization checks here are done based on the Views used to access (display/manipulate) the content. Grok requires you to explicitly grant access to views using the grok.requires() directive.

Contents:

12.5.1 Setup

Imagine a Grok module for holding Contact Info called
contact.py. By default, anyone is able to view the
ViewContact view.

```
from zope import component, interface, schema
import grok

class IContactInfo(interface.Interface):
    """ Interface/Schema for Contact Information """
    first_name = schema.Text(title=u'First Name')
    last_name  = schema.Text(title=u'Last Name')
    email      = schema.Text(title=u'E-mail')


class ContactInfo(grok.Model):
    interface.implements(IContactInfo)

    first_name = ''
    last_name  = ''
    email = ''

class ViewContact(grok.View)
    """Display Contact Info, without e-mail.
```

```
    Anyone can use this view, even unauthenticated users
    over the internet
    """
    def render(self):
        contact = self.context
        return 'Contact: ' + contact.first_name + contact.last_name
```

12.5.2 Defining Permissions and Restricting Access

As all Views in Grok default to public access,
anyone can use the ViewContact view. If you want
to restrict access to a view, you have to explicitly
protect it with a permission.

Define your Grok Permissions by subclassing from the grok.Permission base class. You must use the grok.name directive to give your permission a unique name. It can be any string, but it is strongly recommended to prefix them with the application name.

```
class ViewContacts(grok.Permission):
    grok.name('mysite.ViewContacts')
    grok.title('View Contacts') # optional

class AddContacts(grok.Permission):
    grok.name('mysite.AddContacts')

class EditContacts(grok.Permission):
    grok.name('mysite.EditContacts')

class ViewContactComplete(grok.View)
    """Display Contact Info, including email.

    Only users which have the permission 'mysite.ViewContacts'
    can use this view.
    """"
    grok.require('mysite.ViewContacts')  # this is the security declaration

    def render(self):
        contact = self.context
        return 'Contact: %s%s%s' % (contact.first_name,
                                    contact.last_name,
                                    contact.email)
```

Note The `grok.Permission` component base class was introduced *after* the release 0.10. In earlier versions of Grok a permission was defined using a module level directive, like so:

```
grok.define_permission('mysite.ViewContacts')
```

If you are using `grokproject` this change currently does not affect your installation. In this case use `grok.define_permission` as described above.

12.5.3 Granting Permissions

You can grant permissions to principals with a PermissionManager. For example, if all registered users should have permission to view contact details and to create new contacts, you could grant them the permissions when the user

account is created.

```
from zope.app.security.interfaces import IAuthentication
from zope.app.authentication.principalfolder import InternalPrincipal

# note: the securitypolicy package was moved in Grok 0.12+ from zope.app. to zope.
from zope.securitypolicy.interfaces import IPrincipalPermissionManager

def addUser(username, password, realname):
    """Create a new user.

    create a new user and give it the authorizations,
    ``ViewContacts`` and ``EditContacts``. This example assumes
    you are using a Pluggable Authentication Utility (PAU) /
    PrincipalFolder, which you have to create and register when
    creating your Application.
    """

    pau = component.getUtility(IAuthentication)
    principals = pau['principals']
    principals[username] = InternalPrincipal(username, password, realname)

    # grant the user permission to view and create contacts
    # everywhere in the site
    permission_man = IPrincipalPermissionManager(grok.getSite())

    # NOTE that you need a principal ID. If you are
    # authenticating users with a PAU this is normally the user
    # name prepended with the principals-folder prefix (and the
    # PAU-prefix as well, if set)
    permission_man.grantPermissionToPrincipal (
       'mysite.ViewContacts',
       principals.prefix + username)
    permission_man.grantPermissionToPrincipal(
       'mysite.AddContacts',
       principals.prefix + username)
```

Permissions are granted for the context for which the PermissionManager is created, and – if not explicitly overridden – all its children. The above example grants `View` and `Add` permissions for the complete site, unless a folder down in the hierarchy revokes the permission.

If you want users to be able to edit only their own `ContactInfos`, you have to give them the `Edit` permission only within the context of the `ContactInfo`-object itself

```
class AddContact(grok.AddForm):
    """Add a contact.
    """

    # Only users with permission 'mysite.AddContacts' can use
    # this.
    #
    # NOTE that if you don't protect this Form, anyone -- even
    # anonymous/unauthenticated users -- could add ``Contacts``
    # to the site.
    grok.require('mysite.AddContacts')

    #automagically generate form fields
    form_fields = grok.AutoFields(IContactInfo)
```

```
    @grok.action('Create')
    def create(self, **kw):
        # Create and add the ``ContactInfo`` to our context
        # (normally a folder/container)
        contact = ContactInfo()
        self.applyData(contact, **kw)
        self.context[contact.first_name] = contact

        # Grant the current user the Edit permission, but only in
        # the context of the newly created object.
        permission_man = IPrincipalPermissionManager(contact)
        permission_man.grantPermissionToPrincipal(
            'mysite.EditContacts',
            self.request.principal.id)
        self.redirect(self.url(contact))

class EditContact(grok.EditForm):
    """Edit a contact.
    """

    #only users with permission 'mysite.EditContacts' can use this
    grok.require('mysite.EditContacts')

    form_fields = grok.AutoFields(IContactInfo)

    @grok.action('Save Changes')
    def edit(self, **data):
        self.applyData(self.context, **data)
        self.redirect(self.url(self.context))
```

12.5.4 Checking Permissions

How to check permission in python code.

When generating user interface elements you might want to check that the current logged in principal actually can access a view to which a link refers. You need to do two things: 1 get the view, 2 check permissions on that view. This is how you do it:

```
from zope.component import getMultiAdapter
from zope.security import canAccess

def canAccessView(obj, view_name):
    # obj - is the object you want view
    # view_name - is the grok.View/AddForm/EditForm you want to access
    view = getMultiAdapter((obj, self.request), name=view_name)
    # check if you can access the __call__ method which is equal
    # to being allowed to access this view.
    return canAccess(view, '__call__')
```

If you want to check if the current logged in principal has a specific permission on a specific object or view you can do so by means of the checkPermission method. It is available through zope.security and in a view through self.request.interaction. Note that Grok doesn't allow a simplified way of setting object level permissions. The grok.requires statement is only applicable to views.

```
from zope.security import checkPermission
def justChecking(context):
    # context - the object or view you are checking permissions on
    user_allowed = checkPermission(PERMISSION_NAME, context)

class MyView(grok.View):
    def update(self):
        i = self.request.interaction
        # checking permission on currently viewed object (self.context)
        user_allowed = i.checkPermission(PERMISSION_NAME, self.context)
```

12.5.5 Defining Roles

Permissions can be grouped together in Roles, which makes granting all the permissions for a particular type of user much easier. Defining roles is similar to defining permissions.

As an example, let's group all permissions in two roles: one for normal site members, and one for administrators:

```
class MemberRole(grok.Role):
    grok.name('mysite.Member')
    grok.title('Contacts Member') # optional
    grok.permissions(
        'mysite.ViewContacts',
        'mysite.AddContacts')

class AdministratorRole(grok.Role):
    grok.name('mysite.Administrator')
    grok.title('Contacts Administrator') # optional
    grok.permissions(
        'mysite.ViewContacts',
        'mysite.AddContacts',
        'mysite.EditContacts')
```

Now, if the context here is the site/application, users with the administrator role can edit all ContactInfos, regardless of who the creator is.

```
# note: securitypolicy package moved in Grok 0.12+ from zope.app. to zope.
from zope.securitypolicy.interfaces import IPrincipalRoleManager

role_man = IPrincipalRoleManager(context)
role_man.assignRoleToPrincipal('mysite.Administrator', principalID)
```

12.6 Musical Performance Organizer - Annotated

Author Thomas Richter

The first in a series of tutorials that will build a reasonably complex application, starting from a simple prototype and evolving it into a finished application. The tutorial will touch on many practical aspects of building the application and demonstrate how to combine the various components that are available in Grok into a finished product.

Contents:

12.6.1 Introduction

An overview of the example requirements and a road map for how the tutorial will cover the development of the application.

Contents

- Introduction
  - General Overview
  - Sample Application Requirements
  - Description
  - Entity Outline
  - General Requirements
  - Phased Deliverables
    - Phase 1 - Basic Functions
    - Phase 2 - Improved Usability
    - Phase 3 - Style and Management
  - Development Road Map

General Overview

One of the goals of this tutorial is to give the reader a chance to walk through the process of developing a Grok application. Specific pieces of application development are handled in greater details in other how-to's and tutorials. In this tutorial, we hope to demonstrate how the pieces can come together in a development process.

If you are a seasoned Zope3 developer or even a long-time Python developer, you probably have your own preferred development style and you may disagree with some parts of this tutorial. (If you have ideas for improvements, they are certainly welcome but this is not an exercise in determining which development style, text editor, IDE, template system, or AJAX library is the best or most efficient.)

This tutorial is targeted at those who are new to the Grok and Zope3 framework. They may have been using other web development frameworks and programming languages, or they may have been lucky enough to have found Grok first and are anxious to see what it can do for them.

We will try to *not* make assumptions about your knowledge of Python, Zope3, and Grok. Obviously, it is assumed that you have some software development experience and are able to learn the Python language and follow the development patterns. A great deal of high quality documentation and tutorials exists for most of the topics surrounding this tutorial. Where appropriate we will provide links to resource materials.

It is expected that you have Python version 2.5 installed on your computer. Specifically, a version of CPython in the 2.5.x series from http://www.python.org, not Jython, IronPython, or some other alternative implementation of the language. Python version 2.4 is also acceptable, however, version 2.6 is not yet supported and version 3.0 will not currently work.

Grok will work on any operating system supported by Python which includes Linux, Macintosh, and Windows. There can be some slight differences in things like directory locations depending on platform. We will try to note some of the differences, but understand that Ubuntu Linux is being used for the writing of this tutorial.

Sample Application Requirements

Bookmark This Page

Over the course of this tutorial we will be working from the following set of application requirements. Take some time now to review the requirements in order to get some idea of where we are going. You will be referring to these requirements often, so you may want to create a bookmark to this location.

Description

In some venues, ad-hoc bands of musicians and vocalists come together to perform at scheduled events. The band leader for a given event needs to organize the people involved, develop the list of music they will perform, and distribute sheet music, chord charts, and lyrics.

A leader may be in charge of several performances involving different musicians. Likewise, musicians and vocalists may be involved in several performances lead by different people.

Currently, much of this organization is done via email. As expected, often key people are missed in the email distribution or their email address changes or their account is broken, causing them to miss important information.

Even if they receive all the information, it is still possible for them to mis- file the emails or attachments. Often updates for various performances come "out of sequence" and, if the subject is not very clear (i.e. "Friday's Performance" instead of a distinct date), the recipient can get confused.

Having a web site that lists all the important information and allows them to download the files they need for their preparation would be a benefit to everyone involved.

Entity Outline

- Application Root (contains Performance objects)
  - Performance
    - Date
    - Location
    - Leader
    - Start and End Times (free-form, optional)
    - Musicians (contains Musician objects)
      - Name
      - Email Address
      - Instrument
    - Set List (contains Song objects)
      - Song Title
      - Arrangement Notes
      - Display Order
      - Attachments (contains Resource references)
      - File Name
      - File URL
      - File MD5 Hash (Hidden/Optional)

  * Comments (Contains Comment objects
    - Date
    - Author Email Address
    - Comment Text

General Requirements

- Performances should be organized by date. Since it is possible to perform more than once in a day, an optional, "rough time" can be added to the day to differentiate them. Performances will never happen more frequently than once per hour.
- Performances should be able to be found by URL's that can be understood to refer to a certain performance. Example: http://example.com/TheBandName/2009- 03-05/
- Users should be able to upload files in any format they choose (PDF, DOC, Text, etc). No special handling or MIME types are provided by the application to enhance browser interaction.

Phased Deliverables

Phase 1 - Basic Functions

- All current and future performances are listed Sorted ascending by date.
- Text-field forms

  User hand-types dates, musician names, song names, and display order relying on web browser for any form completion efficiencies.
- Basic file upload support

  Upload file attachments to a static directory or directly into the storage database. Always upload user selected attachments and use a "name-chooser" to prevent overwriting existing files. Deleting an attachment, or the song which contains it, should remove the file from the application, but may require a management operation to reclaim the free space.
- No user authentication, roles or permission support.

  Rely on external security via web proxy configuration or rewrite rules.
- Basic Text Comments Anyone can comment.

  No email support or comment-spam protection.

Phase 2 - Improved Usability

- Enhanced Performance Listing

  Ability to see past performances. More display of summary performance details such as simple musician and song listings.
- Form Widgets

  Use of date-picker, and drop down lists to speed selection of information that already exists in the application. Buttons or "Drag and Drop" to adjust display order.
- Enhanced Upload

Ability to quickly select more than one attachment item. Warn user when an attachment may already exist and offer to link to the existing file rather than uploading another. (This could be done by file name matching or possibly a file content hash comparison.) Only remove underlying file content when no attachments (in any performance) reference the file.

- Basic Roles for Editing

  Leader role can create Performances and add songs to Set list. Assistant role can modify existing song arrangement notes and add/remove attachments.

- Email Support for Comments

  Basic comment-spam protection via capcha or email confirmation request. Notification email with URL of this performance sent to the addresses of all associated musicians.

Phase 3 - Style and Management

- Create a stylized look for the site.

  Use a template system, CSS, and some images to give the site an appealing look. Should be easy to modify the colors and any logos via CSS edits.

- Application Search

  Ability to search or filter performances by song performed or musicians involved.

- Provide Management Functions

  Removal or archiving of past performances. Attachment management to free storage resources.

Development Road Map

We are going to begin by getting the basic development environment in place and test that it works. We will then begin writing some tests which correspond to some of the Phase 1 requirements. From there we will begin to write model and view code to satisfy our tests.

Several iterations will be required to get through the first deliverable. From there, we will move on to the other deliverables.

It will take a significant amount of time to make it all the way through this tutorial, but we will try the keep the amount of work in each iteration manageable.

12.6.2 Setting Up The Environment

Configuring the project environment, understanding where files are located, and how to run application and test code.

Contents

- Setting Up The Environment
  - Installing Grok
    - Creating the Virtual Environment
    - Preparing for the Sample Project
    - Creating the Sample Project using GrokProject
  - Becoming Familiar With The Project
    - Directory and File Structure
    - Configuration Files
  - Run Tests and Start the Web Server

Installing Grok

Instructions for installing Grok can be found in the Grok Tutorial . We are going to use a helper application called "GrokProject" to set up our sample project. This is the recommended way to install a Grok application as it puts in place all the scripts and tools you will need to build, run and deploy your application.

GrokProject is installed by running a tool called "easy_install", which will download GrokProject from a repository on the internet called PyPI (the Python Package Index). Most of the application code that you will install using Grok will come from this source.

While it is very *easy* to use easy_install, there can be problems if you use it to install all the software you need for your Grok project and, perhaps, other Python based software you may use. The versions of different component pieces will sometimes conflict and it can be difficult to figure out why your project fails due to an update of some other package.

In order to avoid these problems we are going to create our sample project in a "virtual environment" which will isolate it from other python libraries. Fortunately, there is an easy way to create this isolated environment using a tool called virtualenv .

Some Linux distributions provide virtualenv as a standard install for the operating system. (On Debian/Ubuntu it is called "python-virtualenv") If this is available for you, you do not need to install "easy_install" to your main Python installation, as virtualenv will include its own copy.

If this option is not available, you will need to install "easy_install" following the directions in the Grok Tutorial . There is a tutorial dedicated to explaining how virtualenv is used with Grok and how to create a project inside the environment.

Creating the Virtual Environment

Create the sample project in your home directory in a new sub-directory, called "grok". (For Windows, use "My Documents\grok". For others use "~/grok".)

Change into this new directory and create the virtual environment and our sample project. If you have difficulties or receive error messages, refer to the tutorial links above as they do provide answers to common problems.

```
$ cd ~/grok
$ virtualenv --no-site-packages virtualgrok
$ source virtualgrok/bin/activate
(virtualgrok)$ easy_install grokproject
(virtualgrok)$ cd virtualgrok
```

Preparing for the Sample Project

The grokproject application will set up your project so that it can be further developed using a tool called "zc.Buildout". This tool creates an environment that allows you to test, debug, and deploy your project code. Grokproject will even run Buildout for you the first time so that the new application is ready to run and display a simple welcome screen in a web browser.

zc.Buildout and Python Eggs

zc.Buildout is a tool used to assemble Python applications from multiple parts. It is very flexible, in that it allows you to pick all types of available applications and components from Python, from Zope, or from other projects and it will download and configure them so that you can use them with your application.
Once you have configured Buildout, the steps required to set up your application for development are very reproducible. Instead of having to instruct other developers to download and install all of your project's dependencies before downloading and configuring your project, you can simply have them run your project's buildout.
http://pypi.python.org/pypi/zc.buildout
You can also use Buildout to package your application so others can install it using easy_install. (Just like you did for with GrokProject.)
Eggs are how Buildout packages your Python application. Eggs are a way of bundling your code with extra information that will allow easy_install to fetch and configure any dependencies your application might have. If you are writing a web application using Grok, it will certainly have some dependencies. Depending on which additional components you choose to use in your project, it will have even more.
When you choose to use additional components in your application, you simply add them to your Buildout configuration and run the Buildout script again. The additional code you need will be downloaded automatically (usually in the form of Eggs!), and made available to your application.
http://peak.telecommunity.com/DevCenter/PythonEggs
The easiest place to find applications and components (packaged as Eggs) for use with Buildout, is the Python Package Index or PyPI. This index archives the version history of thousands of Python projects. If you meet the requirements, you can even upload your application to PyPI to make it easily available to everyone.
http://pypi.python.org/pypi

If you have never run buildout before, it will download between 80 and 90 MB of code to create your initial Grok project. By default, it will create a cache of the source code it downloads in your home directory under ~/.buildout/eggs. This will prevent having to re-download this code every time you run buildout or grokproject.

You can change the location of this cache by creating or altering a file named ~/.buildout/default.cfg. Simply include the following lines:

```
[buildout]
eggs-directory = /path/to/directory
```

Creating the Sample Project using GrokProject

Now you are ready to run grokproject. If this is your first time running grokproject, be prepared to wait several minutes while the necessary source code is downloaded from PyPI. (Creating future projects will be much quicker because the source code will be cached locally.)

```
(virtualgrok)$ cd ~/grok/virtualgrok
(virtualgrok)$ grokproject music
Enter user (Name of an initial administrator user): grok
Enter passwd (Password for the initial administrator user): grok
```

The new sample project is located at ~/grok/virtualgrok/music and now ready for us to try out.

> Note: Now that the project is created, the buildout configuration will point to the virtual Python installation within our environment. We no longer need to source the "activate" script while working with our project.

Becoming Familiar With The Project

Take a moment to become familiar with the sample project you just created by browsing through the project directory.

Directory and File Structure

Under "music" we will find several directories and files.

bin Contains various script files to control your project. The ones that you will use most often are:

- buildout: to rebuild your project if you make changes to its dependencies.
- music-debug: gives you an interactive debug prompt (Python interpreter) with your project environment already configured. You can inspect and modify your stored objects and simulate browser requests.
- paster: to run your project. Initially, it will run in its own web server process, but paster can deploy your application in many ways.
- test: runs every unit test and functional test it can find and reports the results.

etc Contains configuration templates for this project's installation. The template file names end with ".in". These templates are copied to the project's parts/etc directory when buildout is run. If you want to make changes to the configuration that are available the first time you run buildout or are maintained even when you reconfigure your project by running buildout again, you need to modify these files.

parts Contains the object database files and web server log files. It also contains an "etc" subdirectory which has the actual configuration files used when running the application.

- debug.ini and deploy.ini: configuration scripts for paster. They determine thing like what network port and address the web server will be bound to. The debug.ini will present more meaningful error messages through your browser should an error occur. Edits to this file will be lost the next time you run buildout.
- site.zcml: contains some basic security directives for your project. The user name and password you supplied to grokproject are recorded here. (More advanced security schemes can be implemented later to avoid having clear-text passwords on the file system.) Edits to this file will be lost the next time you run buildout.

src Contains the source code for your application and configuration information to be used if you want to deploy your code as a Python egg (music.egg-info).

src/music Contains the source code you write.

- app.py: the starting point for your application. You can place all your source code in this file or break it up into multiple files.
- app.txt: a very simple functional test of the application. More tests can be added to this file or you can add tests to other text files in this directory and the test runner will find them.

src/music/app_templates Contains html template files that grok will associate with your code in order to publish it on the web.

- index.pt: a simple template created by grokproject that presents a welcome message. We will see this template in action when we run our project.

src/music/static A directory where you can place static files that you want available to your application. This may include image files, cascading style sheets, or JavaScript files.

Configuration Files

There are a few configuration files you should be familiar with.

setup.py Located in the main project directory, this file contains metadata about your project including its version and what other packages it requires to run.

buildout.cfg Located in the main project directory, this file allows you to customize the buildout process. You can specify a project-specific location for the cached source code. You can specify other places to look for source code, or you can combine several applications into one buildout processes. Specifics about buildout can be found in the Introduction to zc.buildout tutorial.

versions.cfg Located in the main project directory, this file initially contains a list of specific version of software that are known to work together (aka “Known Good Set” or KGS).

Run Tests and Start the Web Server

Now that we have had a look around, it is time to do something with the application we created. We will start by running the simple tests created with the project.

```
$ cd ~/grok/virtualgrok/music
$ ./bin/test

Running tests at level 1
Running music.FunctionalLayer tests:
  Set up music.FunctionalLayer in 1.523 seconds.
  Running:
...
  Ran 3 tests with 0 failures and 0 errors in 0.158 seconds.
Tearing down left over layers:
  Tear down music.FunctionalLayer ... not supported
```

The tests that were run are located in ~/grok/virtualgrok/music/src/music/tests.py. Take a look at this file, if you would like to see what was tested.

Now it is time to start the web server. By default, the web server will be available at http://localhost:8080 . If you already have a program that is using that port, you will have to assign a new port using parts/etc/debug.ini or parts/etc/deploy.ini.

> Note: These changes will be lost the next time your run buildout, so if you want permanently change the default port, you will need to modify etc/debug.ini.in and/or etc/deploy.ini.in and run buildout again.

```
$ cd ~/grok/virtualgrok/music
$ ./bin/buildout
```

> Note: If you want this server to be accessible from another computer you will have to modify the ‘host’ entry in the above files to your computer’s host name, its IP address, or to 0.0.0.0 (bound to all addresses).

Run the server:

```
$ cd ~/grok/virtualgrok/music
$ ./bin/paster serve etc/deploy.ini
Starting subprocess with file monitor
------
2009-03-06T22:12:40 WARNING root Developer mode is enabled: this is a
security risk and should NOT be enabled on production servers. Developer
```

```
mode can be turned off in etc/zope.conf
Starting server in PID 16637.
serving on http://127.0.0.1:8080
```

Point a web browser at http://localhost:8080 and you will be prompted for a user name and password. Enter the values that you entered when you ran grokproject. (Should be user name = grok and password = grok.)

You are now in the Grok Admin User Interface.

tutorials/musical_performance_organizer/GrokAdminInitial.png

We will explore the Admin UI in greater detail later. For now, understand that this is a place where you can create instances of your application. Each instance will be separate and can be accessed by name as the first part of the URL following the host name.

For the Musical Performance Organizer, several bands could use the same application on the same web server by having separate instances. (There are some problems to consider in order to use that strategy and we will explore those later.) For now, this gives us a quick way to setup and delete entire instances as we are developing our application.

Create an application instance by typing "band" in the "Add Application" text box and press "Create".

You will now see the instance listed in the "Installed Applications".

tutorials/musical_performance_organizer/GrokAdminTestApp.png

Click on the "band (Music)" link to view the instance at http://localhost:8080/band .

You should see the following text:

> Congratulations!
>
> Your Grok application is up and running. Edit music/app_templates/index.pt to change this page.

The application is up and running. Now we need to start modifying it to meet our requirements.

12.6.3 Initial Tests and Model Objects

Begin transcribing the application requirements into tests and perform the first coding cycle to satisfy the tests. This is the first exposure to working with model objects and unit testing.

Contents

- Initial Tests and Model Objects
  - Application Root
  - Requirements for Adding a Performance
  - Creating Tests

Application Root

It's time to start implementing the application requirements. If we review the entity outline we created, we can see that our top-level object is our application root which contains "Performance" objects.

If we look at the class that was generated by grokproject (in the project directory at ~/grok/virtualgrok/music/src/music/app.py), we see that it is called "Music". This was the name we provided to grokproject with the first letter capitalized to match the naming convention for classes.

```
import grok
class Music(grok.Application, grok.Container):
    pass
class Index(grok.View):
    pass # see app_templates/index.pt
```

By looking at this class, we see that it inherits from two base classes. The first is grok.Application. This is what gets the class listed in the Grok Admin UI so that you can make instances of the application as we discussed in the previous tutorial page.

The other base class is grok.Container. This means that the class will be a dictionary-like structure that will contain key-value pairs where the values will be objects and they can be accessed using unique strings as the keys.

There are several things that this container does beyond a standard dictionary. First, it automatically persists objects placed in it to the Zope object database (ZODB). Secondly, it keeps an efficient index of its contents so that it can quickly find objects even when it contains a fairly large quantity. Lastly, it records the object hierarchy to assist code that interprets an URL in order to find a particular object in the database.

Traversal

Other MVC (Model-View-Controller) type frameworks have various ways of mapping an URL to a specific set of data and a method for working with that data. In Grok and Zope3 this concept is called "traversal".
The URL is parsed and a default set of rules tries to map the URL to "content objects" (model objects you defined that are placed in containers), certain object "attributes", or "views" that are able to display particular objects.
TODO: Find the documentation on default traversal and link to it. I read a good explanation once, but can't find it on Zope, Grok, or Google.
(Repoze.BFG has a nice traversal explanation that is somewhat applicable to Grok.)
The default set of traversal rules is quite flexible and can even be extended by providing custom traversal.

Requirements for Adding a Performance

We know from the requirements that we are going to store our performances by date. Since the key is a string, we could let the user enter any sort of date-like string for the performance and use that. The problem will be that, not only will this be hard to sort and read, but some of the strings a user might provide would require some really ugly

encoding to work as part of an URL. (Remember, the URL's are supposed to understandable and a date encoded like "03%2F15%2F2009" is hard to read.)

We also know that the minimum date and time resolution for performances is an hour. A text representation of a date as "yyyy-mm-dd" with an optional "-hh" suffix will provide the resolution we need as well as provide chronological ordering by simply iterating over the container's keys.

Now, how do we enforce this rule? We could do it as part of an HTML form validation or we could try to encapsulate the container object so that performances can only be added through a method that evaluates the user input against the rule. If we do it in the form or the view code, it will have to be functionally tested as opposed to writing a unit test.

Encapsulating to force coding integrity is rather "un-Pythonic" as Python development generally assumes all developers "are adults" and will understand and respect the code they are interacting with. This "development style" generally places more emphasis on evaluating the contents of attributes and parameters when they are about to be used, rather than trying to prevent them from being assigned the wrong type of data in the first place.

There are some sophisticated ways to address this type of data validation issue, but we are going to keep it as simple as possible while still meeting our requirements. We will leave the container object "as-is" and simply add a validation method to the Music class which our view code can call before it adds a new performance to the container. This will also allow us to unit test our validation code.

Remember also, in the Phase 2 deliverables, we will be adding more sophisticated user interface controls. At that point, dates will be entered using some sort of date-picker control and it will be less likely that an invalid date will be entered by a user.

Creating Tests

We will start with some unit tests following the DocTest style. A tutorial on testing is available in the Grok documentation.

We will call our validation method "IsValidKey" and modify the source code at ~/grok/virtualgrok/music/src/music/app.py to read as follows:

```
import grok
class Music(grok.Application, grok.Container):
    def IsValidKey(self, keyVal):
        pass
class Index(grok.View):
    pass # see app_templates/index.pt
```

We do this so that the method exists for our tests to call. We don't want to write the code needed to satisfy our requirements yet. First we want to create some tests based on the requirements.

Create a new directory in the source code directory called "app_tests". We will use this to store all of our unit tests. (They can be located anywhere in the project source code and the test runner will find them. However, we will put them here in order to keep our project organized.)

```
cd ~/grok/virtualgrok/music/src/music/
mkdir app_tests
cd app_tests
```

Using your editor, create a file in this directory called "music.txt" with the following contents:

```
Tests for the Musical Performance Application.
*********************************************
:Test-Layer: unit
```

```
The Music class is a container that will hold all of the information
about "Performances".

When you create a new application instance there should be no objects
in the container::

   >>> from music.app import Music
   >>> performances = Music()
   >>> list(performances.keys())
   []


We are not going to test the pre-existing functionality of a grok.Container.
(That is already handled in its own unit tests.)
However, we do have application requirements that affect the format of
the key used to store performances in the container.

The IsValidKey function needs to return "True" only when a valid string
key of the form "yyyy-mm-dd" with an optional "-hh" suffix is provided.

First we check at the basic string format::

   >>> performances.IsValidKey(None)
   False
   >>> performances.IsValidKey('')
   False
   >>> performances.IsValidKey(0)
   False
   >>> performances.IsValidKey('Not Right')
   False
   >>> performances.IsValidKey('4/1/05')
   False
   >>> performances.IsValidKey('4/1/2005')
   False
   >>> performances.IsValidKey('2008.12.31')
   False
   >>> performances.IsValidKey('2008-3-15')
   False
   >>> performances.IsValidKey('08-3-5')
   False
   >>> performances.IsValidKey('2009--0315')
   False
   >>> performances.IsValidKey('2009-03-15')
   True

Strings that are formatted correctly should fail, if the date
is not valid::

   >>> performances.IsValidKey('2009-13-01')
   False
   >>> performances.IsValidKey('2009-03-32')
   False
   >>> performances.IsValidKey('2009-02-29')
   False

The suffix needs to be a dash, followed by an integer from 00 to 23::

   >>> performances.IsValidKey('2009-03-15Bad')
```

```
False
>>> performances.IsValidKey('2009-03-15-')
False
>>> performances.IsValidKey('2009-03-15-1')
False
>>> performances.IsValidKey('2009-03-15-00')
True
>>> performances.IsValidKey('2009-03-15-23')
True
>>> performances.IsValidKey('2009-03-15-24')
False
```

Now, we can run the application tests as we did in the previous section and see the results. Of course, everything will fail.

We can now begin implementing the IsValidKey method in app.py, until all of the tests pass. There are many ways to accomplish this, so feel free to use your own implementation. You can always re-factor it later.

Here is a sample implementation:

```
import grok
from datetime import datetime
import time

class Music(grok.Application, grok.Container):
    def IsValidKey(self, keyVal):
        if not isinstance(keyVal, basestring):
            return False
        if len(keyVal) in [10, 13]:
            keyParts = keyVal.strip().split('-')
            if len(keyParts) in [3, 4]:
                try:
                    datePart = '-'.join([keyParts[0], keyParts[1], keyParts[2]])
                    newDate = datetime(*(time.strptime(datePart, '%Y-%m-%d')[0:6]))
                except ValueError:
                    return False

                if len(keyParts) == 4:
                    try:
                        newTime = int(keyParts[3])
                        if newTime < 0 or newTime > 23:
                            return False
                    except ValueError:
                        return False

                # Everything checks out.
                return True

        return False

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

If you run the application tests again, everything should pass and you should get a result that looks like the following:

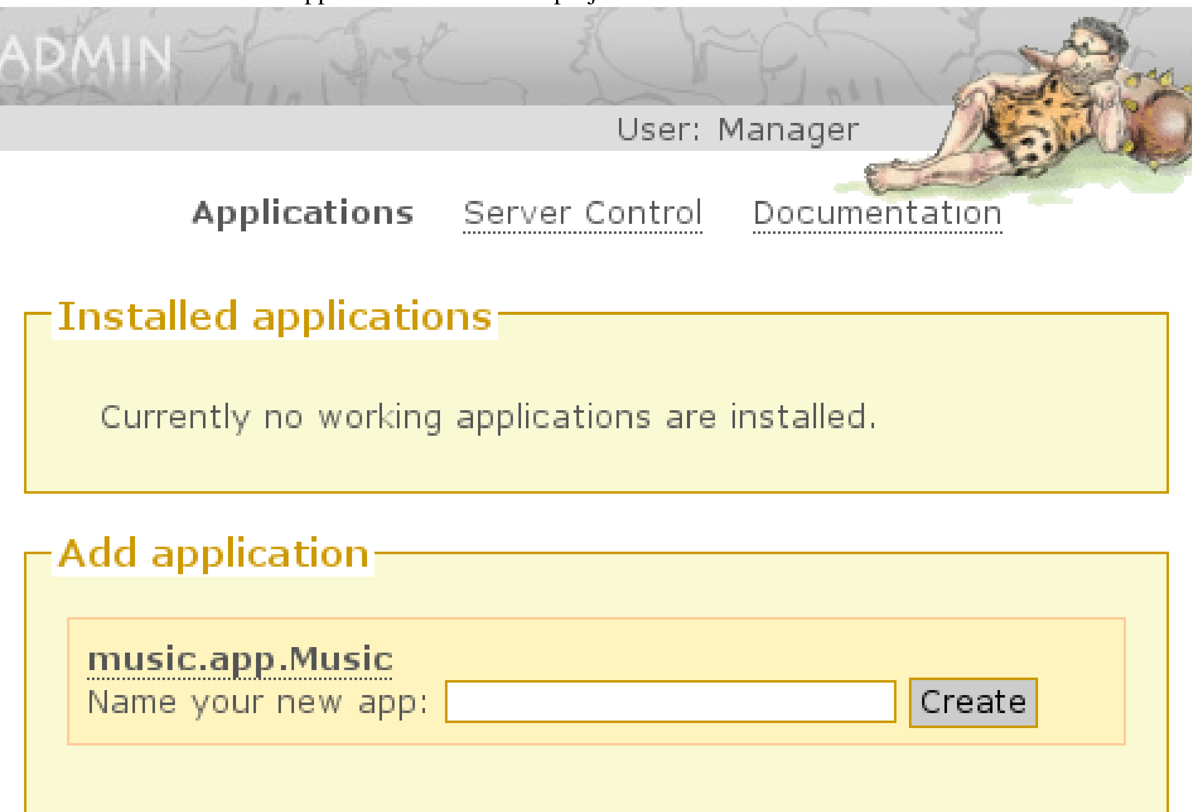
```
Running tests at level 1
Running unit tests:
  Running:
```

```
....
  Ran 4 tests with 0 failures and 0 errors in 0.011 seconds.
```

Next, we will work on creating the performance object and implementing a mechanism to add them to the container.

12.6.4 Initial Grok Admin Screen

The Grok Admin screen as it appears for a brand new project:



12.6.5 Grok Admin Screen with Test App

The Grok Admin Screen after an application instance has been added:

Installed applications

band (Music) [object browser]

Delete Selected Rename

12.6.6 Adding Performances

Create a basic content object which describes a Performance. Also, create a page template and the necessary view code so that a new performance can be added using a web browser.

Contents

- Adding Performances
  - The Performance Class
  - Adding a Performance
  - Editing a Performance

The Performance Class

It is time to create our first content object. If we review the entity outline, we can see that the Performance object is composed of both simple data fields and collections of other objects. We are going to start by implementing the simple string fields of "Location" and "Leader". The performance date will already be encoded as the object's key in the container object.

Since this object will also be a container, we will have it inherit from the grok.Container class also. This will provide a basic content object that will be easy to store in the object database and to render into an HTML page.

Before we work on the performance class directly, we are going to define its interface. Python doesn't generally use interfaces in the way that other languages do. They are used quite a bit in Zope programming for a variety of reasons. The reason we will use them right now, is to allow the auto-generation of forms and validation of user input. Later, you will find that they are valuable when adapting or combining different content objects to present their information in different ways without having to rework existing code.

Interfaces can also be an excellent way to document your code. By gathering them together in one place and surrounding them with meaningful comments, someone reading your code can get a quick overview of what your objects are going to do without having to read through all the implementation details.

For that reason we are going to gather all of our interfaces into one file called "interfaces.py" in the ~/grok/virtualgrok/music/src/music directory. Using your editor, create this file with the following content:

```
import grok
from zope import interface, schema

class IPerformance(interface.Interface):
    """ Represents a distict musical performance.

    This object defines the top level of our content.
    It will also "contain" other collections of content objects.

    location
        A free-form explaination of where the performance will occur.
    leader
        The name of the person co-ordinating a particular performance.
    starttime
        The time the performance is scheduled to begin.
        This information will be an optional, free-form string.
    endtime
        The time the performance is scheduled to end.
        This information will be an optional, free-form string.
```

```
    """

    location = schema.TextLine(title=u"Location")
    leader = schema.TextLine(title=u"Leader")
    starttime = schema.TextLine(title=u"Start Time", required = False)
    endtime = schema.TextLine(title=u"End Time", required = False)
```

All we have done so far is define that a performance will have a location and a leader. The "schema" assignments inform the rendering machinery that these two items can be represented as text lines on forms. We also added two optional string fields for the start and end time of the performance.

Grok allows you to auto-generate three types of forms: display forms, add forms, and edit forms. For add and edit forms, these fields will be rendered as text inputs. For display forms, the content of the field will be output as part of the HTML content.

The title attributes specify the label that will be associated with the HTML control. You can find more about schemas in the Working with Forms in Grok tutorial.

Now create a file called performance.py in the same directory, with the following content:

```
import grok
from interfaces import IPerformance
from zope import interface

class Performance(grok.Container):
    interface.implements(IPerformance)
    location = u''
    leader = u''
    starttime = u'--'
    endtime = u'--'

class Index(grok.DisplayForm):
    form_fields = grok.AutoFields(Performance)
```

Notice how this class uses the interface.implements directive to associated itself with the IPerformance interface. Also, note that we added a simple display form using grok.AutoFields and gave it the Performance class as a parameter.

When there is only one model class in a module (.py file), a view or form class will automatically associate itself with that class. In other words, that model class will become the view class's context. (You will hear and use this idea of "context" frequently.)

If the name of the view to be used in the URL is not explicitly stated, it will be the name of the view or form class in lowercase. When the name is "index", it will be the default view for the content object.

For the Performance object, the auto-generated Display Form will be used to render the contents when the URL specifies its key value in the container.

Now that we have a Performance object to add, we must revisit the Music class and enable its addition to the container.

Adding a Performance

As of yet, we have not seen anything rendered to HTML except the welcome screen generated by grokproject. First, we need to modify the template used to render the page. The main page of the application needs to list the available performances and provide a way to add new performances.

We will create an HTML list of the performances containing links to the URL of each performance. For now, we will provide a text box for the user to enter the date of a new performance. (Later we will implement a "date picker"

control.) Since we know that we can limit the choices for the "hour" portion of the performance key to one of twenty five choices ("None" or -00 thru -23), we will provide an option box for this entry.

Edit ~/grok/virtualgrok/music/src/music/app_templates/index.pt as follows:

```
<html>
<head>
</head>
<body>
  <h1>Performances</h1>

  <p>Available Performances:</p>
  <ul>
    <li tal:repeat="key python:context.keys()">
        <a tal:attributes="href python:view.url(key)" tal:content="python:key">Performance Date</a>
    </li>
  </ul>

  <form tal:attributes="action view/url" method="POST">
    New Performance Date: <input type="text" size="15" label="whatnot" name="NewPerformanceDate" valu
    <br />
    New Performance Hour:
    <select name="NewPerformanceHour" >
        <option value="">--</option>
        <option value="-00">00 / 12AM</option>
        <option value="-01">01 /  1AM</option>
        <option value="-02">02 /  2AM</option>
        <option value="-03">03 /  3AM</option>
        <option value="-04">04 /  4AM</option>
        <option value="-05">05 /  5AM</option>
        <option value="-06">06 /  6AM</option>
        <option value="-07">07 /  7AM</option>
        <option value="-08">08 /  8AM</option>
        <option value="-09">09 /  9AM</option>
        <option value="-10">10 /  10AM</option>
        <option value="-11">11 /  11AM</option>
        <option value="-12">12 /  12PM</option>
        <option value="-13">13 /  1PM</option>
        <option value="-14">14 /  2PM</option>
        <option value="-15">15 /  3PM</option>
        <option value="-16">16 /  4PM</option>
        <option value="-17">17 /  5PM</option>
        <option value="-18">18 /  6PM</option>
        <option value="-19">19 /  7PM</option>
        <option value="-20">20 /  8PM</option>
        <option value="-21">21 /  9PM</option>
        <option value="-22">22 /  10PM</option>
        <option value="-23">23 /  11PM</option>
    </select>
    (Optional)
    <br />
    <input type="submit" value="Add New Performance" name="SubmitButton" />
  </form>
</body>
</html>
```

For the most part this template is a simple HTML form. They key area to note is the unordered list section and the "tal:" attributes it contains. We are stating that we want to repeat list items for each of the items in the "context

object's" list of keys. (Here the "context object" will be the Music class, because that is the model class associated with the Index view in the app.py module shown below.)

We assign the value of each key to a variable called "key" and use that to generate the displayed text and the href attribute of the link tag.

We now need to modify the view (the Index class) used to render the Music class to interact with this template.

Use your editor to modify ~/grok/virtualgrok/music/src/music/app.py to the following:

```
import grok
from datetime import datetime
import time
from performance import Performance #Note that we now import the Performance object in this module.

class Music(grok.Application, grok.Container):
    def IsValidKey(self, keyVal):
        if not isinstance(keyVal, basestring):
            return False
        if len(keyVal) in [10, 13]:
            keyParts = keyVal.strip().split('-')
            if len(keyParts) in [3, 4]:
                try:
                    datePart = '-'.join([keyParts[0], keyParts[1], keyParts[2]])
                    newDate = datetime(*(time.strptime(datePart, '%Y-%m-%d')[0:6]))
                except ValueError:
                    return False

                if len(keyParts) == 4:
                    try:
                        newTime = int(keyParts[3])
                        if newTime < 0 or newTime > 23:
                            return False
                    except ValueError:
                        return False

                # Everything checks out.
                return True

        return False


class Index(grok.View):
    def update(self, SubmitButton=None, NewPerformanceDate=u'', NewPerformanceHour=u''):
        # Check if the submit button was clicked.
        if SubmitButton == 'Add New Performance':
            # Combine the contents of the two form fields to create a possible key.
            NewPerformanceKey = NewPerformanceDate + NewPerformanceHour
            # Check if the new key is valid.
            if self.context.IsValidKey(NewPerformanceKey):
                # Check if the key already exists in the container.
                if not self.context.has_key(NewPerformanceKey):
                    # Add a blank performance to the container.
                    self.context[NewPerformanceKey] = Performance()
                    # Output a confirmation note to the console.
                    print 'Created New Performance: ' + NewPerformanceKey
                    # Redirect the browser to the URL of the new page.
                    self.redirect(self.url(NewPerformanceKey))
```

The Index class inherits from grok.View, which will give it the ability to access model objects and render a template. Its context will be the Music class and it will attempt to use a template called index.pt (which we edited previously).

The update method runs every time the view is accessed and its parameters are automatically filled from the values of form controls in the template. The parameter names simply have to match the "name" attributes of the form controls.

We call the IsValidKey method on the Music class by referring to it through "self.context". If the key is valid and unique, we add a new Performance object to the container using that key and redirect to the URL of the new object. All the details of persisting the new object in the database are handled automatically.

Run the project and try adding a new performance for "2009-03-15".

tutorials/musical_performance_organizer/AddPerformanceForm.png

Restarting After Code Changes

When you change a HTML template file, you do not need to restart your application. If you change some Python code, you do need to restart the application web server for the changes to take effect.
Fortunately, there is a quick way to restart the server. Add the "–reload" option to your paster command as in:
./bin/paster serve –reload etc/deploy.ini
Now, whenever you save a source code file, the application will restart and your changes will take effect. There is a catch - if you save your code and it causes an exception, the server will stop and you will have to manually restart the server after you fix the problem.

If you look at console where you are running the application you should have an output that says, "Created New Performance: 2009-03-15". Your browser should move to a rather uninteresting screen that simply says, "Location", "Leader", "Start Time", and "End Time". These are the titles that where taken from the schema defined in the IPerformance interface. Since we created a blank performance object, the values for these fields are empty strings and dashes and we do not yet have a way to edit them.

Note the URL of our new Performance: http://localhost:8080/band/2009-03-15 The new part of the URL is this object's key value in the container. It follows the format specified in the requirements and is easy to understand.

We do not yet have a way to get back to the main application page. Instead of hitting the "back" button, browse to http://localhost:8080/band directly and see that we now have an entry listed under "Available Performances".

tutorials/musical_performance_organizer/FirstListedPerformance.png

Go ahead and try adding more performances with invalid or duplicate dates. Confirm that they are not added. (We will at some point need to give the user some feedback when an attempt to add a performance fails.)

Editing a Performance

If we want to be able to edit the two fields currently in our performance objects, we can do it by adding two lines of code to performance.py.

```
class Edit(grok.EditForm):
    form_fields = grok.AutoFields(Performance)
```

This will create an editable form view that is accessible by appending "/edit" to the URL of the performance object. We currently do not have a link to take you to this URL, so you will have to adjust it manually.

tutorials/musical_performance_organizer/FirstEdit.png

Try typing some text into the fields and press the "Apply" button. The asterisks mean the fields are required. Try leaving one of the required fields blank and note the validation error. Browse back to the display form and see that your changes have been applied.

While that is quite a bit of functionality from two lines of text, let's make this editing process work a little smoother and look nicer.

We will start by adding a bit more to the edit view so that we have some visual context of which performance we are working on. We will also override the default behavior of the form post action to give the button a new name and redirect the user to the application main page when editing is complete.

```
class Edit(grok.EditForm):
    form_fields = grok.AutoFields(Performance)

    def update(self):
        self.label = u'Edit Performance ' + self.context.__name__

    @grok.action('Save')
    def edit(self, **data):
        self.applyData(self.context, **data)
        self.redirect(self.url(self.context.__parent__))
```

The update method gets called any time the view is rendered. The self.label property will fill a heading in the default EditForm with the description that includes the performance name.

The @grok.action directive overrides the name of the button used to post the form. The applyData method is a quick way to map all of the form fields in the post request to the context of this view, which is the Performance object.

Finally, we redirect to the Performance object's parent, which is the Music application container object.

We can make our display form look a little nicer by adding a custom template. We are basically taking the default template from the grokcore.formlib package which can be found in your "buildout-eggs" cache directory and modifying it slightly to include a heading and links for editing and returning to the parent page.

Save the following in a file called "index.pt" in a new directory called ~/grok/virtualgrok/music/src/music/performance_templates.

```
<html>
<head>
</head>
<body>
  <h1>Performance for: <span tal:content="view/context/__name__">Unique Date</span></h1>
  <table class="listing" border="1" >
    <tbody>
```

```
      <tal:block repeat="widget view/widgets">
        <tr tal:define="odd repeat/widget/odd"
          tal:attributes="class python: odd and 'odd' or 'even'">
          <td class="fieldname" align="right">
            <tal:block content="widget/label"/>:
          </td>
          <td>
            <input tal:replace="structure widget" />
          </td>
        </tr>
      </tal:block>
    </tbody>
    <tfoot>
     <tr class="controls">
        <td colspan="2" class="align-right">
          <a href="edit">Edit Performance</a>
        </td>
      </tr>
      <tr class="controls">
        <td colspan="2" class="align-right">
          <a href="../">Return to List</a>
        </td>
      </tr>
    </tfoot>
  </table>
</body>
</html>
```

We need to instruct our display form to use this template by changing the Index view of the performance module to the following:

```
class Index(grok.DisplayForm):
    template = grok.PageTemplateFile('performance_templates/index.pt')
```

Finally, we want to edit the last line of our Index view code for the Music class so that when we add a new performance we are taken directly to the edit form.

```
class Index(grok.View):
    def update(self, SubmitButton=None, NewPerformanceDate=u'', NewPerformanceHour=u''):
        if SubmitButton == 'Add New Performance':
            NewPerformanceKey = NewPerformanceDate + NewPerformanceHour
            if self.context.IsValidKey(NewPerformanceKey):
                if not self.context.has_key(NewPerformanceKey):
                    self.context[NewPerformanceKey] = Performance()
                    print 'Created New Performance: ' + NewPerformanceKey
                    # Redirect the browser to the URL of the new object's edit page.
                    self.redirect(self.url(NewPerformanceKey) + '/edit')
```

We now have the beginnings of our application. In the next section, we will start adding some collections of other objects to our performance objects and begin writing some functional tests.

12.6.7 Adding the first performance

First use of the basic add form.

Performances

Available Performances:

New Performance Date: 2009-03-15 (Format: yyyy-mm-dd)
New Performance Hour: -- (Optional)
Add New Performance

12.6.8 First listed performance

The first blank performance object added to the container.

Performances

Available Performances:

- 2009-03-15

New Performance Date: (Format: yyyy-mm-dd)
New Performance Hour: -- (Optional)
Add New Performance

12.6.9 First Edit of a Performance

Basic EditForm

Updated on Mar 28, 2009 6:04:05 PM

*Location Carnegie Hall
*Leader Isaac Stern
Start Time 10am
End Time 12pm
Apply

12.6.10 Adding Musicians to a Performance

Add a list of musicians to a performance. Create some functional tests of the application.

Contents

- Adding Musicians to a Performance
  - Musicians
    - Interface
    - Implementation
    - Adding to a Performance
    - Viewing within a Performance
  - Functional Tests

Musicians

Now it is time to start adding some content related to a given performance. We are going to start by adding musicians to a performance. For a musician, we are going to be concerned about three items:, their name, their email address, and the instrument played.

Interface

Append the following to interfaces.py:

```
import re
expr_keyname = re.compile('[a-zA-Z][a-zA-Z0-9_ \.]*$')
check_keyname = expr_keyname.match

expr_email = re.compile(r"^(\w&.%#$&'\*+-/=?^_`{}|~]+!)*[\w&.%#$&'\*+-/=?^_`{}|~]+"
                        r"@(([0-9a-z]([0-9a-z-]*[0-9a-z])?\.)+[a-z]{2,6}|([0-9]{1,3}"
                        r"\.){3}[0-9]{1,3})$", re.IGNORECASE)
check_email = expr_email.match

class IMusician(interface.Interface):
    """ Represents a musician involved in a distict musical performance.

    name
        The name of the musician.  This name will be used to create a
        key for the object's storage in a performance, after is has been
        sanitized of special characters and can be used as a valid, but
        readable, URL.  The user will be limited to entering
        alpha-numeric characters.
    email
        The current email address of the musician.  This may be used in
        the future to contact the musician when something changes on a
        performance that he/she should know about.  The field will be
        validated as a properly formated email address.
    instrument
        The instrument or role that the musician will have in the
        performance.  This is a free-form string that may include more
        than one instrument or role.  (i.e. "piano, vocals")

    """

    name = schema.TextLine(title=u"Name", constraint=check_keyname)
    email = schema.TextLine(title=u"Email Address", constraint=check_email)
    instrument = schema.TextLine(title=u"Instrument")
```

Again, we are using simple text inputs to gather the information. Notice that this time the name and email schema entries specify constraints. These constraints are checked by the regular expressions defined above the class. (These do need to precede the class definitions.)

For more information on constraints, see the Automatic Form Generation howto.

Implementation

Create a new file called musician.py in the source code directory, and add the following code:

```
import grok
from zope import interface
from interfaces import IMusician

class Musician(grok.Model):
    interface.implements(IMusician)
    name=u''
    email=u''
    instrument = u''

class Index(grok.EditForm):
    grok.context(Musician)
    form_fields = grok.AutoFields(Musician)

    def update(self):
        self.label = u'Edit Musician'

    @grok.action('Save')
    def edit(self, **data):
        self.applyData(self.context, **data)
        self.redirect(self.url(self.context.__parent__.__parent__))

    def null_validator(self, action, data):
        return u''

    @grok.action('Cancel', validator=null_validator)
    def cancel(self, **data):
        self.redirect(self.url(self.context.__parent__.__parent__))

    @grok.action('Delete', validator=null_validator)
    def delete(self, **data):
        self.redirect(self.url(self.context.__parent__.__parent__))
        del self.context.__parent__[self.context.__name__]

class Musicians(grok.Container):
    pass
```

The Musician class itself should require no further explanation. Notice that, for now, we are using an EditForm for our default view. We may need to change this later when we implement permissions and roles, but for now this will suffice.

We again add an action called 'Save', but this time we are going to redirect to this object's grandparent instead of its parent. The reason for this will become obvious in a moment.

There are two additional actions called 'Cancel' and 'Delete', which are used as you would expect. We are using a new parameter to the @grok.action decorator which overrides the default form validation behavior by assigning it to a

new custom callable (method) named 'null_validator'. If we didn't override the form validation, the user would have to enter valid data before cancelling their edit or deleting a musician.

Also note how the 'del' command is used against the musician object by referencing the parent object. That is because the parent object will be a container just like the root object that holds the performances. We will use the same technique in the next step to allow us to remove entire performances also.

The last thing we added to this module, is a Musicians class which simply subclasses grok.Container and will be the parent container for the musician objects.

Adding to a Performance

Use your editor to once again edit performance.py. Change the content to the following:

```
import grok
from zope import interface
from interfaces import IPerformance
from musician import Musician, Musicians
from datetime import datetime

class Performance(grok.Container):
    interface.implements(IPerformance)
    location = u''
    leader = u''
    starttime = u'--'
    endtime = u'--'

    def __init__(self):
        super(Performance, self).__init__()
        self['musicians'] = Musicians()

class Index(grok.DisplayForm):
    template = grok.PageTemplateFile('performance_templates/index.pt')

class Edit(grok.EditForm):
    form_fields = grok.AutoFields(Performance)

    def update(self):
        self.label = u'Edit Performance ' + self.context.__name__

    @grok.action('Save')
    def edit(self, **data):
        self.applyData(self.context, **data)
        self.redirect(self.url(self.context))

    def null_validator(self, action, data):
        return u''

    @grok.action('Cancel')
    def cancel(self, **data):
        self.redirect(self.url(self.context))

    @grok.action('Delete', validator=null_validator)
    def delete(self, **data):
        self.redirect(self.url(self.context.__parent__))
        del self.context.__parent__[self.context.__name__]
```

```
class AddMusician(grok.AddForm):
    form_fields = grok.AutoFields(Musician)
    label = "Add Musician"

    @grok.action('Add')
    def add(self, **data):
        musician = Musician()
        self.applyData(musician, **data)
        keyname = musician.name.strip().replace(' ', '').replace('.', '')
        if keyname.isalnum():
            if not self.context['musicians'].has_key(keyname):
                self.context['musicians'][keyname] = musician
                return self.redirect(self.url(self.context))
```

We have now added an "__init__" method to the Performance class. This method acts as a constructor for the object and will be called every time a new Performance object is instantiated. (See note about "Super" below.) Notice that the constructor adds a new instance of the Musicians class to the performance's container using the key name 'musicians'. This name will become part of the URL that will be used to locate musicians in a given performance.

Super

You must add the "super(Performance, self).__init__()" line in order for this to work. Other assignments of default field values can also be made in the __init__ method instead of at the class level, but there are subtle difference in behavior.
There is often some confusion and controversy surrounding where default values are assigned in general, and the use of the super method in particular.
To learn more about assigning default values see, Understanding default values for object database backed attributes.
The short (and incomplete) reason you need to call the super method is that, by placing an __init__ method on the Performance class, you override all of the __init__ methods up the inheritance chain from grok.Container, which causes important code not to run. By calling "super", you cause the necessary code to run.
There is more to the discussion about super and others much more knowledgeable than myself have discussed this in depth. (If you are interested see: http://mail.zope.org/pipermail/grok-dev/2008-October/006509.html and https://bugs.launchpad.net/grok/+bug/162286)

Note that the 'Save' action has been modified to return to the Performance view after editing, instead of redirecting to the parent list of performances. This is because we now have other interesting information to edit from the Performance view.

We also added 'Cancel' and 'Delete' actions to the performance's edit form. However, the null_validator is only used on the Delete action, because we add new performances in an invalid state and we really want the user to add the required information instead of being able to easily cancel out of the form.

The AddMusician view inherits from AddForm and is similar in functionality to the EditForm we have used before. The musician schema validation will also be applied to the add form, however, although we allow spaces and periods in musician's name, we do not want them in our key value or the resulting URL's. We simply strip out the unwanted characters and then do a check to see if the key value already exists. If everything checks out, we add the new musician to the 'musicians' collection and redirect back to the performance page.

As with adding a new performance, we will at some point want to provide feedback if an addition fails. Later, we will allow the user to select musicians from a list which should allow us to avoid conflicts in the first place. Remember, that in this first deliverable, we are trying to provide a simple means for users to evaluate the completeness of the content objects and provide more feedback.

Viewing within a Performance

Rather than navigating to a separate form to view the musicians, we would like to incorporate the list right into the page that displays information about the performance. We will provide links to add a new musician and to edit existing ones.

To do this, we need to modify our default performance page template. Edit ~/grok/virtualgrok/music/src/music/performance_templates/index.pt to look like the following:

```
<html>
<head>
</head>
<body>
  <h1>Performance for: <span tal:content="view/context/__name__">Label</span></h1>
  <table class="listing" border="1" >
    <tbody>
      <tal:block repeat="widget view/widgets">
        <tr tal:define="odd repeat/widget/odd"
          tal:attributes="class python: odd and 'odd' or 'even'">
          <td class="fieldname" align="right">
            <tal:block content="widget/label"/>:
          </td>
          <td>
            <input tal:replace="structure widget" />
          </td>
        </tr>
      </tal:block>
    </tbody>
    <tfoot>
     <tr class="controls">
        <td colspan="2" class="align-right">
          <a href="edit">Edit Performance</a>
        </td>
      </tr>
      <tr class="controls">
        <td colspan="2" class="align-right">
          <a href="../">Return to List</a>
        </td>
      </tr>
    </tfoot>
  </table>
  <br />
  <h2>Musicians</h2>
  <table class="listing" border="1" >
    <thead>
      <tr><td>Musician</td><td>Instrument</td></tr>
    </thead>
    <tbody>
      <tal:block repeat="name python:context['musicians'].keys()">
        <tr>
          <td>
            <a tal:attributes="href python:view.url(context['musicians'][name])"
            tal:content="python:context['musicians'][name].name">Musician</a>
          </td>
          <td tal:content="python:context['musicians'][name].instrument">Instrument</td>
        </tr>
      </tal:block>
    </tbody>
```

```
    <tfoot>
      <tr class="controls">
        <td colspan="2" class="align-right">
          <a href="addmusician">Add Musician</a>
        </td>
      </tr>
    </tfoot>
  </table>
</body>
</html>
```

The big addition is a table to list the musicians. This is similar to how we listed our performances except we are adding table rows instead of list items. Context here still refers to the current Performance object, so context['musicians'] refers to this performance's own list of musicians.

Since the 'musicians' field is itself a container, we can iterate through its keys and assign them to a variable called 'name'. We then use this name to access information about individual musician objects to use in our table. We will list the name and instrument fields and the name field will be a link that will allow us to edit a musician.

In the footer of our table, we place a relative link that will display the addmusician view that we added to the Performance module earlier.

Make sure the web server is running with the latest code and navigate to a performance. Click the "Add Musician" link and test out the form.

tutorials/musical_performance_organizer/AddMusician.png

When you have added some musicians, you will see them listed on the performance page. Try editing and deleting some existing musicians.

tutorials/musical_performance_organizer/MusicianList.png

Functional Tests

While working with the forms in a browser can give us some quick feedback about how our application is working (or not working), we would rather have formal testing plan that we can quickly repeat whenever we reinstall or make changes to our application. This is where functional testing is very useful.

The testing framework will find your functional tests and run them along with your unit tests. This allows you to organize them however you would like. There is a sample functional test in the main source code directory that you can extend or use as a template for your own files.

You will definitely want to visit the URL provided in the sample to read about all of the testbrowser features: http://pypi.python.org/pypi/zope.testbrowser

To create the functional tests we will simply extend the app.txt file at: ~/grok/virtualgrok/music/src/music/app.txt

The original file:

```
Do a functional doctest test on the app.
========================================

:Test-Layer: functional

Let’s first create an instance of Music at the top level:

   >>> from music.app import Music
   >>> root = getRootFolder()
   >>> root['app'] = Music()

Run tests in the testbrowser
----------------------------

The zope.testbrowser.browser module exposes a Browser class that
simulates a web browser similar to Mozilla Firefox or IE.  We use that
to test how our application behaves in a browser.  For more
information, see http://pypi.python.org/pypi/zope.testbrowser.

Create a browser and visit the instance you just created:

   >>> from zope.testbrowser.testing import Browser
   >>> browser = Browser()
   >>> browser.open('http://localhost/app')

Check some basic information about the page you visit:

   >>> browser.url
   'http://localhost/app'
   >>> browser.headers.get('Status').upper()
   '200 OK'
```

The tests to append to the file:

```
   >>> print browser.url
   http://localhost/app

Add a new performance:

   >>> browser.getControl(None, 'NewPerformanceDate').value = '2009-03-28'
   >>> browser.getControl(None, 'SubmitButton', 0).click()
   Created New Performance: 2009-03-28
   >>> print browser.url
   http://localhost/app/2009-03-28/edit
   >>> frm = browser.getForm(None, None, None, 0)
   >>> print frm.getControl(None, 'form.location').value
   >>> print frm.getControl(None, 'form.leader').value

Check out the edit form:

   >>> frm.getControl(None, 'form.location').value = 'Where To Perform'
   >>> frm.getControl(None, 'form.leader').value = 'Who Leads'
   >>> frm.getControl(None, 'form.starttime').value = '10 AM'
   >>> frm.getControl(None, 'form.endtime').value = 'Late Afternoon'
   >>> frm.getControl(None, 'form.actions.save').click()
   >>> print browser.url
   http://localhost/app/2009-03-28
```

```
>>> browser.getLink('Return to List').click()
>>> link = browser.getLink('2009-03-28')
>>> print link.text
2009-03-28
```

View the display form:

```
>>> link.click()
>>> print browser.url
http://localhost/app/2009-03-28
>>> print browser.contents.find('Where To Perform') > 0
True
>>> print browser.contents.find('Who Leads') > 0
True
```

Add a musician:

```
>>> browser.getLink('Add Musician').click()
>>> print browser.url
http://localhost/app/2009-03-28/addmusician
>>> frm = browser.getForm(None, None, None, 0)
>>> print frm.getControl(None, 'form.name').value
>>> print frm.getControl(None, 'form.email').value
>>> print frm.getControl(None, 'form.instrument').value
>>> frm.getControl(None, 'form.name').value = 'Mr. Musician'
>>> frm.getControl(None, 'form.email').value = 'test@example.com'
>>> frm.getControl(None, 'form.instrument').value = 'Piano'
>>> frm.getControl(None, 'form.actions.add').click()
>>> print browser.url
http://localhost/app/2009-03-28
>>> print browser.contents.find('Mr. Musician') > 0
True
>>> print browser.contents.find('Piano') > 0
True
```

Edit the musician:

```
>>> browser.getLink('Mr. Musician').click()
>>> print browser.url
http://localhost/app/2009-03-28/musicians/MrMusician
>>> frm = browser.getForm(None, None, None, 0)
>>> print frm.getControl(None, 'form.instrument').value
Piano
>>> frm.getControl(None, 'form.instrument').value = 'Drums'
>>> frm.getControl(None, 'form.actions.save').click()
>>> print browser.url
http://localhost/app/2009-03-28
>>> print browser.contents.find('Drums') > 0
True
```

Delete the musician:

```
>>> browser.getLink('Mr. Musician').click()
>>> frm = browser.getForm(None, None, None, 0)
>>> frm.getControl(None, 'form.actions.delete').click()
>>> print browser.url
http://localhost/app/2009-03-28
>>> print browser.contents.find('Mr. Musician') > 0
```

```
   False

Delete the performance:

   >>> browser.getLink('Edit Performance').click()
   >>> frm = browser.getForm(None, None, None, 0)
   >>> frm.getControl(None, 'form.actions.delete').click()
   >>> print browser.url
   http://localhost/app
   >>> print browser.contents.find('2009-03-28') > 0
   False
```

As you can see, once you get used to the syntax, it is really easy to write the tests. Run ~/grok/virtualgrok/music/bin/test and you should get results like the following:

```
$ cd ~/grok/virtualgrok/music
$ ./bin/test
Running tests at level 1
Running unit tests:
  Running:
....
  Ran 4 tests with 0 failures and 0 errors in 0.011 seconds.
Running music.FunctionalLayer tests:
  Set up music.FunctionalLayer in 1.502 seconds.
  Running:
..........
  Ran 10 tests with 0 failures and 0 errors in 0.306 seconds.
Tearing down left over layers:
  Tear down music.FunctionalLayer ... not supported
Total: 14 tests, 0 failures, 0 errors in 1.945 seconds.
```

Next, we will move on to adding a song list (often called a set list) to our application.

12.6.11 Adding a Musician

Adding a musician to a performance.

Add Musician

*Name Isaac Stern

*Email Address istern@example.com

*Instrument Violin

Add

12.6.12 Viewing Musicians in a Performance

View the list of musicians within the performance page.

Performance for: 2009-03-15

Location:	Carnegie Hall
Leader:	Isaac Stern
Start Time:	10am
End Time:	12pm
Edit Performance	
Return to List	

Musicians

Musician	Instrument
Alexander Zakin	Piano
Isaac Stern	Violin
Add Musician	

12.6.13 Adding Songs to a Performance

Add a list songs to a performance. Prepare for uploading files that will be linked to the songs.

Contents

- Adding Songs to a Performance
  - Songs
    - Interface
    - Model and View Classes
    - Update Performance
    - Viewing within a Performance

Songs

We are now going to put together the set list for our performance. The first part of this will be similar to how we added the musicians. Later, we will have to add the ability to upload and attach documents related to items in the set list.

Interface

Append the following to interfaces.py:

```
class ISong(interface.Interface):
    """ Represents a song in the set list for a performance.

    title
        The title of the song.  This may include author information, if
        that is useful in identifying the song.  This title will be used
        to create a key for the object's storage in a performance, after
        is has been sanitized of special characters and can be used as
        part of a valid, but readable, URL.
    key
        The key the song will be performed in.  This is a free-form text
        field.
    arrangement
        Notes on how the song will be arranged or the key it will be
        played in.  Examples might be,
        "Verse 1, Chorus, Verse 2, Chorus, Bridge, Chorus"
        or "V1,Ch,V2,Ch,Br,Ch."  It can be any free-form string that
        the musicians will understand.
    order
        The order the songs will be performed within the set.  Also the
        display order of the item within the set list.  This will be an
        integer value.
    """

    title = schema.TextLine(title=u"Song")
    key = schema.TextLine(title=u"Key", max_length=20, required=False)
    arrangement = schema.TextLine(title=u"Arrangement Notes", required=False)
    order = schema.Int(title=u"Order", default=0)
```

You can see description of the fields we are adding for each song and the constraints placed on the input.

Model and View Classes

Create a new file in the "src" directory and name it "song.py". Paste the following code into the file:

```
import grok
from zope import interface
from interfaces import ISong
from zope.app.form.browser.textwidgets import TextWidget

class Song(grok.Container):
    interface.implements(ISong)
    title=u''
    key=u''
    arrangement=u''
    order = 0
```

```
class LongTextWidget(TextWidget):
    displayWidth = 50

class Index(grok.EditForm):
    grok.context(Song)
    form_fields = grok.AutoFields(Song)
    form_fields['title'].custom_widget = LongTextWidget
    form_fields['arrangement'].custom_widget = LongTextWidget

    def update(self):
        self.label = u'Edit Song for ' + self.context.__parent__.__parent__.__name__

    @grok.action('Save')
    def edit(self, **data):
        self.applyData(self.context, **data)
        self.redirect(self.url(self.context.__parent__.__parent__))

    def null_validator(self, action, data):
        return u''

    @grok.action('Cancel', validator=null_validator)
    def cancel(self, **data):
        self.redirect(self.url(self.context.__parent__.__parent__))

    @grok.action('Delete', validator=null_validator)
    def delete(self, **data):
        self.redirect(self.url(self.context.__parent__.__parent__))
        del self.context.__parent__[self.context.__name__]

class SetList(grok.Container):
    pass
```

Because the "Title" and "Arrangement" fields can be quite long, we want to be able to display a larger text box for user input. In order to do this, we need to override the standard TextWidget by creating our own LongTextWidget with a display width of 50 characters. The rest of this code should be familiar from the work we did previously.

Now, we must prepare the Performance class to work with the new song objects.

Update Performance

Near the top of "performance.py", add the following line:

```
from song import Song, SetList, LongTextWidget
```

The Performance class also needs to be modified to add a container to hold the songs:

```
class Performance(grok.Container):
    interface.implements(IPerformance)
    location = u''
    leader = u''
    starttime = u''
    endtime = u''

    def __init__(self):
        super(Performance, self).__init__()
```

```
        self['musicians'] = Musicians()
        self['setlist'] = SetList()
```

Note: This adds a new container for songs to any new performance object. You will only be able to add songs to Performances created after you have added this code. (If this system was already in use and needed to be updated, we would have to write the code differently to allow for updating old performances.) Since we a still developing, it might be best to delete and recreate your application instance using the Admin UI.

Append the following to the bottom of "performance.py":

```
class AddSong(grok.AddForm):
    form_fields = grok.AutoFields(Song)
    form_fields['title'].custom_widget = LongTextWidget
    form_fields['arrangement'].custom_widget = LongTextWidget
    label = "Add Song"

    def cleankey(self, dirtykey):
       cleanstring = []
       padded = False
       for c in dirtykey.strip(' _'):
           if c.isalnum():
               padded = False
               cleanstring.append(c)
           elif not padded:
               if c == ' ':
                   padded = True
                   cleanstring.append('_')
               elif '_-'.find(c) > -1:
                   padded = True
                   cleanstring.append(c)
       return ''.join(cleanstring).strip(' _')


    @grok.action('Add')
    def add(self, **data):
        song = Song()
        self.applyData(song, **data)
        keyname = self.cleankey(song.title)
        if not self.context['setlist'].has_key(keyname):
            self.context['setlist'][keyname] = song
            return self.redirect(self.url(self.context))
```

Before adding the new song to the set list, a key is generated by calling a method that is part of the add song class. This method should remove or replace any characters that we would not want to be part of the URL for this song.

Viewing within a Performance

Similar to the musician list, we would like to incorporate the list of songs right into the page that displays information about the performance. We will provide a links to add a new song and to edit existing ones.

To do this, we need to modify our default performance page template. Edit ~/grok/virtualgrok/music/src/music/performance_templates/index.pt to look like the following:

```
<html>
<head>
  <style type="text/css">
```

```
    table { empty-cells:show; }
  </style>
</head>
<body>
  <h1>Performance for: <span tal:content="view/context/__name__">Label</span></h1>
  <table class="listing" border="1" >
    <tbody>
      <tal:block repeat="widget view/widgets">
        <tr tal:define="odd repeat/widget/odd"
          tal:attributes="class python: odd and 'odd' or 'even'">
          <td class="fieldname" align="right">
            <tal:block content="widget/label"/>:
          </td>
          <td>
            <input tal:replace="structure widget" />
          </td>
        </tr>
      </tal:block>
    </tbody>
    <tfoot>
     <tr class="controls">
        <td colspan="2" class="align-right">
          <a href="edit">Edit Performance</a>
        </td>
      </tr>
      <tr class="controls">
        <td colspan="2" class="align-right">
          <a href="../">Return to List</a>
        </td>
      </tr>
    </tfoot>
  </table>
  <br />
  <h2>Musicians</h2>
  <table class="listing" border="1" >
    <thead>
      <tr><td>Musician</td><td>Instrument</td></tr>
    </thead>
    <tbody>
      <tal:block repeat="name python:context['musicians'].keys()">
        <tr>
          <td>
            <a tal:attributes="href python:view.url(context['musicians'][name])"
            tal:content="python:context['musicians'][name].name">Musician</a>
          </td>
          <td tal:content="python:context['musicians'][name].instrument">Instrument</td>
        </tr>
      </tal:block>
    </tbody>
    <tfoot>
      <tr class="controls">
        <td colspan="2" class="align-right">
          <a href="addmusician">Add Musician</a>
        </td>
      </tr>
    </tfoot>
  </table>
  <br />
```

```
  <h2>Set List</h2>
  <table class="listing" border="1" >
    <thead>
      <tr><td>#</td><td>Songs</td><td>Key</td><td>Arrangement</td></tr>
    </thead>
    <tbody>
      <tal:block repeat="song python:sorted(context['setlist'].values(), key=lambda obj:obj.order)">
        <tr>
          <td tal:content="python:song.order">#</td>
          <td>
            <a tal:attributes="href python:view.url(song)"
            tal:content="python:song.title">Title</a>
          </td>
          <td tal:content="python:song.key">Key</td>
          <td tal:content="python:song.arrangement">Arrangement</td>
        </tr>
      </tal:block>
    </tbody>
    <tfoot>
      <tr class="controls">
        <td colspan="4" class="align-right">
          <a href="addsong">Add Song</a>
        </td>
      </tr>
    </tfoot>
  </table>
</body>
</html>
```

Notice the new table at the end of the template used to display the songs. Since we do not want the songs sorted alphabetically (as they would if we used tried to use the key value derived from the title), but by the value of the "order" field, we need to make this happen.

We could create a new method on our performance model object or in the view to return a sorted list of songs. Instead, we accomplish this right in the template by using Python's built-in "sorted" function. The "lambda" operator allows us to specify the sort condition directly without having to created a new named function. (See the Python Sorting Mini-HOW TO.)

Start the application and create a new performance. The new performance should have a "Set List" section. You should be able to add and edit songs using a form that looks similar to the following:

tutorials/musical_performance_organizer/AddSong.png

Note: If you have problems viewing previously created performances, you may need to reset your application by going to the Grok AdminUI to delete and re-create your test application with the same application name. If we were upgrading an application that was already in use by others, we would create the code necessary to migrate existing stored objects to the new application. Since we are developing the first version of this application, it is easier to recreate our test objects as needed.

12.6.14 Adding a Song

Add Song

*Song

Key

Arrangement Notes

*Order 0

Add

12.6.15 Attaching Files to a Song

Enable users to upload sheet music and other documents related to a song in the set list.

Contents

- Attaching Files to a Song
  - Attachments
    - Adding Components
    - Adjusting the Model
    - Updating the Edit Form
    - Updating the Performance View
  - Functional Tests

Attachments

It is now time to add file attachments to the songs in a set list. To do this, we will need to use some components from the Zope3 component architecture that are not included in our default Grok installation.

We will be using the zope.app.file component, which you will find more fully documented in the How-to titled: Handling file uploads with zope.app.file and zope.file

Adding Components

To add new components to our application we will add them to our project's setup.py file. Edit ~/grok/virtualgrok/music/setup.py and add the line 'zope.app.file' to the "install_requires" property, so the file contents look like this:

```
from setuptools import setup, find_packages

version = '0.0'

setup(name='music',
      version=version,
      description="",
```

```
      long_description="""\
""",
      # Get strings from http://www.python.org/pypi?%3Aaction=list_classifiers
      classifiers=[],
      keywords="",
      author="",
      author_email="",
      url="",
      license="",
      package_dir={'': 'src'},
      packages=find_packages('src'),
      include_package_data=True,
      zip_safe=False,
      install_requires=['setuptools',
                        'grok',
                        'grokui.admin',
                        'z3c.testsetup',
                        'grokcore.startup',
                        # Add extra requirements here
                        'zope.app.file',
                        ],
      entry_points = """
      [console_scripts]
      music-debug = grokcore.startup:interactive_debug_prompt
      music-ctl = grokcore.startup:zdaemon_controller
      [paste.app_factory]
      main = grokcore.startup:application_factory
      """,
      )
```

We now need to run buildout again which will download the necessary software and configure it to be available in our project.

```
$ cd ~/grok/virtualgrok/music
$ ./bin/buildout
```

Adjusting the Model

We will start by editing song.py in the src directory to look like the following:

```
import grok
from zope import interface
from interfaces import ISong
from zope.app.form.browser.textwidgets import TextWidget

import zope.app.file
from zope.app.container.interfaces import INameChooser
from zope.app.container.contained import NameChooser
import urllib

class Song(grok.Container):
    interface.implements(ISong)
    title=u''
    key=u''
    arrangement=u''
```

```
    order = 0

    def __init__(self):
        super(Song, self).__init__()
        self['files'] = FileContainer()

class LongTextWidget(TextWidget):
    displayWidth = 50

class Index(grok.EditForm):
    grok.context(Song)
    form_fields = grok.AutoFields(Song)
    form_fields['title'].custom_widget = LongTextWidget
    form_fields['arrangement'].custom_widget = LongTextWidget

    def update(self):
        self.label = u'Edit Song for ' + self.context.__parent__.__parent__.__name__

    @grok.action('Save')
    def edit(self, **data):
        self.applyData(self.context, **data)
        self.redirect(self.url(self.context.__parent__.__parent__))

    def null_validator(self, action, data):
        return u''

    @grok.action('Cancel', validator=null_validator)
    def cancel(self, **data):
        self.redirect(self.url(self.context.__parent__.__parent__))

    @grok.action('Delete', validator=null_validator)
    def delete(self, **data):
        for file in list(self.context['files'].keys()):
            del self.context['files'][file]
        self.redirect(self.url(self.context.__parent__.__parent__))
        del self.context.__parent__[self.context.__name__]

class SetList(grok.Container):
    pass

class DeleteFile(grok.View):
    grok.context(Song)
    def render(self):
        filename = urllib.unquote(self.request.get('QUERY_STRING'))
        if filename and self.context['files'].has_key(filename):
            del self.context['files'][filename]
            self.redirect(self.url(self.context))

class FileContainer(grok.Container):
    pass

class AddFile(grok.AddForm):
    grok.context(Song)
    form_fields = grok.AutoFields(zope.app.file.interfaces.IFile).select('data')

    @grok.action('Upload')
    def add(self, data):
        if len(data) > 0:
```

```
            self.upload(data)
        self.redirect(self.url(self.context))

    def upload(self, data):
        fileupload = self.request['form.data']
        if fileupload and fileupload.filename:
            contenttype = fileupload.headers.get('Content-Type')
            file_ = zope.app.file.file.File(data, contenttype)
            # use the INameChooser registered for your file upload container
            filename = INameChooser(self.context['files']).chooseName(fileupload.filename, None)
            self.context['files'][filename] = file_

class PrimitiveFilenameChangingNameChooser(grok.Adapter, NameChooser):
    grok.context(Song)
    grok.implements(INameChooser)
    grok.adapts(FileContainer)

    def chooseName(self, name):
        if name.startswith('+'):
            name = 'plus-' + name[1:]
        if name.startswith('@'):
            name = 'at-' + name[1:]
```

As you can see we imported several new items for use in our code. The zope.app.file will provide our storage and NameChooser will assure that our file names do not collide with one another.

Similar to the Performance object, we have added a sub-container to the Song object. Instead of being a new grok.container, it will be a new instance of FileContainer. The delete action for a song has also been modified to remove any songs. (Apparently, the contents of the FileContainer are not cleaned up as automatically as a grok.container.) Failure to remove the files before deleting the songs will result in a error like:

```
TypeError: There isn't enough context to get URL information.
This is probably due to a bug in setting up location information.
```

Notice that we have also added a new "DeleteFile" view. This time, we will not be providing the name of the file to be deleted via a form submission, but rather through a query string. This requires us to properly encode the name of the file for use in an URL and to decode it for use in our application. For this we use the Python "urllib" library.

We will generate a grok.AddForm based on the schema provided by the IFile interface. This will include a text box with a button used to browse for a file, and a second button to upload the selected file.

The upload method will use the NameChooser to fix potentially problematic file names and also to modify the filename by adding a numeric suffix when a particular filename already exists for this song.

Updating the Edit Form

We can no longer use the standard, generated edit form if we want to be able to display a list of file attachments for a song. In the src/music directory, create a new directory called "song_templates" and add the following file named "index.pt" to the directory:

```
<html>
<head>
  <style type="text/css">
    table { empty-cells:show; }
  </style>
</head>
```

```
<body>
<form action="." tal:attributes="action request/URL" method="post"
      class="edit-form" enctype="multipart/form-data">

  <h1 i18n:translate=""
    tal:condition="view/label"
    tal:content="view/label">Label</h1>

  <div class="form-status"
    tal:define="status view/status"
    tal:condition="status">

    <div i18n:translate="" tal:content="view/status">
      Form status summary
    </div>

    <ul class="errors" tal:condition="view/errors">
      <li tal:repeat="error view/error_views">
         <span tal:replace="structure error">Error Type</span>
      </li>
    </ul>
  </div>

  <table class="form-fields">
    <tbody>
      <tal:block repeat="widget view/widgets">
        <tr>
          <td class="label" tal:define="hint widget/hint">
            <label tal:condition="python:hint"
                   tal:attributes="for widget/name">
              <span class="required" tal:condition="widget/required"
              >*</span><span i18n:translate=""
                             tal:content="widget/label">label</span>
            </label>
            <label tal:condition="python:not hint"
                   tal:attributes="for widget/name">
              <span class="required" tal:condition="widget/required"
              >*</span><span i18n:translate=""
                             tal:content="widget/label">label</span>
            </label>
          </td>
          <td class="field">
            <div class="widget" tal:content="structure widget">
              <input type="text" />
            </div>
            <div class="error" tal:condition="widget/error">
              <span tal:replace="structure widget/error">error</span>
            </div>
          </td>
        </tr>
      </tal:block>
    </tbody>
  </table>

  <div id="actionsView">
    <span class="actionButtons" tal:condition="view/availableActions">
      <input tal:repeat="action view/actions"
             tal:replace="structure action/render"
```

```
                  />
        </span>
      </div>
    </form>
      <h2>File List</h2>
      <table class="listing" border="1" >
        <thead>
          <tr><td>File URL</td><td>File Type</td><td>Remove</td></tr>
        </thead>
        <tbody>
          <tal:block repeat="file python:context['files'].keys()">
            <tr>
              <td>
                <a tal:attributes="href python:view.url(context['files'][file])"
                tal:content="python:file">Title</a>
              </td>
              <td tal:content="python:context['files'][file].contentType">Arrangement</td>
              <td>
                <a tal:attributes="href python:view.url(context) + '/deletefile?' + file">Delete</a>
              </td>
             </tr>
          </tal:block>
        </tbody>
        <tfoot>
          <tr class="controls">
            <td colspan="4" class="align-right">
              <form action="./addfile" method="post" class="edit-form" enctype="multipart/form-data">
                <table class="form-fields">
                  <tbody>
                    <tr>
                      <td class="label">
                        <label for="form.data">
                          <span>New Attachment: </span>
                        </label>
                      </td>
                      <td class="field">
                        <div class="widget">
                          <input class="hiddenType" id="form.data.used" name="form.data.used" type="hidde
                          <input class="fileType" id="form.data" name="form.data" size="20" type="file"
                          <input type="submit" id="form.actions.upload" name="form.actions.upload" value=
                      </td>
                    </tr>
                  </tbody>
                </table>
              </form>
            </td>
          </tr>
        </tfoot>
      </table>
</body>
</html>
```

Auto-generated Button Names

When you provide a name for one of your "grok.action" buttons, the value you provide is displayed on the button. It is also used to generate values for the "id" and "name" HTML attributes. This is done by prefixing "form.action." to the lowercased name you provided.
However, if the name contains characters that are invalid for those attributes (this includes something as simple as adding a space to the name), the auto form will substitute a hex representation of the name.
A better way to specify the name you want for the button and still have a reasonable value for the control name, is to use an under-documented features of the "grok.action" directive. By providing a named-parameter called "name" you can specify the control name attribute separately from the display text on the button.
@grok.action('Big name for button', name='ValidNameForAction')

The top portion is the standard edit form, but the bottom portion will render a table that contains the attached files. The "File URL" column will contain a link to download the stored file attachment.

The "Remove" column will construct an URL which will pass the name of a given file to the "DeleteFile" view in the query string.

Notice that in the footer of the table, instead of providing a link to the add form, we actually provide the form controls required to add a file attachment and we post the result to the AddFile auto-form view.

It is important that the control names in this template match those in the auto-form. If you have problems, you can view the generated add form directly, by appending "/addfile" to the URL of song and viewing the HTML source.

The AddFile auto-form will redirect back to the song view after it adds the file. So it will appear to the user like we never left the form. Not quite as seamless as using AJAX, but it is nicer than than bouncing between two different forms when adding several objects. (We will address the use of AJAX in a later tutorial installment.)

tutorials/musical_performance_organizer/AttachFiles.png

Updating the Performance View

To enhance the user experience further, it would be nice to allow the user to download attached files directly from the list of songs in the Performance view. Modify the set list table within the page template src/music/performance_templates/index.pt as follows:

```
<h2>Set List</h2>
<table class="listing" border="1" >
  <thead>
    <tr><td>#</td><td>Songs</td><td>Key</td><td>Arrangement</td><td>Files</td></tr>
  </thead>
  <tbody>
    <tal:block repeat="song python:sorted(context['setlist'].values(), key=lambda obj:obj.order)">
      <tr>
        <td tal:content="python:song.order">#</td>
        <td>
          <a tal:attributes="href python:view.url(song)"
          tal:content="python:song.title">Title</a>
        </td>
        <td tal:content="python:song.key">Key</td>
        <td tal:content="python:song.arrangement">Arrangement</td>
```

```
        <td>
          <tal:block repeat="file python:song['files'].keys()">
            <a tal:attributes="href python:view.url(song['files'][file])"
              tal:content="python:file">File</a><br />
          </tal:block>
        </td>
      </tr>
    </tal:block>
  </tbody>
  <tfoot>
    <tr class="controls">
      <td colspan="5" class="align-right">
        <a href="addsong">Add Song</a>
      </td>
    </tr>
  </tfoot>
</table>
```

We have added another repeat block in the right-most column of the table which will display a list of file attachment links within each table row. It will be unlikely to have more than one or two attachments per file.

tutorials/musical_performance_organizer/PerformanceWithAttachments.png

Functional Tests

12.6.16 Attaching Files

Edit Song for 2009-03-01

*Song	Sample Song
Key	C
Arrangement Notes	1,Ch,2,Ch,Br,Ch
*Order	10

Save Cancel Delete

File List

File URL	File Type	Remove
ChordChart.doc	application/msword	Delete
sg247529.pdf	application/pdf	Delete
New Attachment: Browse... Upload		

12.6.17 Performance with File Attachments

Musician	Instrument
Alexander Zakin	Piano
Add Musician	

Set List

#	Songs	Key	Arrangement	Files
10	Sample Song	C	1,Ch,2,Ch,Br,Ch	ChordChart.doc sg247529.pdf
20	Other Song	D	1,Ch,2,Ch,Ch	Song.pdf
Add Song				

12.6.18 Adding Comments to a Performance

Contents

- Adding Comments to a Performance
  - Comments
    - Interface
    - Implementation
    - Page Template Changes
  - Functional Tests

Comments

We have almost implemented all of the Phase 1 requirements. (See entity outline) However, we still need to provide a basic commenting function.

One approach would be to use annotations, which are a means of adding extra information to any content object without explicitly changing the object itself. All grok Model and Container objects are "annotate-able" by using grok.Annotation.

However, annotations are best used for information that will be applied to many classes of objects, but is accessed less frequently than the content of the object itself. Since, we only want to add comments to our Performance class and we will be displaying the comments every time someone views a performance, an Annotation would not be the best choice in this case.

Instead, we will follow our previous pattern of adding a new collection of content objects directly to our performances. While we will not yet be adding authentication and authorization requirements to our comments, we will be adding some extra data about the comments and who authored them to our content class.

This will pave the way for adding security and change tracking features in the future, and it will also allow us to learn a bit more about using the request object to access HTTP header information.

Interface

We will begin by adding a new interface definition. Append the following to our existing interfaces.py module:

```
class IComment(interface.Interface):
    """ Represents a comment involved in a distict musical performance.

    datePosted
        The date the comment was originally posted.
    whoPosted
        The person who originally posted the comment.
        If the user is not authenticated, the browser agent
        IP address is used.
    email
        The email address of the user posting the comment.
    text
        The text of the comment.
    dateModified
        The last date and time the comment was modified.
    whoModified
        The person who last modified the comment.
        If the user is not authenticated, the browser agent
        IP address is used.
```

```
    """

    datePosted = schema.TextLine(title=u"Posted", readonly=True)
    whoPosted = schema.TextLine(title=u"Posted By", readonly=True)
    email = schema.TextLine(title=u"Email", constraint=check_email)
    text = schema.Text(title=u"Comment")
    dateModified = schema.TextLine(title=u"Modified", readonly=True)
    whoModified = schema.TextLine(title=u"Modified By", readonly=True)
```

Notice that four of the six entries are read-only. The first two will be filled in when adding the comment. The other two will be updated any time the comment is edited. The form will only pass back user-entered data for the email address and comment text. The email address will be checked that it formatted in a valid fashion, however, no validating email will be sent to the address to confirm that the author can be contacted.

We will be using the HTTP headers to determine the IP address of unauthenticated users. For now, this will be the only type of user. While IP address is not a very sure way of identifying someone we will use it for now.

Implementation

In order to implement the new interface, we will create a file called "comment.py" in the src/music directory. Add the following to the new file:

```
import grok
from zope import interface
from interfaces import IComment
from datetime import datetime

class Comment(grok.Model):
    interface.implements(IComment)
    datePosted=u''
    whoPosted=u''
    email=u''
    text=u''
    dateModified=u''
    whoModified=u''

class Index(grok.EditForm):
    grok.context(Comment)
    form_fields = grok.AutoFields(Comment)

    def update(self):
        self.label = u'Edit Comment for ' + self.context.__parent__.__parent__.__name__

    @grok.action('Save')
    def edit(self, **data):
        self.applyData(self.context, **data)
        self.context.dateModified = datetime.now().strftime("%Y-%m-%dT%H-%M-%S")
        self.context.whoModified = self.request.getHeader('REMOTE_ADDR')
        self.redirect(self.url(self.context.__parent__.__parent__))

    def null_validator(self, action, data):
        return u''

    @grok.action('Cancel', validator=null_validator)
    def cancel(self, **data):
        self.redirect(self.url(self.context.__parent__.__parent__))
```

```
    @grok.action('Delete', validator=null_validator)
    def delete(self, **data):
        self.redirect(self.url(self.context.__parent__.__parent__))
        del self.context.__parent__[self.context.__name__]

class Comments(grok.Container):
    pass
```

Notice that we have created a simple edit view for the comment, but no display view. As with our other content, we will integrate the display of comments into our Performance view.

The "self.applyData" statement will transfer the content of all the form fields to the object attributes, however, remember that the user can not edit the fields marked as "read-only". These attributes will be modified after the form data is applied. The "dataModified" attribute receives a string representing the current date and time. The "whoModified" attribute receives a string representation of the user's IP address from the request object.

The request object contains information about the several aspects of the HTTP request. (See the Grok Developer's Notes for more details.)

We are interested in a particular environment variable from the request: REMOTE_ADDR. This contains the IP address provided to web server in the HTTP request.

In order to add new comments, we will append the following code to performance.py:

```
class AddComment(grok.AddForm):
    form_fields = grok.AutoFields(Comment).select('email','text')
    label = "Add Comment"

    def update(self):
        self.label = u'Add Comment for ' + self.context.__name__

    @grok.action('Add')
    def add(self, **data):
        comment = Comment()
        self.applyData(comment, **data)
        comment.datePosted = datetime.now().strftime("%Y-%m-%dT%H-%M-%S")
        comment.whoPosted = self.request.getHeader('REMOTE_ADDR')
        keyname = comment.datePosted
        if not self.context['comments'].has_key(keyname):
            self.context['comments'][keyname] = comment
            return self.redirect(self.url(self.context))
```

This is very similar to the edit view code, except that we are populating the other read-only attributes. Also, notice that we are storing the comments in our container based on the date and time we calculated. This will prevent duplication as long as comments are not added to a particular performance faster than once per second.

> Note: This could be a concern for a site with heavy user traffic, but it is a limitation we are willing to live with for this for the practical usage of this example. We will, however, need to be aware of this limitation when doing functional testing, as the testing mechanism could easily add several comments per second.

The Performance class also needs to be modified to add a container to hold the comments:

```
import grok
from zope import interface
from interfaces import IPerformance
from musician import Musician, Musicians
from datetime import datetime
from song import Song, SetList, LongTextWidget
```

```
from comment import Comment, Comments
import time

class Performance(grok.Container):
    interface.implements(IPerformance)
    location = u''
    leader = u''
    starttime = u''
    endtime = u''
    timezone= unicode(time.strftime("%Z", time.localtime()))

    def __init__(self):
        super(Performance, self).__init__()
        self['musicians'] = Musicians()
        self['setlist'] = SetList()
        self['comments'] = Comments()
```

A new "timezone" attribute was also added to the Performance class so that we can notify the user what timezone the server is using when determining timestamps on the comments. An instance of this application is unlikely to be used across wide geographic regions and we do not have enough reliable information about an anonymous user's preferred timezone settings to adjust the times correctly, so local time will be sufficient. (Hopefully, the application is hosted by a provider that is geographically close to the people using the application. Further enhancements could certainly be made to make the timezone configurable, but they will not be covered as part of this tutorial.)

Page Template Changes

We still need to update the template at src/music/performance_templates/index.pt, by appending the following within the body following the "Set List" table:

```
<h2>Comments</h2>
<table class="listing" border="1" >
  <thead>
    <tr><td>Date (Timezone: <span tal:content="python:context.timezone">TimeZone</span>)</td><td>Name
  </thead>
  <tbody>
    <tal:block repeat="key python:context['comments'].keys()">
      <tr>
        <td>
          <a tal:attributes="href python:view.url(context['comments'][key])"
          tal:content="python:context['comments'][key].datePosted">Date Posted</a>
        </td>
        <td tal:content="python:context['comments'][key].whoPosted">Who Posted</td>
        <td><pre tal:content="python:context['comments'][key].text">Comment Text</pre></td>
      </tr>
    </tal:block>
   </tbody>
  <tfoot>
    <tr class="controls">
      <td colspan="3" class="align-right">
        <form action="./addcomment" method="post" class="edit-form" enctype="multipart/form-data">
          <table class="form-fields">
            <tbody>
              <tr>
                <td class="label">
                  <label for="form.email">
                    <span>Email Address</span>
```

```
                    </label>
                  </td>
                  <td class="field">
                    <div class="widget">
                      <input class="textType" id="form.email" name="form.email" size="20" type="text"
                      <input type="submit" id="form.actions.add" name="form.actions.add" value="Add" cl
                  </td>
                </tr>
                <tr>
                  <td class="label">
                    <label for="form.text">
                      <span>Comment</span>
                    </label>
                  </td>
                  <td class="field">
                    <div class="widget">
                      <textarea cols="60" id="form.text" name="form.text" rows="5" ></textarea>
                  </td>
                </tr>
              </tbody>
            </table>
          </form>
        </td>
      </tr>
    </tfoot>
</table>
```

Similar to our file attachment form, this template will list the existing comments and provide form fields and a button for submitting a new comment.

tutorials/musical_performance_organizer/AddComment.png

The email address the user provides is not included in the comment listing as access to this information will eventually be hidden from the public.

Functional Tests

TBD

This concludes the first part of the Musical Performance Organizer tutorial.

12.6.19 Adding a Comment

Add a new comment and view the list of existing comments.

Comments

Date (Timezone: CDT)	Name	Comment
2009-06-30T21-34-38	192.168.1.10	This is a sample comment. Any sort of information can be typed here, including HTML tags which will be properly encoded.

Email Address [ ] Add

Comment

CHAPTER

THIRTEEN

LICENSE

This work is licensed under the Creative Commons Attribution-ShareAlike 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/3.0/ or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.

If you need a copy of the Grok Community Documentation licensed differently (for instance because you're planning to write a book or similar), please contact grok-dev@zope.org.

CHAPTER

FOURTEEN

COPYRIGHT

Copyrights for the Grok Community Documentation are by

```
Grok Community and Contributors.
```

CHAPTER

FIFTEEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*