Grok; A component oriented approach to web applications

Table of Contents

Contents

List of Figures

List of Figures

1 Introduction

Grok is based on the Zope Component Architecture (ZCA), which is just a big name for a Python global component registry. The way components are normally registered, were one to use Zope 3 but not Grok, is to use an XML dialect called ZCML (*Zope Configuration Markup Language*). Instead of ZCML, Grok lets one use directives embedded in Python source to accomplish the same thing, bypassing the need for ZCML. This approach is much friendlier to the coder, and makes the source easier to read and maintain.

What the ZCA buys us, is the ability to write modular code with minimal inter-dependencies. The use of components fosters code re-use and reduces the need for using inheritance or mix-ins to accomplish relationships between Python objects.

This article is about Grok, and how it relates to the ZCA. We shall describe components and component interfaces and how they are used by Grok to provide a framework for developing web applications. We will finally show the implementation for a full featured web application, and discuss the approaches taken to reduce the coding overhead.

1.1 Foreword

Many of the Grok tutorials, and in fact much written about Zope & Grok in general, make the often mistaken assumption that the reader is already conversant with the Zope architecture. This assumption leads to seemingly incomprehensible documentaton for the newcomer. Panic sets in, and an immediate and automatic knee-jerk discounting by the reader of the merits of the platform.

Such comments as "*With Zope, the simple things are hard*" abound. Others describe the "*Z shaped learning curve*" of Zope. In reality, Grok (and Zope) are no harder to learn than learning another platform, except for the fact that it takes a non-conformist approach to how web platforms should be made.

For these reasons, this article makes no assumption about the reader regarding prior knowlege, and with the hope that the reader will approach Grok & Zope without prejudice.

2 What is a component?

In essence, a component is a Python callable (function or class) which *implements* a defined *interface*.

Which leads to the next question; what is an *Interface*?

Well, an *Interface* describes a set of methods or attributes which may be implemented independently in concrete classes elsewhere. The *implementor* of an interface may be made available to other areas of the application by using the global component registry[1].

2.1 Some examples of components

Imagine we have a user for which we want to define a Python class. A user has a *name*, a *surname*, and an *email* address.

```
from zope.interface import Interface, Attribute
class IUser(Interface):

     name = Attribute('name')
     surname = Attribute('surname')
     email = Attribute('email')
```

Effectively, the above declares a basic description for a user as an interface. There are no implementation details included in an interface, since the intent is to define the implementation elsewhere.

```
class User(object):

     grok.implements(IUser)
     def __init__(name=None, surname=None, email=None):
          self.name = name if name else ”
          self.surname = surname if surname else ”
          self.email = email if email else ”
```

When the Python module is loaded, classes are scanned by the Python interpreter, and any executable code in the class definition is executed. This means that without us having to do anything, the instruction *grok.implements(IUser)* in the above class will execute, resulting in the *User* class being registered with the component framework as an implementor of the IUSER interface. Neat, huh?

So, what does this buy us?

[1]Not to muddy the waters here, but one may also have a Local component registry which stores site specific components.

We can say that the class User is a *factory* which produces instances of IUser. The ZCF way of saying this, is that the *class* User *implements* IUser, while the object *user=User()* *provides* the interface IUser. We can test this by:

```
>>> IUser.implementedBy(User)
True
>>> user = User()
>>> IUser.providedBy(user)
True
>>>IUser.implementedBy(user)
False
```

We can retrieve the attribute names defined by IUser by taking it's list:

```
>>> list(IUser)
['surname', 'name', 'email']
```

2.2 Runtime checking of interfaces

There is nothing in the ZCA to ensure that *User* actually does implement IUser. The ZCA is purely declarative, and if a class declares that it implements a given interface, the ZCA assumes that it correctly does so. If this is not the case in practice, errors may be generated at runtime where attributes or methods of the object might be accessed without having first been defined[2].

2.3 Adapters and Utilities

The ZCA provides a number of different types of registered components, but most of them can be boiled down to either *Utilities* or *Adapters*.

Utilities are simply registered factories. The *User* class shown above is an unregistered utility. To register the utility, we need to associate the *User* class with a registry which we can later query.

```
>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()
>>> gsm.registerUtility(user, IUser)
>>> from zope.component import getUtility
>>> user = getUtility(IUser)
>>> IUser.providedBy(user)
True
```

[2]With Zope schemas, a type of interface, validation of content for individual attributes may be enforced, but it is necessary to code specifically for that feature. Interfaces themselves are not enforced by either the language or the framework.

Grok provides another way of defining a global utility, by subclassing the *grok.GlobalUtility* class. For example:

```
>>> class MyUserUtility(grok.GlobalUtility):
...     grok.implements(IUser)
...     def __init__(self, name=None, surname=None, email=None):
...      self.name = name if name else ”
...      self.surname = surname if surname else ”
...      self.email = email if email else ”
```

Adapters are objects which are initialised from one interface but provide another, or return instances of a provided interface when given a different one in a __call__. This is incredibly useful, since we often need do nothing other than define an adapter in order to extend the framework in some way.

```
>>> class IAccount(Interface):
...     accountId = Attribute('accountId')
...     password  = Attribute('password')
...
...     def check_passwd(passwd):
...         ”' Checks password validity and returns boolean ”'
>>> class Account(object):
...     grok.implements(IAccount)
...     def __init__(self, user):
...         self.accountId = 0
...         self.password = ”
...     def check_password(self, passwd):
...         return hash.md5(passwd).text() == self.password
>>> class UserAccount(grok.Adapter):
...     ”' Find the account for a user ”'
...     grok.context(IUser)
...     grok.implements(IAccount)
...
...     def __new__(self, user):
...         return Account(user)
```

One may simply call an interface to retrieve an instance of the appropriate adapter:

```
>>> acct = IAccount(user)
>>> isinstance(acct, Account)
True
```

2.4 Multi Adapters

A multi adapter is similar to an adapter, except that it takes as argument more than just a single object. In other words, it provides an instance of an interface by consuming multiple other objects.

For example, assume we had some sort of persistent [3]store for accounts;

```
>>> class AccountStore(grok.Container):
...     ''' Stores user accounts '''
```

and we stored our account instance in it;

```
>>> store = AccountStore()
>>> store[user.name] = acct
```

We could define a multi adapter which returns our account when given a user and an instance of our account store;

```
>>> class UserAccountStore(grok.MultiAdapter):
...     ''' Retrieve a user account using the account store '''
...     grok.adapts(IUser, AccountStore)
...     grok.implements(IAccount)
...
...     def __new__(cls, user, accountStore):
...         if user.name in accountStore:
...             return accountStore[user.name]
```

A multi adapter cannot be called by simply calling the interface. Instead, we must use the *getMultiAdapter()* or *queryMultiAdapter()* functions as defined in the module *zope.component*. The difference between these functions is that on failure *getMultiAdapter()* generates an exception, while *queryMultiAdapter()* returns FALSE:

```
>>> from zope.component import getMultiAdapter
>>> acct = getMultiAdapter((user, store), IAccount)
>>> isinstance(acct, Account)
True
```

2.5 Events

There is a special kind of adapter in the ZCA, called a *subscription* adapter. These implement the well known listener pattern; one or more adapters are

[3]This is by example only- AccountStore is not persistent even though we are using a grok.Component. More on this later3.1.

registered for a specific interface, and may subsequently be iterated using the *zope.component.subscribers(InterfaceName)* method. To register a subscription adapter, one uses

```
>>> from zope.component import getGlobalSiteManager
>>> gsm = getGlobalSiteManager()
>>> gsm.registerSubscriptionAdapter(Adapter1)
>>> gsm.registerSubscriptionAdapter(Adapter2)
>>> ...
```

Handlers are similar to subscription adapters except that they do not provide any specific interface, but are Python callables. A callback function can be easily declared to be a handler:

```
>>> @zope.component.adapter(IMyCallbackInterface)
>>> def handler(event):
...     ''' handles events of type IMyCallbackInterface '''
```

Handlers can then be registered with the ZCA, and called using the *notify()* function:

```
>>> zope.component.provideHandler(handler)
>>> ob = ImplementerOfIMyCallbackInterface()
>>> zope.event.notify(ob)
```

With Grok, the functional equivalent of defining and registering the handler would be:

```
>>> @grok.subscribe(IMyCallbackInterface)
>>> def handler(event):
...     ''' handles events of type IMyCallbackInterface '''
```

We can then notify handlers of an event with

```
>>> grok.notify(event)
```

When one takes into account the flexibility of interfaces, including marker interfaces, it is easy to see how powerful this simple mechanism may become. When we later (**??** and **??**) discuss SQLAlchemy and the DSN configuration, we will show how the use of handlers can help us in practice.

3 Using components in a web framework

So, we have a way to declare components; how can we use that to make a web framework?

To do that we need to add a little persistence. There are a few ways of doing this, but one way is to use the Zope Database (ZODB).

3.1 Using ZODB

The ZODB is a hierarchical persistent object database. ZODB stores two kinds of Python objects, namely containers and models.

Containers (GROK.CONTAINER) are objects which contain other objects, and look and behave similar to a Python *dict*. Models (GROK.MODEL) are persistent objects which may be used to store data. There are refinements to these ideas, where for example an ORDEREDCONTAINER defines an *updateOrder()* method which may be used to change the order or stored items. The *persistent* module also defines PERSISTENTDICT and PERSISTENTLIST classes which may be used interchangeably with LIST and DICT types, but are persistently stored in the ZODB.

Rules for persistence To be persisted in the ZODB database, an object must:

1. Inherit from *persistent.Persistent* or one of it's subclasses
2. Objects must be related to each other in a hierarchical structure, and to the ZODB root.
3. Where mutable attributes (eg. dict, list) of an object are not persistent, the special *_p_changed* attribute for the object should be set to TRUE where such attributes have changed.

3.1.1 Making a web application based on ZODB

A *grok.Application* implements the IAPPLICATION interface, and is a class which holds the configuration and entry point in the traversal hierarchy for an application. Mixing this class with a *grok.Container* (an ICONTAINER) allows a simple way to further extend the traversal hierarchy.

```
>>> class MyApp(grok.Application, grok.Container):
...     ''' An application which can be registered with the ZMI '''
```

The *Zope Management Interface* (ZMI) is accessable directly after a default Grok installation, and is used to create and manage any instances of IApplication it may find through introspection. So, when you define a *grok.Application*, your application class is made known to the ZMI. The site administrator can then use the ZMI to create one or more persistent instances of your application entry point.

If your IApplication instance is also an IContainer, then adding objects to the container automatically extends the site hierarchy, and makes use of the ZODB as a persistent store.

3.2 Using SQLAlchemy

While ZODB is useful for many things, true extensibility and power offered by a relational database is available through the amazing *SQLAlchemy* object relational mapper (ORM).

An ORM is a time saver, cutting through reams of otherwise boilerplate code one would need to map your RDBMS in Python. While it is possible to declare and instantiate your database tables and structure in Python directly using SQLAlchemy, it is arguably better to manage the structure where it belongs-in the database itself. SQLAlchemy provides a way to derive your Python classes representing tables directly from the database through a process called *'reflection'*.

A module called *megrok.rdb* maps SQLAlchemy declarative table objects directly to container classes, making a RDBMS completely accessible and immediately usable in a Grok application.

3.2.1 A simple example, using megrok.rdb

The mapping of a *SQLAlchemy* configuration may look like a bit of black magic, but is really rather straightforward. The following maps a *Postgresql* database. An engine factory is a utility which produces a database engine instance given a dataset name (DSN). A scoped session factory is a utility which provides a session. The *metadata* is completed by the function *rdb.setupDatabase()*, which is called after the IEngineCreatedEvent is posted.

```
import grok
from megrok import rdb
from z3c.saconfig import (EngineFactory, GloballyScopedSession)
from z3c.saconfig.interfaces import (IEngineFactory,
                            IScopedSession, IEngineCreatedEvent)
DSN = "postgresql://user1@localhost:5432/firstdb"
#engine_factory = EngineFactory(DSN, echo=True)
engine_factory = EngineFactory(DSN, echo=False)
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_metadata = rdb.MetaData()
@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)
```

Aside from the above boilerplate configuration section, which need only be done once, some other things are needed to define and use models from a relational database. For example, to make use of *megrok.rdb*, we also need to define a few tables:

```
class platforms(rdb.Model):
    ''' This table lists the platforms supported by our products
    '''
    grok.implements(IPlatforms)
    rdb.metadata(metadata)
    rdb.reflected()
```

The *rdb.reflected()* directive tells *SQLAlchemy* to examine the existing database table and map the table attributes (columns) to class attributes for the defined class, including any primary key constraints or foreign key relationships.

Using *reflection* has many benefits, and possibly the greatest of which is the ease with which columns may be added to tables, or relationships changed without affecting existing code.

rdb.metadata(metadata) specifies which metadata subset to use for the database. There are two defined during configuration: one called *skip_create_metadata*, and the other called *create_metadata*. We point the variable *metadata* to either one or the other. The *create_metadata* may be used when one intends for a class declaration to create new tables rather than just referencing them.

We use *grok.implements(...)* to further specify the interface which this table class implements. The platforms class defined here may be imagined as a factory which produces objects which are the individual rows in the table.

3.2.2 Models and Containers;

Where an *rdb.Model* is the equivalent of the ZODB based *grok.Model*, the *rdb.QueryContainer* is the equivalent of a *grok.Container*. The *rdb.QueryContainer* presents itself as a dict, and this class may be used directly as a *traversable* object.

Aside from *rdb.QueryContainer*, there is also an *rdb.Container*, which may be used to represent collection classes for one to many relationships.

4 Model, View, and almost but not quite Controller

The process of *traversal* in Zope and Grok involves walking the hierarchical tree from the site *root* in order to identify an object. The names in the URL are extracted, and assumed to correspond to the keys of items in an ICONTAINER, or the names of attributes or methods marked as *traversable* in the class definition, or otherwise the keys defined inside a *grok.Traverser* subclass.

A *data object* identified through traversal is called a *Model context*.

In traversal, the last name identified may be the name of a *model context*, but sometimes may be the name of a *view*. If the last name in the URL identifies a *model*, the default *view* name is assumed to be *'index'*. Where a *model* is not the end result of traversal, the last name is assumed to be the name of a *view*.

Where the view name might potentially conflict with a model context name, Grok allows the view name to be prefixed with a double @ symbol to remove ambiguity. Eg. '*@@index*' is equivalent to the view name *'index'*.

A URL processed in this way through traversal is associated with a *'request'* object, being an instance of a *zope.publisher.interfaces.browser.IBrowserRequest* interface.

4.1 The result of traversal

So, traversal gives us a *Model* object, a *View* name and a *request* object. From this, Grok needs to produce a response to be sent back to a browser. While the *Model* provides the contextual data, the *View* provides the means transform the data to a visual representation of the data.

Views in Grok, are instances of an *Adapter* (see 2.3), or rather, instances of a *Multi Adapter* (2.4).

```
   >>> class Index(grok.View):
...     grok.context(MyApp)
...
...     def render(self):
...         return '<p>My App Index</p>'
```

Another way of specifying the above view, might have been:

```
   >>> from grokcore.view.interfaces import IGrokView
>>> from zope.publisher.browser import IBrowserRequest
        >>> class index(grok.MultiAdapter):
...     grok.adapts(MyApp, IBrowserRequest)
...     grok.implements(IGrokView)
...
...     def __call__(self):
...        args = get_args_from(self.request)
...        self.render(**args)
...
...     def render(self):
...         return '<p>My App Index</p>'
```

However, I am sure one would agree that the *grok.View* example is friendlier.

4.2 So where is the controller?

In the traditional definition of the *Model-View-Controller* (MVC) pattern, the *controller* is the only means to updating the *model.* The controller is normally an object manipulated by user interaction and applied to the model prior to updating the view.

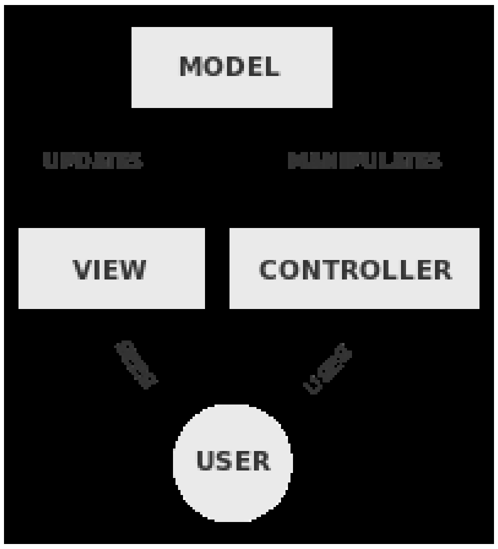


Figure 1: Traditional MVC Architecture (as per Wikipedia)

In a web application, the bit that updates the *model* would ideally be the *request*, or rather the data or logic which may accompany or result from the request. This data is often provided as a result of *user interaction* at the browser end, for example filling out a form, and and consequent issuance of a request.

In many other web frameworks, the *controller* has become an *abstraction* between the *view* and the *model*, which in effect repackages data from the model for easier dereferencing in the view implementation, and prevents the view from directly updating the model. This description does not gel with, and is not in agreement with the spirit and intention of the MVC pattern. It at best loosely fulfills the MVC requirement that only the controller may update the model, and does so at the expense of losing definition of precisely what the function of the controller is.

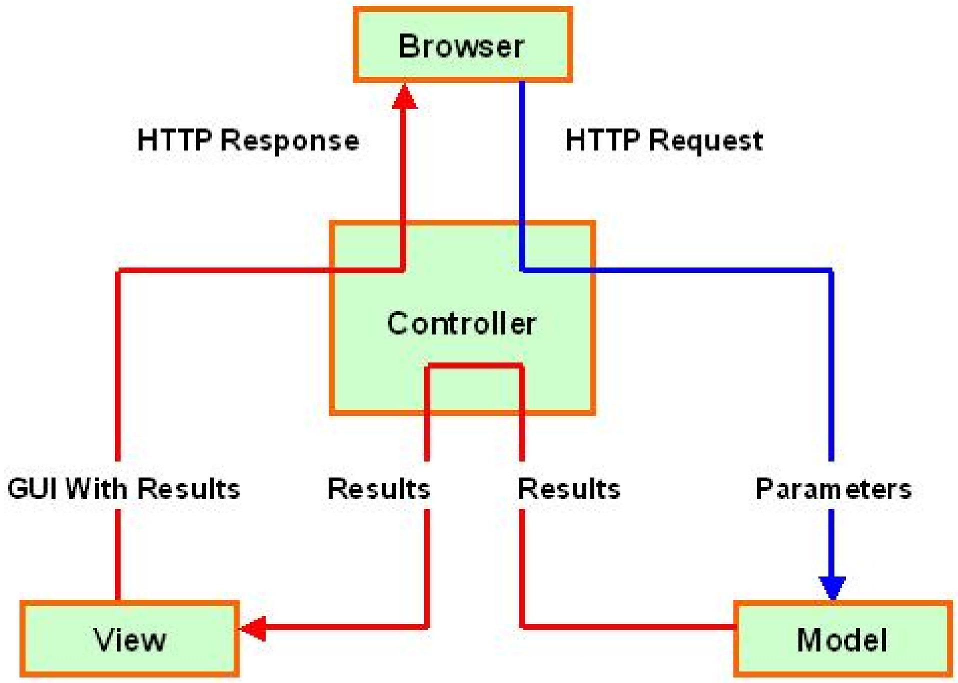


Figure 2: MVC as defined by the web

In *Grok*, the controller may be compared to the request itself, an instance of ZOPE.PUBLISHER.BROWSER.IBROWSERREQUEST as built by the Zope publisher. We update the model data by traversing the URL path found in the request, matching the path to the persistent *Model* (traversing ZODB or other), and finally calling the *update()* method defined by the associated *View* object for the identified *model context*. Any data contained in the request is passed as optional arguments to the *View*'s *update()* method. Having the data *context* immediately available, the *View* object is in an ideal position to perform the required alterations to the *Model* prior to rendering the visual representation of the data (i.e. the *view*).

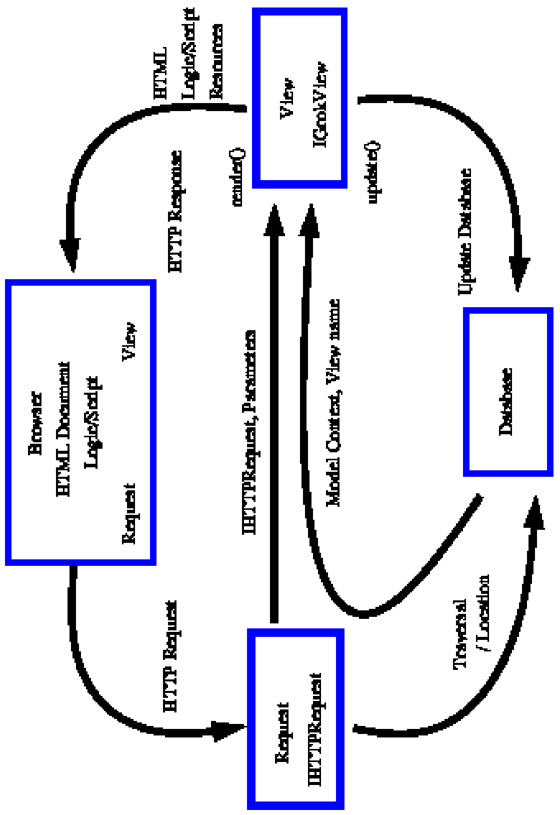


Figure 3: The Grok web model

4.2.1 So, is Grok MVC?

In a nutshell, no it is not. MVC requires that the logic for the view and that of the controller be implemented as separate objects. In Grok, the logic for both update and view resides in a single *View* object. Moreover, it is quite easily shown that even those web frameworks that indeed profess to be MVC based are only able to claim this by perverting the definition of MVC itself.

Grok is a *component based web framework*. It makes use of a component architecture for tasks such as data storage and retrieval, matching data components to view components, application design, representation and layout, permissioning, authentication, automatic form generation, and so on and so forth.

The reality is, that real MVC does not fit the web model very well, since in MVC the user is generally expected to "see" the *view*, and "manipulate" the *controller*, but the user of a web application does not in reality have access to these things if they are part of a web application hosted on a server far away.

There has been much written about MVC and it's benefits, but unfortunately each framework vendor understands the term differently, and about the only thing they seem to agree on is that the architecture should have a model, a view, and a controller.

In answer to the question "What then is MVC?", one might best answer simply: "MVC is a very popular architecture".

5 Schemas

A schema is a definition for the structure, types, names and description for data elements, including their scope. XML schemas and Data Type definitions (DTD's) define the structure of an XML document conforming to the schema. Relational database schemas define the structure, tables, columns and attribute types of elemenst within tables in a relational database. Zope schemas are a kind of *Interface* which defines the structure, types and descriptions for data elements in classes which would be stored in the ZODB.

Zope *schemas* may also be used to generate data entry forms automatically. Having the definition and description for class attributes, as well as their types, allows the association of HTML widgets with each attribute. Such *widgets* may be combined and presented as a data entry form.

Note that Zope schemas may be associated with any class, including SQLAlchemy mapped database classes and tables. Doing so allows Grok/Zope to automatically generate forms for viewing, editing or adding database records.

5.1 Fields in a schema

Casting back to the beginning of this document, we defined the interface for a user as:

```
from zope.interface import Interface, Attribute
class IUser(Interface):

     name = Attribute('name')
     surname = Attribute('surname')
     email = Attribute('email')
```

Schemas are a kind of interface, just like the above definition for an IUser, except that there is a whole lot more flexibility allowed for describing what the attributes actually mean. To turn the above interface into a schema based interface, with the immediate ability to generate an automatic data entry form for it, we could do the following:

```
from zope.interface import Interface
import zope.schema as schema
class IUser(Interface):

     name = schema.TextLine(title=u'First Name:',
                           description=u'A user name',
                           min_length=2, max_length=32,
                           readonly=False, required=True)
     surname = schema.TextLine(title=u'Last Name:',
                              description=u'Your last name or surname',
                              min_length=2, max_length=50,
                              readonly=False, required=True)
     email = schema.TextLine(title=u'Email:',
                            description=u'Your email address',
                            required=False)
```

Of course, there are many more types of fields than a simple *TextLine*, and one may even with a bit of work define one's own derived field types and associated widgets.

5.1.1 Validation

There are a few smart things that schema interfaces give us that we would not otherwise have had. For example, for classes which implement the schema, we can have automatic *validation*. To get this feature for all of our schema fields, we must define our attributes as instances of *FieldProperty*:

```
   >>> from zope.schema.fieldproperty import FieldProperty
>>> class User(object):
...     grok.implements(IUser)
...     name = FieldProperty(IUser['name'])
...     surname = FieldProperty(IUser['surname'])
...     email = FieldProperty(IUser['email'])
        ...
...     def __init__(self, name=None, surname=None, email=None):
...      self.name = name
...      self.surname = surname
...      self.email = email
>>> IUser.implementedBy(User)
True
>>> user = User(u'Bob', u'TheBuilder', u'bob@thebuilder.org')
>>> IUser.providedBy(user)
True
>>> try:
...     user = User('Bob')
... except Exception, e:
...   print 'Error:{}'.format(str(e))
Error:('Bob', <type 'unicode'>, 'name')
```

In other words, if we try to instantiate a User with anything other than a unicode name, it fails with an exception. It becomes impossible to assign attributes to such a class which fail to meet the requirements for the field.

The kind of exception generated is a *ValidationError*, which means it has a *doc()* method. In the case where we try to assign a name which is too short, we get the message: "*Value is too short*".

```
        >>> from zope.schema import ValidationError
>>> try:
...     user = User(u'B')
... except ValidationError, e:
...   print 'Error:{}'.format(e.doc())
Error:Value is too short
```

We can also do our own thing when it comes to validation. For example, if we wanted to check that the email address provided at least looks like one, we could do:

```
>>> from zope.schema import ValidationError
>>> class EmailValidationError(ValidationError):
```

```
...     ''' EMail address does not appear to be valid. '''
>>> def validEmail(email):
...  import re
# Note: this may fail for some valid email addresses!
...
if re.match(r"^[A-Za-z0-9\.\+_-]+@[A-Za-z0-9\._-]+\.[a-zA-Z]*$", email):
...  return True
...  raise EmailValidationError
```

Now we define a constraint for the email field in the IUser interface:

```
>>> class IUser(Interface):
          ... etc ...
...     email = schema.TextLine(title=u"Email:", required=True,
...                             constraint=validEmail,
...                             description=u'Your email address.')
```

Creating a user with a valid email address succeeds as normal:

```
>>> user = User(u'Bob', u'TheBuilder', u'bob@thebuilder.org')
>>> IUser.providedBy(user)
True
```

But trying to pass in an invalid email address will fail with an exception, as per our definition:

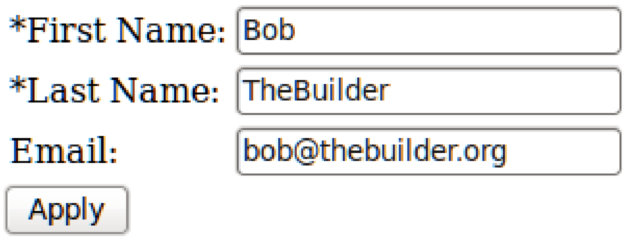
```
>>> try:
...     user = User(u'Bob', u'TheBuilder', u'bob.thebuilder.org')
... except ValidationError, e:
...  print 'Error:{}'.format(e.doc())
Error: EMail address does not appear to be valid.
```

5.1.2 Form generation

Of course, a really really smart thing we can do with Zope schemas is automatic form generation. The simple case is, as one might expect, simple:

```
class User(grok.Model):
    grok.implements(IUser)
    ....
class UserForm(grok.EditForm):
    grok.context(User)
```

First, we change the class *User* to inherit from *grok.Model* rather than *object*, then define a *UserForm* as a sub-class of *grok.EditForm*, and specify a *grok.context(User)* directive, which tells Grok to configure *UserForm* with the appropriate fields for the class *User*. Because the class *User* implements the *IUser* interface, the fields from *IUser* are interpreted to produce the form fields.



*First Name:	Bob
*Last Name:	TheBuilder
Email:	bob@thebuilder.org

Figure 4: An automatically generated edit form

Page templates When the HTML is produced for automatic forms, a default page template is used to render the output. Page templates in Grok are normally either *ZPT* (Zope Page Template) or *Chameleon* templates, but one is free to choose from a variety of other supported templating tools, eg *Mako*.

There is plenty of good documentation around for *ZPT/Chameleon*, so we won't revisit the syntax in detail here.

5.2 Reviewing Automatic Forms

So, how exactly did Grok generate that edit form from the schema? We already know about the page template which is associated automatically with a *grok.EditForm*. For reference, the default edit form template looks like this:

```
<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>
```

One can see where **<div class="widget" tal:content="structure widget">** replaces the div content with a widget, which was defined as a loop variable in the **<tal:block repeat="widget view/widgets">** statement, which loops over the widgets property of the view (or form). Where do these widgets come from?

Well, when Grok "*grokked*" the class *UserForm*, the data context class *User* was associated with the form, and since *User* implements the IUSER interface, the schema fields from IUSER could be associated with the *UserForm* view. This association is produced as an attribute of *UserForm* named *form_fields*.

The *form_fields* attribute can also be overridden directly, providing exceptional flexibility in deciding which fields should be used in the form, and how they should be displayed.

Producing a *widget* from a *form field*, which is derived from a *schema field*, is rather simple since we have a whole component framework at our disposal. A widget implements either a IDisplayWidget or a IInputWidget interface, and depending on whether the field is to be read-write or readonly, we find the widget by searching for it in the component framework:

```
widget = getMultiAdapter((Field, Request), IInputWidget)
```

Here, *Field* would refer to a schema field such as a *zope.schema.TextLine*, and the *Request* would be an IBrowserRequest.

An IInputWidget renders an HTML snippet which is normally an <input /> tag, although this is far from the only option. Widgets map schema fields to HTML, your data context to default form values and request data to the data context.

5.2.1 Other schema fields and default widgets

Apart from the *schema.TextLine*, we also have several other schema fields available, and if these are not enough, it is quite possible to define your own schema fields and asociated widgets, or to change the widget shown for a given field.

Table 1: Schema fields and default widgets

The field types defined in *zope.schema* are:

Field	Used for	Default Widget
Field	A generic schema field base class	None
SourceText	Preformatted text	<textarea />
Bytes	Raw data blob	<input type=”file” />
Text	Multi-line unicode text	<textarea />
ASCII	To contain multiline ASCII character data	<textarea />
BytesLine	Single line of raw character data	<input type=”text” />
TextLine	A single line of unicode text	<input type=”text” />
ASCIILine	Single line of ASCII character data	<input type=”text” />
Bool	A boolean value	<input type=”checkbox” />
Int	An integer value	<input type=”text” />
Password	A password field	<input type=”password” />
Float	Maps to a Python Float	<input type=”text” />
Decimal	Maps to the Python Decimal type	<input type=”text” />
Datetime	Contains datetime.datetime	<input type=”text” />
Date	Contains a datetime.date	<input type=”text” />
Timedelta	Contains a datetime.timedelta	<input type=”text” />
Time	Contains a datetime.time	<input type=”text” />
Choice	Choose from a range of values	<select />
InterfaceField	Maps to a zope interface	<input type=”text” />
Tuple	Maps to a Python Tuple	<select multiple />
List	Maps to a Python List	<select multiple />
Set	Maps to a Python Set	<select multiple />
FrozenSet	Maps to a Python FrozenSet	<select multiple />
Dict	Maps to a Python Dict	
Object	Maps to an arbitrary Python object	
URI	A Uniform Resouce Identifier	<input type=”text” />
Id	Maps to a Python ID	<input type=”text” />
DottedName	For example, a module name	<input type=”text” />

A rather complete list of widgets available for mapping to various fields may be found at: `http://bluebream.zope.org/doc/1.0/manual/schema.html`

5.2.2 Vocabularies and Choices

Imagine your goal is to produce a drop down list in a form allowing a user to select from a group of countries. Where does the list come from? Or, say, a list of fruits should be displayed and when the form is submitted we want the field value to be an index into the list of fruits. This is the province of the vocabulary.

A simple *Choice* field may be defined as follows:

```
gtype = Choice(title=u'Graph Type',
               values=['Scatter', 'XY', 'DateY'])
```

where this would be rendered as a <SELECT /> field returning a field value for one of the choices. In this case, no vocabulary is necessary; or rather, the *values* list is converted automatically into a *SimpleVocabulary* for you.

```
>>> from zope.publisher.browser import TestRequest
>>> from zope.formlib.widgets import SelectWidget
>>> from zope.schema import Choice
>>> gtype = Choice(__name__='gtype', title=u'Graph Type',
>>>                values=[u'Scatter', u'XY', u'DateY'])
>>> gtype = gtype.bind(object())
>>> request = TestRequest(form={'field.gtype':u'Scatter'})
>>> widget = SelectWidget(gtype, gtype.vocabulary, request)
>>> widget.hasInput()
True
>>> widget.getInputValue()
u'Scatter'
>>> print widget().replace(' ', '\n  ')
<div>
<div
  class="value">
<select
  id="field.gtype"
  name="field.gtype"
  size="5"
  >
<option
  selected="selected"
  value="Scatter">Scatter</option>
<option
  value="XY">XY</option>
<option
  value="DateY">DateY</option>
</select>
</div>
<input
  name="field.gtype-empty-marker"
  type="hidden"
  value="1"
  />
</div>
```

Lets take a closer look at the above code:

- In this simple example case, we first create a *Choice* schema field. If the schema field were defined as part of an interface, the __name__ argument would be automatic but in this demonstration we have to specify it. The *gtype.vocabulary* is created automatically when specifying a *values* argument to *Choice()*.
- We then *bind* the field, which associates a data object (or context) with the field, and allows the field to get or set data in the bound object. This is normally done automatically during rendering; the context is the same as the *form* context.
- The *TestRequest()* is an easy way for us to make an IBrowserRequest within the Grok/Zope test framework.
- We can now create a *SelectWidget* using the schema field and the test request. During form rendering, this would be done by the framework looking for a registered adapter that returns an IInputWidget for the field.
- The point of all this is to demonstrate what happens when the *SelectWidget* renders itself. The *SelectWidget* is just a *view* on a data context like any other view, and produces HTML.

Sometimes though, the choice may be a bit more complicated. A vocabulary is defined as a named Global Utility which implements the IVocabularyFactory interface. For example:

```
class Countries(grok.GlobalUtility):
    grok.implements(IVocabularyFactory)
    grok.name(u'list_countries')
    countries = [('au', 'Australia'),
                 ('uk', 'United Kingdom'),
                 ('us', 'United States of America'),
                 ....
                 ('za', 'South Africa')]
    def __call__(self, context):
        terms = [SimpleVocabulary.createTerm(value, token, title)
                 for value, (token, title) in enum(countries)]
        return SimpleVocabulary(terms)
```

A term has a *value*, a *token* and a *title*. The *title* is what is displayed to the user in an option. The *token* will be used as an option name in the HTML **<select><option name="token" /></select>** tag. The value is what is mapped as a Python field value when the form is submitted. In the above, we know the value will be an *integer* offset into the vocabulary list.

To use this vocabulary, one might define a choice field as:

```
country = Choice(title=u'Country:',
                 vocabulary=u"list_countries")
```

Take special note of the *context* argument to the method *__call__(self, context)*. This *context* will be the context of the *field*, which is the same as the context for the *view* (or *form*) which we specify. This means that we can respond with context sensitive vocabularies which may depend on a whole range of properties specific to the data being viewed.

It is very much worth the time spent to ensure a good understanding of vocabularies in Grok/Zope and how they are used.

5.2.3 The *Object* Schema Type, and Sequence Widgets

A rather interesting aspect to automatic form generation is that of how to handle lists (or collections) of objects. For example, how might one go about defining a tabular (or spreadsheet) view containing rows corresponding to table or query rows, and columns corresponding to table columns? How would one do this using automatic (CRUD) form generation in Grok?

Well, the good news is that yes, it can be done. The bad news is that no, it is not easy (unlike the case for some other frameworks), and the mitigating factor is that the code resulting from such an enterprise may be entirely re-usable.

An *object* schema field directly identifies a normal Python object which, if the object implements another schema interface may have a form rendered for the object itself. Almost like a form within a form. Some other schema fields, such as *set*, *frozenset* or *list*, are *collection* fields. The value type contained in such collections may be *simple* types such as strings or integers, but then again, they may also be *objects*. If the contained value type is an object, then a widget may be generated for it using an *ObjectWidget*, and sequences (sets, lists, etc) of these objects may be represented by a *SequenceWidget*.

A widget definition may be replaced in the special method *setUpWidgets* in a form class as follows:

```
class MyForm(grok.EditForm):

    grok.context(...)
    form_fields = ...
    def setUpWidgets(self, ignore_request=False):
        super(self, MyForm).setUpWidgets(ignore_request)
        widget = CustomWidgetFactory(ObjectWidget(MyObjectClass), ...)
        self.widgets['my_object_widget_name'].custom_widget = widget
```

The widget will be rendered as a sub form within it's own HTML <FIELDSET /> tag, and may be styled seperately. Arguments passed to *CustomWidgetFactory* are passed to the widget at creation time. For form fields which are sequences of objects, one might do:

```
class MyForm(grok.EditForm):
grok.context(...)

    form_fields = ...
    def setUpWidgets(self, ignore_request=False):
        super(self, MyForm).setUpWidgets(ignore_request)
        seq_widget = SequenceWidget(ObjectWidget(MyObjectClass), ...)
        widget = CustomWidgetFactory(seq_widget)
        self.widgets['my_object_widget_name'].custom_widget = widget
```

Sequence widgets are rendered in edit forms with controls for adding or deleting instances of the object to the collection.

6 The anatomy of a Grok web application

To recap, at this stage we should know about Zope *Interfaces*, and *Components* (2.1) which implement interfaces. We also should know about *Models*, being data components, and *Views* being visual components (4.1). We should also know about *schema* fields and how they map to *widgets* which are simple examples of *views*.

We have learned a great deal about how *forms* may be derived from *schema* interfaces and how *form fields* reference *schema fields* which are used to produce *widgets* during automatic form generation.

Although we have touched on *grok.Application*, *grok.Container* and *grok.Model* (3.1), we have not visited these in any real depth, and we have yet to discuss how to go about building a real web application. Regarding views, *grok.View* is still relatively unknown to us, and *grok.Viewlet* is still completely foreign.

There is a lot of tutorial and reference material available for Grok. Even given all of this material though, there is a wide gap between an introductory tutorial and what is involved in building a large site based upon Grok. We will try to breach this gap here.

6.1 The Core

At the heart of a Grok application, is a single class which serves as the entry point for the application. This class, called a *grok.Application*, implements an IApplication and ISite interface (see 3.1.1).

Now, one might expect a *grok.Application* to be a large class with many attributes and methods, and IApplication to specify many things that an application must do to comply. Instead, we find IApplication to be defined as follows:

```
class IApplication(Interface):
    """Interface to mark the local site used as application root.
    """
```

Yes, that's it. Nothing else.

The definition of a *grok.Application* component is not much better:

```
class Application(Site):
    """Mixin for creating Grok application objects.
    When a :class:‘grokcore.content.Container‘ (or a
    :class:‘grokcore.content.Model‘, though most developers
    use containers) also inherits from :class:‘grokcore.site.Application‘,
    it not only gains the component registration abilities of a
    :class:‘grokcore.site.site‘, but will also be listed in the
    Grok admin control panel as one of the applications
    that the admin can install directly at the root of their Zope
    database.
    """
    implements(IApplication)
```

So, at this point we start getting suspicious. Someone's having us on, right? Lets go look at *grok.Site* then, which a *grok.Application* inherits.

```
class BaseSite(object):
    """Mixin to grok sites in Grok applications.
    It’s used to let different implementation of sites to exists, and
    still being grokked correctly.
    """
class Site(BaseSite, SiteManagerContainer):
    """Mixin for creating sites in Grok applications.
    When an application :class:‘grok.Model‘ or :class:‘grok.Container‘
    also inherits from :class:‘grokcore.site.Site‘, then it can
    additionally support the registration of local Component
    Architecture entities like :class:‘grokcore.site.LocalUtility‘ and
    :class:‘grok.Indexes‘ objects; see those classes for more
    information.
    """
```

Isn't this frustrating? It really looks like there is nothing much there. Quite a bit of documentation, explaining what these classes are supposed to accomplish. A *SiteManagerContainer* looks promising, and by the name seems to be a component which the site manager might use to contain instances of your site. Let us see. *SiteManagerContainer* is imported from *zope.site.site*, and is defined as:

```
class SiteManagerContainer(Contained):
    """Implement access to the site manager (++etc++site).
    This is a mix-in that implements the IPossibleSite
    interface; for example, it is used by the Folder implementation.
    """
    zope.interface.implements(zope.component.interfaces.IPossibleSite)
    _sm = None
    def getSiteManager(self):
        if self._sm is not None:
            return self._sm
        else:
            raise ComponentLookupError('no site manager defined')
    def setSiteManager(self, sm):
        if zope.component.interfaces.ISite.providedBy(self):
            raise TypeError("Already a site")
        if zope.component.interfaces.IComponentLookup.providedBy(sm):
            self._sm = sm
        else:
            raise ValueError('setSiteManager requires an IComponentLookup')
        zope.interface.directlyProvides(
            self, zope.component.interfaces.ISite,
            zope.interface.directlyProvidedBy(self))
        zope.event.notify(interfaces.NewLocalSite(sm))
```

Ok, since we are now getting into the code for the Zope Framework, let us stop right there. What have we learned by this?

By inheriting from *grok.Application*, your application will tie itself into the Zope site manager, and thus make itself known as an IPossibleSite, so that the site manager can create instances of it. We also know (as per inline documentation) that a *grok.Application* inherits from a *grok.Site*, which apparently provides us with the ability to register local component infrastructure like *LocalUtility* and a site index.

A *LocalUtility* is a ZCA utility just like any other except that it's scope is specific to the current application. This type of utility is used to implement features such as authentication.

6.1.1 Surprisingly lightweight

If you are wondering where the beef behind the application is hidden, you are in for a disappointment. The reason why there does not appear to be much there, is that ... there is not much there!

The core functionality of a Grok or Zope application lives within components registered with the application framework. This means that the detail is implemented where it needs to be, and not as part of some massive monolithic framework.

6.1.2 Zope vs CGI

CGI (Common Gateway Interface) is the first basic web protocol. The web server (eg. Apache) would translate the URL and identify a resource (like an HTML file) and return it as a HTTP response. CGI allowed for executing server side scripts- literally anything such as bash, perl, tcl, awk executable code, passing request data as script arguments or through the app environment. The script would then return valid HTML which got packaged as a response.

The problems with CGI involved overheads which would be repeated with every request- opening/closing databases, searching file systems, startup and teardown code. This led to CGI being a rather inefficient protocol.

Application servers such as Zope provide a partial answer to the inefficiencies inherent in a stateless request/response protocol such as HTTP.

For example, it is unnecessary to repeatedly open and close a database with Zope (and other application servers) with each request. There are some things though, which still need to happen repeatedly, since HTTP is, after all, stateless.

6.1.3 The Z-Object Publishing Environment

The concept behind Zope, is to publish objects with which users may interact. Views of those objects would be rendered as HTML and returned as an HTTP response.

The Zope publisher is responsible for a variety of things:

- Build the request (IBrowserRequest) object from the HTTP request
- Populate request data, decoding, naming or converting as necessary
- Resolve URL path and identifying *Model* and *View* objects (traversal)
- User authentication, and session management

- Call the *View*, passing the request parameters
- Package the response into a HTTP response, handling any exceptions raised

The *traversal* step is generally where web coders generally start taking issue with Zope, even though traversal is almost identical to the way CGI would identify file system resources or scripts.

The reason for this, is that many modern web frameworks do not assume a direct correspondence between resource location and site hierarchy. Instead, a separate configuration (*routing*) exists whereby URL paths are mapped to resources (models) and views. Web coders are familiar and content with the configuration based approach to resource location, since this gives a great deal of flexibility - eg. dropping views in at arbitrary locations.

In the case of Zope/Grok and the use of the hierarchical ZODB as a data store, it makes little sense to divorce site URL path map from resource location. It actually simplifies things to combine these concepts and removes the need for maintaining a seperate site configuration map.

For those who insist on routing for URL dispatch, there is support for the *Traject* library (The Traject Library) in the module *megrok.traject*.

6.2 The Site, Pages and Views

There are a few concepts around Grok which take a bit of getting used to. A rather big part of it is the use of adaptive views.

Typically, web sites need to be *consistent*. It is seldom that a web site will use completely different fonts, layout, navigation or colours on each different page produced for the site. This means that for a given site, much of the HTML code will remain the same. Resources loaded such as javascript, images and CSS are often (but not always) identical between pages of a site.

One approach to building a Zope/Grok web site is to use Page Template *Macros*. A macro allows one to define a *skeleton* page consisting of a bunch of slots to fill. Each slot has it's own name. Depending on the *context* identified by *traversal*, a view can specify how each of the slots should be filled.

Another way to build a Grok site, is to make use of *Viewlets*. A view for a *context* is defined to source content snippets from numerous "content providers", otherwise known as "*Viewlet managers*". One or more *Viewlets* are defined to provide content snippets to these viewlet managers, depending on the context and view.

Regardless of approach to building a skeleton site layout, whether using *Macros* or *Viewlets*, a *View* (or page) is filled in piecemeal so as to reduce the amount of duplication required to produce a consistent and maintainable multi-page web site.

6.2.1 Sharing a skeleton view between contexts

Defining an application and a default view is rather trivial:

```
class app(grok.Application, grok.Container):

    ''' My application '''

class Index(grok.View):

    ''' My application view '''
    grok.context(app)
```

Now, defining a page template for our index view would involve creating a file called '*index.pt*' in the directory *app_templates* (*app* is the Python module name used as a prefix):

```
<html>

    <head></head>
    <body>

        <p> some stuff here </p>

    </body>

</html>
```

When *traversal* provides us with an instance of *app*, our default view named *index* will render the '*index.pt*' page template.

Adding other models to our site is equally trivial. Let us add two new model classes A, and B, and add these to our application:

```
class A(grok.Model):

    ''' a class "A" model '''

class B(grok.Model):

    ''' a class "B" model '''

class app(grok.Application, grok.Container):
    ''' My application '''
    def __init__(self):
        self['a'] = A()
        self['b'] = B()

...
```

Traversal of */app* provides an instance of class app(), */app/a* gives us an instance of A, and */app/b* provides us with an instance of B.

Now the question is: How can we ensure that *A*, *B* and *app* all share exactly the same index view? Remember that it does not really help us to have context sensitive *Viewlets* if we end up having to use different *View* definitions for each.

This question is one obstensibly not answered in the many tutorials for Grok, and the answer is itself not immediately apparent unless one has already used Zope/Grok extensively.

The answer is, of course, to use the component framework. First of all, we need to define what is called a "*Marker*" interface. This is an interface without a body, but which will be used to mark those models for which we want to produce the default index page. Then, instead of *grok.context(app)*, we tell our view that it's context is the marker interface:

```
class ISitePage(Interface):
    ''' A marker for models that get the
        default site page '''
class A(grok.Model):
    ''' a class "A" model '''
    grok.implements(ISitePage)
class B(grok.Model):
    ''' a class "B" model '''
    grok.implements(ISitePage)
class app(grok.Application, grok.Container):
    ''' My application '''
    grok.implements(ISitePage)
    def __init__(self):
        self['a'] = A()
        self['b'] = B()
class Index(grok.View):
    ''' My application view '''
    grok.context(ISitePage)
```

Now, whenever traversal lands us on a model that implements ISitePage, that model will also inherit the default index view!

6.2.2 Viewlets

Once we have understood how marker interfaces can gain us re-use of a view for different URL's, the world of content providers and *Viewlets* start opening up and making sense.

A *content provider* can be seen as similar to filling page template macro slot. In the page template, one refers to such content with tal:content="structure provider:my_provider". For example (our *index.pt* view page template shown previously):

```
<html>
<head></head>

    <body>

        <h1 tal:content="structure provider:my_provider"> My header </h1>
        <p> some stuff here </p>
    </body>
</html>
```

In the source, one would define the content provider as a viewlet manager:

```
class my_provider(grok.ViewletManager):

    grok.context(ISitePage)
```

Once declared, the viewlet manager will work, but will not return any content until you define one or more viewlets. *Viewlets* are like views in some ways, and can have their own page templates too. They can refer to the view within which they are embedded by addressing the special *view* attribute. They are also multi adapters just like views, but in the case of a *Viewlet*, they adapt the *context*, *provider* and *request*:

```
class SiteHeaderViewlet(grok.Viewlet):

    grok.context(app)
    grok.provider(my_provider)
    grok.view(Index)
    grok.name('header')
    def render(self):
        return "Main app header"
```

```
application = grok.getApplication()
view = getMultiAdapter((context, request), name='index')
prov = getMultiAdapter((context, request, view), name='my_provider')
viewlet = getMultiAdapter((application, prov, request), name='header')
```

The above shows the relationships between *views*, *providers* and *viewlets*.

The properties of viewlets are:

- They normally contain snippets of HTML rather than whole pages.
- May be specific to a given view by name. *grok.view(Index)* limits the viewlet to just the *Index* view.
- More than one viewlet may be defined for the same provider (slot), eg. menu items or tabbed interfaces.
- May be controlled by context. Defining *grok.context(ISitePage)* instead of *grok.context(app)* in our example would make the viewlet render for all pages rather than just the main *app*.

6.3 Exploring Traversal

One of the most criticised aspects to Zop/Grok in recent times is the way *traversal* is used to identify models and views. Since traversal is directly related to the ZODB, which is a hierarchically organised persistent object database, the main complaint is that "*not everything can be coorced into a hierarchy*", which of course, is completely true.

The very complaint though speaks volumes about how misunderstood traversal actually is.

6.3.1 The case for traversal

Thinking about it, something missed in todays information-centric world is the maxim:

A model is not always data, but may instead represent data.

In other words, what should be made very clear is that

T*raversal is more about identifying models than identifying data.*

What is indisputable, is the fact that hierarchies often do model real world scenarios; a book consisting of parts consisting of chapters; a city consisting of municipal areas consisting of suburbs; A taxonomy eg. a *domain* -> *kingdom* -> *phylum* -> *class* -> *order* -> *family* -> *genus* -> *species*. Hierarchies are undebatably one of the better ways of organising masses of information.

So, in our taxonomy shown, is the *kingdom* data?

Not at all: *kingdom* merely refers to the class of data, or kind of data, and is not data in and of itself. The animal *kingdom* consists of a great many animals, not all of which may be represented in your data. Relationships between such data classes may be defined as related tables in a relational database, or as models in a ZODB having parent-child relationships.

With ZODB, classes are models, while class instances (objects) are data. An instance of *kingdom* is *animal*. Similarly in a relational database, the table (the container) is a model, but individual rows in the table are data. The methods associated with a ZODB model represent logic in the same way which stored functions might do the same for an RDBMS.

Since a model class which results from traversal represents data rather than containing it, one might as easily use models based upon an RDBMS as one would where the data is stored in the ZODB, and it is often a good choice to do so.

While it is true that not everything fits a hierarchy, especially not data, one thing that absolutely does fit a hierarchy is a web site. In other words, it is always possible to map out your site as a set of URL's in a rather strict hierarchical manner.

The gains in performance offered by traversing the ZODB from a given root is well worth the effort in properly planning a site map.

6.3.2 Transitory objects

A transitory object is one which is not persistent, and who's life span is limited to the processing of a single request. Such objects may nevertheless form the basis of a view, and if location information is provided to the object, Grok views can generate URL's to such objects.

6.3.3 Session Management

When dealing with data in the web model, there is first persistent data, which is permanently stored for as long as it remains relevant. Then there is transitory *data* which might be passed to and fro between a browser and the server, but only lives in one or the other at a time. A server may store long-lived information in browsers as cookies, and retrieved whenever it needs to. Finally, there is the session.

A browser session is created whenever a browser makes a request to the server. If there are no further requests, the session expires and any data associated with the session is lost. We can store information in the session between requests for as long as 20 minutes after the last browser interaction. Grok/Zope automatically provides such session containers, which behave like a dict, and can be retrieved by calling *ISession(grok.BrowserRequest).*

7 Putting it together

We have seen how Grok advertises a site based upon an instance of a *grok.Application* [section 3.1.1], how the Zope *publisher* uses *traversal* [sections §**??**, §**??** and §**??**] to deliver up instances of *models*, *containers* and *view* names, and how we may use the *Zope Component Architecture* (ZCA) [] to register *views*, *viewlets* and *viewlet managers* to build site components in a reusable manner.

The way *Zope* schemas [section §5] are a natural extension to interfaces, and how these may be used for field validation[section 5.1.1], or the production of automatic forms has been discussed in some detail [section 5.1.2]. We have explained the way schema fields relate to widgets, and how widgets are treated as views by the architecture [section 5.2], and the flexibility that this brings to web page design.

Even though complex sites may be build using this architecture, we have also shown that the architecture itself is reasonably lightweight.

The next section of this article describes Grok in greater detail by implementing and discussing an example of a non-trivial application.

7.1 A book library

Our example application will implement a *book library* from ground up, including the storage and searching by field (eg *title, author*) for books in the library. We will discuss the code every step of the way.

7.2 Getting started: *interfaces.py*

The conventional name for the Python module in which we define our interfaces and schemas, is *interfaces.py*. This module generally does not need to include much, but will itself be imported into just about any other module.

In the design of our library, we will specify that a library is a collection of books. Books have many fields such as *title isbn* and *subject*, but a special case is a *list* of *authors*, not just one. A book also has a *publisher* and *publication date*, and we would like a place where we could write some *notes*.

```
class IBook(Interface):
    ''' This defines the fields we store for a book '''
    title = schema.TextLine(title=u'Title:', description=u'This book title')
    authors = schema.List(title=u'Authors:', description=u'Authors for this book',
                          value_type=schema.Choice(title=u'Author:', description=u'An author',
                                                   vocabulary=u'BookAuthors'),
                          default=[])
    subject = schema.TextLine(title=u'Subject:', description=u'The subject of this book')
    publisher = schema.TextLine(title=u'Publisher:', description=u'The name of the publisher')
    pub_date = schema.Date(title=u'Publication date:', description=u'The date of publication')
    isbn = schema.TextLine(title=u'ISBN:', description=u'ISBN Number', constraint=check_isbn,
                           required=False)
    notes = schema.Text(title=u'Notes:', description=u'General notes',
                        required=False)
    def authorStrings(): #@NoSelf
        ''' The concatinated list of authors '''
```

The method a*uthorList()* is defined with no *self* parameter since we are only creating an *interface* here, and not an implementation.

Zope insists on the use of unicode strings when defining interface names. It is an error to omit the leading *'u'* for strings.

Note that the authors field is a list of *Choices*, where the *Choice* field uses a vocabulary called BookAuthors. The vocabulary will be used to populate the field with the available values. More about this later.

Take a look at the isbn field; it specifies a *constraint=check_isbn*. The *check_isbn* function raises a ValidationError if the field does not match a valid ISBN:

```
class ISBNInvalid(schema.ValidationError):
    ''' This does not appear to be a valid ISBN number. '''
def check_isbn(isbn):
    import re
    if re.match(r"^[-0-9xX ]{13}$)(?:[0-9]+[- ]){3}[0-9]*[xX0-9]$", isbn):
            return True
        raise(ISBNInvalid)
```

The next schema class we want to define, is an IAuthor. This one is a lot simpler:

```
class IAuthor(Interface):
    ''' Authors implement this interface '''
    name = schema.TextLine(title=u'Name:', description=u'The name of this author')
    born = schema.Date(title=u'Date of birth:', description=u"This author's birth date",
                       required=False)
    died = schema.Date(title=u'Date of death:', description=u"If deceased, the date",
                       required=False)
    country = schema.Choice(title=u'Country of origin:',
                            description=u"Where is this author from?",
                            vocabulary=u'Countries',
                            required=False)
```

We again use a *Choice* in this interface, this time for the *country.* The vocabulary name for the *country* field is "Countries".

Finally, we would like to be able to search our ZODB collections to locate books (and authors) since we can expect our collection to grow rather large. To this end we define an ISearch interface which we will use for identifying searches and defining an input field for a search string:

```
class ISearch(Interface):
    ''' A simple search box '''
    search = schema.TextLine(title=u'Search:', description=u'A search field')
    def do_search(self, **kw):
        ''' Searches for specified field values '''
    def results(self):
        ''' Returns the results of the search '''
```

We will cover the use of the *ISearch* interface in great detail later.

7.2.1 Marker interfaces

In section 6.2.1 we discussed the use of marker interfaces to share view skeletons for a site. For our new book library, we define a marker interface called ISiteIndex which we will use whenever we want a *grok.Container* or *grok.Model* to render the site:

```
class ISiteIndex(Interface):
    ''' Marker for models which render the site index '''
```

7.3 The site layout; *layout.py*

Out site will be laid out with three areas; a *title* area above everything, a *left navigation* area and a *content* area. We will want to render the *index* page for every instance of a ISiteIndex implementor.

For each of the aforementioned areas, we define a viewlet manager which, one will recall, defines a slot which we can fill later with some content:

```
class HeaderMgr(grok.ViewletManager):
    ''' Slot for section headers '''
    grok.context(ISiteIndex)
class ContentMgr(grok.ViewletManager):
    ''' Slot for section content '''
    grok.context(ISiteIndex)
class LeftNavMgr(grok.ViewletManager):
    ''' Slot for left navigation bar '''
    grok.context(ISiteIndex)
```

Notice that all three of these viewlet managers define a context of an *ISiteIndex*, which means that they will render only for objects which provide that interface.

The index view is defined as a class with a related page template. The class definition also specifies that it will render for an *ISiteIndex* provider:

```
class Index(grok.View):
    ''' The site index '''
    grok.context(ISiteIndex)
    deleting = False
```

The *deleting* attribute is used later to switch between choices of edit and delete views. Our page template for the site (*layout_templates/index.pt*) is rather simple. It contains all the usual boilerplate stuff:

```
<!doctype html >
<html lang="en-US" style="height: 100%" itemtype="http://schema.org/WebPage"
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">
<head>
<title tal:content="context/site_title | nothing">Publication Archiving Tool</title>
<meta charset="utf-8" />
<meta content="width=device-width" name="viewport" />
<meta content="width=device-width, initial-scale=1" name="viewport" />
<!--link rel="shortcut icon" tal:attributes="href static/favicon.ico" type="image/x-icon" /-->
<link rel="stylesheet" tal:attributes="href static/style.css" type="text/css" />
```

```
<link rel="stylesheet" tal:attributes="href static/prettytable.css" type="text/css" />
</head>
<body>
<div id='leftCol'>
        <div tal:content="structure provider:leftnavmgr" />
</div>
<div id="topDiv">
<h1 tal:content="structure provider:headermgr" />
</div>
<div id="navDiv">
</div>
<div class='contentFrame'>
<div>
<div class="content" tal:content="structure provider:contentmgr" />
</div>
</div>
</body>
</html>
```

7.3.1 The Zope Page Template syntax

Note how there are several attributes in the otherwise normal XHTML source, which have a namespace starting with TAL: These attributes are *template* tags which allow one to fill in the content from the *view*. To avoid getting bogged down in detail here, we will simply say that good references are already available for the *Zope Page Template* (ZPT) language.

A brief cheat sheet though:

- *tal:content="..."* replaces the content of the tag with what is specified,
- *tal:replace="..."* replaces the whole tag with the content,
- *tal:repeat="varname iterable"* loops the current tag over the iterable using the varname,
- *tal:condition=""* renders depending on the condition and
- *tal:attributes="attr val; ..."* defines HTML attributes.
- *tal:define="varname value; ..."* defines variables scoped inside the current tag.
- tal:omit-tag="" omits the current tag from the HTML.

- *<tal:loop="varname iterable">body</tal:loop>* loops the body over the iterable using the varname.

The template for *index.pt* shows some of these in use.

When using the syntax

tal:content="..." or *tal:replace="..."*,

the content would normally be structured for display "*as is*". So for example, if the content was HTML, the left angle bracket would be replaced with &LT; etc. To prevent this, and render the *structure* instead, one uses:

tal:content="structure ...".

One may use standard python in the body of the template by prefixing with the name python: eg.

*tal:content="python: 'answer={}'.format(12*12)"*

In the *tal:content="structure provider:..."*, the phrase *"provider:..."* is replaced with the content of the viewlet manager (i.e. content provider) by that name.

7.3.2 Static Resources (*resources.py*)

Our static resources are dereferenced with the variable *'static'* for example:

```
<link rel="stylesheet" tal:attributes="href static/style.css" type="text/css" />
```

This is not completely provided by Grok itself, but rather provided due to our use of the *fanstatic* (not *fantastic*) library. The content of the *resources.py* is simply:

```
from fanstatic import Library, Resource
library = Library('camibox.pat', 'static')
```

This file is referenced from the main Grok installation setup.py, where the entry points for the static resource directories are specified:

```
entry_points={
        'fanstatic.libraries': [
            'pat          = camibox.pat.resource:library',
            ...
        ]
}
```

By this mechanism, the name for the resource directory 'static' is made available to the template.

Not used in this example project, but fanstatic resources are very flexible indeed. To include *jquery* in your site, one might do something similar to the following (in *jquery.py*):

```
from fanstatic import Library, Resource
library      = Library('cbjquery', 'jquery')
smoothness   = Resource(library, '1.7.2/css/smoothness/jquery-ui-1.8.21.custom.css')
jquery       = Resource(library, '1.7.2/js/jquery-1.7.2.min.js')
jqueryui     = Resource(library, '1.7.2/js/jquery-ui-1.8.21.custom.min.js',
                        depends=[jquery, smoothness])
```

Telling your view or viewlet to use the library would be done like this:

```
from jquery import jqueryui
class MyView(grok.View):

    ...
    def update(self):
        jqueryui.need()
```

After defining an entry point for *jquery.py*, this would automaticaly add the required links to your HTML page to import the javascript and css for *jquery*, the *theme* and the *jqueryui* library.

7.4 The Countries vocabulary (*countries.py*)

When presenting fields to be completed for an *author*, the *country* field is validated against the *Countries* vocabulary as specified in the *IAuthor* interface. When providing a *select* widget as produced from the *Choice* field, the options are produced from the *Countries* vocabulary.

A vocabulary in Zope/Grok is defined as a global utility, as described in section 5.2.2 on page 26. In our app, we define CountryVocabulary as follows:

```
import grok
from zope.schema.interfaces import IVocabularyFactory
from zope.schema.vocabulary import SimpleVocabulary
g_countries = [
{'timezones': ['Europe/Andorra'],
'code': 'AD', 'continent': 'Europe', 'name': 'Andorra',
'capital': 'Andorra la Vella'},
...
{'timezones': ['Asia/Dubai'],
'code': 'AE', 'continent': 'Asia', 'name': 'United Arab Emirates',
'capital': 'Abu Dhabi'},
{'timezones': ['Europe/London'],
'code': 'GB', 'continent': 'Europe', 'name': 'United Kingdom',
'capital': 'London'},
]
class CountryVocabulary(grok.GlobalUtility):
    grok.implements(IVocabularyFactory)
    grok.name(u'Countries')
    def __call__(self, context):
        terms = [SimpleVocabulary.createTerm(c['code'], c['code'], c['name'])
                 for c in g_countries]
        terms.sort(key=lambda term: term.title)
        return SimpleVocabulary(terms)
```

The country list is quite easily sourced from the web.

7.5 The Dashboard (*pat.py*)

When defining a site in Grok, we do so by subclassing from the class grok.Application. Often a grok.Application is also a grok.Container. In the case of our example app, we display a dashboard with a summary of the state of the library as an intro page, and allow navigation to either the authors maintenance page or the library itself.

The definition for the main application is straightforward:

```
class PAT(grok.Application, grok.Container):
    ''' Putting it All Together: A book library'''
    grok.implements(ISiteIndex, ISiteRoot)
    site_title = u'Publication Archiving Tool'
    def __init__(self):
        super(PAT, self).__init__()
        self['authors'] = Authors()
        self['library'] = Library()
    def authCount(self):
        return len(self['authors'])
    def bookCount(self):
        return len(self['library'])
```

Since PAT implements an ISiteIndex, we get the main index.pt rendering, which in turn expects to fill slots for the title, the navigation and content areas. So we add viewlets for each:

```
class Header(grok.Viewlet):
    ''' Fills the slot for the dashboard header '''
    grok.viewletmanager(HeaderMgr)
    grok.context(PAT)
class Content(grok.Viewlet):
    ''' Fills the dashboard content slot '''
    grok.viewletmanager(ContentMgr)
    grok.context(PAT)
class LeftNav(grok.Viewlet):
    ''' Fills the left navigation slot '''
    grok.viewletmanager(LeftNavMgr)
    grok.context(PAT)
```

The names of the corresponding page templates are just the lowercase of the class names. For now, we do not provide anything for the navigation slot.

7.5.1 The header (*pat_ templates/header.pt*)

```
<h1>Book Library: Dashboard</h1>
```

7.5.2 The content (*pat_ templates/content.pt*)

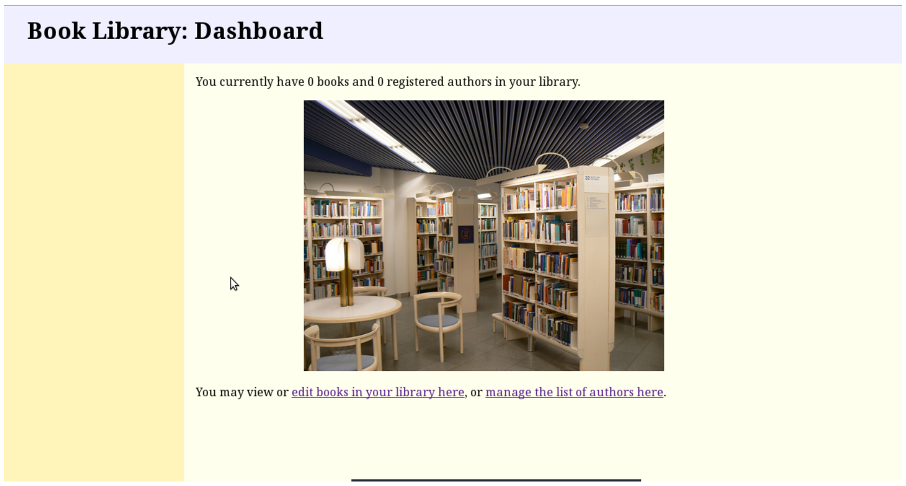
```
<p>You currently have <span tal:content="context/bookCount" /> books
and <span tal:content="context/authCount" /> registered authors in your library.</p>
<div style='max-width:800px; margin:auto'>
<img tal:attributes="width string:500px; src static/shelves.jpeg; alt string:Shelves" />
</div>
<p>You may view or <a href="library">edit books in your library here</a>,
    or <a href="authors">manage the list of authors here</a>.</p>
```

The template should be completely self explanitory other than for the special variable *context*. This refers to the view context, and we know that this provides the methods *bookCount()* and *authCount()*.

With a bit of CSS formatting, our index view for the dashboard renders like figure 5:

7.6 Authors

We come to the first of two relatively complex modules. Complex because this module defines the model for an *author*, the *authors* collection, a catalogue for the books in the authors collection, edit and deletion forms and various viewlets for plugging in content for the views. The other complex module will be the library itself, and by the time we get to it, much about that module will already have been explained.



Book Library: Dashboard

You currently have 0 books and 0 registered authors in your library.

You may view or edit books in your library here, or manage the list of authors here.

Figure 5: The Dashboard view

7.6.1 The *AuthorVocabulary*

We expect *AuthorVocabulary* to return a list of authors as we have defined them. This list is available from an instance of *Library*, which defines a method *authorList()* which returns them. We can define *AuthorVocabulary* as follows:

```
class AuthorVocabulary(grok.GlobalUtility):
    ''' A vocabulary of all the authors we know about '''
    grok.implements(IVocabularyFactory)
    grok.name(u'BookAuthors')
    def __call__(self, _context):
        terms = []
        app = grok.getApplication()
        authors = app['authors']
        for author in authors.values():
            title = author.mkTitle()
            token = author.key()
            terms.append(
                SimpleVocabulary.createTerm(author, token, title))
        return SimpleVocabulary(terms)
```

The phrase *grok.getApplication()* retrieves a reference to the nearest application, in this case an object of class *PAT*. We can get the authors list from there.

7.6.2 The Author Model

An instance of an author is stored in the ZODB within the *Authors* container (section 7.6.3). We first decare *Author* to be an implementor of the IAuthor interface, and also ISiteIndex.

```
class Author(grok.Model):
    ''' An author implements the IAuthor interface '''
    grok.implements(IAuthor, ISiteIndex)
...
```

The first provides us with the schema field definitions, and the second with an inherited index view.

For each field in IAuthor, we define a *FieldProperty* attribute. These attributes ensure that assignment will validate the field values whenever they change:

```
name    = FieldProperty(IAuthor['name'])
born    = FieldProperty(IAuthor['born'])
died    = FieldProperty(IAuthor['died'])
country = FieldProperty(IAuthor['country'])
```

If validation fails, we are informed by means of a *schema.ValidationError* exception.

None of the fields in the author model are safe to use as keys. An author name is not a unique identifier. Neither are any of the rest really. Perhaps the name and year of birth might approach uniqueness. However, since we do not want to insist on a birth date for an author, it is best that we use a sequence. For purposes of reference it is also convenient to store the key in the object itself.

```
    _key = None
    def key(self, val=None):
        if val: self._key = int(val)
        if self._key: return str(self._key)
```

This method may be used to update or retrieve the key value. The sequence itself will be discussed when we review the *Authors* container (section 7.6.3).

Our initialiser provides us with the ability to directly assign (and validate) all fields upon iniialisation. The importance of this in relation to the correct operation of the Zope Catalog will be discussed later.

```
    def __init__(self, name=None, born=None, died=None, country=None):
            super(Author, self).__init__()
            if name: self.name = name
            if born: self.born = born
            if died: self.died = died
            if country: self.country = country
```

The *mkTitle()* method is for convenience, and is intended to summarise all the information we have on an author in a single line of text.

```
    def mkTitle(self):
        title = self.name
        if self.born:
            title = title + '({}'.format(self.born.year)
            if self.died:
                title = title + '-{})'.format(self.died.year)
            else:
                title += ')'
            if self.country:
                title = title + '[{}]'.format(self.country)
        return title
```

7.6.3 The Authors collection

```
class Authors(grok.Container):
    ''' All of our authors are part of this collection '''
    grok.implements(ISiteIndex, ISearch)
```

By implementing ISiteIndex, we know *Authors* gets a site index rendered. This lets us define header, content and navigation viewlets for the container.

ISearch provides us with some smart abilities. Firstly, forms based upon the implementor of ISearch give us a search entry box. Secondly and most important, ISearch may be used in conjunction with a catalogue of indexes to give us a nicely paged search and result list. More on that later.

```
    site_title = u'Authors'
    search = FieldProperty(ISearch['search'])
```

The content of *site_ title* is rendered as a page title as per the index.pt template we saw earlier. Defining *search* as a *FieldProperty* is not necessary, but it is more correct and self documenting to do so.

```
    _seq = 0
    def next_seq(self):
            self._seq += 1
            return self._seq
```

Here we define a persistent sequence, initialised to zero and incrementing each time we call *next_seq()*.

```
        grok.traversable('new')
        def new(self):
            return Author()
```

EditForms and AddForms

Grok provides us with an *EditForm* as well as an *AddForm*. Conceptually, an instance of the model need not exist prior to the use of an AddForm, but most for an EditForm. By defining the method *new()*, we gain two things:

1. We now need only an *EditForm*, which simplifies some things, and
2. The object returned by the method automatically gains a *__parent__* attribute, being the container.

By stating that the method *new()* is traversable, the method is callable via the url: */pat/authors/new*

```
        def delete(self, key):
            if key in self.keys(): del self[key]
```

The *delete()* method is again for convenience only.

7.6.4 The *AuthorIndex* Catalogue

A catalogue (*zope.catalog*) is a collection of indexes pertaining to a single object type, here defined by grok.context(IAuthor). Effectively, whenever anything which implements the IAuthor interface is added to a container ***for the very first time***[4], that object's fields will be indexed by the catalogue.

```
    class AuthorIndex(grok.Indexes):
        ''' This catalogues and indexes all of our authors '''
        grok.site(ISiteRoot)
        grok.context(IAuthor)
```

[4]The object created by *new()* was given a __parent__, but not added to the container. This state would prevent the required *IObjectAdded* event from firing were we to add the object to the container, and skip the catalog document addition. The workaround is to apply form data to an entirely new object and add that to the container instead of the one we created with new().

```
        grok.name('authorindex')
        name = grok.index.Text()
        born = grok.index.Field()
        died = grok.index.Field()
        country = grok.index.Text()
```

Note that we declare the site as *ISiteRoot*. By doing this, we are saying that we want the object which implements that interface to act as the site for our catalogue, and simultaneously avoid circular module imports.

The *grok.context(IAuthor)* tells the catalogue which kinds of object to look for by monitoring the IObjectAdded event, and *grok.name('authorindex')* gives the catalogue a name.

For each attribute we define, we specify the kind of index to use. A *grok.index.Text()* indexes words in the text, and allows subsequent arbitrary partial queries. The *grok.index.Field()* indexes exact values, so for example one would have to search for the exact text in a field to produce a match. Not used here is *grok.index.Value()* which provides support for matches using the greater or less than operators.

7.6.5 AuthorSearch; a *PagedSearchAdapter*

Here's an interesting definition:

```
class AuthorSearch(PagedSearchAdapter):
    ''' An adapter which does library searches and returns results '''
    grok.adapts(Authors, grok.IBrowserRequest)
    catalog = 'authorindex'
```

This class inherits from *PagedSearchAdapter*, imported earlier. We specify that it adapts two kinds of classes, being *Authors*, and a *request*. So in other words, it's a *zope.component.multiAdapter*. Not shown here, is the interface implemented by *AuthorSearch*, which is an ISearch.

We will take a detailed look at *PagedSearchAdapter* later (section 7.7), but for now, lets move on.

7.6.6 EditAuthor; an edit form

At some point, we know that we will want to edit an *Author*. This form provides the ability to do exactly that. Since we are just using the default page template for edit forms, we need not bother with defining our own. Grok+*zope.formlib*

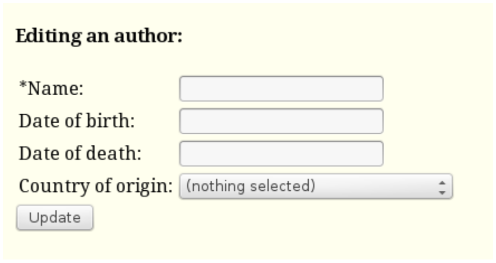


Figure 6: The author edit form

will look at the fields defined by the context (IAuthor), and produce a form for us using the default widget for each schema field.

```
class EditAuthor(grok.EditForm):
    ''' A form letting us define an author '''
    grok.context(IAuthor)
    def author_registered(self, act=None):
        author = self.context
        container = self.context.__parent__
        if author.key() in container: return True
    @grok.action(u'Update')
    def update_author(self, **data):
        author = self.context
        container = self.context.__parent__
        self.applyData(author, **data)
        key = author.key()
        if key not in container:
            key = str(container.next_seq())
            author = container[key] = Author(**data)
            author.key(key)
            self.redirect(grok.url(self.request, author))
    @grok.action(u'Delete', condition=author_registered)
    def delete_author(self, **data):
        author = self.context
        self.redirect(grok.url(self.request, author, 'delete'))
```

When rendered using the default form template, it looks like figure 6:

Of special interest, the *@grok.action* decorator defines *actions*, which becomes a list of *submit* buttons. The first argument is the button title, and further arguments control other aspects of the action. Take a look at the *u'Delete'* action: the *condition=author_registered* calls the *author_registered()* method, which means that the delete button will never appear unless the object is already in the container.

Also, take special note of the rigmarole in the *update_author()* function. First, we apply the incoming request *data* to the current *context*. We then retrieve the *key*, and if the key is not in the container, we create a whole new *Author* object and add that instead of just adding the context which is already an *Author*. Why?

Well, it turns out that adding objects to the container which already have a *__parent__* attribute does not trigger the IObjectAdded event that our catalogue is waiting for, and so the object never gets indexed. This sort of thing might be construed as a bug, except for the fact that the writers of Grok expected us to use a *grok.AddForm* and not a *grok.Editform* to add completely new objects to the container.

7.6.7 A delete view for an author

```
class Delete(grok.View):
    ''' A view to delete an author '''
    grok.context(Author)
    def render(self):
        index = getMultiAdapter((self.context, self.request), name=u'index')
        index.deleting = True
        return index()
```

When hitting the *'delete'* button, we navigate to the *delete* view. Instead of a template, this view has just a *render()* method. We locate the *index* view for our author using *getMultiAdapter*, set the *deleting* attribute and return the rendered *index()* view. This renders the whole site view, but with our *delete* form for the author instead of the *edit* form (see **??** and **??**).

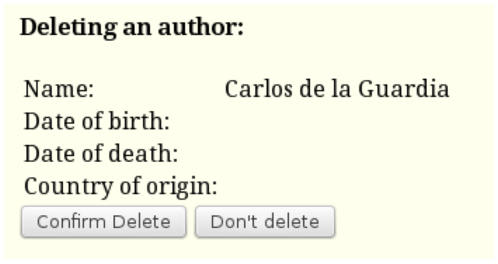
7.6.8 A deletion confirmation form

Grok provides a DisplayForm which effectively renders the form as read-only. The problem with using such forms for confirmation of an action, is that the form is read-only: Thus, actions will not be performed correctly.

Luckily we have an alternative. We can render our form as an edit form, but with all read-only fields (figure 7). Since it's an edit form, actions are excuted properly:

```
class DelAuthor(grok.EditForm):
    ''' An author deletion confirmation '''

                                    grok.context(IAuthor)
```


Deleting an author:

Name:	Carlos de la Guardia
Date of birth:	
Date of death:	
Country of origin:	

Figure 7: Author delete form

```
form_fields = grok.Fields(IAuthor, for_display=True)
@grok.action(u'Confirm Delete')
def delete_author(self, **data):
    author = self.context
    container = self.context.__parent__
    container.delete(author.key())
    self.redirect(grok.url(self.request, container))
@grok.action(u"Don't delete", validator=lambda *a, **k: {})
def cancel(self, **data):
    author = self.context
    self.redirect(grok.url(self.request, author))
```

7.6.9 A search form

Remember our *AuthorSearch* multi adapter we defined in section 7.6.5? Well, we are still not quite ready to look at the implementation for *PagedSearchAdapter*, but the following shows how we go about doing our searches.

The *update()* method is called first, an sets up the *SearchForm.search* attribute to point at a *do_search()* method inside the adapter. Each *grok.action()* defines a button which, when clicked, submits the form and allows us to use the content of the *search* field (*data['search']*) to perform the appropriate search (figure 8).

```
class SearchForm(grok.EditForm):
    ''' This form performs a search and returns a list of authors '''
    grok.context(Authors)
    form_fields = grok.Fields(ISearch)
    search = None
    def update(self):
        self.search = getMultiAdapter((self.context, self.request),
                                      ISearch).do_search
    @grok.action('Name')
```

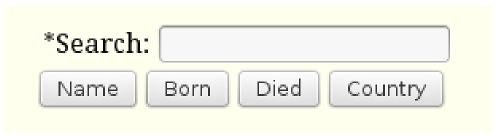


Figure 8: The search form for authors

```
    def find_name(self, **data):
        self.search(name=data['search'])
    @grok.action('Born')
    def find_born(self, **data):
        self.search(born=data['search'])
    @grok.action('Died')
    def find_died(self, **data):
        self.search(died=data['search'])
    @grok.action('Country')
    def find_country(self, **data):
        self.search(country=data['search'])
```

7.6.10 Header, Content and Navigation viewlets for the Authors container

The header is really simple. It's just a single template line:

```
class Header(grok.Viewlet):
    ''' Authors section header '''
    grok.viewletmanager(HeaderMgr)
    grok.context(Authors)
```

Template (*author_templates/header.pt*):

```
<h1>You are defining Authors</h1>
```

The content is a bit more involved. As is apparent, if a search was done earlier, it uses the result from the search:

```
class Content(grok.Viewlet):
    ''' Authors section content '''
    grok.viewletmanager(ContentMgr)
    grok.context(Authors)
    auth_results = None
    def update(self):
        search = self.search = \
            getMultiAdapter((self.context, self.request), ISearch)
        if 'prevPage' in self.request: search.prevPage()
        if 'nextPage' in self.request: search.nextPage()
```

Note how the *update()* method for viewlets does not receive named arguments for request variables, and we have to test the content of the *request* structure directly to see whether a *prev* or *next* button was pressed.

Template (*author_templates/content.pt*):

```
<h4>You may filter authors by any of the following fields:</h4>
<div style='margin:1em' tal:content='structure context/@@searchform' />
<div style='float:left' class='prettyTable'
tal:define='rows viewlet/auth_results | nothing'
tal:condition='rows'>
<table>
<thead>
<tr>
<th>Name</th>
<th>Born</th>
<th>Died</th>
<th>Country</th>
<th />
<th />
</tr>
</thead>
<tbody>
<tr tal:repeat='row rows'>
<td tal:content='row/name' />
<td tal:content='row/born' />
<td tal:content='row/died' />
<td tal:content='row/country' />
<td><a tal:attributes="href python:view.url(row)">
<button>Edit</button></a>
</td>
<td>
              <a tal:attributes="href python:view.url(row, 'delete')">
<button>Delete</button></a>
</td>
</tr>
</tbody>
</table>
</div>
<div style='float:none; clear:both' />
```

The first thing to point out, is the instruction tal:content='structure context/@@searchform' which replaces the content of the div with that of the search form (The view SearchForm). This is a neat way of plugging views together.

The next thing is tal:define='rows viewlet/auth_results | nothing' which defines the variable *rows* equivalent to *auth_results* calculated in the viewlet *update()*

method. We then do tal:condition='rows' to make the table render conditional on whether we have a valid *rows* variable.

The navigation is again a simple thing:

```
class LeftNav(grok.Viewlet):
    ''' Authors section navigation '''
    grok.viewletmanager(LeftNavMgr)
    grok.context(Authors)
```

The template(*author_templates/leftnav.pt*)

```
<p><a tal:attributes="href python:viewlet.view.url('new')">
  <button>Add a new author</button>
</a></p>
<p><a tal:attributes="href python:viewlet.view.url('../library')">
  <button>Edit the Library</button>
</a></p>
<p><a tal:attributes="href python:viewlet.view.url('..')">
  <button>View the Dashboard</button>
</a></p>
```

Here we use some simple Python snippets to build urls for the *href* links.

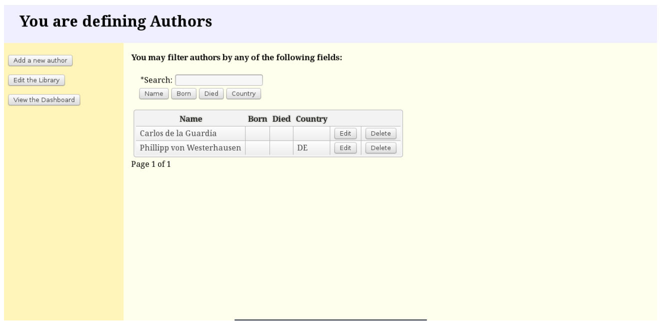
7.6.11 Header, Content and Navigation viewlets for the Author Model

Just as with the *Authors* collection, which gains *content*, *header* and *navigation* viewlet slots by reason of implementing the ISiteIndex interface, the *Author* Model likewise implements ISiteIndex and gets the default index view.

Declarations for the viewlet classes are minimalist:

```
class Author_Header(grok.Viewlet):
    ''' Author editor header '''
    grok.viewletmanager(HeaderMgr)
    grok.context(Author)
class Author_Content(grok.Viewlet):
    ''' Author editor content '''
    grok.viewletmanager(ContentMgr)
    grok.context(Author)
class Author_LeftNav(grok.Viewlet):
    ''' Author editor navigation '''
    grok.viewletmanager(LeftNavMgr)
    grok.context(Author)
```

The header viewlet (*authors_templates/author_header.pt*) is really simple as always:


You are defining Authors

You may filter authors by any of the following fields:

*Search:

Name	Born	Died	Country		
Carlos de la Guardia				Edit	Delete
Phillipp von Westerhausen			DE	Edit	Delete

Page 1 of 1

Figure 9: The *Authors* page

```
<h1>Author Detail</h1>
```

You will remember how the *layout.Index(grok.View)* class sets a *'deleted'* attribute dependent on request data. The content viewlet (*authors_templates/author_content.pt*) for the *Author* model uses this attribute to decide which form to display:

```
<div tal:condition="not:viewlet/view/deleting">
    <h4>Editing an author:</h4>
    <div tal:content="structure context/@@editauthor" />
</div>
<div tal:condition="viewlet/view/deleting">
    <h4>Deleting an author:</h4>
    <div tal:content="structure context/@@delauthor" />
</div>
```

The term tal:content="structure context/@@editauthor" replaces the content of the *div* tag with whatever the *EditAuthor* view renders (see section 7.6.6). Since *EditAuthor* is an automatic form, this depends on the current state of the *Author* model instance, and the schema fields provided by IAuthor. The same thing goes for *DelAuthor*.

The page template for navigation is as follows:

```
<p><a tal:attributes="href string:..">
   <button>Show Authors</button></a></p>
<p><a tal:attributes="href string:../../library">
   <button>Edit the Library</button></a></p>
```

7.7 The paged search adapter

Earlier (section 7.6.5), we skimmed over the processing of a catalogue (*zope.catalog*) search, having shown a class definition for *AuthorSearch* which derives from a

PagedSearchAdapter. We showed how we use this adapter for search:

```
self.search = getMultiAdapter((self.context, self.request),
                              ISearch).do_search
self.search(name=data['search'])
```

and how we access the results

```
search = getMultiAdapter((self.context, self.request), ISearch)
self.auth_results = search.results()
```

7.7.1 Defining a grok.MultiAdapter

In this section we will discuss the implementation of the adapter (*pagesearch*.py). We declare the class *PagedSearchAdapter* as a *grok.MultiAdapter*, which adapts a generic INTERFACE and grok.IBROWSERREQUEST.

As the directives state, the adapter implements an ISEARCH. Another way of saying this, is that given an interface and request, the adaptor produces an ISEARCH instance.

```
class PagedSearchAdapter(grok.MultiAdapter):
    ''' An adapter which does searches and returns results '''
    grok.adapts(Interface, grok.IBrowserRequest)
    grok.implements(ISearch)
```

7.7.2 The grok.baseclass() directive

```
    grok.baseclass()
```

The directive *grok.baseclass()* indicates that the class will be used as a base from which to derive other classes. We saw earlier how class *AuthorSearch* was derived from this class.

7.7.3 Initialisation

```
    search  = ''  # The search field (unused)
    limit   = 10  # Limits results to this many items
    offset  = 0   # Position from which we display items
    nPages  = 0   # Number of pages in current search result
    catalog = ''  # Override with applicable catalog name
    def __init__(self, context, request):
        self.context = context
        self.request = request
        self._getStatus()  # Read session if it exists
```

The attributes are given default values; we default to 10 rows per page and an empty catalog string. The initialisation refreshes the object instance with any current state.

```
def _calc_nPages(self):
    return 1 + int((len(self.results(noLimits = True))-1)/self.limit)
```

7.7.4 The Grok/Zope session

```
def _getStatus(self):
    session = ISession(self.request)['SearchResults']
    if self.__class__ not in session: session[self.__class__] = {}
    status = session[self.__class__]
    self.nPages = status['nPages'] if 'nPages' in status else self._calc_nPages()
    self.offset = status['offset'] if 'offset' in status else 0
```

Taking a closer look at *_getStatus()*, we see that it uses another adapter to retrieve an ISession from the *request*. Apparently an ISession acts like a dict; we create a session variable called *'SearchResults'*, and then use the class name to locate a *status*. We save *nPages* and *offset* in *status*, being the only two variables we need to implement a pager.

Sessions [section 6.3.3] are automatically associated with a remote browser making a request, and is a feature provided by the Grok/Zope framework. Sessions are cookie based, and will expire after 20 minutes of inactivity. Any data stored in a session is persisted for as long as the session stays alive.

7.7.5 Recording state in the session

```
def _setStatus(self, results=None):
    session = ISession(self.request)['SearchResults']
    if self.__class__ not in session: session[self.__class__] = {}
    status = session[self.__class__]
    if results is not None:
        status['results'] = results
        self.nPages = self._calc_nPages()

    status['offset'] = self.offset
    status['nPages'] = self.nPages
```

The *_setStatus()* method stores session data and performs the opposite function from *_getStatus()*. We see here that search results are also stored with the session. The next part is just boilerplate code to implement a pager.

```
def next_page(self):
    if self.page() < self.nPages: self.offset += self.limit;
    self._setStatus()
def prev_page(self):
    if self.page() > 1: self.offset -= self.limit;
    self._setStatus()
def page(self):  # current page #1 = first
    return 1 + int(self.offset/self.limit)
```

7.7.6 Doing searches and saving results

The *do_search()* method takes arguments in the form fieldname=searchspec, ... and allows us to search multiple fields simultaneously.

```
def do_search(self, **kw):
    try:
        catalog = getUtility(ICatalog, name=self.catalogName)
        results = catalog.searchResults(**kw)
        self.offset = 0
    except:
        results = []
    self._setStatus(results=results)
```

7.7.7 How catalogues work

A *zope.catalog* is a *LocalUtility*, associated with a model or container which implements an ISite. A *grok.Application* is an ISite, and when we defined the catalogue, we specified the site where the data for the catalogue should be stored, using the *grok.site()* directive. When indexes associated with the index are updated, these indexes are stored persistently with the site. When we do getUtility(ICatalog, name=self.catalogName), the utility which is returned is a persistent structure [5]. So whatever is stored in the catalog lives persistently in the ZODB.

Instead of storing weak references to the actual documents being indexed, *zope.catalog* indexes store integer references. Another utility, *IIntIds(catalog)* - (see *zope.intids*), is associated with the catalogue, which provides an integer ID to weakref lookup for each document. This greatly simplifies the task of maintaining indexes and reduces processing overheads.

[5]This has some rather large implications about when *LocalUtility* structures are added to the ZODB, and what happens if they are changed. In a nutshell, if you want to add them to an existing project, there is a world of pain.

So basically, if you are using a *LocalUtility*, make sure you get it all right before you start adding data to your site.

7.7.8 Retrieving stored results

The do_search(self, **kw) function locates the catalogue using getUtility(ICatalog, name=self.catalogName) and then performs the search using the catalog.searchResults(**kw) call. Results are just stored in the session for later access.

```
def results(self, noLimits = False):
    session = ISession(self.request)['SearchResults']
    status = session[self.__class__]
    results = []
    if 'results' in status:
        results = list(status['results'])
    if not len(results):
        results = self.context.values()
    return results if noLimits else results[self.offset:self.offset+self.limit]
```

If results are not available in the session, we just return the full set of values in the container.

7.7.9 A visual component for the paged search adapter.

For all models which implement an ISearch, we would like to display a page counter, and potentially prev/next buttons to allow for navigation up and down through the list of items being displayed. For our app, this includes the *Authors* and *Library* collections. Since the *Pager* class is general in nature, we added it to *layout.py*. We could also have placed it in *pagedsearch.py*, which is arguably a better place for it.

Remembering that viewlets are displayed for a specific viewlet manager, we can define a viewlet to be included in the content manager as follows;

```
class Pager(grok.Viewlet):
    ''' Displays a pager for models implementing ISearch '''
    grok.viewletmanager(ContentMgr)
    grok.context(ISearch)
    grok.order(2)
    def update(self):
        search = getMultiAdapter((self.context, self.request), ISearch)
        if 'prevPage' in self.request: search.prevPage()
        if 'nextPage' in self.request: search.nextPage()
        self.nPages = search.nPages
        self.currPage = search.page()
        self.prevPage = self.currPage > 1
        self.nextPage = self.currPage < self.nPages
```

The *grok.order()* directive is a way or rearranging content snippets, and tells the viewlet manager to display our pager viewlet after the rest of the content. We could switch this pager to render in the navigation pane instead of the content pane by simply changing the directive to: *grok.viewletmanager(LeftNavMgr)*

The *update()* method is always called before a render, which in this case is provided by a template (*layout/pager.pt*);

```
<div>
<div>Page <span tal:replace='viewlet/currPage' />
     of <span tal:replace='viewlet/nPages' /></div>
<div>
<a tal:condition='viewlet/prevPage'
tal:attributes='href string:.?prevPage=True'>
<button>Prev Page</button></a>
<a tal:condition='viewlet/nextPage'
tal:attributes='href string:.?nextPage=True'>
<button>Next Page</button></a>
</div>
</div>
```

7.8 The library

Having discussed the implementation for the *Authors* collection, a great deal of our implementation for a *Library* will be very familiar. Where our *Authors* class was a collection of *Author* model instances, our *Library* class is a collection of *Book* model objects. Both *Library* and *Book* implements ISiteIndex, and so inherit the main site index view. This means that we get viewlet slots for *header*, *content* and *navigation* for both *Library* and *Book*.

7.8.1 The Book model

The *Book* model definition is approached in almost exactly the same way as the *Author* model (section 7.6.2).

```
class Book(grok.Model):
    ''' A book is defined by an IBook interface '''
    grok.implements(IBook, ISiteIndex)
    site_title = u'Book Detail'
    authors   = FieldProperty(IBook['authors'])
    title     = FieldProperty(IBook['title'])
    subject   = FieldProperty(IBook['subject'])
    publisher = FieldProperty(IBook['publisher'])
    pub_date  = FieldProperty(IBook['pub_date'])
    isbn      = FieldProperty(IBook['isbn'])
    notes     = FieldProperty(IBook['notes'])
    _key = None
    def key(self, val=None):
        if val: self._key = int(val)
        if self._key: return str(self._key)
    def __init__(self, authors=None, title=None, subject=None, publisher=None,
                 pub_date=None, isbn=None, notes=None):
        super(Book, self).__init__()
        if authors: self.authors = authors;
        if title: self.title = title;
        if subject: self.subject = subject
        if publisher: self.publisher = publisher
        if pub_date: self.pub_date = pub_date
        if isbn: self.isbn = isbn
        if notes: self.notes = notes
    def authorStrings(self):
        if self.authors and len(self.authors): return "|".join([a.name for a in self.authors])
        return ""
```

Where the *IBook* interface defines that we should have an *authorStrings()* method, we implement it here in the class definition for the *Book* object. It simply returns a concatenated string of authors separated by a pipe (|) symbol.

7.8.2 Delete Viewlet for books

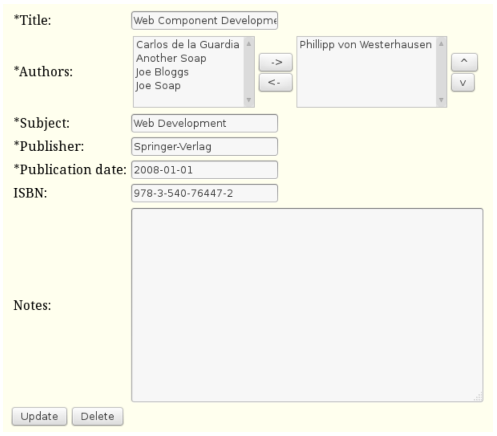
This is a replica of the equivalent function for deleting authors.

```
class Delete(grok.View):
    ''' A view to delete a book '''
    grok.context(Book)
    def render(self):
        index = getMultiAdapter((self.context, self.request), name=u'index')
        index.deleting = True
        return index()
```

7.8.3 Content, header and navigation for books

Our class definitions for *content*, *header* and *navigation* slots are uncomplicated:

```
class Book_Header(grok.Viewlet):
    ''' Book editor header '''
    grok.viewletmanager(HeaderMgr)
    grok.context(Book)
class Book_Content(grok.Viewlet):
    ''' Book editor content '''
```



*Title: Web Component Developme

*Authors: Carlos de la Guardia, Another Soap, Joe Bloggs, Joe Soap | Phillipp von Westerhausen

*Subject: Web Development

*Publisher: Springer-Verlag

*Publication date: 2008-01-01

ISBN: 978-3-540-76447-2

Notes:

Figure 10: Edit view for a book

```
    grok.viewletmanager(ContentMgr)
    grok.context(Book)
class Book_LeftNav(grok.Viewlet):
    ''' Book editor navigation '''
    grok.viewletmanager(LeftNavMgr)
    grok.context(Book)
```

Header template (library_templates/book_header.pt):

```
<h1>Book Detail</h1>
```

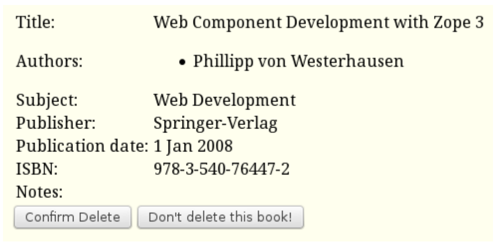
Content template (library_templates/book_content.pt):

```
<div style='margin:1em'>
<div tal:condition='not:viewlet/view/deleting'
         tal:content='structure context/@@editbook' />
<div tal:condition='viewlet/view/deleting'
         tal:content='structure context/@@deletebook' />
</div>
```

The content template simply renders an edit form or a delete form dependent on the state of the *'deleting'* view attribute. This was the same behaviour for the *Author* content viewlet.

Navigation template (library_templates/book_leftnav.pt):

```
<p><a tal:attributes="href string:..">
    <button>Show the Library</button></a></p>
<p><a tal:attributes="href string:../../authors">
    <button>Edit Authors</button></a></p>
```



Title:	Web Component Development with Zope 3
Authors:	• Phillipp von Westerhausen
Subject:	Web Development
Publisher:	Springer-Verlag
Publication date:	1 Jan 2008
ISBN:	978-3-540-76447-2
Notes:	

Figure 11: Delete view for a book

7.8.4 Edit and Delete forms for books

The same logic as described for authors, applies to editing books.

```
class EditBook(grok.EditForm):
    ''' A book editor '''
    grok.context(Book)
    form_fields = grok.Fields(IBook)
    def book_in_library(self, _act=None):
        book = self.context
        library = book.__parent__
        return book.key() in library
    @grok.action(u'Update')
    def update_book(self, **data):
        if not self.book_in_library():
            library = self.context.__parent__
            key = str(library.next_seq())
            book = library[key] = Book(**data)
            book.key(key)
            self.redirect(grok.url(self.request, book))
        self.applyData(self.context, **data)
    @grok.action(u'Delete', condition=book_in_library)
    def delete_book(self, **data):
        book = self.context
        self.redirect(grok.url(self.request, book, 'delete'))
```

The *book_in_library()* method checks to see whether the current book is already in the library. It is used in *update_book()* as well as a condition to the *delete_book()* action. You can see the book edit form in action in figure 10.

The deletion form, figure 11, displays all fields as read-only, and asks for confirmation:

```
class DeleteBook(grok.EditForm):
    ''' A book deletion confirmation '''
    grok.context(Book)
    form_fields = grok.Fields(IBook, for_display=True)
    @grok.action(u"Confirm Delete")
    def delete(self, **data):
        library = self.context.__parent__
        key = self.context.key()
        if key in library:
            self.context.authors = []
            del library[key]
        self.redirect(grok.url(self.request, library))
    @grok.action(u"Don't delete this book!", validator=lambda *a, **k: {})
    def cancel(self, **data):
        book = self.context
        self.redirect(grok.url(self.request, book))
```

7.8.5 The Library collection

```
class Library(grok.Container):
    ''' A library is a collection of books '''
    grok.implements(ISearch, ISiteIndex)
    site_title = u'Library'
    search = FieldProperty(ISearch['search'])
    _seq = 0
    def next_seq(self):
        self._seq += 1
        return self._seq
    grok.traversable('new')
    def new(self):
        return Book()
```

A *Library* is a *grok.Container* for *Book*, just as *Authors* was a container for *Author*. It implements the ISiteIndex and ISearch interfaces, which means that it gets a search box and the main site view. As with *Authors*, a book does not really have an identifying key, so we implement a sequence field (def next_seq(self):...) .

The same bit of cheating (grok.traversable('new'), def new(self):...) as we did previously, lets us roll *edit* and *add* views into just an *edit* view.

7.8.6 A catalogue for the Library

```
class BookIndex(grok.Indexes):
    ''' A catalogue of books in our library '''
    grok.site(ISiteRoot)
    grok.context(IBook)
    grok.name('bookindex')
    title         = grok.index.Text()
    subject       = grok.index.Text()
    publisher     = grok.index.Text()
    pub_date      = grok.index.Field()
    isbn          = grok.index.Text()
    authorStrings = grok.index.Text()
```

The nature and use of a catalogue was described in section 7.6.4, where it pertains to an Author model. Here, we define a catalogue for books, containing a number of indexes we can use to search.

Note *authorStrings*, which is defined as a method in the IBook interface, and implemented in the *Book* model as a concatenation of all the authors. This is indexed just as any other attribute would be.

7.8.7 A paged search adapter for the Library

```
class LibrarySearch(PagedSearchAdapter):
    ''' An adapter which does library searches and returns results '''
    grok.adapts(Library, grok.IBrowserRequest)
    catalog = 'bookindex'
```

Having discussed the PagedSearchAdapter earlier (see section 7.6.5), the implementation here should be obvious.

7.8.8 A search form for books in the library

```
class SearchForm(grok.EditForm):
    ''' Searches our book catalogue '''
    grok.context(Library)
    form_fields = grok.Fields(ISearch)
    search = None
    def update(self):
        self.search = getMultiAdapter((self.context, self.request), ISearch).do_search
    @grok.action('Title')
    def find_title(self, **data):
        self.search(title=data['search'])
    @grok.action('Subject')
    def find_subject(self, **data):
        self.search(subject=data['search'])
    @grok.action('Authors')
    def find_authors(self, **data):
        self.search(authorList=data['search'])
    @grok.action('Publisher')
    def find_publishers(self, **data):
        self.search(publisher=data['search'])
    @grok.action('ISBN')
```

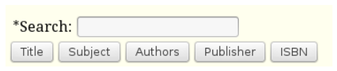


Figure 12: The search form for a book

```
        def find_isbn(self, **data):
            self.search(isbn=data['search'])
```

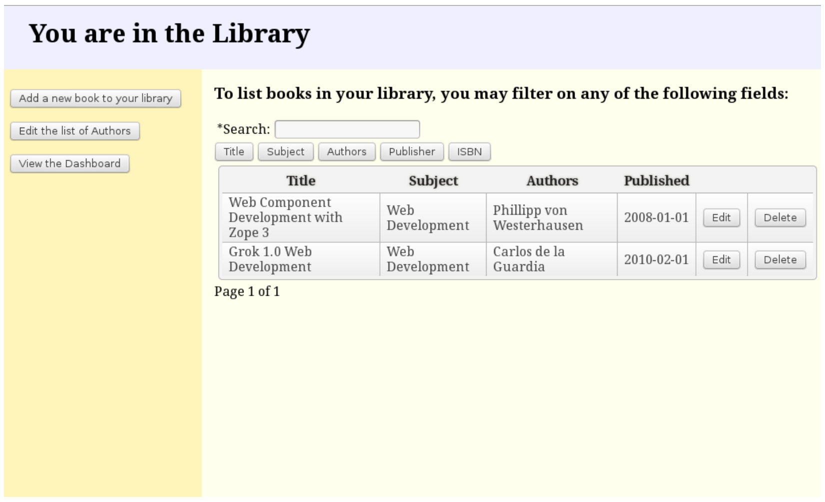
The *Book* search form differs from that of the *Author* (section 7.6.9) only in the fields available for search. The result of this view may be seen in figure 12 .

7.8.9 The Header, Content and Navigation for the Library

```
    class Header(grok.Viewlet):
        ''' Library section header '''
        grok.viewletmanager(HeaderMgr)
        grok.context(Library)
    class Content(grok.Viewlet):
        ''' Library section content '''
        grok.viewletmanager(ContentMgr)
        grok.context(Library)
        def update(self):
            search = self.search = \
                getMultiAdapter((self.context, self.request), ISearch)
            if 'prevPage' in self.request: search.prevPage()
            if 'nextPage' in self.request: search.nextPage()
        def book_results(self):
            return self.search.results()
    class LeftNav(grok.Viewlet):
        ''' Library section navigation '''
        grok.viewletmanager(LeftNavMgr)
        grok.context(Library)
```

Again we have an almost direct copy from the implementation of the *Authors* collection. For *Authors* we had a method called *auth_results()*. Here the method is *book_results()*.

Normally when faced with similar looking boilerplate code, one would start looking for ways to refactor the code. However, the page templates for these viewlets are slightly different from those for *Authors,* and although it is quite possible to make generic templates and a single set of viewlets to handle both *Authors* and the *Library* collection, there are few benefits to doing so.



You are in the Library

To list books in your library, you may filter on any of the following fields:

*Search:

Title	Subject	Authors	Published		
Web Component Development with Zope 3	Web Development	Phillipp von Westerhausen	2008-01-01	Edit	Delete
Grok 1.0 Web Development	Web Development	Carlos de la Guardia	2010-02-01	Edit	Delete

Page 1 of 1

Figure 13: The Library View

7.8.10 The header, content and navigation templates for the Library collection

There is again not much to say about these templates, due to their similarity to those for the *Authors* collection. At this stage what they produce should be self evident to the reader.

The result of these viewlets are shown when rendering the whole Library page as in figure 13 .

library_templates/header.pt

```
<h1>You are in the Library</h1>
```

library_templates/content.pt

```
<h3>To list books in your library, you may filter on any of the following fields:</h3>
<div tal:content="structure context/@@searchform" />
<div style='float:left' class='prettyTable'
tal:define='rows viewlet/book_results'
tal:condition='rows'>
<table>
<thead>
```

```
<tr>
<th>Title</th>
<th>Subject</th>
<th>Authors</th>
<th>Published</th>
<th />
<th />
</tr>
</thead>
<tbody>
<tr tal:repeat='row rows'>
<td tal:content='row/title' />
<td tal:content='row/subject' />
<td tal:define='authors row/authors'>
<div tal:repeat='auth authors'
                         tal:content='auth/name' />
</td>
<td tal:content='row/pub_date' />
<td><a tal:attributes="href python:view.url(row)">
<button>Edit</button></a>
</td>
<td><a tal:attributes="href python:view.url(row, 'delete')">
<button>Delete</button></a>
</td>
</tr>
</tbody>
</table>
</div>
<div style='float:none; clear:both' />
```

library_templates/leftnav.pt

```
<p><a tal:attributes="href python:viewlet.view.url('new')">
    <button>Add a new book to your library</button></a></p>
<p><a tal:attributes="href python:viewlet.view.url('../authors')">
    <button>Edit the list of Authors</button></a></p>
<p><a tal:attributes="href python:viewlet.view.url('..')">
    <button>View the Dashboard</button></a></p>
```

8 Epilogue

There are many topics left uncovered in this article, such as authentication and access control, backing up the database and other administrative functions, or in depth discussions of the page template macro facility. To cover the whole of Grok & Zope would mean that this article would instead quickly become a book.

The intention was that by first discussing the *Zope Component Architecture*, the foundation would be laid for better understanding of how the Zope/Grok framework provides the facilities that it does.

A topic for which there is a glaring lack in documentation or working examples, is the use of the Zope catalogue. This is probably by reason that in many projects, *zope.catalog* is unnecessary or worked around. Showing catalogue use in a non-trivial project was rewarding, but at the same time frustrating as it took a while to realise that adding an already parented object to a catalogue failed to generate the required events for the catalogue to index the document. This is precisely the sort of problem one might hope to avoid by referring to this article.

Without prior knowledge of the ZCA, and such esoterics as marker interfaces, the newcomer to Zope/Grok has no immediate pointers as to how to structure his site for component re-use. At the very least, this article should breach the gap in understanding left by entry-level tutorials as to how a large Grok/Zope site might be structured.