LearningJourney - introduction in Zope 3

Release 1.0

January 16, 2009

Contents:

Chapter 1

General presentation

Zope 3 is a complete framework that makes it possible to develop web applications using Python. Much of this framework can be used in any Python application, independent of what represents, historically, a Zope application.

The development of Zope 3 began in 2001-2002, in part as a reaction to the experience gained through developing the Zope 2 application server. The first version (Zope 3.0) was published in 2004.

The design of the infrastructure that represents Zope 3 was largely centered on the needs of big companies and the clients of the Zope Corporation, having in mind applications with a big level of complexity. As such, Zope 3 can be used in developing complex application, its best use being in such environments.

The main components that make it possible to develop Zope applications are:

- publishing a graph of objects through various protocols, tipically http
- an infrastructure similar to MVC for page publishing
- a security system that can be applied to any object, including pages
- integrates the Zope Component Architecture (ZCA)

Typically, a Zope application would include:

- integration with zc.buildout, to build the application infrastructure
- files to configure the server, maybe even integration with PasteScript
- a module that starts the application (startup.py)
- code packages and ZCML that will implement the desired application

Chapter 2

The philosophy of the Zope 3 ecosystem

When Zope 2 was published, it was an extremely inovative product (in that period, CGI was still considered high end). Still, some of the choices and assumptions made in the design of Zope 2 are not valid anymore, so Zope 3 was developed, partly, as a reaction to the perceived shortcommings of Zope 2.

Thus, in Zope 3 there are several important ideas:

It is *pythonic* It follows the mainstream conventions of code structuring, it uses Python libraries, it keeps a simple, easily understandable code structure.

Promotes interoperatibility The patter of adaptation makes it possible to safely use external libraries, the WSGI standard is implemented and made easily available, there is flexibility in the choice of the persistency medium, etc.

No more Zope programmers There are no more odd things in Zope 3 development that would require splitting from the rest of the community

Clean code The code should be clean, elegant and tested, the persistent objects shouldn't be overloaded with functionality that doesn't belong to them, logically

A testimony of the flexibility of the Zope 3 libraries is the inclusion in various applications and frameworks, among which are Plone, repoze.bfg, z3ext, etc. Most of the packages from Cheeseshop (PyPI) are producced by the Zope communities (Plone, Zope 3, Grok, zc.buildout).

Chapter 3

Strong points, weak points

In order to understand Zope 3 it is useful to know its advantages and disadvantages. These are:

3.1 Strong points

3.1.1 Makes possible development of complex applications

One of the advantages of Zope 3 is that it offers a way to manage the complexity of applications. One of the solutions used in the industry to build complex and extensible applications is to use an architecture that is based on components. Examples include XPCOM (Mozilla), Java Beans (Sun), COM (Microsoft) and Gumbo Flex (Adobe). The solution offered by Zope 3 is to integrate the `zope.interface` and `zope.component` libraries, which together form the **Zope Component Architecture** - ZCA

3.1.2 Using ZCA means flexibility and plugability

Most of the Zope 3 libraries use ZCA and this makes it possible to achieve great flexibility in changing the behaviour of libraries: injection points cover almost of the Zope operations.

3.1.3 It offers new and inovative solutions for web development

Just like Zope2 was an inovative product when it appeared and managed to make itself used and supported by a large community for more then 10 years, the innovations brought by Zope3 offer a strongly argumented solution to most of the tipical problems that are met by a web developer.

3.1.4 WSGI Compatibility

Just as Pylons, Turbogears and others, Zope 3 offers the possibility to be integrated with WSGI libraries "out of the box". By integrating with PasteScript it is possible to use, very easy, filters and other WSGI middleware.

3.1.5 Adaptability to new trends

One of the latest trends is developing relatively small projects, by small teams. In this context the preferences go to "agile" solutions and conventions over explicitness, so that the productivity is maximized for the ammount of written code. To cover this segment there is **Grok**, a framework that offers a level of convetions over the explicitness preferred by Zope 3. For those that prefer the minimalism, there is **repoze.bfg**, a framework similar to Pylons, but that integrates the Zope libraries.

3.1.6 Experienced developers

Some of the Zope developers have been working for over 12 years with the Zope components and some of its predecesors (Bobo, Principia, etc). One of the advantages that is given by the persistence and tenacity of this people is the experience that they offer in developing new versions of concepts and libraries that will offer solutions for the Zope 3 platform.

3.1.7 Reusable Zope libraries

The knowledge gained in programming on the Zope platform can be reused in developing other forms of Python applications: the advantages of the component based architecture can be applied to any other Python application.

3.2 Weak points

3.2.1 The developer/users community

Although the Zope 3 developers community is relatively big and strongly expanding thanks to its adaption by Plone, in general Zope has a smaller exposure and weaker marketing then other web frameworks, such as Django or Ruby on Rails.

3.2.2 Needs experienced developers, is relatively hard to learn

Zope 3 is not PHP. By using advanced concepts and the vast amount of solutions offered, Zope 3 is relatively dificult to learn by beginner programmers. With certainty, a beginning web developer would find it easier to understand PHP than to use efficiently the Zope 3 infrastructure. A programmer that never had to deal with Zope technologies has to learn the templating language, the system architecture, the ZCA way of thinking, the ZCML dialect of XML, concepts such as viewlets, form libraries, etc. Still, to be productive, one doesn't need to deeply learn all the components, just the way the libraries used work.

3.2.3 Predominant use of ZCA

Although it doesn't have foreign or difficult to learn concepts for a programmer with knowledge of OOP, ZCA needs some accomodation period to understand the way it can be used to the benefit of the developer.

3.2.4 Documentation problems

This is a problem of perception and marketing. Zope 3 doesn't have, at this moment, an attractive site that will centralize an exhaustive documentation set, as other webframeworks have (for example, Django). But there are several books dedicated to Zope 3, up to date with the latest changes, and the usage of `zope.interface` throught the codebase ensure that there is a good documentation of the API. Thus, the Zope API is well documented and is available straight from any Zope 3 application or from the site http://apidoc.zope.org. The text files that document the various libraries are gathered in a Zope Book, available in the same place.

3.2.5 Integration with relational databases

Although there are strong solutions for integration with relational databases (especially with ORM systems such as STORM or SQLAlchemy), the fact that they don't come by default with Zope, which offers an implicit integration with ZODB, can be perceived as a weakness of Zope.

Chapter 4

Builtin features

4.1 Web publishing

The main role of Zope is to publish webpages. Although Zope can work with more then the http protocol, many of the Zope libraries deal with this problem. The way in which Zope achieves web pages publishing is to associate pages to a graph (tree) of objects. Most of the times these objects are persistent, coming from ZODB, but they can be just as well "transient" objects.

4.2 Object persistency

By default the user data is saved in an object database called ZODB. Lately, with the appearance of the ORM libraries also appeared libraries that make it possible to publish objects that represent database data. The main problem that the ORM to Zope integration libraries solve is that of the transaction mechanism. If an error appears in the code that generates the page or processes the user data, the data will not be saved in the database, thus preserving the integration of the data.

4.3 API for developing applications

Zope 3 enjoys a rich ecosystem of libraries that cover most of what is necessary to develop web applications. The Zope API is well documented and stable. In case one of the libraries is deprecated, the backwards compatibility is maintaned. Thanks to the extensive usage of zope.interface in most of Zope's libraries, the API is self-documenting, and thanks to the extensive usage of the adaption design pattern, extending and changing the default behaviour of those libraries is easy to achieve. Just the same, it is easy to write code that makes it easy for third parties to extend.

4.4 Extensible templating system

Although Zope can use any templating engine, the default integration is with the ZPT engine. There are two implementations for ZPT: the classic one and **z3c.pt**, a new library, compatible with the ZPT syntax and integrated with Zope, but that offers a better rendering speed by a factor of at least 10 times. In Zope, the ZPT templating system can be extended with new expression types.

4.5 Templating models

Zope offers several high level models of integrating content into templates: the **content providers**, that insert content according to the the type of context, the **viewlets**, an extensible form of management of the content providers and **pagelets**, which separate the content of an html fragment (page or view) from the layout used for that piece of content.

4.6 Indexing and search system

Zope offers an integrated system of indexing objects, offering by default indexes for Text and arbitrary values (Field). Writing a new type of index is not a complicated task, thanks to the simple API that needs to be implemented. This makes it possible to easily integrate specialized search systems such as Lucene or Xapian.

4.7 Object adnotation

One of the innovative concepts integrate by Zope 3 is that of object annotation. This mechanism, facilitated by ZCA, allows annotating any object (persistent or transient) with data, so that its functionality can be extended.

4.8 Internationalization

Zope 3 offers an internationalization mechanism compatible with the **gettext** standard. This mechanism has extensions for ZPT and there is an easy to use API from Python. Also, there are mechanism to localize (format numbers, dates and time) and also a database with regional differences that are involved in localization. There are tools to extract the messages from zcml, zpt and python files and to integrate them in already generated po files. To internationalize content there are several extensions created by the Zope Community.

4.9 Events

A very easy and elegant way of providing insertion point to extend functionality, events are also used in Zope 3. During normal processing, there are a large number of events defined and triggered. Among the most important are the ones connected to the object lifecycle (creating, adding, modifying and erasing objects).

4.10 Web forms

Zope offers two libraries to generate, validate and process web forms, and the Zope community has developed a new package (z3c.form), which offers a similar, but more advanced and flexible compared to the provided Zope form library.

4.11 Security

Security is one aspect where Zope is very strong. The system integrated in Zope abstracts the interaction of external agents (users) and offers a mechanism based on permission, groups and roles. Implicit, there are two levels of security checks: per published page and per object.

4.12 Code test libraries

Zope makes it possible to automatically test the code (testing is strongly encouraged by the Zope community). There is integration with the test mechanisms offered by default by Python, and there is also the possibility to have integration and functional tests, to test the entire system.

4.13 Other protocols

Zope offers a simple way to queue messages for sending, including in asynchronous mode. Also, it is possible to publish the objects through various other protocols, such as WebDAV, FTP and XMLRPC.

Chapter 5

Differences to Zope 2

The programmers that have developed on the Zope 2 platform need to take the following into consideration:

5.1 Radical change in concepts

Although some concepts from Zope and CMF are kept, there are enough changes that the two systems can be considered completely different. Even so, a Zope 2 programmer will have a shorter and easier learning period.

5.2 Through the web development

TTW development is discouraged in Zope 3. The facilities provided by Zope 3 for TTW are limited and need to be enabled explicitely.

5.3 Acquisition

It is limited and explicit (even more, it is not used by the community). Some of the problems that acquisition solved are not longer valid (TTW development), or solve through other paradigms: local utilities, registered at site levels and an inteligent system of layers and skins implemented in Python code.

5.4 RestrictedPython

Once TTW development is no longer used, RPython is not necessary. This makes it possible to have unrestricted access to objects in ZPT (the objects are still wrapped in security proxies, so the objects need an explicit declaration of security settings).

Chapter 6

An example application

LearningJourney is a minimal Zope 3 example application. It offers a sistem similar with a basic blogging, multi-user, system. It has the following facilities:

- self-registering of users, with a system to check the emails
- a personal content area, protected by roles and permissions, in which the user can create personal content
- a simple type of content with a text field that can be edited with **TinyMCE** and a tag field that is used to clasify the record
- search on the site using the Zope catalog
- a page used to browse all the records, groupd by the tags.

6.1 Generating the application skeleton

The skeleton of the application was generated using ZopeSkel, a project that extends **PasteScript** with a new template (**zope_app**) that can be used to generate a basic Zope 3 application. This application is a WSGI app which can be configured using one of the two `ini` files from the root of the app - one for the debug mode and one for the production mode.

To start the app, one of the following commands can be used:

```
# bin/paster serve debug.ini
# bin/learningjourney-ctl fg
```

Both versions will start the applicatio in debug mode.

6.2 The application structure

The generated structure is that of a python package developed using setuptools. In the root is the `setup`.py` module, which configures the package, while the code itself sits in the ``learningjourney` namespace from the `src` folder. In this namespace there are 5 packages, of which the most important are `app` and `ui`. The `app` package hosts the models and the logic of the application, independent of their publishing as web pages. The `ui` package contain everything that pertains to the web pages published by the application (pages, images, CSS files, etc). The `startup.py` module is the one that defines the application server and is used by the scripts from the `bin` folder to start the app.

6.3 ZCML files

In the root of the application there are several zcml files: `site.zcml`, `apidoc.zcml` and `custom-security.zcml`. These are XML files that contain instructions to configure the components used inside the application and the rest of the Zope libraries. These configuration operations have a direct equivalent in Python code, but separating them in XML serves the following purposes:

- allows the site managers to configure the application without needing to know how to program (it is assumed that XML, as a language, is more familiar to the site admins)
- the ZCML configuration system does not allow accidental overriding of the configured components. To be able to override, an explicit mechanism needs to be used
- in general, it makes the developers' job easier, by making it easy to identify the components used in a package, without needing to read the code.

A basic example of an xml file:

```xml
<configure xmlns="http://namespaces.zope.org/zope"
           xmlns:browser="http://namespaces.zope.org/browser"
           package="learningjourney"
           i18n_domain="learningjourney">

   <include file="configure.zcml" />
   <include package=".ui" />
   <includeOverrides file="overrides.zcml" />

   <configure package="z3c.widget.tiny">
       <browser:resourceDirectory path="lib/resources" />
   </configure>

</configure>
```

The first tag, `configure`, serves as a grouping directive and specifies the package for which the grouped directives apply. It defines the default XML namespace as `http://namespaces.zope.org/zope` and the namespace with the **browser** prefix as `http://namespaces.zope.org/browser`.

The `include` directive is used to include a package (the `configure.zcml` file from that package will be read and configured) or a file, specified by a complete path, relative to the folder of the current file. In case a package name is used, this can be spelled as an "absolute" python package, with the complete name of the package, or as a "relative" package, in which case the name will start with one dot, for a sub-package in the current package, or two dots, for a package in the parent package. See PEP 328 http://www.python.org/dev/peps/pep-0328/#id10 for more details.

The `includeOverrides` directive (not shown here) is used to identify a file whose configuration will have priority in case it conflicts with other configuration directives.

A complete reference of the ZCML dialect is published on the APIDOC website, at http://apidoc.zope.org/++apidoc++/. Other ZCML directives will be discussed further in this text.

6.4 The basics of Zope Component Architecture

The interfaces are one of the fundamental mechanisms of Zope 3. They are objects that specify (document) the behaviour of other objects that declare to "provide" them. An interface specifies behaviour with the help of:

- informal documentation: the docstrings
- defining how methods and attributes should look like
- invariants, which are conditions the object should meet to "provide" that interface.

Thus, an interface describes the characteristics of an object, not its capacities. It describes what the object does, not how it does that. To understand how an object provides what the interface defines, it is necessary to read the definition of that object, its class.

The use of interfaces can be said to belong to a design pattern. One of the recommandations made in the "bible of patterns" - **Design Patterns** is to "program not for an implementation, but for an interface". Defining an interface helps in understanding the involved systems and is one of the first steps in using the ZCA.

An example of interface:

```
from zope.interface import Interface, Attribute, implements, provides

class IBoardgame(Interface):
   """A boardgame"""

   title = Attribute("title")
   description = Attribute("description")

   def borrow(to_person):
       """Allows changing the owner by borrowing to another person"""
```

We notice:

- interfaces are classes that inherit `zope.interface.Interface` (directly or through a chain of inheritance)
- attributes are defined as instances of `Attribute`
- methods don't have to be implemented, just defined and documented. That's why defining the self parameter in the method signature is not needed.

A class that will implement the interface can look like this:

```
class Boardgame(object):
    implements(IBoardgame)

    title = u""
    description = u""

    def borrow(self, to_person):
       self._location = "At this moment the game is at " + to_person
```

We notice the declaration `implements(IBoardgame)`, which registers the class `Boardgame` as implementing the interface `IBoardgame`. Just as many other things in Python, this declaration is based on a "gentlemen's agreement", as it doesn't force the programmer to really implement the interface. This means that it cannot be used, by itself, to implement "static typing" system in Python. It can be used, however, to determine the capabilities of an object:

```
>>> game = Boardgame()
>>> IBoardgame.providedBy(game)
True
>>> IBoardgame.implementedBy(Boardgame)
True
>>> list(providedBy(game))
[IBoardgame]
```

Notice: a class implements (will have `implements()` in code), an object will provide the interface (`provides()` can be asserted on it).

`zope.interface` provides an API to decorate with interfaces even objects and classes that come from external libraries:

```
>>> classImplements(Boardgame, IBoardgame)
```

and the equivalent in zcml:

```
<class class=".app.Boardgame">
    <implements=".interfaces.IBoardgame" />
</class>
```

or, direct on objects:

```
>>> alsoProvides(game, IBoardgame)
```

Once the interfaces are known, the definition for a **component**, as understood by Zope, is simple: a component is an object that provides at least an interface. Most of the classes in Zope are written so that they become component once they are instantiated.

6.5 Other features of ZCA

`zope.component` is a library based on `zope.interface` that introduces several types of components, in fact an implementation of several design patterns. The 4 base components are:

- adapters
- utilities
- subscribers
- handlers

6.5.1 Adapters

Adapters are an implementation of the AOP (aspect oriented programming) model. It helps, using the interfaces, to obtain an aspect of an object. In the process of adaptation are involved the adapted object and the interface that determins the aspect that we are seeking for that object.

An example: let's suppose we have a container that holds various objects. Some objects are audio files (mp3), some are images, some are text files, etc. We are looking to display the specific dimension of each of these objects. For the audio files we will display the size in seconds, for images the size in pixels, etc.

The solutions that would not involve `zope.component` would be:

- Building a class that would know how to extract the information from each type of object. This system is not flexible: to add a new type of object we need to add code to the class that will know how to extract the info from the object
- Implementing, by each object type, of a special method that would return the information that will be displayed. This means that the objects need to know in what type of systems they will be integrated, which leads to an overload with functionality for those objects.

The solution offered by adapters is elegant, but more complex. For each type of object there is a component that adapts the object and extract the information from each object.

We will have an interface `IDisplaySize` that defines the way the size for objects is displayed:

```
class IDisplaySize(Interface):
    """Provides information about the size of objects"""

    get_size():
        """Display the size, this info is intended for users"""
```

The code that displays the sizes of the objects will iterate through the objects of the container and build an adapter for each object:

```
>>> size = getAdapter(IDisplaySize, obj).get_size()
```

Notice that the object that is build by calling `getAdapter` is a component that provides the interface `IDisplaySize`. Through the construction of the adapter we can get an implementation specific to each type of context:

```
class Mp3DisplaySize(object):
    zope.component.adapts(IMp3File)
    zope.interface.implements(IDisplaySize)

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

    def get_size(self):
        sound_length = extract_track_size(context)
        return "%s seconds" % sound_length

class ImageDisplaySize(object)
    zope.component.adapts(IImage)
    zope.interface.implements(IDisplaySize)
    def __init__(self, context):
        self.context = context

    def get_size(self):
        width, height = get_image_size(context)
        return "%s x %s px" % (width, height)
```

Thus, acording to the type of context (`IMp3File` or `IImage`), will be selected a class that will be used in building the adapter, retrieving this way an object that is different for each type of context and that will know how to extract the information from it.

The construction `getAdapter(...)` can be shortned with:

```
>>> IDisplaySize(obj).get_size()
```

There are also **multiadapters**, which adapt more then one object to a given interface. The most common met example of multiadapters are the pages (or views), which adapt the request - information received from the user, along with the context object, to an information of type `IBrowserPublisher`, which will be returned to the users. The pages are registered as multiadapters with a name, so that we don't need to specify the interfaces to which we adapt the two objects, because there will be just one with that name and type of request:

```
>>> view = getMultiAdapter((context, request), name='index.html')
>>> page_content = view()
```

The rest of components (utilities, subscribers, handlers) will be covered later.

6.6 Publishing a simple page

Zope 3 has several ways to develop and publish web pages. The simples page doesn't need Python code, just a ZPT file. Pages are components which are registered with the ZCA.

```
<browser:page
    name="about.html"
    for="*"
    template="pt/about.pt"
    permission="zope.View"
    />
```

Notice the name of the page that will be used in the publishing of that page, the path to the template used, permission which the user needs to have to access it and the attribute `for`, which says for what type of contexts the page is available (in this case, all).

In case there are pages with the same name, because adaptation is used to determine the class that will be used to generate the page, it is possible to discriminate by using the type of context by class or interface and the layer (skin) per which the page is registered.

The context is determined by a dotted name to a class, an interface or the asterix, which means "all types of context" (or `zope.interface.Interface`).

6.7 Publishing resources

Resources (images, CSS and JS files) are registered with the help of two zcml tags from the **browser** namespace: `browser:resource`, which registers a single resource (text file or image) and `browser:resourceDirectory`, which can be used to register a folder to act as container for resources. For resources which are different according to the language, the `browser:i18n-resource` tag is used.

The path to these resources is generated as relative to the local site (details about sites will follow), in the form `http://localhost/mysite/@@/styles.css`

To compute this path, in the template is used an expression such as: `<script tal:attributes="src context/++resource++myscripts/jquery.js" />`

Examples of resources and their registration can be found in the files `src/learningjourney/ui/configure.zcml` and `src/learningjourney/widget/addremove/configure.zcml` and the template file `src/learningjourney/ui/site/pt/layout.pt` and `src/learningjourney/widget/addremove/widget.pt`

6.8 Persistent objects

To store the data in ZODB, we use classes that inherit from the `Persistent` class. These are very simple classes, as can be noticed from the `learningjourney.app.userhome.LearningEntry` example: inherit `Persistent`, implement the `ILearningEntry` interface, write the methods and attributes defined by the interface.

The last thing, to complete the integration with Zope, is the need to declare the security of that object. In case this doesn't exist, access to the attributes of the object will be forbidden (there will be `Forbidden` errors). This security declaration is done in zcml:

```
<class class=".userhome.LearningEntry">
    <require
        permission="zope.View"
        interface=".interfaces.ILearningEntry" />
    <require
        permission="zope.ManageContent"
        set_schema=".interfaces.ILearningEntry" />
</class>
```

For direct access to the object, unproxied by the security mechanism, the following piece of code can be used:

```
>>> from zope.proxy import getProxiedObject
>> obj = getProxiedObject(someobj)
```

Within the `require` tag the `interface` attribute is used to designate an interface that defines methods and attributes available for reading, with the indicated permission. The `set_schema` attribute is used to asign a permission to change the attributes from indicated interface.

In the case of containers, there are two interfaces involved in specifying the required permissions: `IReadContainer` and `IWriteContainer`. Example:

```
<class class=".site.Application">
  <require permission="lj.ModifyContent"
    interface="zope.app.container.interfaces.IWriteContainer" />
  <require permission="zope.View"
    interface="zope.app.container.interfaces.IReadContainer" />
  <allow attributes="getSiteManager" />
  <require permission="zope.ManageServices" attributes="setSiteManager" />
</class>
```

To restrict the types of objects that can be added in a container or the type of containers where an object can be added, you can use the constraints available from `zope.container.constraints`. An example is in `learningjourney.app.userhome`.

Other elements that can appear in this declaration are:

- the <allow> tag, which can be used to to set the access to certain attributes defined by an interface, or directly name the attributes as public (which will require the `zope.Public` permission)
- the `attributes`, `set_attributes` and `like_schema` options allow setting permissions at the level of attributes or copying the security settings of another class
- the `<implements>`, as specified earlier, allows augumenting a class with an extra interface. Objects derived from that class will provide that interface.

6.9 Interaction with the user through the request object

We saw earlier how a simple page can be created using just a template. Usually, we need a bigger level of logic and iteraction with the user. To achieve this, we can use a class to generate the page:

```
class HelloWorldPage(object):

    def __init__(self, context, request):
        self.context = context
        self.request = request

    def __call__(self)
        return u"Hello world"
```

```
<browser:page
    name="hello.html"
    class=".pages.HelloWorldPage"
    for="*"
    permission="zope.View"
    />
```

Notice that the published page is the result of calling the class instance - the `__call__` method is run. Also, the class is a multiadapter, so in the `__init__` it receives the context and the request. This class will be used as a *mixin* to generate, at *runtime*, a new type of object that will generate the class, so it's not really necessary to declare the `__init__` method, as it is not necessary to inherit from `BrowserPage` either.

It is also possible to combine a class with a ZPT template, as it is demonstrated for the `explore.html` (`ExplorePage`), associated with the `learningjourney.app.interfaces.ILearningJourneyApplication` objects, defined in the `learningjourney.ui.site.page` module.

In the templates associated with a page class, the instance of that class and its attributes can be accessed through the `view` variable. `context` and `request` can be accessed too.

Because the **request** object from the page offers access to the data associated with the GET and POST methods that were used to access the page, simple form pages can be implemented using a method like this:

```
class SampleForm(object):
    def __call__(self):
        if 'submit' in self.request.form:
            return u'Hello, %s' % self.request.form.get('name', 'John Doe')
        else:
            return ViewPageTemplateFile('sampleform.pt')()
```

The template, `sampleform.pt` will contain:

```
<form method="POST">
    <input type="text" name="name" />
    <input type="submit" name="submit" />
</form>
```

Using the **request.response** object, several parameters can be changed for the response returned to the visitor (for example, set some headers, implement a redirect, etc). A simple example of such a form can be found in the class

> `EntryDeletePage` from `learningjourney.ui.homefolder.page` or the class `DashboardRedirect` from the same module.

A convention used in Zope 3 to tell the traversing mechanism that it deals with a page or a view is to prefix the name of the page, in the URL, with two @ signs. Together they form two "eyes", a way of saying it's a "view".

Example: `http://localhost/@@index.html`

By using this convention the traversing mechanism is shortcuted, so that it will not try to find an object called `index.html` in the traversed container, and instead it will directly build a page `index.html` for the context object.

In templates pages can be inserted in other pages directly, like this:

```
<div tal:content="structure item/@@detail" />
```

where item is an object (can be persistent or not, for example the context, or an object retrieved from a `tal:repeat` construct), and `detail` is the name of the view registered and available for that object. When the `detail` view is built, it will have the `item` object as context.

6.10 Augumenting interfaces with information about the type of attributes

Another important library from Zope 3 is `zope.schema`, an extension of the `zope.interface` library, which allows a more details specification of the type of attributes defines in interfaces. In this way, the interfaces become "schema" and can be used in various tasks: validate object values, generate automated forms, generate pages to view objects, etc. By using the zope.schema extension, the attributes become "fields". Using `zope.interface.fieldproperty.FieldProperty` it is possible to implement an automatic validation of the values of an object, based on the schema (see, for an example, the implementation of `learningjourney.app.userhome.LearningEntry`.

`zope.schema` defines various types of fields. Some of them are text lines (`TextLine`), dates (`Date`), boolean values (`Bool`), list of objects (`List`), etc. It is possible to write new type of fields.

An example of schema/interface is `learningjourney.app.interfaces.ILearningEntry`. This interface is used to automatically implement two forms:

- an add and create form
- an edit form

Both are found in the module `learningjourney.ui.homefolder.page`

6.11 Sites and global registries

By default, the ZCA stores its component registrations in a global registry. The results of configuration specified in ZCML are always recorded in the global registry. It is possible to create a local registry, at the level of a persistent container in ZODB. This container will be called a "site" and in its local registry will recorded component registrations that are "local" to that site, which will have precedence over those recorded in an upper level or in the global registry. Not all types of components can be recorded in a local registry, just the utilities. An example of creating a local site manager is the code in `learningjourney.app.event.configure_site`. Local sites are also targeted as the base path in generating locations for browser resources (CSS, images, JS).

6.12 Utilities

Another type of component defined by `zope.component` are utilities. There are two types of utilities: global utilities and local utilites. They behave in a manner similar to **singletons** from the design patterns. Based on an interface, it is possible to get the unique object that is registered as an utility.

To register a global utility the tag `<utility>` is used. Its registration includes the interface the utility will be registered for, the name of the utility (it is possible to have several utilities registered for the same interface, but with different names), and the component that will be used in building the utility.

```
<utility
    provides="ILanguageNegociation"
    component=".app.LanguageNegociation"
    permission="zope.Public" />
```

To make an utility local you need a persistent object, which you'll register as a utility using `zope.component.registerUtility` or the functionality available in the ZMI on the "Registration" page.

Examples of how to create and register local utilities can be found in `learningjourney.ui.search.page.SearchPage`.

6.13 Indexing and searching objects

Because searching all objects for certain values of an attribute can be an expensive operation, Zope offers a solution to index and catalog objects according the the values of predetermined attributes and methods.

The catalog is a persistent object registered as a local utility for the `zope.app.catalog.interfaces.ICatalog` interface. Thus, the catalog can be retrieved like this:

```
>>> catalog = zope.component.getUtility(ICatalog)
```

In the catalog indexes are added (example in `learningjourney.app.event`), which records which interface and attribute it indexes. These indexes record, as a reference to the indexed object, an **intid** - an unique integer, guaranteed to be unique, assigned to all persistent objects by an intid utility registered for the `zope.app.intid.interfaces.IIntIds` interface.

The object indexing is performed usually when there are added or erased from containers (based on the `ObjectAddedEvent` and `ObjectRemovedEvent` event, and also when the `ObjectModifiedEvent` is triggered in code. Example:

```
>>> obj.title = u"My title"
>>> zope.event.notify(ObjectModifiedEvent(obj))
```

When indexing an object, each index from the catalog will try to index the object by adapting the object to the interface they're assigned, and retrieving the value for the attribute or method assigned.

An example of how to perform a search can be found in `learningjourney.ui.search.page`.

6.14 The layers and skinning system

The CMF libraries from Zope 2 define the concept of

- **layers**, where the resources and pages are declared
- **skins**, which group the layers to create a cohesive functionality and look for the website.

These concepts are present in Zope 3, but in a easy to implement, simplified form. A layer is determined by an interface that inherits `zope.publisher.browser.interfaces.IBrowserRequest`. Once a layer is defined, browser components (pages, views and resources) can be registered on that layer using the `layer` attribute in zcml. An example of layer is found in the __init__ module from `learningjourney.ui`.

A skin is a layer that has been assigned the `IBrowserSkinType` type, with the name that the skin will have. An example of a skin can be found in `learningjourney/ui/configure.zcml`.

To select the skin that is used, a special namespace traverser is used in the URL, like:

```
http://localhost:8080/++skin++lj/Application/@@index.html
```

For **LearningJourney** the `z3c.layer.minimal` package has been used as base layer. This layer is a minimal layer, with just the error pages defined. Having a minimal layer as the base means the pages that are published for a site can be fully controled.

6.15 Using macros in templates

Because Zope 3 doesn't directly publish templates and doesn't attach them to all types of context, by default, using macros in Zope 3 is a little more difficult.

The default method is to register pages with macros in a list of macro pages, available in a page called `standard_macros`. Their use can be observed in the `learningjourney.ui.site` package.

The Zope Community has developed an extension package called `z3c.macro` which offers new **TAL** expression and a zcml extension that make it very easy to register and use macros and their templates.

6.16 UI internationalization

At the level of templates, the internationalization is "classic", just like Zope 2, using the attributes from the i18n namespace. Also, in the ZCML files, a message domain can be specified for the titles and labels defined in that file.

In Python code, a `MessageIdFactory` is used to build new ids for messages. Look for an example in the `learningjourney.i18n` package, and its usage in `learningjourney.app.interfaces`. To change the language for the site, there's a namespace traverser, which can be used like this:

```
http://localhost:8080/++skin++lj/++lang++ro/Application/@@index.html
```

Another method of changing the user's language is by using the request in a traverser, using the `IBrowserPreferredLanguages` interface, thus being able to store the user language preference in a cookie or session.

6.17 Security, permissions

The security sistem in Zope is relatively complex, but powerful and flexible. At its base are the principals (objects that represent users) and interactions between users and the system. The **principal** objects are built on every request, usually based on the information extracted from the request. The global utility that generates **principal** objects is registered for the `zope.app.security.interfaces.IAuthentication` interface. The implementation offered by Zope is called **Pluggable Authentication Utility** (PAU), implemented in `zope.app.authentication`. It offers a system of plugins that can control how the authentication credentials are extracted from the request and where the users come from (the users source). An complete example of building an authentication system, complete with user source, can be found in `learningjourney.app.event`. Once the principal object has been created, it is stored in the request and can be used for various tasks: displaying the user id, checking if the curent user is authenticated, etc. You can see more of the authentication system in action in the `learningjourney.ui.authentication` package.

There are several ZCML directives that define types of principals (authenticated, unauthenticated, etc), which can be seen in the site.zcml fiel from the root of the application. At runtime the principals are build based on the information extracted from request and checked against the users source. Zope offers a source of users in the form of a container (`PrincipalFolder`), which stores objects of type `IInternalPrincipal`. Note: these users are information about principals, not principals in themselves. The source of users and the credentials extraction plugins are objects stored in PAU, which is configured with an explicit list of active plugins extraction and user source plugins. This type of configuration can be observed in `learningjourney.app.event`

Access to objects and their properties is protected by permissions, as could be noticed in the *Persistent objects* chapter. To ease the management of permissions, the concept of roles is used: to a principal, in a context, can be assigned a certain role. This role is granted permissions. To assign a role to a principal (user), the `IPrincipalRoleManager` adapter is used:

```
>>> IPrincipalRoleManager(context).assignRoleToPrincipal('lj.Owner',
                                                          request.principal.id)
```

Declaring new permissions, roles and granting permissions to roles can be seen in the `custom-security.zcml` file from the root of the app.

Chapter 7

Some recipes for tipical problems

7.1 Using marker interfaces to define capabilities

A marker interface is an “empty” interface, that usually inherits directly `zope.interface.Interface`. Using these interfaces, we can define capabilities and implement them in an abstract, reusable way. Declaring that a class implements the capability means that we can also benefit from a default implementation, which can be overriden if needed. Example:

Let’s assume that we want a reusable solution to add comments. Each comment will be stored in objects of type `IComment`, and these comments will be stored in object annotations. In a CMS environment, for what object types the *Add comment* button will be shown? To take as a clue the fact that the object is annotatable is not enough: not all annotatable objects need to be commentable. The solution is simple: we will create a marker interface:

```
class IHasComments(Interface):
    """This type of objects can be commented"""
```

The adapter used for the factory that generated the object addnotation can be:

```
class IComments(Interface):
    """Object comments"""

class Coments(BTreeContainer):
    implements(IComments)
    adapts(IHasComments)

    def add_comment(self, *args, **kwds):
        self[u"Comment %s" % len(self)] = Comment(*args, **kwds)
```

If we have an object of type BlogEntry, it is sufficient to mark it as being “commentable”:

```
class BlogEntry(Persistent):
    implements(IBlogEntry, IHasComments)
```

A viewlet that presents a fragment with the comments for the current object with the “Add comments” button, can be like this:

```
class CommentsViewlet(BaseViewlet):
    @property
    def available(self):
        return IHasComments.providedBy(self.context)
```

The code that adds the comments to objects can be like:

```
IComments(some_blog_entry).add_comment(...)
IComments(some_other_object_type).add_comment(...)
```

7.2 Object adnotation

Another concept introduced by Zope 3 with the purpose of maintaing object as "clean" as possible is the annotation. With the annotation we can attach additional data to objects. For example, let's suppose we have objects of type `Boardgame`:

```
class IBoardgame(Interface):
    name = zope.schema.TextLine(title=u"Game name"

class Boardgame(Persistent):
    implements(IBoardgame)

    name = u""
```

We want to attach objects with the dates when the game was used. This dates sit in objects of type `UsageInfo`:

```
class IUsage(Interface):
    date = Date(title=u"Date of usage")

class Usage(Persistent):
    date = None

    def __repr__(self):
        return "Used at %s" % self.date
```

The list will be implemented by annotating the `Boadgame` objects with information of type `IUsageInfo`:

```
class IUsageInfo(Interface):
    usages = List(  title=u"Dates when used",
                    value_type=Object(title=u"Usage",
                                      schema=IUsage)
                  )

class UsageInfo(Persistent):
    implements(IUsageInfo)
    adapts(IBoardgame)

    usages = None
```

Objects of type UsageInfo will be stored in the annotation for objects of type `IBoardgame`. The annotation itself is an adapter that is built using the `factory` function from `zope.annotation.factory`.

```
>>> from zope.annotation.factory import factory
>>> usage_info_annotation = factory(UsageInfo)
>>> zope.component.provideAdapter(usage_info_annotation)
```

Usually, the configuration of that last line is done in ZCML, like so:

```
<zope:adapter factory=".annotations.usage_info_annotation" />
```

Finally, using the annotation in code is simple:

```
>>> usage_info = IUsageInfo(some_boardgame)
>>> usage_info.usages.append(Usage(datetime.datetime.now()))
>>> print usage_info.usages
```

Adnotarea nu este ceva care sa fie in mod implicit asigurat tutoror obiectelor persistente. Pentru ca adnotarea sa fie disponibila pentru un obiect, acesta trebuie sa fie adaptabil la interfata `IAnnotations` (de exemplu, ar putea sa existe un adaptor care sa stocheze adnotarea obiectelor intr-un RDB). In cazul obiectelor persistente bazate pe ZODB, acestea pot fi facute usor adaptabile prin marcarea claselor acestora ca implementand interfata `zope.annotation.interfaces.IAttributeAnnotatable`. Exista un adapter care adapteaza acest tip de obiecte la interfata IAnnotations prin stocarea adnotarilor intr-un atribut `__annotations__`.

7.3 DublinCore

The `DublinCore` information is handled with the help of the `zope.dublincore` library. By default, the DC information is stored for all persistent annotatable objects. Because of this, to benefit from `DublinCore`, objects need just to be marked with their class as implementing `IAttributeAnnotatable`.

DublinCore represents a lot of attributes and information and because of this, if the default behaviour needs to be modified, the implementation process for this interface needs to be optimized. There are two ways of achieving this:

The first, we can declare an attribute as being a `DublinCore` property:

```
from zope.dublincore.property import DCProperty
class Book:
    implements(IBook)
    name = DCProperty("title")
    authors = DCProperty("creators")
```

Second way involves manually creating the DC adapter using a *factory* and in this case we can also specify a mapping between the DC fields and the object attributes.:

```
dc_annotation = partialAnnotatableAdapterFactory({
                                'name':'title',
                                'author':'creators'
                                })
```

The adapter is configured this way:

```
<zope:adapter
    for=".interfaces.IBook"
    factory=".annotations.dc_annotation"
    provides="zope.dublincore.interfaces.IZopeDublinCore"/>
```

7.4 Creating relationships between objects

Let's say we have a folder with images from racings, and we wish to assign a list of images to object of type Pilot, Team or Stadium:

```
class IHasImages(Interface):
   """marker, objects have pointers to images"""

class Pilot:
   implements(IHasImages, IAttributeAnnotatable)
   #it is wrong to have IHasImage inherit IAttributeAnnotatable
   #HasImages says something about behaviour, IAttributeAnnotatable is
   #already about how you implement the IAnnotations

class Team(Persistent):
   implements(IHasImages, IAttributeAnnotatable)

class Stadium(Persistent):
   implements(IHasImages, IAttributeAnnotatable)

class PicturesAlbum(BTreeContainer):
   contains(IImage)

class IPicturesInfo(Interface):
   images = List(title=u"Image",
                 value_type=Relation(title="Relation")
                 )

class PicturesInfoAnotation(Persistent):
    implements(IPicturesInfo)
    adapts(IHasImages)

    images = None

from zope.annotation.factory import factory
annotation_factory = factory(PicturesInfoAnotation, 'images_pointers')
```

The code that uses this annotation is simple:

```
IPicturesInfo(pilot_instance).images.append(img)
del IPicturesInfo(team_instance).images[somename]
```

7.5 Internationalising content with z3c.language.switch

The following implementation is based on a real implementation for a Zope 3 based portal. The queryAttribute method of the II18n interface was redefined to introduce a mechanism that allows getting a value (for the default language) instead of an empty value when there is no translation for that language/value.

```
class GameI18NInfo(Persistent):
    """The game content object"""

    name = FieldProperty(IGame['name'])
    description = FieldProperty(IGame['description'])
    promo_message = FieldProperty(IGame['promo_message'])


class Business(I18n, BTreeContainer, Contained):
    """ """
    _factory = BusinessI18NInfo
    _defaultLanguage = 'en'

    implements(IGame)

    name = I18nFieldProperty(IGame['name'])
    address = I18nFieldProperty(IGame['address'])
    description = I18nFieldProperty(IGame['description'])
    promo_message = I18nFieldProperty(IGame['promo_message'])

    def queryAttribute(self, name, language=None, default=None):
        #override so that we never return empty stuff
        value = super(Game, self).queryAttribute(name, language, default=None)
        if value is None:
            negotiator = getUtility(INegotiator, context=self)
            lang = negotiator.serverLanguage
            try:
                value = self.getAttribute(name, language=lang)
            except KeyError:
                pass
        if value is None:
            #try to return something meaningful
            langs = self.getAvailableLanguages()
            if langs:
                lang = langs[0]
            try:
                value = self.getAttribute(name, language=lang)
            except KeyError:
                pass
        if value is not None:
            return value
        else:
            return default
```

7.6 Changing the traversing with z3c.traverser

The following example uses a case where a series of images are stored in an folder that is set as annotation for objects of type `Game`. Because of the fact that the annotation is not directly traversable, we publish the images on the web as being in a *virtual folder* called *images* by changing the traversing mechanism for objects of type `Game`. The `z3c.traverser` introduces a plugin based mechanism that allows changing the traversing, per object. The first zcml registration changes the publisher for `IBusiness` object types, while the second registers the plugin for this publisher (this mechanism is detailed in the `z3c.traverser` documentation).

```
<view
  for=".interfaces.IGame"
  type="zope.publisher.interfaces.browser.IBrowserRequest"
  provides="zope.publisher.interfaces.browser.IBrowserPublisher"
  factory="z3c.traverser.browser.PluggableBrowserTraverser"
  permission="zope.Public">
  setup plugin traversal for the IGame
</view>

<subscriber
  for="lovely.reviewportal.app.interfaces.IGame
       zope.publisher.interfaces.browser.IBrowserRequest"
  provides="z3c.traverser.interfaces.ITraverserPlugin"
  factory=".traversing.GameTraverserPlugin" />
```

```
class GameTraverserPlugin(ContainerTraverserPlugin):
    """Traversing to games/images will return the annotation of
    GameImagesAlbum for the Games:
    """

    def publishTraverse(self, request, name):
        if name == "images":
            images = IGamesImagesAlbum(self.context)
            proxied_images = LocationProxy(images, container=self.context,
                                           name="images")
            return proxied_images
        if name == "promotions":
            promotions = IPromotions(self.context)
            proxied_promotions = LocationProxy(promotions,
                                               container=self.context,
                                               name=name)
            return proxied_promotions

        subob = self.context.get(name)
        if subob is None:
            raise NotFound(self.context, name, request)
        return subob
```

7.7 Advanced templating methods

In the case of a "classic" Zope 2 website, the way the templates need to be created is obvious: a template for the layout of the site, macros and slots to fill the templates and, maybe, the CMF based skin overriding mechanism. Plone is one example of application that has been using this mechanism succesfully.

With Zope 3, considering the amount of choices that can be made, this mechanism is not so clear. We will analyse several of these choices, in the context of a website for a multi-national company.

7.7.1 Inserting HTML content directly

This method of including content from another page (let's call it template for now) is similar to the `include()` from PHP. In Zope we have the advantage that the "template" is aware of the object to which it is applied. Example:

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

Simple, but with a few problems: on each page of the site will need to be copied the base structure of the website and then have the specific parts of the page inserted in clearly marked areas. For a site with more then just a few pages this method makes it difficult to change the basic structure of the site, because all the pages on the site will need to be updated.

7.7.2 METAL: macros and slots, just like classic Plone

The classic method of separating the layout of the site from the content of the page is to use the METAL extension, with macros and slots. For example:

First we have the template of the site, called `template.pt`.

```
<html metal:define-macro="page">
    <head metal:define-slot="header">
        <title>Some title</title>
    </head>
    <body metal:define-slot="body">
        Body content comes here
    </body>
</html>
```

Then we want to make the macro available. We will develop a page called `view_macros` and we will add it to the `page_macros` tuple of the view `standard_macros`. This view is a special browser view that implements the interface `zope.interface.common.mapping.IItemMapping` (see, for details `standard_macros.py` from `zope.app.basicskin` and `zope.app.rotterdam`). This view has a list of names of pages that contain macros and a list of aliases aliasuri between macros. Finally, the macro is included in the page:

```
<html metal:use-macro="context/@@standard_macros/page">
    <head metal:fill-slot="header">
        <title>MyTitle</title>
    </head>
    <body metal:fill-slot="body">
        Content here...
    </body>
</html>
```

Another method of "finding" the macros inside a template is to put a reference to the template with macros, from the page class:

```
class MainPage(BrowserPage):
    macros = ViewPageTemplate('/path/to/macros.pt')
```

Inside the template associated with MainPage, we can insert the macro with:

```
<div metal:use-macro="view/macros/some_macro">
```

7.7.3 z3c.macro: simplified registration of macros

Using z3c.macro, the registration of new macros becomes an easy task, that doesn't involve redefining or change a page class. For example, to register the `page` macro from `template.pt`, we need the following zcml:

```
<configure xmlns:z3c="http://namespaces.zope.org/z3c">
    <z3c:macro template="template.pt" name="page" />
</configure>
```

The macro is then inserted in pages with:

```
<html metal:use-macro="macro:page">
...
</html>
```

7.8 Content providers

Taking the example of the multinational company, let's suppose that we have a navigation menu for the site. If we develop the site just with macros, we woud write a macro that will be inserted in the main template. What if we want to change the navigation menu just for a few pages from the site. There are two solutions:

- insert a lot of logic in the macro, that will check all the special cases. Ugly, complicated, we want to avoid this case
- restructure the menu as a view that can be defined per context. This solution works in the case that we want to redefine the menu per type of context, but not if we want to modify the menu based on the page where it appears.

The solution offered by Zope 3 is solving the problems of the second point: define a new type of view that takes into account also the view where it is inserted: the **content provider**. Thanks to interfaces, it is easy to redefine the content of the provider based on type of context, request (skin) and page (interface or the class implementing the page). For example, in our site we could define the navigation menu that would be inserted on each page like this:

```
from zope.contentprovider.interfaces import IContentProvider
from zope.publisher.interfaces.browser import IDefaultBrowserLayer
from zope.publisher.interfaces.browser import IBrowserView

class MainSiteNavigation(object):
    implements(IContentProvider)
    adapts(Interface, IDefaultBrowserLayer, IBrowserView)

    def __init__(self, context, request, view):
        self.context = context
        self.request = request
        self.__parent__ = view
    def update(self):
        pass

    render = ViewPageTemplateFile('navigation.pt')
```

This content provider will be registerd with:

```
<adapter
    factory=".browser.MainSiteNavigation"
    name="main_site_navigation" />
```

To override the provider, for example for objects of type PressRelease, we will use:

```
class PressReleasesNavigation(object):
    adapts(IPressRelease, IDefaultBrowserLayer, IBrowserView)
    render = ViewPageTemplateFile('press_releases_navigation.pt')


<adapter
    factory=".browser.PressReleasesNavigation"
    name="main_site_navigation" />
```

This way, using the content providers allows breaking the web pages into reusable components. Using the registration of this multiadapter with a name, it can be inserted into templates using the expression `provider`:

```
<div tal:content="structure provider:lj.MyProvider" />
```

To generate the content of the content providers, a two phase rendering process is used. First the `update` method is called, then the `render` method is used to actually generate the content of that content provider.

7.8.1 Viewlets and viewlet managers

Viewlets represent a step forward in the direction taken by the content providers: a viewlet manager is actually a content provider that agregates “mini-views” and inserts them in pages. These mini-views are the viewlets that are defined as multiadapters for context, request (layer), view and the viewlet manager interface. Using the viewlet mechanism we can decouple the content from the template or the context where it will be inserted: we can control what “boxes” appear on each page by adding registration of viewlets, no longer needing to edit macros, templates or a lot of code.

The `zope.viewlet` package offers two new ZCML tags: `browser:viewletManager` and `browser:viewlet`. When the viewlet manager is registered, it is also possible to specify a class that will generate it and a template that will be used to render the viewlets, so it is possible to control how the viewlets are sorted and how are inserted into pages.

Although can be tempting, the idea of transforming an entire page and site in a structure based on viewlets is dangerous: forms in viewlets are hard to implements, pages will be hard to define and manage (there is no clear picture, in code, over what appears on the page). I recomended that at least the main part of the page to not be defined with viewlets, but with pagelets, as described lower.

A practical example:

- we define a marker interface for the viewlet manager:

  ```
  from zope.viewlet.interfaces import IViewletManagere
  class IExtraStuffBox(IViewletManager):
      '''Viewlets for the extra stuff box'''
  ```

- we register the viewlet manager:

  ```
  <browser:viewletManager
      name='zope3tutorial.ExtraStuffBox'
      provides='.interfaces.IExtraStuffBox'
      permission='zope.View'
      layer='.demoskin.IMySkin'
      />
  ```

- we insert this in the main template of the site:

  ```
  <div tal:replace="structure provider:zope3tutorial.ExtraStuffBox">
      A box for extra stuff
  </div>
  ```

- write a viewlet:

  ```
  >>> class SizeViewlet(object):
  ...
  ...     def __init__(self, context, request, view, manager):
  ...         self.__parent__ = view
  ...         self.context = context
  ...
  ...     def update(self):
  ...         pass
  ...
  ...     def render(self):
  ...         return size.interfaces.ISized(self.context).sizeForDisplay()
  ...
  >>> zope.component.provideAdapter(
  ...     SizeViewlet,
  ...     (IFile, IDefaultBrowserLayer,
  ...      zope.interface.Interface, interfaces.IViewletManager),
  ...     interfaces.IViewlet, name='size')
  ```

  In ZCML, the viewlet registration will be:

  ```
  <browser:viewlet
      name="size"
      for="IFile"
      manager="interfaces.IViewletManager"
      class="SizeViewlet"
      permission="zope.View"
      />
  ```

- The viewlet can be declared also just using a template

  ```
  <browser:viewlet
      name="fortune"
      for="*"
  ```

```
            manager='.interfaces.IExtraStuffBox'
            template='fortune.pt'
            layer='.demoskin.IMySkin'
            permission='zope.View'
            />
```

7.8.2 Separate the template registration from the view class with z3c.viewtemplate

Continuing with our study case, let's assume that this company has multiple websites, generated by the same Zope application, one for each country with the content being almost identical, but layouts and templates slightly different. In this case, the multiple skins applied to the base application could work, with the only problem being that it would require subclassing to redefine, per skin, the template used for a view class.

One of the possible solutions is provided by the z3c.viewtemplate package, which allows registering the template separately from the page and thus makes it possible to simply redefine the used template, per browser layer.

As an example, let's change the front page of one of the skins, to add a new column. We have the `MainSitePage` class for the front page, with the template `main_site.pt` and we wish to change the template. To benefit from the z3c.viewtemplate package, we will need to change the MainSitePage class to have something like:

```
class MainSitePage(object):
    template = RegisteredPageTemplate()

    def __call__(self):
        return self.template()
```

or we can simply inherit `BaseView` from `z3c.viewtemplate`:

```
class MainSitePage(BaseView):
    ...
```

Then we can register the template, per layer:

```
<browser:template
    for=".browser.MainSitePage"
    template="main_page.pt"
    layer=".SkinLayerOne" />
```

We can also override the templates for viewlets, if we use a superclass like:

```
class BaseViewlet(object):

    template = RegisteredPageTemplate()

    def render(self):
        return self.template()
```

In practice, the structure will look like:

- a main template for the site that will provide the layout and will insert the viewlet managers. This layout will be provided as macro called `page`
- the site pages will use the `page` macro and will insert their content in the macro slots

Using an inteligent inheritance mechanism, it is possible to reduce to a minimum the necesity to define new templates.

7.8.3 z3c.template, an improved version of z3c.viewtemplate

z3c.template is a package similar to z3c.viewtemplate (allows separating the page code from the template registration), but also allows separating the layout of a page from its content and "main" template.

Let's assume that we implement a website with z3c.template and we have a page for press releases. For each page we will have a template for the layout and one for the content, but we can skin defining the template for the layout if we inherit a base class. The layout template will contain:

```
<html>
<head>
    <title tal:content="view/title" />
<head>
<body>
    <div tal:replace="view/render" />
</body>
</html>
```

This template will be registered:

```
<configure xmlns:z3c="http://namespaces.zope.org/z3c">
    <z3c:layout template="main_template" for=".interfaces.ISitePage" />
</configure>
```

We need a browser view that will know how to use the layout template and the content template:

```
class SitePage(BrowserPage):
    zope.interface.implements(ISitePage)

    template = None
    layout = None

    title = None

    def update(self):
        pass

    def render(self):
        if self.template is None:
            template = zope.component.getMultiAdapter(
                     (self, self.request), IContentTemplate)
            return template(self)
        return self.template()

    def __call__(self):
        self.update()
        if self.layout is None:
            layout = zope.component.getMultiAdapter((self, self.request),
                         interfaces.ILayoutTemplate)
            return layout(self)
        return self.layout()
```

Our page class for the press releases will inherit the `SitePage` class:

```
class PressReleaseViewPage(SitePage):

    @property
    def title(self):
        return u"Press release: " + self.context.title
```

We can then inherit the template for content and page:

```
<configure xmlns:z3c="http://namespaces.zope.org/z3c">
    <z3c:template template="press_review_view.pt" for="IPressReview" />
</configure>
```

Although the mechanism is easy to implement and understand, it involves some work in supporting forms, and why write the `SitePage` class, when there is a package that already has this? This is...

7.8.4 z3c.pagelet

This package introduces a new type of browser page: the pagelet. A pagelet is a page with a separate template for layout: the layout is defined using the mechanisms introduced by `z3c.template`, but the `SitePage` class is no longer necessary as it is provided by the package. Inside the layout template, where the real content needs to be inserted, the pagelet provider is inserted:

```
<div tal:replace="structure provider: pagelet" />
```

Other “goodies” included in the package:

- integration with the form classes from `zope.formlib`
- using z3c.skin.pagelet we have a base skin that can be used as a start for a pagelet based website, including all the exceptions as pagelets

To register a pagelet the following ZCML is used (notice the similarity with the page registration):

```
<z3c:pagelet
    name="index.html"
    for=".interfaces.PressRelease"
    class=".views.IndexPagelet"
    layer=".interfaces.ICompanyWebsiteLayer"
    permission="zope.View"
    />
```

7.8.5 Other packages useful in templating

- z3c.pt: an implementation of ZPT templates that is faster (by ~ 10 times)

- z3c.macroviewlet: define macros in templates as viewlets, allowing easy registration of a website that is completely made from viewlets
- z3c.formui: integrate the pagelets with the z3c.form library

7.9 Some tips

- the Zope source code is simple and easy to understand. If there is no documentation, read the source code, starting with the interfaces and then the configure.zcml file, to undersand what that package does. Use the APIDOC and Zope book tools to understand the documentation. Use the introspection page for object to understand their structure.
- use namespaces
- separate the "backend" part from the "frontend" part in different sub-packages: *app* and one of *browser*, *skin* or *ui*.
- keep a balance between the necesity to separate into packages and a number too high of packages, with deep interdependencies
- use the Zope Community packages (z3c), they are often newer and more flexible then the ones that Zope has. Example: z3c.form, z3c.pagelet, z3c.table, etc.
- when you design a library or reusable components, it is very easy to create an infrastructure based on the ZCA, with multiple insertion points. A balance needs to be maintained, to prevent the package from imposing a too high requirement when the library is used (for example: implement an adapter for this interface, another utility for that interface, etc). It can happen that, when the developer wants to use that component, to be easier to completely override the entire component. Make sure that you ensure one unique "override" point, that would make it possible to override the entire mechanism. But:
- try to make the components generally available, to be reusable. On the long term, this is one of the advantages of using Zope 3: components are easy to develop so that they are reusable
- it's not necessary to have a big level of integration with ZMI
- don't start a big application as your first on Zope 3, start with a small project first

Chapter 8

The Zope 3 ecosystem

The size of the Zope 3 extension packages ecosystem is a clear proof of the platform vitality. Even more, because of the WSGI compatibility, it is possible to extend this ecosystem to include web development libraries that have no apparent connection with Zope. There are 3 large actors in this ecosystem: the Zope Corporation, the Zope 3 Community (o non-formal community, responsible for the creation of a big part of Zope 3) and the Plone developer community - although their Zope 3 packages that are fully compatible with Zope 3, with no Plone dependencies, are smaller in numbers.

8.1 The Zope packages

The libraries that are most likely to be used in a Zope 3 application are:

- zope.annotation
- zope.contentprovider
- zope.viewlet
- zope.copypastemove
- zope.event
- zope.securitypolicy
- zope.formlib
- zope.interface
- zope.schema
- zope.app.catalog
- zope.app.container

8.1.1 Packages from the zope.* namaspace

zope.annotation allow attaching data to objects. Offers an implementation which stores this data in a special attribute of the objects

zope.browser contains a few interfaces used in other packages

zope.cachedescriptors allows memoizing attributes

zope.component implements ZCA. Offers adapters, utilities, subscribers, handlers. In implementation are used registers in which components are registered, ofering also the posibility to create local registries, that will have priority before the global registries (using the sites mechanism)

zope.configuration defines an extensible configuration system. Implements the bases of ZCML

zope.contentprovider provides the posibility to componentize the page structure with dynamic pieces of content, which can be reused and connected to the type of context, request and page unto which they are inserted

zope.contenttype utilities to determine the type of files, extends the mimetypes standard module

zope.copypastemove support for copy, paste and move operations for objects. Generates events when those operations are executed

zope.datetime definitions and utilities to manipulate time and date objects

zope.deferredimport support for optimizing python import operations, which helps in faster loading of applications

zope.deprecation defines and API useful in marking modules or functions as deprecated

zope.documenttemplate a templating engine for the DTML syntax

zope.dottedname allows solving Python objects using dotted names

zope.dublinecore general implementation of DublinCore, offers an implementation that stores that data in annotation

zope.error global implementation of local and global utilities for error reporting

zope.event implements a notification system using a subscribers system that defines handlers for those events

zope.exception contains definitions for basic exceptions and utilities to format exceptions

zope.filereprezentation defines interfaces used in representing objects through various protocols, such as WebDav or FTP

zope.formlib a general form library that will generate, validate and automatically process web forms

zope.hookable makes it possible to have global, explicit, monkey-patching at runtime

zope.i18n base implementation of internationalization in zope, implements translation domains, message catalogs, a zcml extension to register message catalogs and has utilities for internationalization of messages such as money, dates, numbers, etc.

zope.i18nmessage implements internationalizable messages

zope.index implements indexes as BTree structures, optimized to index several types of data: fields with values, text and list of values. These indexes are used in the implementation of the catalog

zope.interface the base library in Zope 3, allows defining, using and querying objects and the interfaces that they provide. Represents one of the important pieces in ZCA

zope.lifecycleevent defines a series of events that an object can be involved (creating, erasing, changing, adding in a container)

zope.location defines a method of physically locating objects by using their name and their parent, also has utilities to locate them. Defines utilities to place objects in a particular location in the objects tree by proxying them with LocationProxy. This proxy is very important as it allows publishing non-persistent objects through the web, and also allows creating URL patterns that don't match the physical structure of the database

zope.minmax defines a method of solving conflicts from the MVCC infrastructure using values that come from a policy that favors minimal or maximal values of objects involved in the conflict

zope.pagetemplate implements pages that use template with TAL syntax implementeaza pagini ce folosesc template-uri cu sintaxa TAL

zope.proxy a C based implementation of proxies. Is used by the other libraries to implement location and security proxies

zope.publisher implements publishing mechanisms for content through http, ftp and xmlrpc protocols, defines the involved components (request, response, skins, internationalization, etc) and has WSGI support for them

zope.rdb integrates with relational databases by creating and maintaing global connections with them

zope.schema extends zope.interface to implement specific types of attributes (for example, numbers, lines of text, list of values, etc). Here are defined the vocabularies - utilities that return a list of values to display options in user interfaces. One of the base packages for Zope when it comes to implementing user interfaces

zope.security defines a security system tha uses principals and permission to restrict access to objects and their attributes. Contains integration with ZCML for permission definitions and security policies

zope.securitypolicy defines Zope's security policy, which extends the security infrastructure with roles and group, to achieve a greater flexibility. Defines the way roles and permissions are stored on objects using maps and security managers. Contains integration with ZCML for role definitions and granting permissions to roles and principals.

zope.sendmail defines global utilities that can be used to send emails (synchronous, queued) and has zcml integration to define them

zope.sequencesort utilities to sort lists

zope.server contains an implementation for ftp and http servers. ZServer (this implementation) is one of the fastest Python HTTP servers

zope.session implements session and utilities to identify clients

zope.size defines the way size information is extracted from objects, contains a basic implementation of this

zope.structuredtext an engine to change Structured Text into HTML

zope.tal an implementation of TAL templating syntax

zope.tales contains TAL extensions to introduce new types of expresions to traverse objects

zope.testbrowser contains an http browser that can be programmed, useful in defining functional texts

zope.testing contains utilities that can be used in defining tests for Zope application and packages. Contains support for unit tests and integration tests

zope.testrecorder contains a web proxy that can be used in recording the http communication, useful in writing integration tests

zope.thread defiens a way to create and manage per thread local objects. This is a simple extension of the thread standard module

zope.traversing defines utilities and traversing namespaces, also has utilities to compute absolute URLs for objects, taking into consideratio the application's virtual hosting

zope.viewlet extends the concept of content provider with that of viewlet and viewlet managers, to allow defining an area where viewlets can be inserted, which are dependent on the context type and the area where they are inserted. Defines, also, several standard viewlet managers

zope.xmlpickle pickle based serialization to and from xml

8.1.2 Packages from the zope.app namespace

zope.app.apidoc makes it possible to introspect objects with an autogenerated page that presents the provided interfaces, attributes and documentation. Offers a ZCML extension to register documentation for the so-called "Zope Book"

zope.app.aplicationcontrol Offers the possibility to control the application by the user (stop/restart/ database pack, etc)

zope.app.appsetup a way to configure and build Zope applications

zope.app.authentication a complex system, based on plugins, to extract authentication data and to authenticate users (including the source of users)

zope.app.basicskin a simple skeleton skin

zope.app.boston a complete skin, similar to Rotterdam, based on viewlets

zope.app.broken integrates broken objects from the database so that they can be viewed and debugged

zope.app.cache a global utility that can store data for caching purposes

zope.app.catalog a complete indexing and searching solution for objects. Has two types of indexes, based on zope.index: text and field

zope.app.component Extensions of zope.component. Integrates the various Zope components (views, layers, security, utilities, etc) and defines zcml extensions for them.

zope.app.container A BTree based implementation of containers, defines how they are traversed and also has support for preconditions (what types of objects can be added, etc)

zope.app.content Defines the IContentType interface, which can be used to mark other interfaces as having that type. Defines a vocabulary that lists these interfaces.

zope.app.dav Has a webdav server integrated with the rest of Zope infrastructure, has integration with ZCML

zope.app.debug has a debugger that can be started from the application controller script

zope.app.dependent provides a method to mark objects in the graph as being dependent on each other

zope.app.dtmlpage has a content type based on DTML templates

zope.app.exception pages for exceptions, generating the equivalent in HTTP exceptions

zope.app.external minimal integration for what is needed to build an external editor

zope.app.file two new types of content: File and Image, also with integration for their external representation (IFileRepresentation)

zope.app.folder offers a folder content type, integrated with the management interface and its representation as external object (IFileRepresentation)

zope.app.form the old form library for Zope, it is considered outdated because of the dependency on ZCML and inflexibility. All these considered, the widget model is also used by zope.formlib and thus remains an important library

zope.app.ftp a view model for FTP

zope.app.generations migrationi for database when the persistent objects schema is changed

zope.app.homefolder has a homefolder implementation for users

zope.app.http integration with the http protocol: exceptions, methods, traversing methods and how views are solved

zope.app.i18n implements a persistent i18n messages catalog and ZCML extensions to register translation folders

zope.app.i18nfile internationalizable images and files (as content)

zope.app.interface a vocabulary for interfaces that an object provides

zope.app.interpretor an untrusted Python code interpretor

zope.app.intid a utility that associates unique ids to objects from the database

zope.app.keyreference an adapter for persistent objects that extracts their low level id from the database, based on which an intid can be associated

zope.app.locales contains translation and utilities for Zope 3 software

zope.app.locking an implementation for object access locking, to be used, for example, in WebDav

zope.app.module persistent Python modules (Python code in ZODB)

zope.app.onlinehelp an infrastructure for help and documentation and a ZCML extension that makes it possible to register help by any extension package

zope.app.pagetemplate integrates ZPT templates with the view classes, offers a ZCML expression that makes it possible to register new types of ZPT expressions

zope.app.preference implements a method of storing preferences for each user and contains integration with ZCML for recording groups of preferences

zope.app.preview a simple way of previewing the site in an iframe

zope.app.principalannotation a method to associated data with principals, through a persistent annotation

zope.app.publication integrated methods of content publishing, has a ZCML extension to define new methods of publishing, for each HTTP verb used

zope.app.publisher implements objects/pages that can be used in publishing content. Implements ZCML extensions to register pages and menus in ZCML

zope.app.pythonpage implements a type of persistent content that can interpret Python

zope.app.renderer a library useful in transforming from one type of content into another one (for example, from rest or stx in html)

zope.app.rotterdam the most used Zope skin, implements a complex management interface

zope.app.schema integrates zope.schema with the security infrastructure, has a registry for vocabularies

zope.app.security implements the security infrastructure of Zope, has base methods to authenticate users and generate principals, offers zcml extensions to declare global principals and groups

zope.app.securitypolicy implements pages that can be used to make local changes (in the database) of the security settings

zope.app.server implements http and wsgi servers, makes it possible to configure and to control them

zope.app.session has support for session through association of data with a client

zope.app.sqlscript implements content objects that can execute sql commands

zope.app.testing has utilities to create tests

zope.app.tree implements a tree widget that can be used in user interfaces to represent graph of objects

zope.app.twisted implements a server based on Twisted with configuration and control

zope.app.undo implements pages to control and execute undo steps

zope.app.winservice an service integrated with WinNT based operating systems

zope.app.workflow a method of workflow and support for persistent definitions of workflows

zope.app.wsgi a WSGI application, used in building WSGI based Zope applications

zope.app.xmlrpcintrospection offers introspection for the XMLRPC module, by implementing a few new methods

zope.app.zapi groups some of the most used functions, for the convenience of developers

zope.app.zcmlfiles a few zcml files that group, by theme, the Zope packages, allowing selective loading

zope.app.zopeappgenerations migrates the database from older versions of Zope

zope.app.zopetop a Zope skin, not very popular as it is an older version

zope.app.zptpage implements persistent templates in the database

8.2 Other packages

There are aproximatively 300 extension packages on svn.zope.org, not including the namespaces zope.* and zope.app.*. Also, the Zope community is the biggest producer of egg packages (following the PyPI statistics). Among those packages, there are solutions for:

handling big files storages z3c.extfile, z3c.blobfile

reusing views through template customization z3c.template

separate the content of a view from its layout z3c.layout, z3c.pagelet

advanced, extensible forms z3c.form

memoizing (caching) views lovely.responsecache

headers for caching z3c.caching z3c.responseheaders z3ext.cacheheaders

ETAG support z3c.conditionalviews

various fields and widgets z3c.schema.* z3c.widget.*

integration with RDBs STORM zope.sqlalchemy, z3c.saconfig, ore.alchemist

automatic inclusion of resources (CSS, JS) zc.resourcelibrary z3c.resourceinclude

resource concatanation z3c.resourcecollector

local ZCA configuration z3c.baseregistry

faster templating engine z3c.pt

workflow hurry.workflow

language negociation, content internationalization z3c.language.*

configuration framework, specific to each package z3c.configurator

application development zc.buildout, pb.recipes.pydev (Eclipse integration for buildout)

execution of asynchronous tasks lovely.remotetask, zc.async

session sharing for ZEO clients lovely.session

flexible generation of tables and listings z3c.table, zc.table

greater control over the cataloging and indexing process z3c.indexer

8.3 Other web resources

Zope 3 wiki Although doesn't seem too atractive, the Zope 3 wiki represents, at this moment, the official site and is a good place to find pointers to other Zope 3 sites. Location://wiki.zope.org/zope3

The new zope.org Although it is not yet officially launched, the new website has already enough new and intersting pages (for example, the Getting Started Guide, updated to the latest method of installing Zope 3). Location: http://new.zope.org/

A Comprehensive Guide to Zope Component Architecture A document filles with examples and explanation of the Zope Component Architecture, configuration with zcml and the way to use in any Python application. Location: http://www.muthukadan.net/docs/zca.html

3rd party packages reference guide A guide/reference of extension pacakges from svn.zope.org. Location: http://wiki.zope.org/zope3/Zope3PackageGuide

What's new in Zope 3.3 A document that discusses the changes brought by Zope 3.3 (when the component simplification took place). Useful for a short presentation of skins, vocabularies and factories. Location: http://kpug.zwiki.org/WhatIsNewInZope33

Zope 3 Book Although seems old (it was written before the component architecure simplification), a good deal of what is there is still true, or can be easily adjusted to today's code. Location: http://wiki.zope.org/zope3/Zope3Book

Worldcookery The Downloads section contains code licensed under GPL. This code, in the absence of the book (which is highly recommended), helps in understanding Zope. Location: http://worldcookery.com/Downloads

Documentation for z3c.form z3c.form includes its documentation in its source code. An HTML version is available at http://www.carduner.net/docs/z3c.form/

Chapter 9

The future of the Zope 3 platform

The main purpose of the Zope 3 platform, that to build a component based web framework, has been achieved. Zope right now is built on mature libraries, hardened by years of deployment in various online applications. They are stable and will not require any extensive changes in the near future. The developers that use this platform are experienced developers, usually involved in complex projects that drive the development of new conex libraries or refining the existing ones. The number of developers using the Zope 3 platform is rising, thanks to:

- migrating to Zope 3 of some popular systems such as Plone
- the Repoze initiative, with its offer of Zope libraries repackaged and ready to use in any other web framework
- new CMS projects (z3ext, vudo, hivurt, etc), which run of the Zope libraries

9.1 New development directions

Although the core is considered to be complete, the Zope Community keeps improving the platform. Several areas are considered to be essential to be on the roadmap:

- Decrease the level of interdependencies between the code Zope packages. This will lead to a better usage of resources (less RAM, etc) and the complexity as it is apparent to the developers
- Promote alternative solutions to those offered by Zope by default: compatibility with WSGI, integration with RDBs through ORMs, etc.
- Lowering the entry levels for new developers, through the Grok platform, which is a simple and powerful way to developer web applications on the Zope 3 platform
- Repoze.bfg, a new independent webdevelopment framework that is based on libraries from Zope 3, WebOb and WSGI, has a strong integration of Zope 3 concepts, but with the intended purpose of reducing the interdependence among components. It wants to be agnostic about the choices and offer multiple paths to solving some key problems (such as persistency, template engine, etc).
- the **chameleon** templating engine has a better performance and offers several syntax implementations (ZPT, Genshi) and integration with some of the key Zope and Zope Community packages.